Горобец Андрей Владимирович

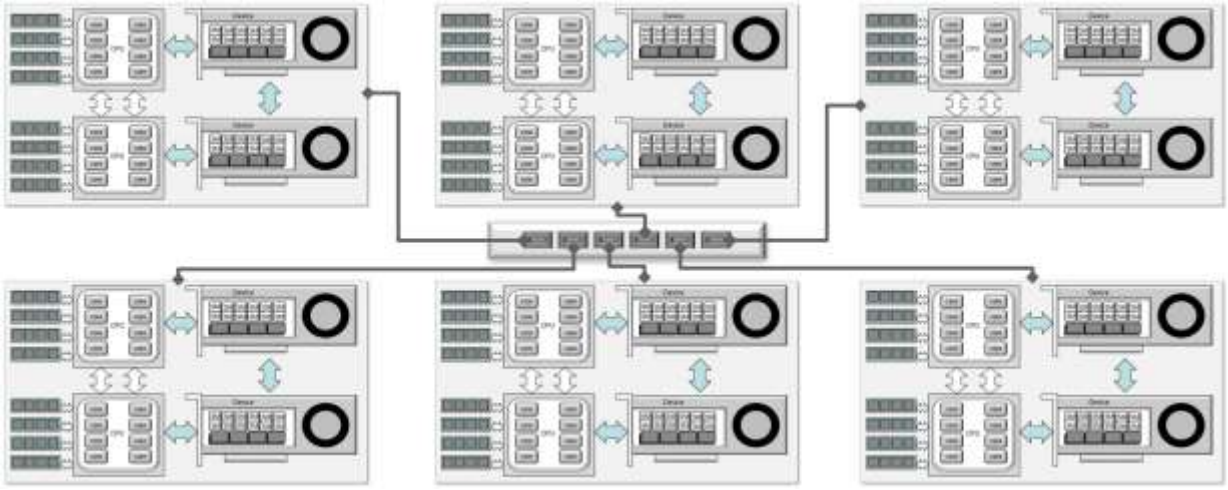
# Параллельные методы решения задач

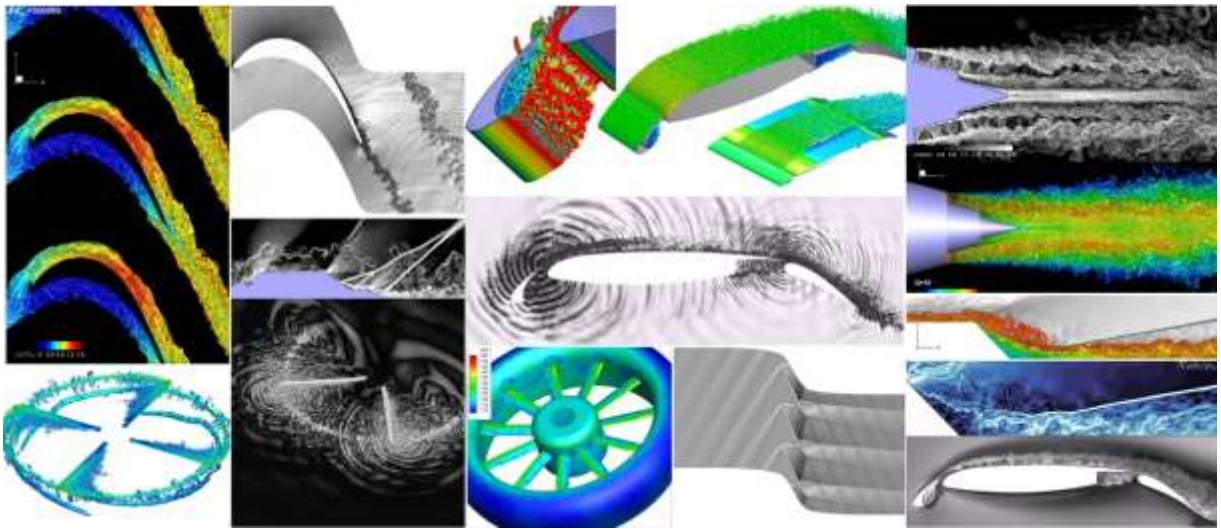
Учебный курс для студентов магистратуры
факультета Вычислительной математики и кибернетики
Московского государственного университета имени М. В. Ломоносова.

Курс лекций подготовлен при поддержке Московского центра фундаментальной и прикладной математики

## Аннотация

Данный курс посвящен проблемам параллельной реализации вычислительных алгоритмов с учетом особенностей архитектуры современных высокопроизводительных систем. Изучается многоуровневая модель параллельных вычислений, сочетающая распараллеливание с распределенной и общей памятью, парадигму потоковой обработки, векторизацию. Изучаются способы повышения производительности вычислений на многоядерных центральных процессорах и массивно-параллельных ускорителях, таких как графические процессоры. Параллельные технологии демонстрируются в основном применительно к ресурсоемким сеточным методам для предсказательного суперкомпьютерного моделирования. Рассматривается ряд репрезентативных алгоритмов и программ для решения вычислительных задач, связанных с вычислительно-емкой обработкой больших массивов данных. Изучаются современные технологии параллельного программирования, формируются навыки разработки параллельного программного обеспечения, а также выполнения расчетов на суперкомпьютере.





# Предисловие

Этот курс про параллельные высокопроизводительные вычисления в сфере предсказательного суперкомпьютерного моделирования. То есть, про обработку большого объема данных, которая в основном заключается в выполнении с этими данными арифметических операций с числами с плавающей точкой. Это такие приложения, для которых важна скорость работы программы. Программа обычно выполняется на сотнях, тысячах, а то и десятках тысяч процессорных ядер. Такие расчеты расходуют много, очень много часов процессорного времени. Часы эти стоят много, очень много денег. А также расходуют много электроэнергии. Поэтому важно, чтобы программа работала как можно быстрее и использовала вычислительные ресурсы эффективно. Курс как раз об этом.

Данный материал предназначен для студентов магистратуры, обучающихся по специальности Прикладная математика и информатика, а также для всех других студентов, аспирантов и вольнонаемных разработчиков по этой и смежным специальностям.

Предполагается, во-первых, что читатель владеет языками программирования C и/или C++, знает хотя бы в общих чертах основные принципы распараллеливания с распределенной и общей памятью, знаком со стандартами MPI и OpenMP. Во-вторых, предполагается, что читатель владеет английским языком и умеет обращаться с поисковыми системами в сети интернет. Соответственно, чем меньше уровень знаний по первому пункту, тем больше потребуется мастерство по второму.

Также предполагается, что у читателя имеет место дефицит времени. Являясь студентом магистратуры, он наверняка вынужден вступить в противоборство с множеством учебных курсов, входящих в программу обучения, а также выполнять научную работу на спецсеминаре. Тактично умолчим о том, что делать все это читателю приходится в свободное от своего основного занятия время (коим, конечно же, является работа на частичную или полную занятость в сфере информационных технологий). Поэтому сокращение времени на передачу данных из пособия в читателя представляется актуальным. Для ускорения понимания будут обильно использоваться картинки, рисунки, графики, диаграммы и прочие графические образы. При этом картинки, а также таблицы, не будут иметь номеров и подписей, а будут располагаться ровно по месту использования в тексте. Это сокращает объем и ускоряет загрузку данных в церебральное процессорное устройство (ЦПУ) читателя. Такая же штука с ссылками на источники. Они в основном будут приводиться в тексте по месту использования, не требуя от читателя лонг-джампа к списку литературы. Будут ссылки на ресурсы в сети Интернет, с которыми дело такое, сегодня они есть, а завтра нет. Если ссылка уже “протухла”, придется использовать поисковые системы.

Поскольку магистратура обычно является заключительным этапом подготовки начинающего специалиста к профессиональной карьере, важно развивать навыки и рефлексы, характерные работе в IT сфере. Поэтому в данном материале могут использоваться разные жаргонизмы и метафорические выражения. Куда уж в IT без профессионального жаргона? В тексте будут использоваться различные англоязычные термины, названия, зачастую без перевода на русский. Предполагается, что владение английским хотя бы на уровне чтения – это must для такой IT элиты, как вы, уважаемые читатели.

В курсе присутствуют примеры исходного кода и практические задания. Если читатель имеет цель прокачать навыки по тематике курса, а не просто избежать с минимальными усилиями отчисления из вуза, то к выполнению практических заданий стоит отнестись

ответственно и с энтузиазмом. Сами понимаете, без отработки на практике знания не заходят. Это как осваивать приемы рукопашного боя по видео в интернете, сидя на диване. Бесполезно.

Также в рамках данного курса хотелось бы устранить у читателя хотя бы небольшую часть тех типичных ошибок, которые часто совершаются на начальном этапе карьеры, от которых у работодателя может в глубине души возникать желание приложить сотрудника лицом об клавиатуру, причитая: "Ну как так!? Чему вас там учили в ваших университетах!?" Хотя есть небезосновательные опасения, что получится ровно наоборот…

# ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

Не знаю, зачем оно тут нужно, все расшифровывается в тексте. Но, говорят, так принято.

| | |
|---|---|
| AGU | Address Generation Unit |
| AKA | Also Known As |
| ALU | Arithmetic and Logic Unit |
| AOS | Array Of Structures |
| API | Application Programming Interface |
| CAD | Computer-Aided Design |
| CAS | Column Address Strobe |
| ccNUMA | cache-coherent NUMA |
| CFD | Computational Fluid Dynamics |
| CG | Conjugate Gradients |
| CPI | Cycles Per Instruction |
| CPU | Central Processing Unit |
| CSR | Compressed Sparse Row |
| CU | Compute Unit |
| CUDA | Compute Unified Device Architecture |
| DDR | Double Data Rate |
| DM-MIMD | Distributed Memory MIMD |
| DRAM | Dynamic RAM |
| ECC | Error-Correcting Code |
| FLOP | FLoating point OPeration |
| FLOPS | FLoating point Operations Per Second |
| FMA | Fused Multiply-Add |
| FP FMA | Floating Point FMA |
| FPGA | Field-Programmable Gate Array |
| FPU | Floating Point Unit |
| GDDR | Graphics Double Data Rate |
| GP GPU | General Purpose GPU |
| GPU | Graphics Processing Unit |
| HBM | High-Bandwidth Memory |
| HMC | Hybrid Memory Cube |
| HPC | High Performance Computing |
| HPCG | High Performance CG |
| HPL | High Performance Linpack |
| ILP | Instruction-Level Parallelism |
| IPC | Instructions Per Cycle |
| IPS | Instructions Per Second |
| LLC | Last Level Cache |
| LSU | Load-Store Unit |
| LU | Lower-Upper decomposition |
| MCDRAM | Multi-Channel DRAM |
| MIMD | Multiple Instruction streams, Multiple Data streams |

| | |
|---|---|
| MISD | Multiple Instruction streams, Single Data stream |
| MPI | Message Passing Interface |
| NaN | Not a Number |
| NAS | Network Attached Storage |
| NUMA | Non-Uniform Memory Access |
| OoO | Out-of-Order execution |
| QA | Quality Assurance |
| RAM | Random Access Memory |
| RAS | Row Address Strobe |
| RISC | Reduced Instruction Set Computer |
| SCP | Secure Copy Protocol |
| SDR | Single Data Rate |
| SDRAM | Synchronous DRAM |
| SIMD | Single Instruction stream, Multiple Data streams |
| SISD | Single Instruction stream, Single Data stream |
| SM | Stream Multiprocessor |
| SM-MIMD | Shared memory MIMD |
| SMT | Simultaneous MultiThreading |
| SOA | Structure Of Arrays |
| SRAM | Static RAM |
| SSH | Secure Shell |
| TBP | Theoretical Bounded Performance |
| TPP | Theoretical Peak Performance |
| UI | User Interface |
| URS | Unified Reservation Station |
| VLIW | Very Long Instruction Word |
| К.О. | Капитан Очевидность |
| ОС | Операционная Система |
| ПЛИС | Программируемая Логическая Интегральная Схема |
| СЛАУ | Система Линейных Алгебраических Уравнений |
| ЦКП | Центр Коллективного Пользования |

# СОДЕРЖАНИЕ

# 1 Введение

## 1.1 Эволюция суперкомпьютерной архитектуры

Чтобы не нарушать традиции, начнем со статистики. Производительность суперкомпьютеров (все еще) интенсивно растет. За последние десять лет, с 2010 по 2020 год, пиковая производительность самого мощного суперкомпьютера в рейтинге Top 500 увеличилась в 220 раз. Описания топовых систем и подробная статистика доступны по ссылке https://top500.org.

Но за счет чего происходит этот рост производительности, какая у этой медали обратная сторона? Мощностей все больше, но пользоваться HPC (High Performance Computing) системами становится сложнее. Упомянутый выше рейтинг строится на основе результатов по бенчмарку HPL (High Performance Linpack). В этом тесте производительности решается (очень) большая системка линейных уравнений с плотной матрицей параллельным прямым методом на основе LU (Lower-Upper) разложения. Объем данных $O(N^2)$, вычислительная сложность $O(N^3)$. Конечно, при таком соотношении объема данных и вычислений, когда вычислений сильно больше, есть над чем поработать, что пооптимизировать. За счет достижения высокой локальности доступа к данным удается снимать аж под 90% от пика. Но такой тест слишком "удобен" для матчасти. Гораздо удобнее, чем абсолютно подавляющее большинство реальных приложений. Из-за такого удаления от реальности в качестве альтернативы, точнее, полной противоположности, появился бенчмарк HPCG (High Performance Conjugate Gradients). Этот тест даже, пожалуй, излишне жестокий по отношению к матчасти. Тут система с разреженной матрицей решается итерационным методом, для чего используется компот из метода сопряженных градиентов, многосеточного метода, симметричного метода Гаусса-Зейделя. Подробнее тут: https://www.hpcg-benchmark.org/ Результаты там, конечно, печальные. Гордо и с большим трудом достигаются значения в 1–2% от пика. Это куда ближе к реальности. Но как же мы дошли до жизни такой?

А было оно примерно так. Сначала побеждали силы добра. Бурно росла производительность процессоров за счет повышения тактовой частоты. Десятки МГц, сотни МГц, несколько ГГц. Это светлая сторона силы. Далее рост закончился, поскольку слишком уж сильно растет энергопотребление с дальнейшим увеличением частоты. Стала расти производительность процессора при фиксированной частоте. Процессор за такт стал делать все больше и больше операций за счет параллелизма уровня инструкций. Размножился конвейер команд, появились multiple issue pipeline и прочая суперскалярность (с этими словечками мы далее будем разбираться подробнее). Затем обильно пророс out-of-order – внеочередное исполнение инструкций, в процессорах появились более сложные планировщики (scheduler) типа многопортовых reservation station. Процессор научился на одном такте задействовать множество разнородных исполнительных устройств. Очень кстати подросла пропускная способность оперативной памяти. В итоге производительность заметно выросла и без роста частоты. Вроде бы всё хорошо. Светлая сторона эволюции архитектуры. Сейчас начнется темная сторона.

Производительность многопроцессорной системы стала расти не столько за счет увеличения производительности процессора, сколько за счет многократного увеличения их количества. Рост числа процессоров в системе усложняет ее эффективное использование: требуются все более масштабируемые алгоритмы, изощренные схемы обменов данными,

упрощение алгоритмов в пользу параллельной эффективности, но зачастую в ущерб общей вычислительной стоимости.

Дальше процессоры стали многоядерными – multicore. Сначала двухъядерные, потом четырёхъядерные, и понеслось. С мультикорами еще можно было иметь дело, используя те же подходы к распараллеливанию с распределенной памятью, как и в случае одноядерных процессоров. Небольшие доработки обмена данными между параллельными процессами решали проблему. Затем процессоры стали существенно-многоядерные – manycore. Это когда в одном процессоре уже десятки ядер. Это все усложнило. Потребовалось сочетать распараллеливание с распределенной и общей памятью (с этими всеми распараллеливаниями мы далее подробно разберемся). Пришлось применять многоуровневое распараллеливание, усложнять балансировку, оптимизировать доступ к памяти в многопоточном режиме, учитывать NUMA (Non-Uniform Memory Access) фактор, и решать много других проблем.

С переходом на мэникоры число процессорных ядер в суперкомпьютерах дико выросло. В крупных системах это число давно перевалило за миллион. Естественно, лишь очень узкий класс алгоритмов может задействовать такие ресурсы, а сами задачи, которые такие алгоритмы могут решать, часто даже и близко не оправдывают использование таких вычислительных мощностей. Конечно, нас никто и не заставляет использовать такие машинки целиком. Это крупные системы коллективного доступа с большим количеством пользователей. Нас вполне устроит эффективно использовать хотя бы их малую часть. Но число процессоров и в этой "хотя бы их малой части" становится все больше и больше. Темная сторона набирает силу.

Началась гонка производительности. Захотелось делать устройства со все более высокой пиковой производительностью. Но как? Частота не растет. Out-of-order уже нехило прокачали, навороченные планировщики в ядре и так умеют эффективно задействовать множественные исполнительные устройства. Пропускная способность оперативной памяти уперлась и не очень хочет дальше расти (с причинами тоже будем разбираться). Что же делать? Правильно. Расширять векторные расширения. Искать потерянные ключи надо там, где проще – под фонарем. Исполнительные устройства для арифметики с плавающей точкой, так называемые FMA (Fused Multiply-Add) unit-ы научились выполнять команду сразу для двух наборов аргументов. Потом для четырех, потом для восьми... Современное ядро с двумя 512-битными FMA юнитами делает по 32 операции с плавающей точкой за такт (с этим всем тоже поразбираемся). Получается, пиковая производительность выросла многократно. А что произошло с эффективностью, то есть соотношением между фактически получаемой на практике производительностью и теоретической пиковой производительностью? Правильно – она многократно уменьшилась для подавляющего большинства приложений. Понадобился еще один уровень распараллеливания – векторизация, который все сильно усложняет, но обычно вообще мало что дает в силу ограничений пропускной способности памяти. И это соотношение пиковой производительности и пропускной способности памяти неуклонно растет. То есть даже очень-очень высокопараллельный алгоритм, очень-очень хорошо сделанный, с многоуровневым распараллеливанием, с векторизацией, все равно будет претендовать лишь на жалкие проценты от пиковой производительности. Потому что просто не сможет настолько быстро выгребать данные из памяти, чтобы обсчитывать их на такой скорости. Вот такая она – темная сторона. Дальше – темнее.

Архитектуры стали гибридными. В суперкомпьютере стали сочетаться вычислительные устройства принципиально различной архитектуры – центральные процессоры и массивно-

параллельные ускорители-сопроцессоры. Наибольшее распространение в этой роли получили графические процессоры GPU (Graphics Processing Unit). И тут уже начинается настоящий мрак. GPU используют параллельную парадигму потоковой обработки, которая уж очень сильно ограниченная, то есть в нее вписывается лишь узкий класс алгоритмов. Отдельное от процессора адресное пространство памяти, а точнее там их несколько разных видов памяти, сложная архитектура, еще более сложное распараллеливание и программная реализация. При этом остаются те же проблемы – пиковая производительность GPU еще более многократно превосходит возможности пропускной способности памяти, а достигаемый процент от пика стал еще более жалким. Но многократный выигрыш в скорости по сравнению с центральными процессорами часто вполне достижим, и это очень заманчиво. Поэтому приходится переделывать параллельные алгоритмы, портировать коды, еще больше их усложняя, осваивать гетерогенные вычисления, чтобы задействовать разнородные вычислительные ресурсы.

Продолжаем наше погружение во мрак HPC. За счет чего GPU могут быть так сильно быстрее процессора? Основная причина – GPU имеет свою встроенную память. У нее в разы более высокая пропускная способность, чем у памяти процессора. Сейчас (на момент написания текста) это обычно HBM2 (High-Bandwidth Memory), выдающая бэндвис (*рус. жарг.* от bandwidth) порядка 1 TB/s за счет очень широкой шины. Ну и где-то рядом по скорости GDDR6, берущая не столько широтой шины, сколько эффективной частотой. Поскольку память интегрирована на устройстве, она может себе позволить широкую шину. За один такт передается многократно большее количество байт данных. Процессор такую ширину себе пристроить не может, так как его память живет отдельно на материнской плате (на него просто не влезет столько ножек, их и так уже обычно несколько тысяч).

Процессоры-мэникоры тоже осознали преимущество набортной памяти и начали наносить ответный удар. На них тоже появилась интегрированная память высокой пропускной способности. Первый широко распространенный пример тому – Intel Xeon Phi с набортной памятью MCDRAM (Multi-Channel DRAM). Более диковинные многоядерные сильно векторные устройства типа NEC SX-Aurora TSUBASA тоже обзавелись быстрой памятью HBM2 на борту. И вот, наконец, на вершине топа самых мощных суперкомпьютеров (см. https://www.top500.org/lists/top500/2020/06/) мы видим машинку на мэникорах (https://www.top500.org/system/179807/). 48-ядерные процессоры A64FX на основе архитектуры ARMv8 имеют на борту память HBM2 на 1 TB/s. Да, и там тоже используется векторное расширение на 512 бит, как и на Intel Xeon Phi. Набортная память хоть и быстрая, но ее – мало. Сейчас характерный размер – это 32 ГБ. Для такого мощного устройства это ни о чем. Поэтому, похоже, нас может ждать гибридизация памяти, как в случае Intel Xeon Phi – сочетание большой памяти низкой пропускной способности и ограниченной по объему памяти высокой пропускной способности. Программировать станет еще сложнее. Хотя еще есть надежды на спасение от гибридной памяти (с этим мы еще поразбираемся далее).

Очевидно, мэникоры тоже встали на темную сторону. Гонка за производительностью продолжается. Те системы и устройства, которые упоминаются выше в тексте, это по состоянию на сейчас. Какой может быть зоопарк вычислительных архитектур и суперкомпьютеров, когда вы будете это читать, даже страшно пытаться представить. Но хуже того. Этот зоопарк архитектур еще и порождает зоопарк средств разработки. Производители навязывают свои проприетарные фреймворки (*рус. жарг.* от framework), применимые только

на их устройствах (как CUDA от NVIDIA, например). Это еще более усложняет положение разработчика параллельных программ.

Системы усложняются, разработка усложняется, парадигмы параллельного программирования сужаются, множество совместимых с архитектурой алгоритмов тает на глазах. Резюмировать все вышесказанное можно очевидным выводом: когда пиковая производительность является самоцелью, в пределе будет система с бесконечно высокой производительностью, на которой, наконец, нельзя будет посчитать абсолютно ничего.

Звучит печально, но посмотрим на это с другой стороны. Высокопроизводительные вычисления очень востребованы. Множество отраслей промышленности и областей науки плотно сидит на софте для предсказательного моделирования. Как правило, очень дорогом софте. Вычислительный эксперимент на суперкомпьютере, про который мы далее поговорим, стал важным инструментом научно-технического прогресса. Так достаточно пафосно сформулировано? Для пущей мотивации вспомним также и смежные области – компьютерная графика (там многое близко к рассматриваемым в данном курсе сеточным методам), игрушки, базы данных, прочие задачи, связанные с обработкой данных большого объема. Там сильно важны параллельные технологии и производительность программного обеспечения. Значит, HPC – важное направление, знания в котором высоколиквидны.

А если что-то востребовано, но это что-то становится всё сложнее? Это значит, HPC программисты, параллельщики, хоть их работа и усложняется, становятся более ценными. При этом повышение порога вхождения снижает конкуренцию. Отметим этот факт, сделаем соответствующие выводы и, хитро улыбаясь, продолжим изучение материала.

## 1.2 Виды параллелизма

Наверное, все уже знают таксономию Флинна, которая классифицирует параллельные вычислительные архитектуры, используя четыре категории:

1) **SISD** – Single Instruction stream, Single Data stream;
2) **SIMD** – Single Instruction stream, Multiple Data streams;
3) **MISD** – Multiple Instruction streams, Single Data stream;
4) **MIMD** – Multiple Instruction streams, Multiple Data streams.

Далее нас будут интересовать только три вида: SISD, SIMD, MIMD.

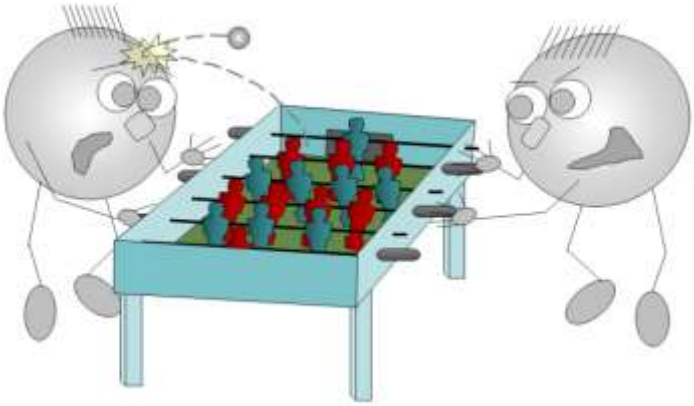
**SISD –** а разве это параллельная архитектура? Существует расхожее мнение, что нет. Тех, кто придерживается такого мнения, "тычем носом" в понятие ILP – Instruction-Level Parallelism. С эти понятием далее будем подробно разбираться.

**SIMD** – это обычно имеется в виду векторный параллелизм, когда одна машинная команда выполняется для множественных наборов аргументов.

**MISD** – а что это за такая дичь? Действительно, сложно подобрать примеры такой системы. С некоторой натяжкой к этому виду можно отнести, например, компьютеры цифровой электродистанционной системы управления самолета. Эти компьютеры (например, на Airbus по два каких-нибудь Elevator Aileron Computer, на Boeing по три Primary Flight Computer, у Сухой Суперджет их тоже три штуки) работают параллельно, выполняют независимые потоки команд, но получают на вход идентичные наборы данных с органов управления. Один из компьютеров управляет полетом, другие его проверяют, сравнивают его результаты со своими результатами. В случае возникновения расхождений, если вдруг компьютер глюканет и решит покрутить "бочку", управление переходит к резервному компьютеру.

**MIMD** – наиболее распространенный вид параллелизма, к которому, во-первых, относятся многоядерные процессоры, многопроцессорные серверы, то есть параллельные системы с общей памятью, и, во-вторых, кластерные системы с распределенной памятью, то есть множественные компьютеры, объединенные сетью, у которых нет общего адресного пространства оперативной памяти. В первом случае это подвид **SM-MIMD** (Shared Memory MIMD), во втором – **DM-MIMD** (Distributed Memory MIMD).

Для лучшего понимания можно обратиться к примерам из реальной жизни. Показательным примером SIMD является настольный футбол, в котором на одной ручке крутится сразу несколько футболистов.



А вот несколько дорожных рабочих, копающих яму – это MIMD. Ну, или вот более знакомый москвичам пример: несколько рабочих выполняют разнородные операции – один отбойным молотком отковыривает недавно уложенный асфальт, другой выкорчевывает только недавно поменянные бордюры, третий циркуляркой пилит очередную плитку. Самый настоящий MIMD. А вот если, например, один землекоп свяжет вместе сразу 4 лопаты и будет этим копать – это будет SIMD расширение землекопательного процессора.


## 1.3 Зоопарк вычислительных архитектур

В высокопроизводительных вычислениях используются разные виды устройств. Некоторые достаточно диковинные. Приводить в пример конкретные современные модели, конечно, смысла мало. Через совсем небольшое время это станет перечислением музейных экспонатов. Можно попробовать в общих чертах изобразить общую суть.

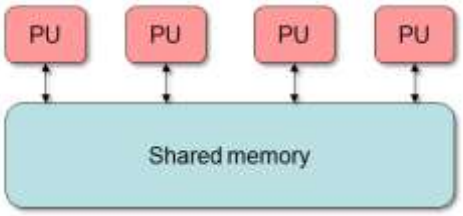
- Многоядерные серверные процессоры – CPU (Central Processing Unit). Параллелизм MIMD с общей памятью. Внутри ядер SISD параллелизм ILP – суперскалярность, внеочередное исполнение и VLIW; параллелизм SIMD – векторные расширения. Широко используются процессоры Intel, AMD. Есть еще процессоры IBM архитектуры Power, а также процессоры разных производителей на основе архитектуры ARM. Среди отечественных архитектур – Эльбрус, например.
- Существенно-многоядерные устройства, aka мэникоры (*жарг*. от manycore) с набортной памятью высокой пропускной способности. Параллелизм такой же, как и в первом случае. Можно было бы просто сказать Intel Xeon Phi. Раньше их называли майками (*жарг.* от MIC – Many Integrated Core архитектуры Intel), теперь больше фаями (*жарг.*

от Phi). Но, во-первых, эти устройства уже "ушли в закат". А, во-вторых, уже можно привести и другие аналоги, например, ARM-овые Fujitsu A64FX. Помимо набортной памяти, такие устройства имеют более простые ядра, но большего количества. Можно сказать, берут не умением, а числом. Эти два вида эволюционируют навстречу друг другу: в обычных CPU число ядер увеличивается в сторону мэникоров, а в мэникорах ядра становятся мощнее, что в пределе сходится к одному и тому же. Отличительным нюансом остаются гигабайты набортной памяти.

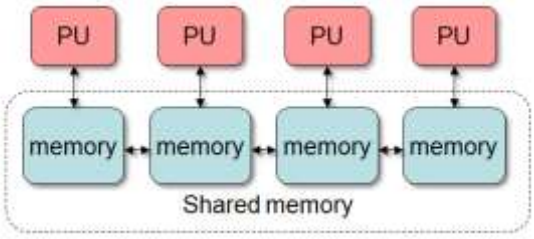
- Ultra-manycore устройства. Принцип примерно тот же. MIMD параллелизм. Но это уже сотни, а то и тысячи ядер. Но уже, возможно, совсем простых ядер. И со всякими другими упрощениями, такими как, например, отсутствие кэш-когерентности и прочие неприятности, осложняющие программирование. Например, долгое время держался на первой строчке мирового рейтинга суперкомпьютер Sunway TaihuLight на процессорах Sunway SW26010, не к ночи будь он помянут. Этот рекордно бесполезный (субъективное мнение автора) суперкомпьютер выдал аж 0.3% от пика на тесте HPCG. Потом удалось довести до 0.4%, собрав, видимо, по всему миру лучших программистов в машинных кодах. Другой пример – многотысячеядерники PEZY-SC. Вот тут есть про разные поколения с диаграммками https://en.wikichip.org/wiki/pezy/pezy-scx
- Графические процессоры общего назначения – GP GPU (General Purpose GPU). Тут параллелизм заметно отличается. По Флинну это ближе всего к SIMD, но не SIMD, потому что еще и MIMD, но не совсем MIMD. Какой-то непонятный SIMD-o-MIMD. Или MIMD-o-SIMD. Назовем это устройствами, работающие в парадигме потоковой обработки. С этим мы будем разбираться долго и мучительно. В этом виде (сейчас) наиболее распространены устройства NVIDIA и AMD.
- Векторные процессоры. По сути, те же многоядерники, возможно, с набортной памятью, но с очень широкими векторными расширениями. Пример такой архитектуры – устройства NEC SX-Aurora TSUBASA (192 операции с плавающей точкой за такт) или их предшественники SX-ACE.
- Системы на кристалле – SoC (System-on-a-Chip). Вообще много что можно назвать SoC. Имеются в виду CPU с интегрированным графическим процессором. Некоторые устройства имеют GPU, поддерживающие вычисления в формате двойной точности, на которых вполне можно считать. У меня в ноутбуке такая штуковина. Можно задействовать в вычислениях и ядра процессора и ресурсы GPU в качестве вычислительного сопроцессора (например, как тут – G.Oyarzun, R. Borrell, A. Gorobets, F. Mantovani, A. Oliva. Efficient CFD code implementation for the ARM-based Mont-Blanc architecture. Future Generation Computer Systems – The International Journal of eScience. Volume 79, Part 3, 2018. Pages 786-796. https://doi.org/10.1016/j.future.2017.09.029).
- Вычислительные ускорители FPGA (Field-Programmable Gate Array). Или программируемая логическая интегральная схема (ПЛИС). В таких устройствах вычислительные подпрограммы "вживляются" в реконфигурируемую микросхему, также имеющую на борту быструю память. Такие устройства в данном курсе не рассматриваются. Подробности можно найти тут, например: Лацис А.О. Параллельная обработка данных. Издательство: М.: Академия. 2010. ISBN: 978-5-7695-5951-8. Там во второй части, главы 11 – 13.

## 1.4 Иерархия параллелизма

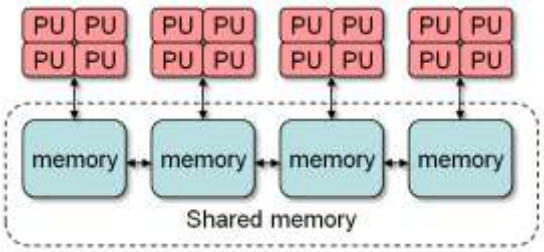
Современный суперкомпьютер сочетает различные виды параллелизма. Получается такая иерархия из нескольких уровней. Ну вот, возьмем, например, самое простое – один узел суперкомпьютера, обычный, не гибридный, то есть без ускорителей. В нем есть сколько-то процессорных ядер, у которых общая оперативная память. Это можно изобразить как систему с общей памятью – SM-MIMD: несколько PU – Processing Unit, то есть обрабатывающие устройства, соединены с общей памятью.



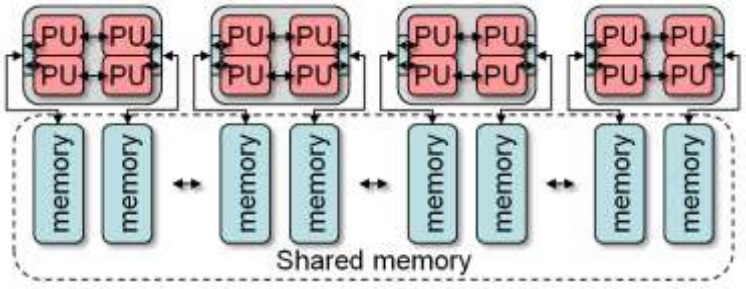

Все просто. Но мы-то знаем, а если не знаем, то скоро узнаем, что доступ к памяти неоднородный. На узле обычно не один процессор. А у каждого процессора свой контроллер памяти. Чтобы память была общей, имеется соединение между памятью на разных котроллерах по некоторой шине. Какая-то часть памяти ближе получается "ближе" к процессору, какая-то – "дальше". Возникает неоднородность – NUMA фактор. Нарисуем по-другому:



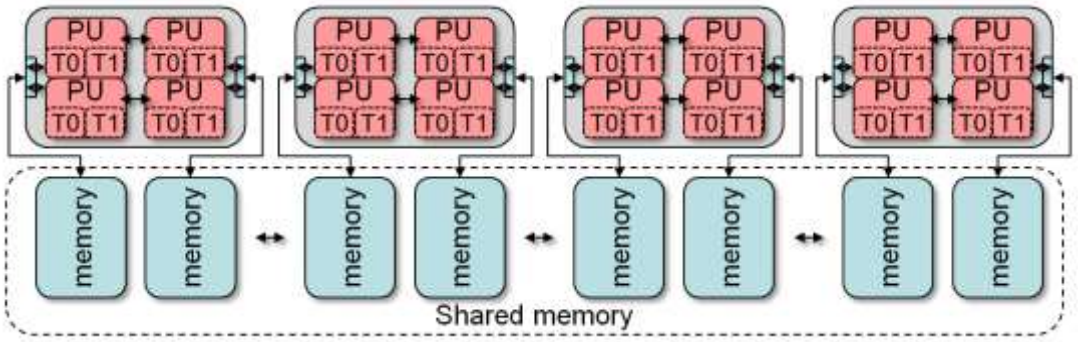

Так уже ближе к жизни. Но мы забыли, что процессоры многоядерные. Дорисуем ядер:



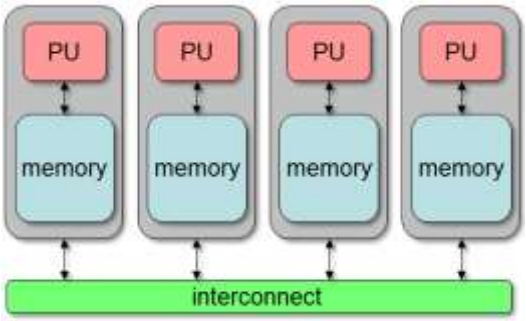

А еще вспомним, что у процессоров обычно более одного контроллера памяти, а ядра в процессорах связаны между собой по некоторой топологии, и не все ядра одинаково близки ко всем контроллерам. Еще приблизим картинку к реальности:



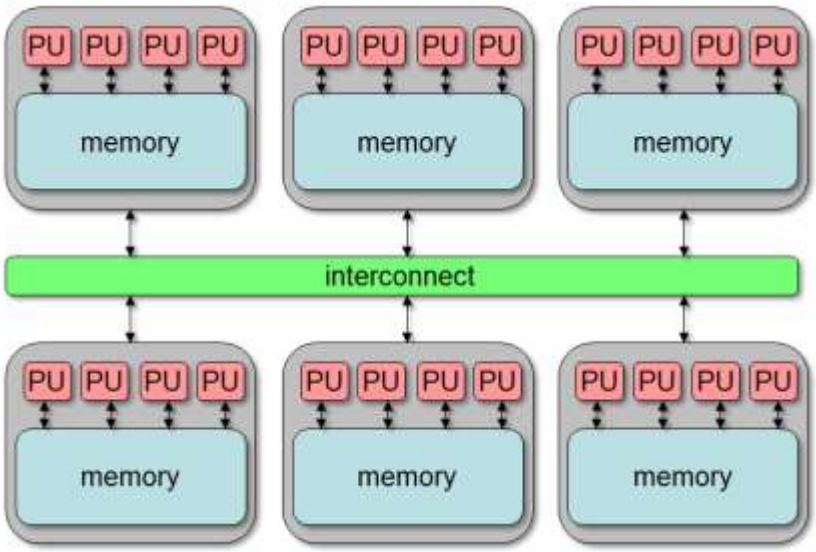
Как-то так, что ли? Ой, да, вспомним еще, что ядра – многопоточные, могут аппаратно поддерживать выполнение несколько потоков в режиме SMT (Simultaneous MultiThreading – одновременная многопоточность). Дорисуем каждому PU по паре T – threads. Получится вот какая-то такая совсем уже жуть:



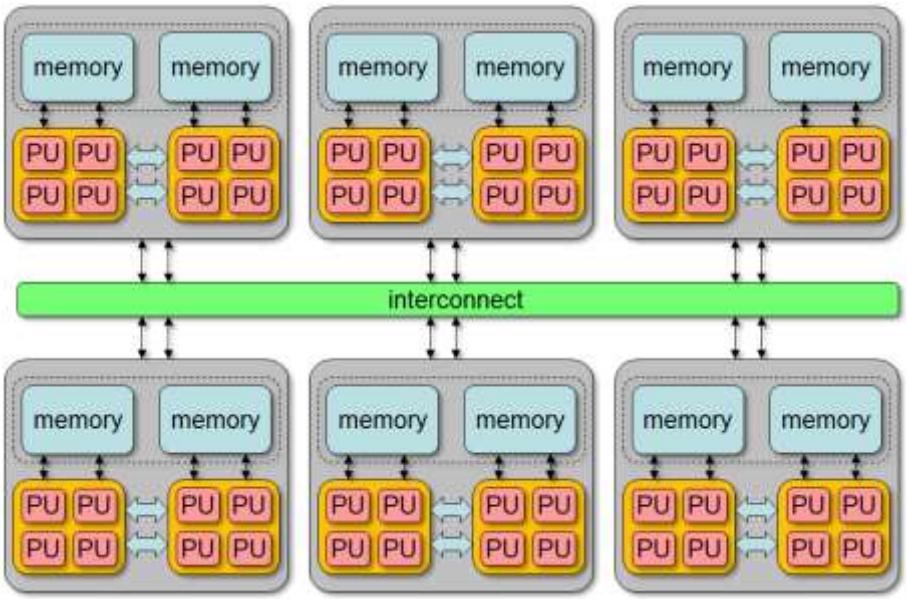

Вот уже внутри узла получается что-то похожее на какую-то иерархию. А теперь посмотрим на систему "сверху", с верхнего уровня параллелизма. Попробуем изобразить весь суперкомпьютер, то есть кластерную вычислительную систему. Это будет параллельная система с распределенной памятью – DM-MIMD. Нарисуем несколько PU, но теперь у каждого своя память, а соединены эти PU по коммуникационной сети:



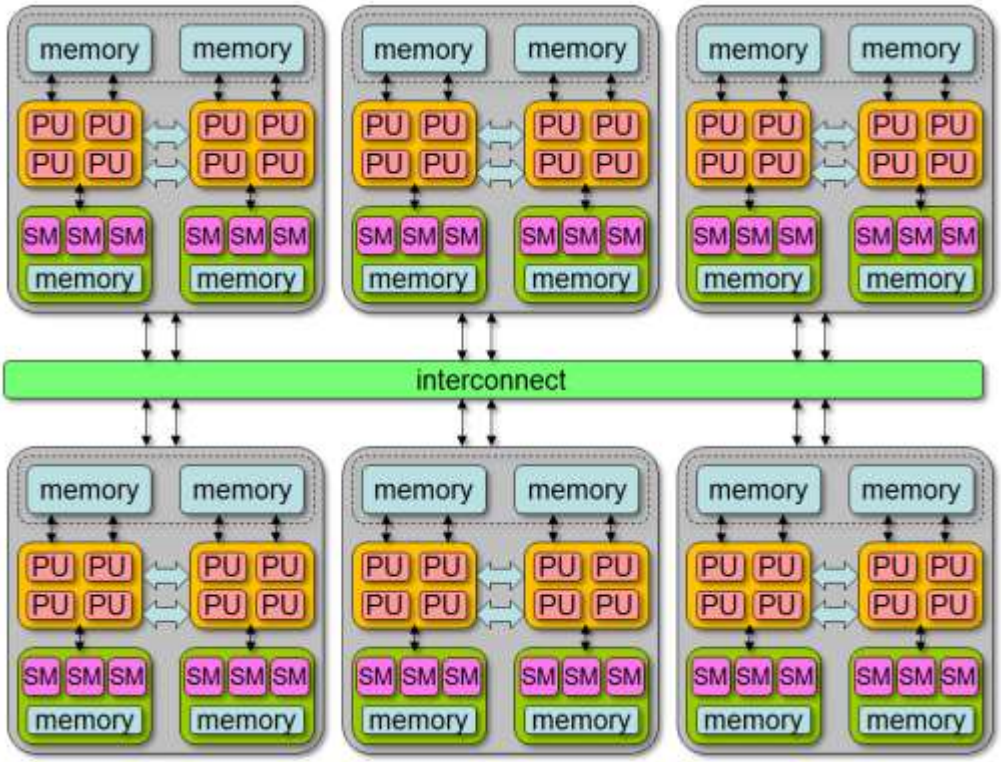

Теперь вспомним, что узлы – многоядерные. Дорисуем им по много PU:



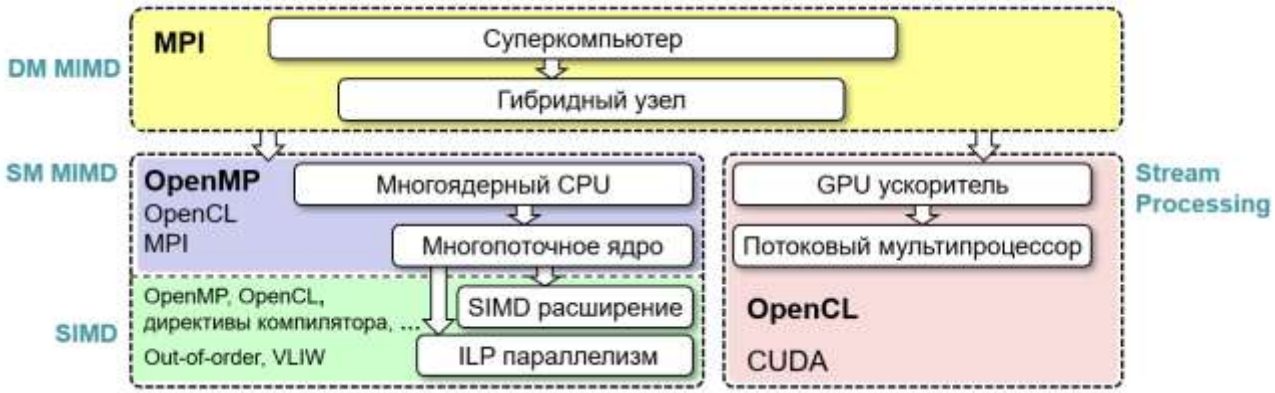
А теперь вспомним, что узлы – многопроцессорные, и память там общая, но с NUMA. А еще на узлах обычно по несколько сетевых адаптеров, поэтому нарисуем узлам хотя бы по две стрелочки в интерконнект:




А теперь самое страшное. Вспомним, наконец, что суперкомпьютеры в основном гибридные, и дорисуем узлам ускорители. Пусть это будут такие зелененькие штучки с множественными SM (Stream Multiprocessor). Да, не забудем, что ускорителям надо дорисовать свою собственную память. Получится нечто такое, на чем, пожалуй, лучше остановить творческий порыв в изобразительном искусстве:



В соответствии с иерархией параллелизма вычислительной системы выстраивается параллельный алгоритм. Там тоже получается иерархия, в которой будут разные параллельные модели и виды параллелизма, разные средства разработки. Получается целый software stack из разных фреймворков, интерфейсов программирования, библиотек:




На верхнем уровне работает DM-MIMD распараллеливание, в котором MPI (Message Passing Interface) обеспечивает взаимодействие между процессами параллельной программы на разных узлах суперкомпьютера. Внутри узлов используется многопоточное SM-MIMD распараллеливание средствами, например, OpenMP. Внутри ядер на нижнем уровне работает SIMD распараллеливание. На ускорителях работает свое отдельное распараллеливание средствами, например, OpenCL или CUDA. Со всеми этими уровнями параллелизма будем подробно разбираться в следующих главах.

## 1.5 Вычислительный эксперимент

Рост производительности вычислений, естественно, открывает все более широкие возможности для применения вычислительного эксперимента в промышленности и науке. Вычислительного эксперимент – это когда в компьютерном расчете предсказываются характеристики какого-то изделия в реальности, особенности протекания какого-то физического процесса. Расширение возможностей предсказательного суперкомпьютерного моделирования, в свою очередь, разгоняет научно-технический прогресс. Самолеты весят меньше, летают тише, долетают дальше, потребляют меньше топлива, становятся более безопасными. Тем же курсом следуют и все прочие вертолеты, теплоходы, автомобили, ракеты-носители космического назначения. Благодаря научному и инженерному вычислительному софту нефть добывается быстрее, медицина лечит лучше, атомные станции становятся безопаснее, и много чего становится лучше, быстрее, выше, сильнее.

Как это работает? Предположим, конструктор конструирует новую модель автомобиля. Он думает, а что будет, если куда-то сюда приварить железку, а отсюда две железки выпилить? Станет ли оно прочнее и легче? Можно построить прототип и разбить его на краш-тесте. Дорого и долго. А можно взять инженерный софт для расчета прочности и деформации твердого тела, понажимать кнопки, пошевелить мышкой и получить результат: кузов сложился, а двигатель пролетел через салон в багажник. О, нет, непорядок. Не будем выпиливать ту железку. И не нужен дорогой краш-тест. Положим, другой конструктор придумывает новый самолет. Он думает, а что будет с аэродинамическими силами, если крыло вот так вот немного изогнуть, немного сместить и, цитируя классика, воткнуть и там два раза повернуть? Можно изготовить такую модель и продуть в аэродинамической трубе. А можно

взять инженерный софт, понажимать кнопки, пошевелить мышкой и получить результат: подъемная сила упала, сила сопротивления выросла, а критический угол атаки стал такой маленький, что самолет свалится в штопор на первой воздушной ямке. Нет, непорядок. Не будем так гнуть крыло. Конструктор расстроен, так попробовал изогнуть крыло – хуже, сяк выгнуть – еще хуже. Что же делать? Тогда конструктор берет всю кучу мыслимых вариантов, как крыло можно погнуть, подвинуть, повернуть, и что-либо еще нехорошее с ним сделать. Он задает софту изменяемые параметры геометрии крыла и диапазоны этих изменений, запускает серию расчетов и идет домой отдыхать. Компьютер за него будет пробовать так крыло погнуть, сяк, туда сместить, сюда, обсчитает сотни вариантов изменения геометрии и выберет лучшие. Для каждого варианта геометрии будет выполнен расчет, который предскажет аэродинамические характеристики крыла. Это и есть HPC (High Performance Computing). А выбирать варианты геометрии будет специальный софт – оптимизатор. Он будет крутить какой-нибудь алгоритм, по которому будет варьировать параметры геометрии. Например, генетический алгоритм. В параметры текущего варианта геометрии вносятся произвольные изменения, обсчитывается "поколение" – набор разных вариантов изменений, из них выбираются лучшие варианты, и эти варианты "оставляют потомство". На следующем поколении, полученном из лучших вариантов предыдущего, опять произвольным образом вносятся изменения в параметры геометрии. Так гоняются массовые расчеты на суперкомпьютере, поколение за поколением, оптимизируется геометрия крыла. А конструктор в это время культурно отдыхает в баре с друзьями. Примерно так происходит НИОКР (научно-исследовательские и опытно-конструкторские работы) в куче отраслей промышленности. Фармацевты и химики тоже обычно не ковыряют вручную пинцетом разные варианты молекул, за них это делает суперкомпьютер. Он переберет миллионы вариантов молекул, просчитает миллионы вариантов химических реакций и скажет – вот, всё перепробовал, опять получилось $C_2H_5OH$. Нефтяник тоже не будет сверлить скважины где попало, чтобы понять, как лучше выкачать нефть из тамошних глубоких недр. Он загонит результаты геологоразведки в инженерный софт. Софт по этим данным построит модель всяких там пластов, учтет всю эту пористость, проницаемость, геометрию, где там нефть, куда пихать воды, переберет варианты и просчитает, куда надо правильно забуриться. Атомщик, конструирующий атомную станцию, тоже не станет каждый раз проверять на реальном реакторе, а хватит ли выбега ротора турбогенератора для питания главного циркуляционного насоса? А то мало ли, вдруг не хватит. Он возьмет расчетный софт, всячески аттестованный и сертифицированный, и промоделирует на компьютере происходящие на реакторе процессы. Можно привести еще много примеров, но мы же боремся за краткость изложения. Да и общий смысл, наверное, начинает прослеживаться. Далее рассмотрим конкретный вид приложений – CFD (Computational Fluid Dynamics), а именно моделирование механики сплошной среды сеточными методами.

## 1.6 Суперкомпьютерное моделирование сеточными методами

В данном разделе рассказывается, как работает суперкомпьютерное моделирование на примере задач вычислительной газовой динамики. Этот раздел можно считать научпопом и при отсутствии любопытства пропустить. Вопросы на экзамене по содержанию этого раздела не предусмотрены.

### 1.6.1 Как это работает

Предположим, что читатель не проходит обучение на кафедре Вычислительных методов или Математической физики. Исходя из этого предположения, максимально упростим пояснения о том, как это всё работает. Пусть моделируется течение жидкости или газа. Это течение моделируется некоторой системой уравнений. Ну ладно, специально для читателей с вышеупомянутых кафедр, течение сжимаемого вязкого газа описывается системой безразмерных уравнений Навье – Стокса, которая в дивергентной форме (в виде законов сохранения) относительно вектора консервативных переменных $\mathbf{Q}=\left(\rho,\rho u,\rho v,\rho w,E\right)^T$ – плотности, трёх компонент импульса и полной энергии, имеет вид

$$\frac{\partial \mathbf{Q}}{\partial t}+\frac{\partial \mathbf{F}_1(\mathbf{Q})}{\partial x}+\frac{\partial \mathbf{F}_2(\mathbf{Q})}{\partial y}+\frac{\partial \mathbf{F}_3(\mathbf{Q})}{\partial z}=\frac{\partial \mathbf{F}_1^{NS}(\mathbf{Q})}{\partial x}+\frac{\partial \mathbf{F}_2^{NS}(\mathbf{Q})}{\partial y}+\frac{\partial \mathbf{F}_3^{NS}(\mathbf{Q})}{\partial z}.$$

Конвективные потоки:

$$\mathbf{F}_1(\mathbf{Q})=\left(\rho u,\ \rho u^2+p,\ \rho uv,\ \rho uw,\ u(E+p)\right)^T,$$
$$\mathbf{F}_2(\mathbf{Q})=\left(\rho v,\ \rho vu,\ \rho v^2+p,\ \rho vw,\ v(E+p)\right)^T,$$
$$\mathbf{F}_3(\mathbf{Q})=\left(\rho w,\ \rho wu,\ \rho wv,\ \rho w^2+p,\ w(E+p)\right)^T,$$

где полная энергия $E=\rho\left(u^2+v^2+w^2\right)/2+\rho\varepsilon$, а $\varepsilon$ – внутренняя энергия.

Вязкие потоки:

$$\mathbf{F}_1^{NS}=\left(0,\tau_{xx},\tau_{xy},\tau_{xz},u\tau_{xx}+v\tau_{xy}+w\tau_{xz}-q_x\right)^T,$$
$$\mathbf{F}_2^{NS}=\left(0,\tau_{yx},\tau_{yy},\tau_{yz},u\tau_{yx}+v\tau_{yy}+w\tau_{yz}-q_y\right)^T,$$
$$\mathbf{F}_3^{NS}=\left(0,\tau_{zx},\tau_{zy},\tau_{zz},u\tau_{zx}+v\tau_{zy}+w\tau_{zz}-q_z\right)^T.$$

Компоненты вязкого тензора напряжений $\tau_{ij}$ и вектора теплового потока $q_i$ имеют вид:

$$\tau_{xx}=\mu\left(2\frac{\partial u}{\partial x}-\frac{2}{3}\mathrm{div}\mathbf{u}\right),\quad \tau_{yy}=\mu\left(2\frac{\partial v}{\partial y}-\frac{2}{3}\mathrm{div}\mathbf{u}\right),\quad \tau_{zz}=\mu\left(2\frac{\partial w}{\partial z}-\frac{2}{3}\mathrm{div}\mathbf{u}\right),$$

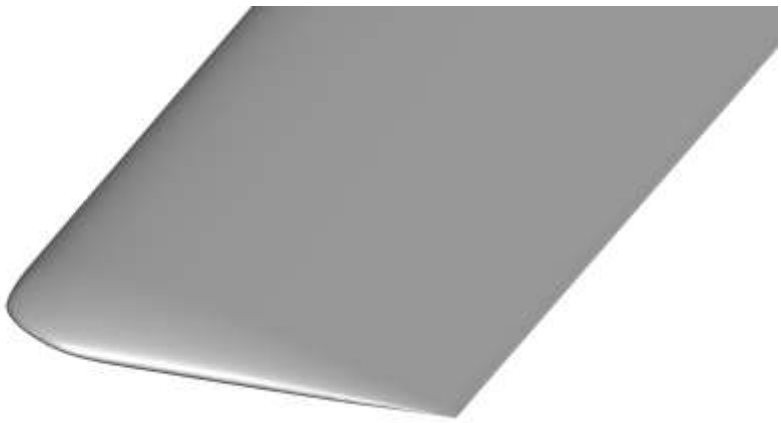
$$\tau_{xy}=\tau_{yx}=\mu\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right),\quad \tau_{xz}=\tau_{zx}=\mu\left(\frac{\partial u}{\partial z}+\frac{\partial w}{\partial x}\right),\quad \tau_{yz}=\tau_{zy}=\mu\left(\frac{\partial v}{\partial z}+\frac{\partial w}{\partial y}\right),$$

$$q_x=-\frac{\gamma\mu}{\mathrm{Pr}}\frac{\partial\varepsilon}{\partial x},\quad q_y=-\frac{\gamma\mu}{\mathrm{Pr}}\frac{\partial\varepsilon}{\partial y},\quad q_z=-\frac{\gamma\mu}{\mathrm{Pr}}\frac{\partial\varepsilon}{\partial z},$$

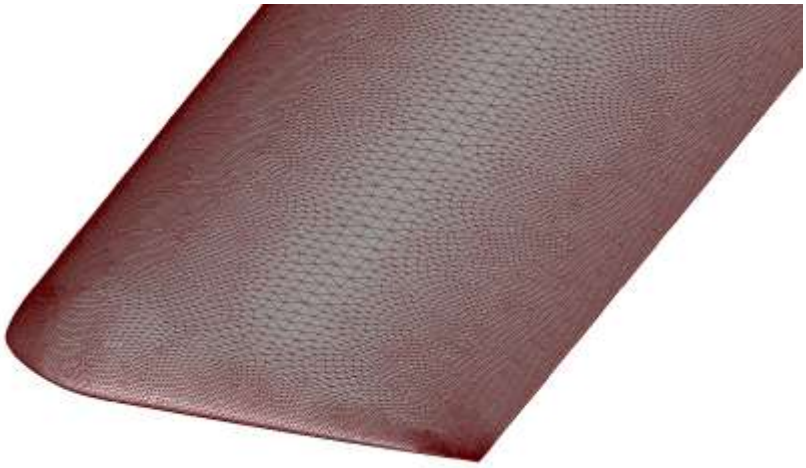
где $\mu$ – коэффициент динамической вязкости, $\mathbf{u}=(u,v,w)$ – вектор скорости.

Система уравнений Навье – Стокса замыкается уравнением состояния совершенного газа $p=\rho\varepsilon(\gamma-1)$, где $\gamma$ есть показатель адиабаты. Вроде так. Наверняка где-нибудь ошибся в этих формулах. Ну да ладно. Пожалуй, хватит формул. Перейдем к сути.

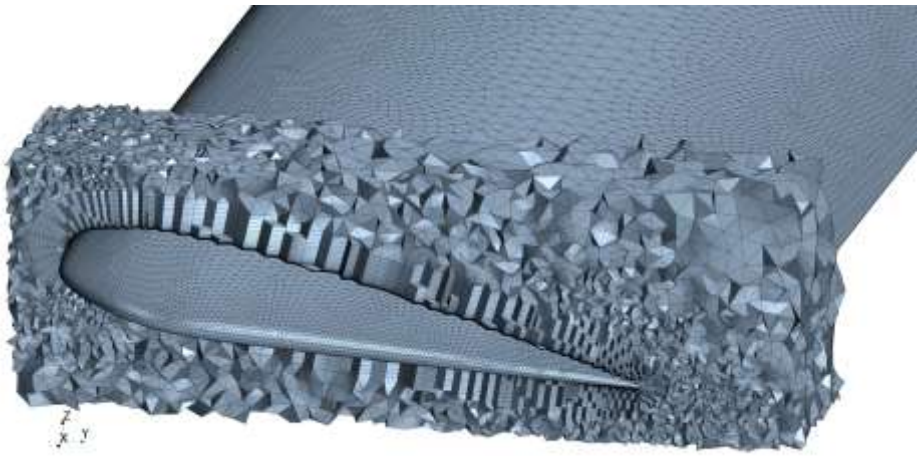
Пусть моделируется течение газа вокруг твердого тела. Пусть этим телом будет, например, лопасть несущего винта вертолета. Конструктор построил геометрию лопасти в CAD (Computer-Aided Design) системе. Вот, например, ниже на картинке показана законцовка этой лопасти.



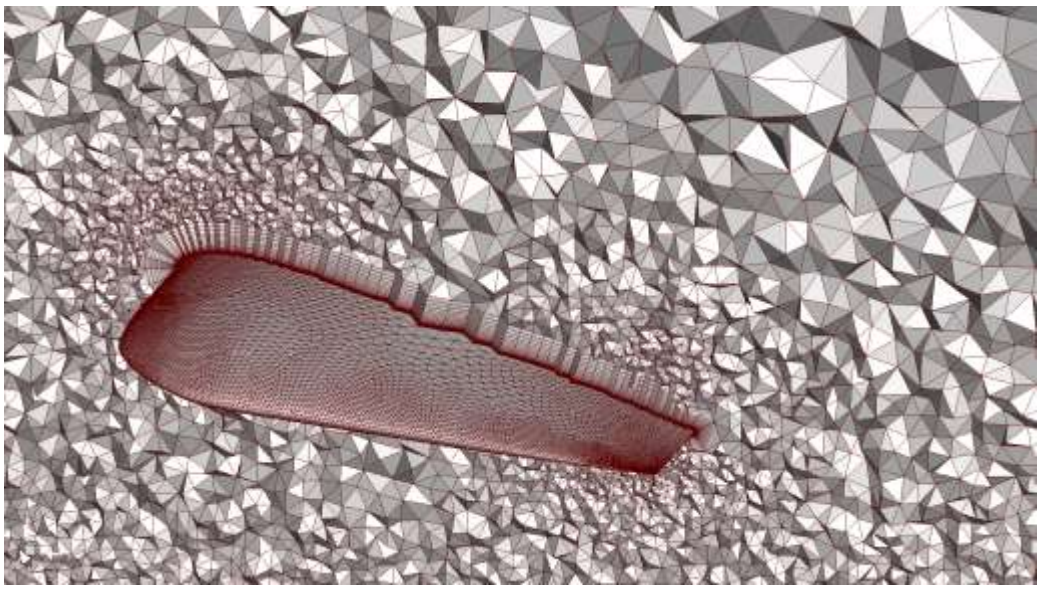
Затем геометрия подается на вход сеточному генератору. Эта программа построит на поверхности лопасти триангуляцию. У вас же был курс машграфа? Это почти как в компьютерной графике, только сетка получается намного мельче, чем нужно для графики.



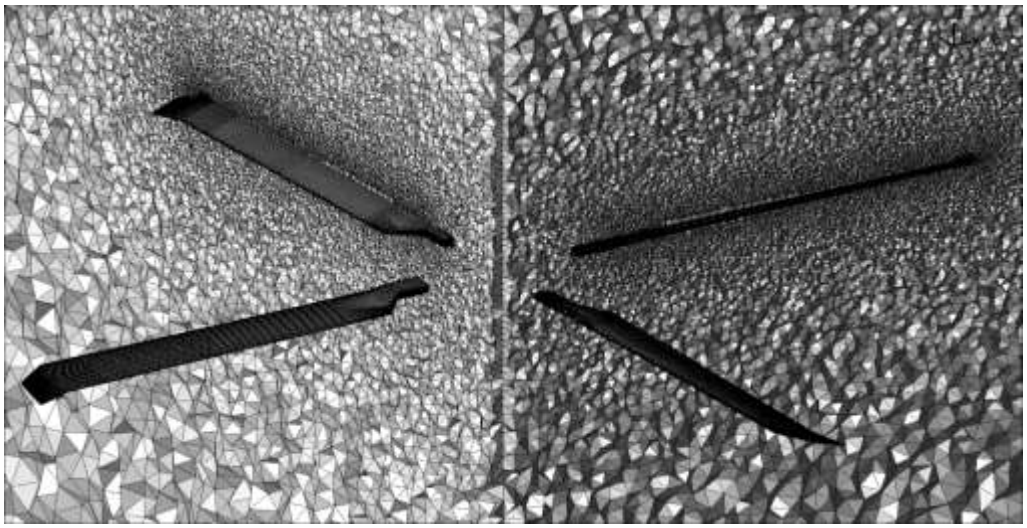
От триангуляции формируется объемная сетка, покрывающая расчетную область. Сетка состоит из сеточных элементов – тетраэдров, призм, пирамид, гексаэдров, например. Сеточными элементами заполняется весь моделируемый объем вокруг лопасти.



Вблизи поверхности лопасти – зона пограничного слоя, где сетка становится очень мелкой. В этой зоне образуется сдвиговый слой, ламинарное пристеночное течение переходит в отрывное турбулентное. Там требуется очень маленький шаг сетки по ортогональному поверхности направлению. Видно, как сильно уменьшается высота призм вблизи поверхности.

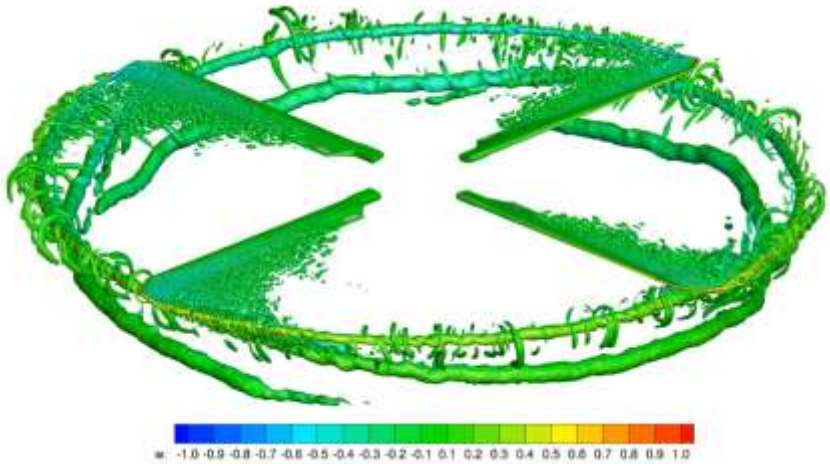


Так весь объем вокруг винта поделился на маленькие объемчики – сеточные элементы.

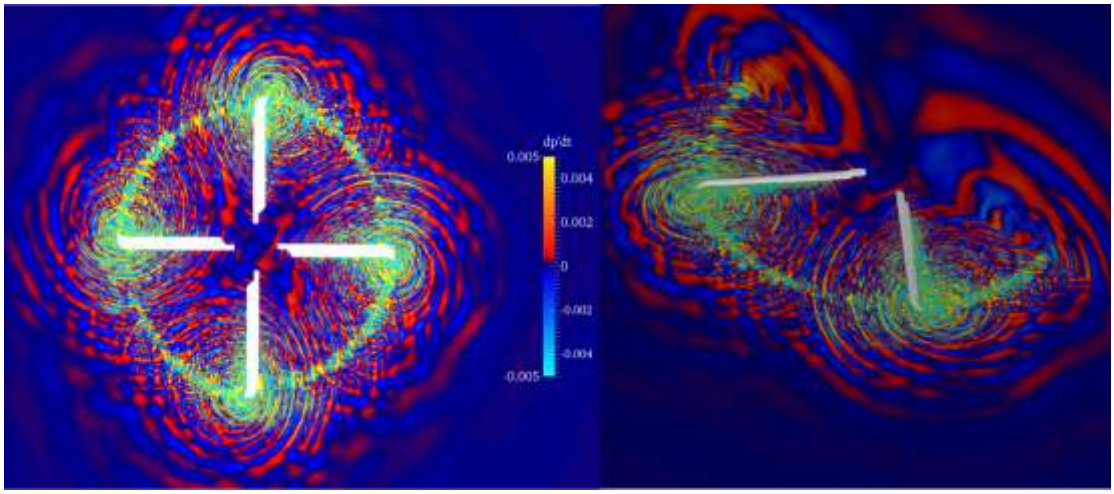


На этой сетке выполняется дискретизация уравнений Навье – Стокса, описывающих течение. В каждой ячейке сетки задаются физические величины – плотность, три компоненты импульса и полная энергия. Даже для простоты пусть это будет плотность, три компоненты вектора скорости и давление. Получается, как ПЗС матрица в цифровом фотоаппарате, в которой в каждом пикселе заданы три компоненты цвета. Ну а тут точно также, только не цвета и не на плоскости, а в объеме (и не в покер, а в преферанс, и не выиграл, а проиграл). Ячейками сетки могут быть либо сами сеточные элементы, либо ячейки могут строиться вокруг сеточных узлов некоторым образом, но сути это не меняет.

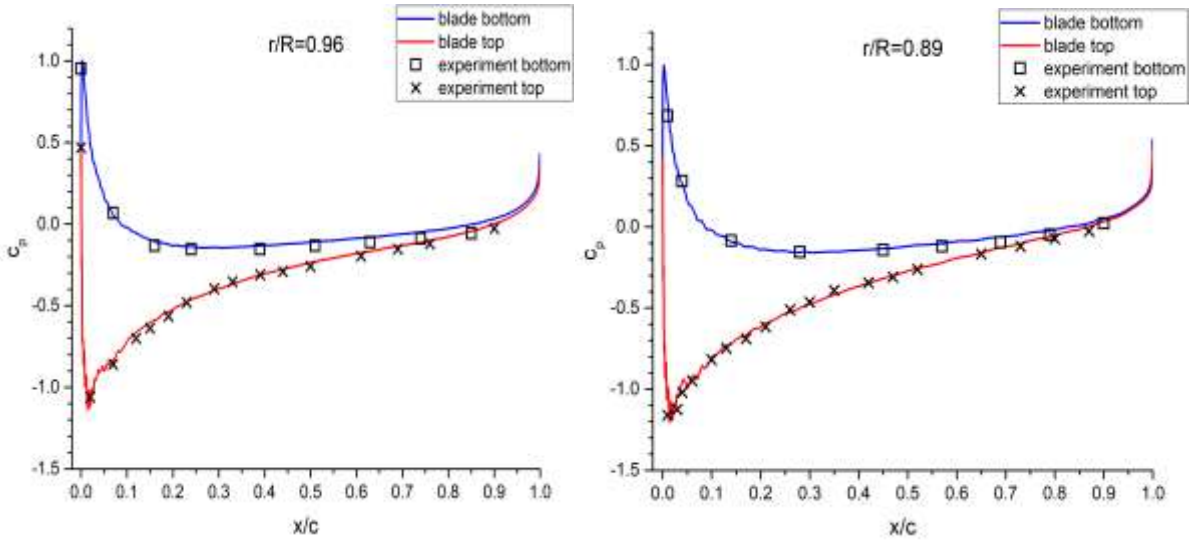
Итак, у нас есть миллионы ячеек сетки, в которых заданы 5 переменных. Выполняется интегрирование по времени: время продвигается вперед на какую-то (очень) малую величину. Для всех ячеек сетки проводится расчет, в котором определяется, сколько жидкости или газа втекло-вытекло через каждую из граней ячеек. Узнав, сколько из каждой ячейки куда перетекло, вычисляем значение наших физических величин в каждой ячейке в новый момент времени, т.е. на новом временном слое. Продвигаясь вперед такими шагами по времени, моделируется динамика течения, получается турбулентное течение вокруг вращающегося несущего винта вертолета.



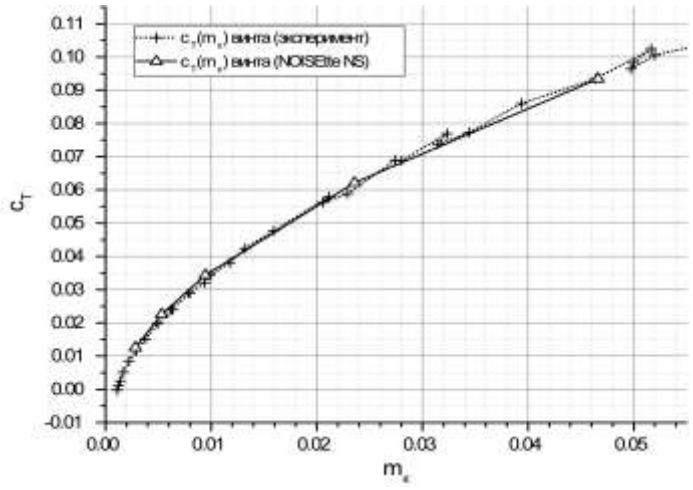
Воспроизвелись даже акустические поля, можно получить шум, возникающий от взаимодействия турбулентных структур с законцовкой лопасти.

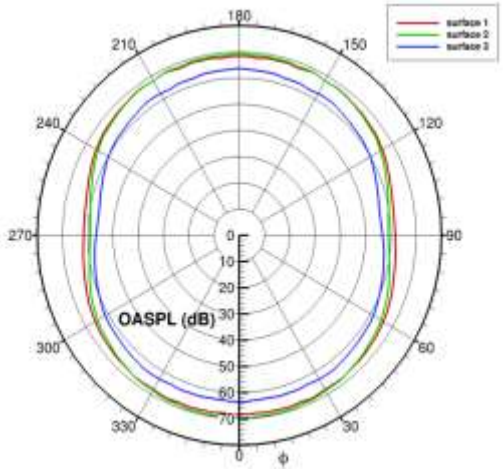



Конструктор, вернувшись на работу после тяжелых выходных, открыл результаты расчета и получил аэродинамические и акустические характеристики своего винта. Распределение коэффициента давления по поверхности лопасти,



аэродинамические силы – коэффициент тяги винта при различных углах установки лопасти, поляры винта (соотношение коэффициента тяги и момента – график ниже слева), диаграммы направленности шума (справа) и т.д.



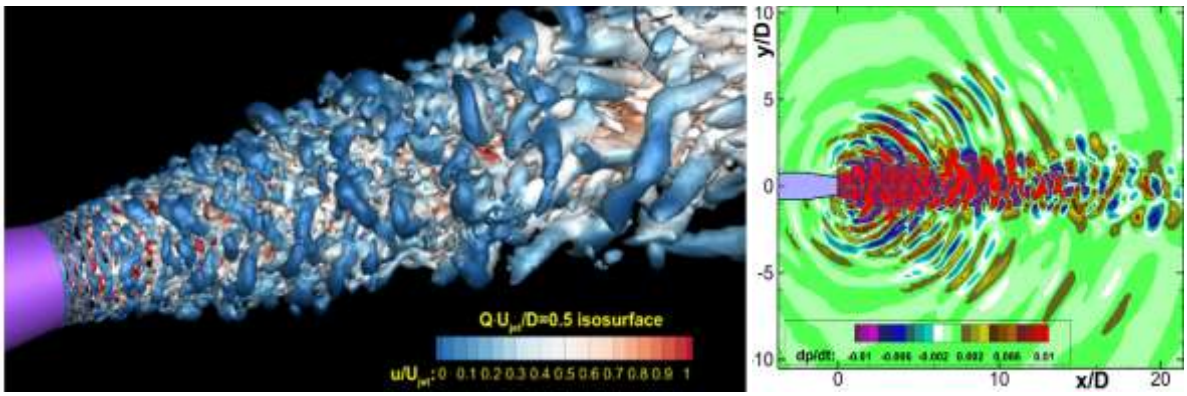




Довольный результатами конструктор, конечно, потом посмотрит, сколько прожглось процессорного времени на суперкомпьютере, посчитает, сколько это стоит, и подумает с тоской, что лучше бы заказал изготовление модели для натурного эксперимента. Но на то мы и изучаем этот курс, чтобы расчеты с вашей помощью становились эффективнее и дешевле.

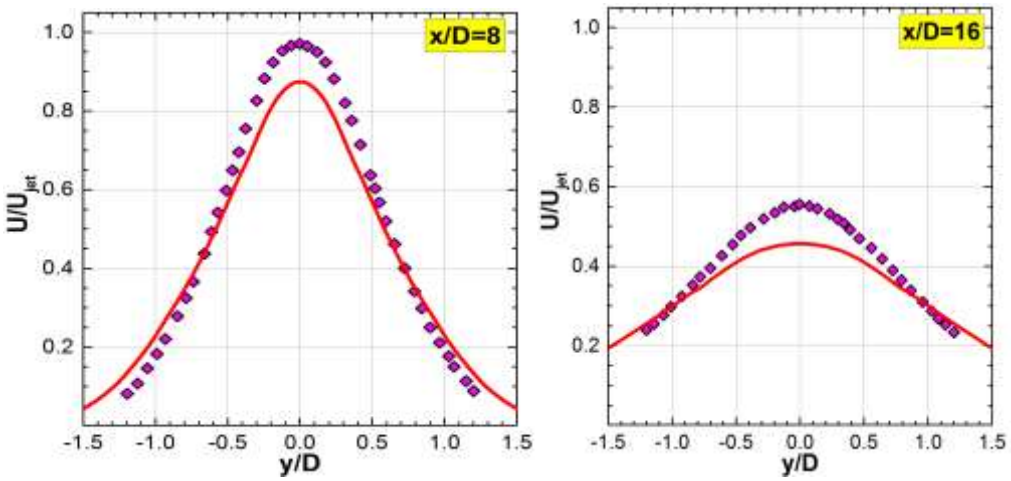
### 1.6.2 Пространственное разрешение и стоимость расчета

Почему, чтобы узнать правду, всё-таки зачастую проще продуть модель в трубе или сгонять полетать летающую лабораторию? Разберемся, откуда берется такая ресурсоемкость вычислительного эксперимента и зачем нужны суперкомпьютеры.

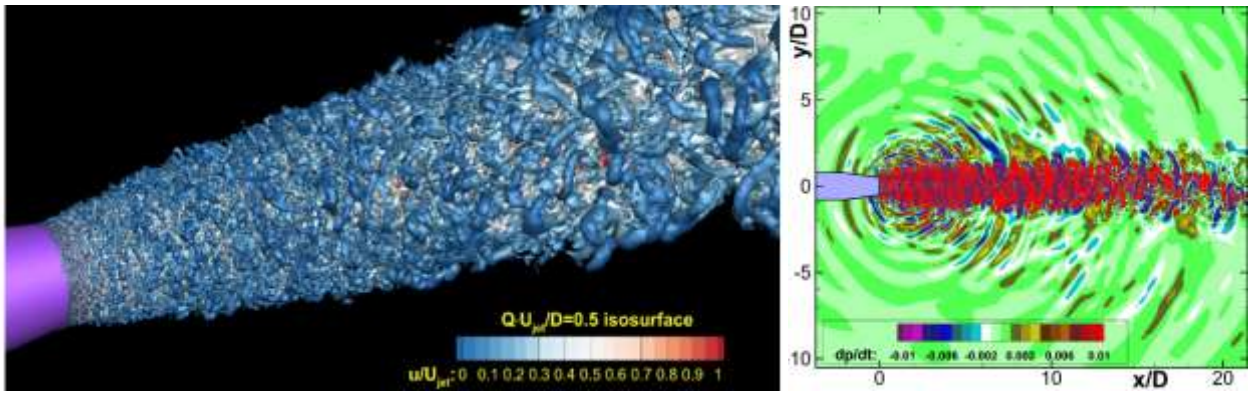
Чем лучше пространственное разрешение сетки, т.е. чем больше ячеек, этих "пикселей", в которых заданы переменные, тем точнее описывается течение и ближе к реальности результат. Вот, например, считаем мы зачем-то реактивную струю ($Re=1.1\cdot10^6$, $M=0.9$), взяли сетку на полтора "мегапикселя" – из 1.5 млн ячеек. Посчитали, получили какую-то картину течения.



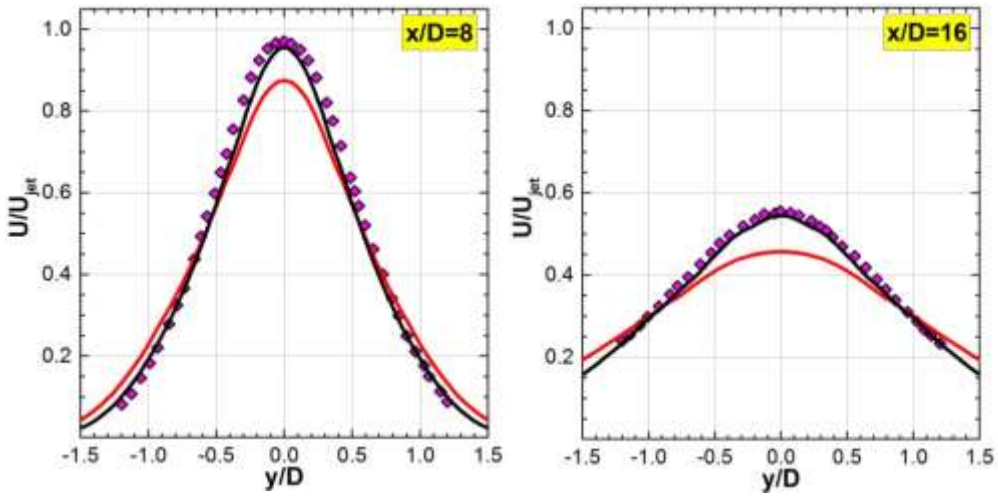

Видим турбулентные структуры, даже какие-то звуковые волны на правой картинке. Но это модельная конфигурация, у нее есть эталонное решение. Сравним его результатами расчета.




Вот, например, показан профиль скорости по линии, перпендикулярной оси струи, на разном удалении от сопла. Что-то результаты вообще не в дугу. Красная линия расчета сильно далека от ромбиков эксперимента. Берем сетку поподробнее, на 9 "мегапикселей". Сразу видно, что появились более мелкие структуры течения.




А что стало с графиками? Красная линия – было на сетке 1.5 млн ячеек, черная линия – стало на сетке из 9 млн ячеек.

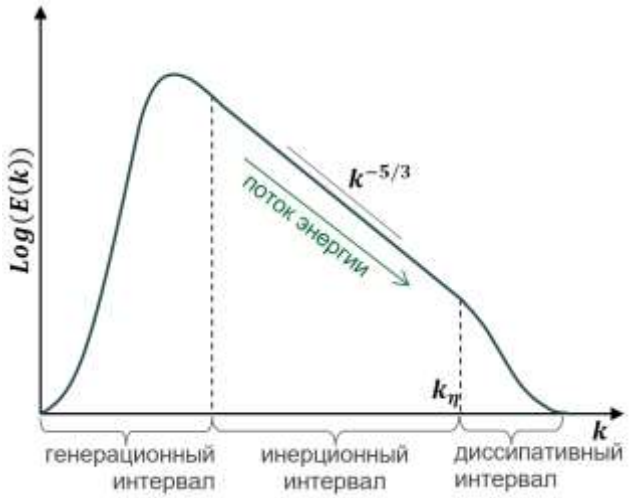



Результаты расчета хорошо прилегли на эксперимент. Дело было в недостаточном для данного физического процесса сеточном разрешении. Для более сложных конфигураций нужны десятки миллионов ячеек. Для расчетов крупных частей летательного аппарата, таких как крыло самолета с выпущенной механизацией, могут потребоваться сотни миллионов, а то

и миллиарды ячеек. Чтобы численно воспроизвести течение вокруг самолета или вертолета целиком (и получить близкий к реальности результат, а не просто никчемные цветные картинки!), может понадобиться сетка из десятков миллиардов ячеек и десятки (а то и сотни) миллионов процессорных часов. А откуда берутся требования к сеточному разрешению?

### 1.6.3 Моделирование турбулентности

Турбулентное течение характеризуется наличием вихрей различных размеров и структуры. В нем происходит переход кинетической энергии потока в энергию сначала крупных (когерентных) вихрей, и далее по каскаду – от крупных вихревых структур к более и более мелким. Ниже приведена очень классическая картинка, характеризующая турбулентный поток, – график зависимости кинетической энергии турбулентности $E$ от волнового числа $k$ (чем больше это $k$, тем мельче вихри):

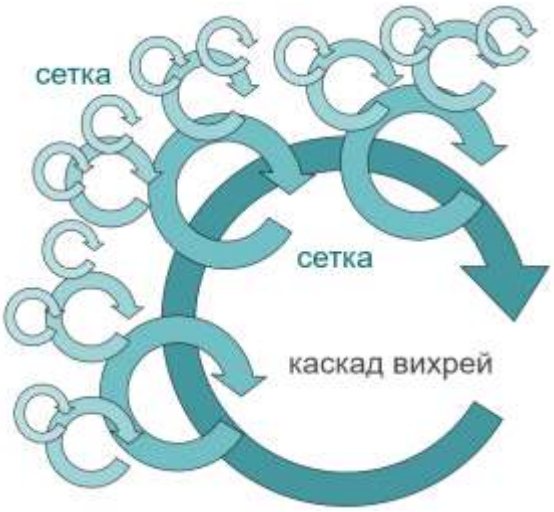



Крупные вихри распадаются на более мелкие, те на еще более мелкие, те на еще-еще совсем более мелкие. Таким образом, масштабы течения уменьшаются, пока не дойдут до так называемого колмогоровского масштаба ($k_\eta$), на котором кинетическая энергия движения жидкости или газа за счет вязкости среды переходит в тепло, то есть во внутреннюю энергию.

Получается, чем больше в изучаемом течении перепад масштабов, тем больше нужно сеточных ячеек, чтобы всех их разрешить. Течение характеризуется безразмерным параметром – числом Рейнольдса $\mathrm{Re}=U\cdot L/\nu$, где $U$ – скоростной масштаб, $L$ – некоторый характерный размер, $\nu$ – кинематическая вязкость среды. Это число, по сути, определяет соотношение связанных с движением инерционных сил (в числителе), и вязких сил (в знаменателе). Кроме того, оно в некотором смысле характеризует перепад масштабов от крупных структур до самых мелких, после которых происходит диссипация в тепло. Чем больше размер объекта ($L$), или чем больше скорость течения ($U$), или чем меньше вязкость среды ($\nu$) – тем больше это число. Соответственно, тем более высокое нужно пространственное и временное разрешение, тем выше ресурсоемкость расчета.

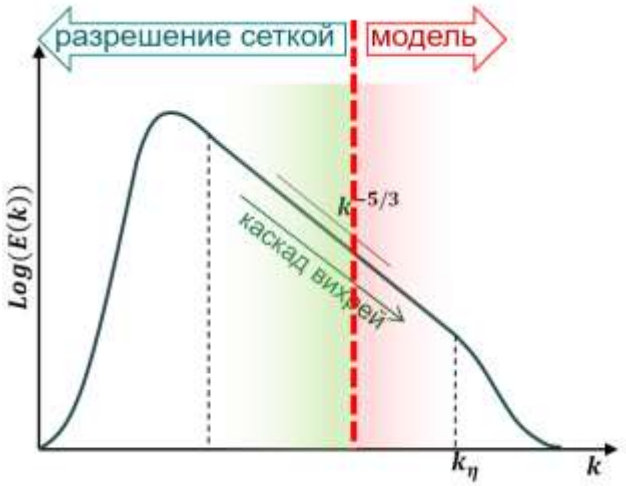
Тут будет только очень кратко общий смысл. Более подробно про все это математическое моделирование турбулентности можно почитать, например, тут: Гарбарук А. В. Моделирование турбулентности в расчетах сложных течений: учебное пособие / А. В. Гарбарук, М. Х. Стрелец, М. Л. Шур – СПб: Изд-во Политехнического университета, 2012. – 88 с. https://cfd.spbstu.ru/agarbaruk/doc/2012_garbaruk-strelec-shur_modelirovanie-turbulentnosti-v-raschetah-slozhnyh-techenij.pdf

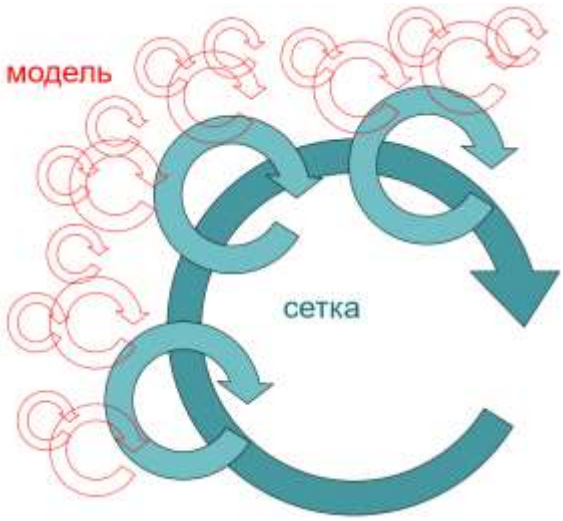
**Прямое численное моделирование – DNS** (Direct Numerical Simulation). Если мы будем разрешать по пространству (то есть, сеточно) и по времени все релевантные масштабы течения, то есть все динамические процессы, то это будет DNS. Это самый дорогой подход, но и самый точный.

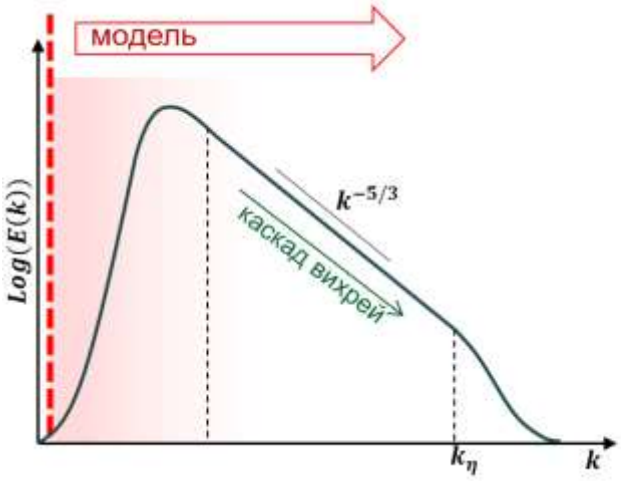

Расчеты с помощью метода DNS, как правило, носят фундаментальный характер. Обычно это штучные эталонные расчеты, часто рекордные, в которых исследуются модельные конфигурации течения, сферические кони в турбулентном вакууме. Целью таких расчетов является изучение физического процесса, получение надежных эталонных данных для разработки и валидации моделей турбулентности. Ну, чтобы было, откуда взять ромбики на графиках, с которыми сравнивать свои кривые не в дугу результаты расчетов.

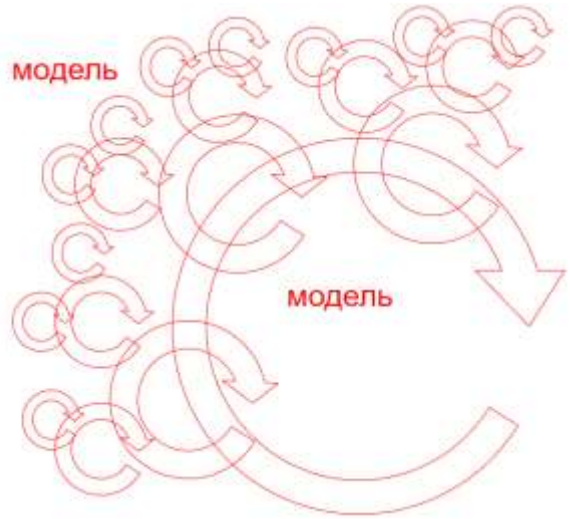
**Метод крупных вихрей – LES** (Large Eddy Simulation). Пусть сеткой мы можем разрешить только относительно крупные структуры, а на мелкие структуры у нас не хватит вычислительных ресурсов. Обрубим в каком-то месте график энергетического спектра (по оси X, внутри инерционного интервала). Влияние мелких структур будем имитировать некоторой моделью, которая воспроизведет переход энергии дальше по каскаду, не разрешая сами вихри. Модель подсеточного масштаба заменит действие вихрей, меньших по размеру, чем ячейки сетки, некоторой вычисляемой величиной – вихревой вязкостью, с добавлением которой энергия с крупных вихрей будет соответствующим образом отводиться.



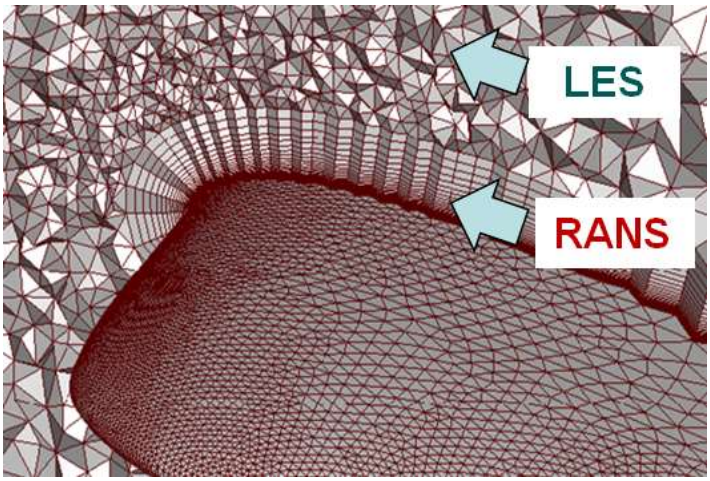



**Осредненные уравнения – RANS** (Reynolds Averaged Navier – Stokes). Если описание вообще всех масштабов течения доверить некоторой специальной модели, называемой полуэмпирической, то получится стационарный расчет RANS. Все масштабы течения мы отдали на откуп модели и получили стационарное осредненное решение – как если бы в каждой ячейке был датчик измерений физических величин, и мы бы усредняли его показания на длительном интервале времени.

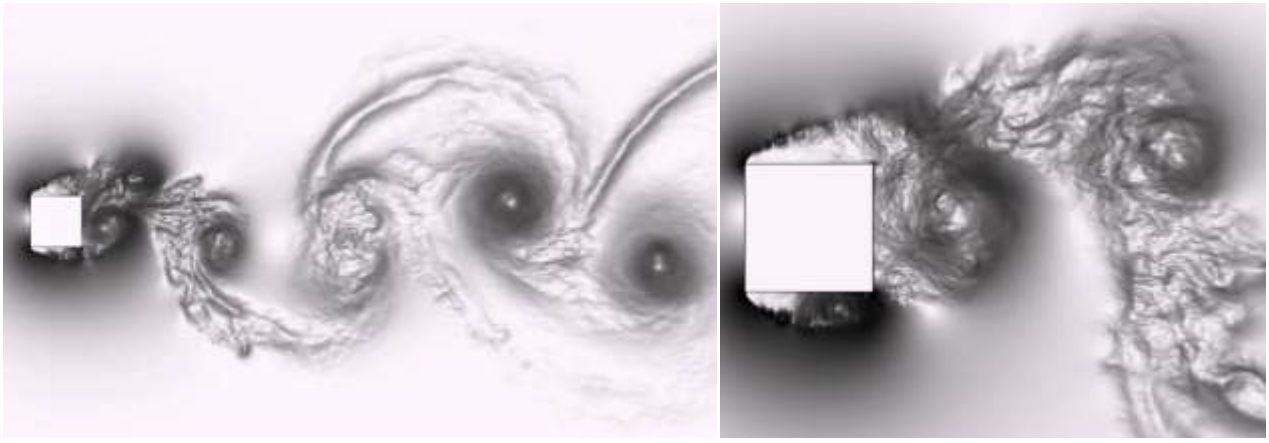



**Метод отсоединенных вихрей – DES** (Detached Eddy Simulation). Это гибридный подход, сочетающий LES и RANS. В пристенной области, то есть совсем вблизи поверхности твердого тела, для корректного моделирования сдвигового слоя (пограничного слоя, aka погранслоя) методу LES нужен очень маленький шаг сетки во всех направлениях. Это обеспечить крайне затруднительно и ресурсоемко. Поэтому в пристенной области будем использовать RANS, который работает на анизотропных сетках (то есть высокое разрешение нужно только по одному направлению, ортогональному стенке). В остальной же области будем использовать LES. Так выходит большая экономия по числу ячеек сетки в погранслое.

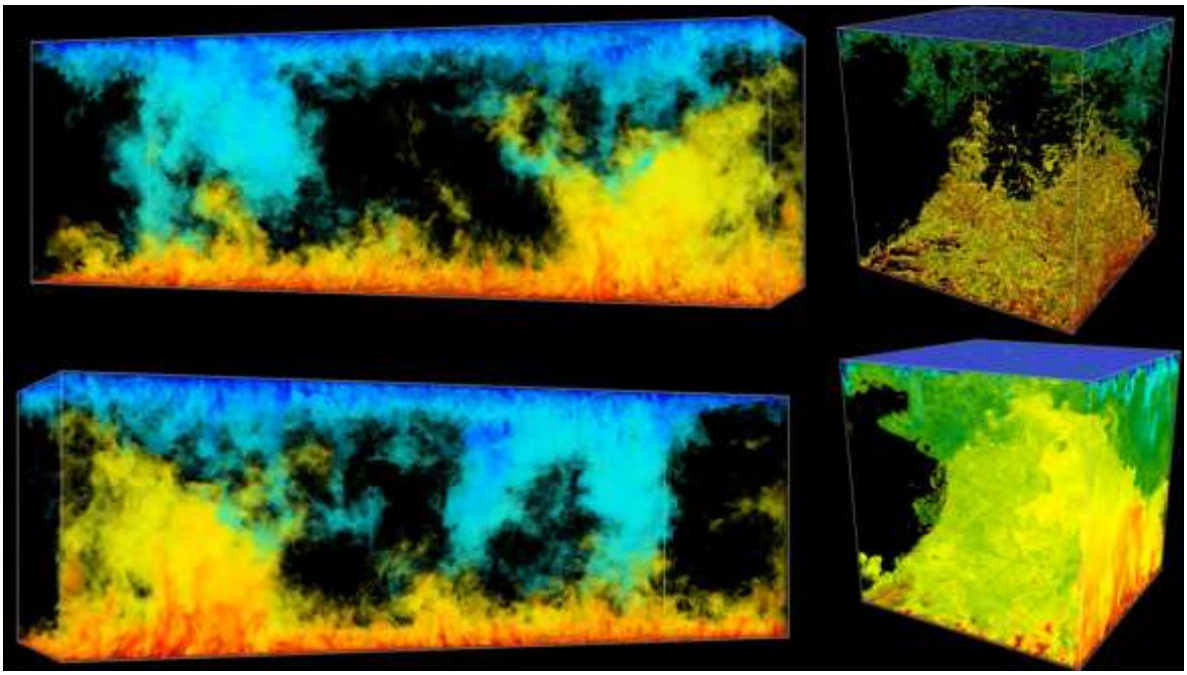



Получается, что самый дорогой подход – это DNS. Вот, например, ниже на картинке приведен расчет течения вокруг квадратного цилиндра (Re=22000, сетка 320 млн ячеек). Показана моментальная картина для поля давления в центральном сечении. Понятно, что расчет делался не для того, чтобы улучшить аэродинамическую форму квадратного цилиндра. Изучается физика течения вокруг тела с острыми кромками, на которых возникает отрыв, неустойчивость Кельвина–Гельмгольца, слои смешения, образование вихревого следа.



Картинка отсюда: https://doi.org/10.1016/j.compfluid.2015.09.013

Или вот, например, серия расчетов конвекции Рэлея–Бенара. Расчеты были до числа Рэлея $10^{11}$. Самый большой расчет (рекордный на тот момент) был на сетке из 6 миллиардов ячеек. Чтобы разрешить течение до мельчайших турбулентных структур и получить данные об эволюции топологии течения понадобилось 30 млн ядрочасов на суперкомпьютере.



https://doi.org/10.1103/PhysRevFluids.5.024603 https://doi.org/10.1063/1.5005842

Для реальных задач аэродинамики такое пространственное разрешение непозволительно, моделирование напрямую уравнениями Навье – Стокса неприменимо. Нужны дополнительные модели и уравнения. LES сильно (на порядки) дешевле DNS, особенно в случае отрывных течений. Гибридный подход DES многократно дешевле LES. RANS сильно дешевле DES.

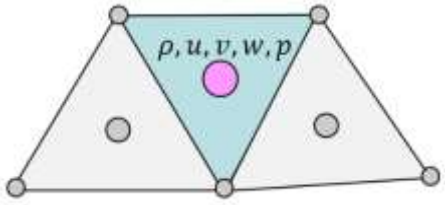
Подходы DNS, LES, DES называют вихреразрешающими, поскольку вихри хоть какого-то масштаба, хотя бы самого крупного, воспроизводятся напрямую в процессе расчета. Раз течение воспроизводится в динамике, то такие расчеты называют нестационарными, а также – точными по времени. Для получения осредненной картины течения, то есть статистики течения, нужно усреднять моментальные картины течения на достаточно длительном временном интервале. RANS – это стационарный расчет. Он дает сразу осредненную картину

течения. Поэтому в нем нельзя получить никаких нестационарных характеристик – ни пульсаций давления на поверхности, ни спектров шума, ни вибрационных нагрузок, и т.д. Можно получать интегральные величины типа подъемной силы, сопротивления, моментов.

Основная масса расчетов промышленных задач – это RANS, поскольку ресурсоемкость таких расчетов очень низкая. Но точность таких расчетов зачастую невысока, потому что турбулентные структуры течения сильно зависят от задачи, от геометрии, от конфигурации течения. Особенно сильно зависят крупные когерентные структуры. Подобрать параметры модели так, чтобы она подходила ко всем течениям невозможно. Настроишь модель на течение вокруг крыла, она не будет работать на течении вокруг шасси, и т.д. Особенно плохо дело с отрывными течениями. Поэтому, разрешая хотя бы крупные структуры методом LES, можно намного повысить точность результата. А мелкие структуры ведут себя уже более однородно и куда более схожи между собой в разных конфигурациях течения.

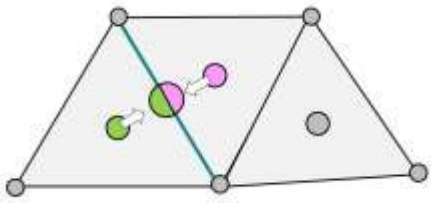
### 1.6.4 Точность численной схемы

Рассмотрим проблему на простейшем примере. В самом упрощенном виде получается примерно следующий алгоритм. Пусть у нас есть сетка. Для простоты изображения – двухмерная треугольная. Рассмотрим маленький фрагмент сетки из нескольких элементов – треугольников. Тот факт, что на треугольниках заданы физические (или консервативные) переменные, изобразим кружочками в центрах масс треугольников.

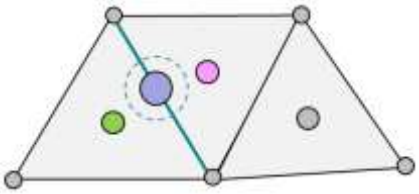



На сетке заданы сеточные функции для плотности, давления (или полной энергии), компонент вектора скорости (или импульса). Они описывают состояние моделируемой среды, в данном случае – сжимаемого газа, в какой-то момент времени. Чтобы узнать, как будет дальше развиваться течение газа, надо сделать (небольшой, очень небольшой) шаг по времени и найти сеточные функции на следующем временном слое. То есть надо для каждой ячейки (треугольника) посчитать, сколько втекло и вытекло газа через его границы. В двухмерном случае границы – это ребра треугольника, в трехмерном – грани тетраэдра, например. Вычислив потоки через границы элементов, мы найдем для каждого элемента значения переменных на новом временном слое.

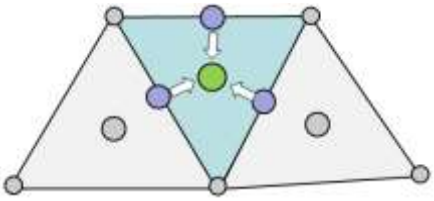
Чтобы посчитать потоки через грани, нам надо выполнить реконструкцию – узнать значения переменных в центрах масс граней. Для этого их надо туда как-то хитро интерполировать. Можно построить полином, можно нагородить разные интерполяционные конструкции, и много еще чего. Изобразим этот этап так:



После реконструкции мы для грани, разделяющей два элемента, назовём их – левый и правый, узнали значения переменных в центре этой грани со стороны каждого из элементов. На картинке выше это отмечено как розовый и зеленый полукружки в центре ребра. Теперь нам из этих двух наборов переменных надо получить один набор. Для этого решается задача Римана о распаде разрыва. Делает это специальная подпрограмма (*жарг.* – распадник). Мы получили значения потоков через грань (синий кружок на картинке ниже). Так надо сделать для всех граней между элементами.



Теперь осталось просуммировать потоки с граней в элементы, чтобы найти новые значения переменных в элементах. Поток с грани добавляется с плюсом в один элемент, и с обратным знаком в другой.



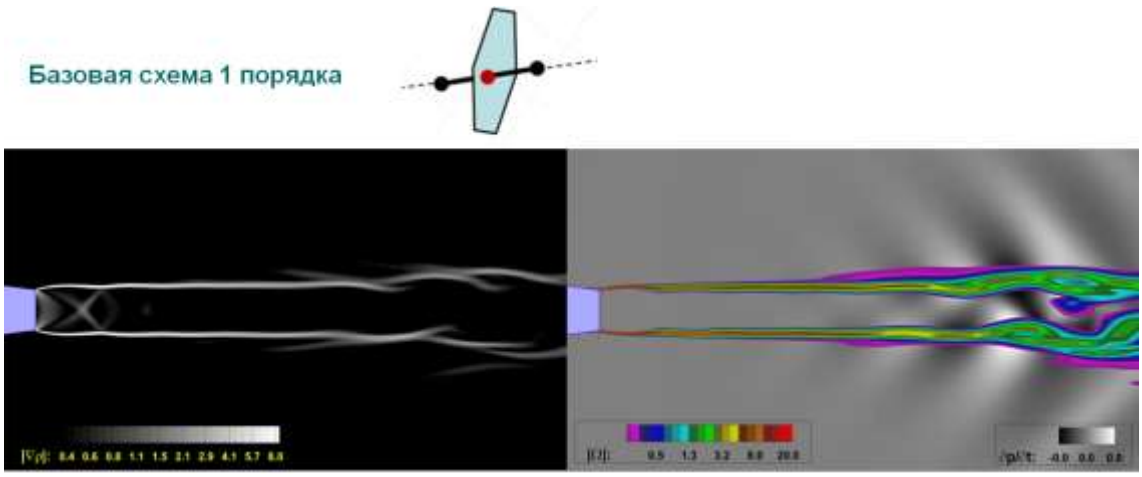
Получили следующий алгоритм:

1) реконструкция значений на гранях по значениям в ячейках (цикл по набору граней);
2) расчет потока через грань – решение задачи Римана о распаде разрыва (цикл по набору граней);
3) переход на новый временной слой: суммирование потоков через грани в ячейки (цикл по набору ячеек).

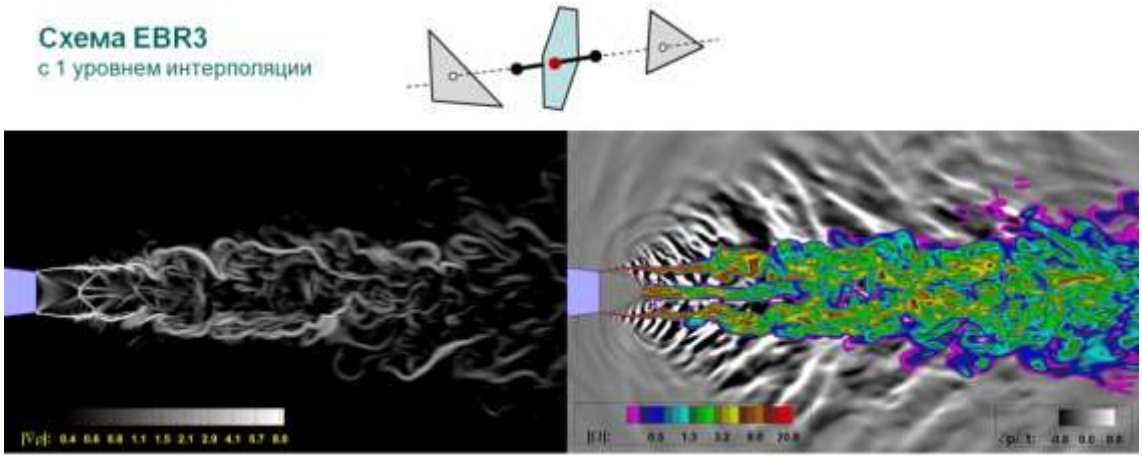
На каждом из этапов получается обработка набора из (очень) большого количества сеточных объектов (граней, ячеек и т.д.). Далее с помощью распараллеливания, то есть, распределяя эту обработку между множественными вычислительными устройствами, мы будем многократно ускорять расчет.

Но интересно, насколько же важна точность реконструкции на этапе 1? Рассмотрим примерчик – моделирование трехмерной реактивной струи, вытекающей из сопла круглого сечения. На трех картинках ниже показаны поля течения в центральном сечении. Поля показаны двумя способами: слева (на черном фоне) визуализация по градиенту давления, справа цветом выделены турбулентные структуры, а на сером фоне показаны акустические возмущения (поле производной давления по времени). Также показаны схемы интерполяционных конструкций, принцип работы которых в данном контексте не сильно важен. Важно, что, когда мы наворачиваем мудреную интерполяцию по значениям в соседних ячейках, сильно повышается точность.

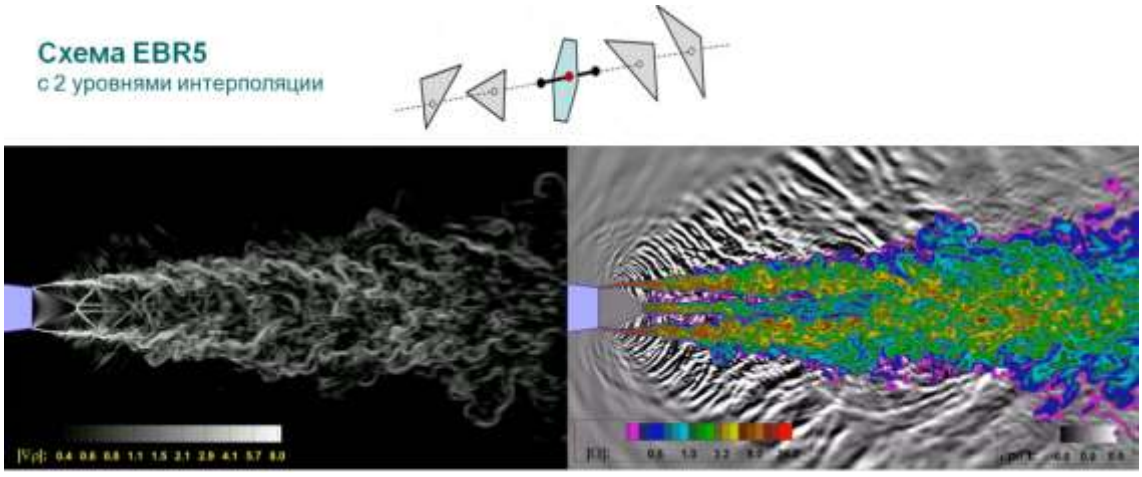
Если реконструкцию вообще не делать, просто брать на грань значения напрямую из ячеек, получится как на первой картинке ниже: ни турбулентности, ни акустики, вязкое ламинарное течение.




Задействуем схему повышенной точности с интерполяционными конструкциями EBR3. Результат показан на второй картинке ниже. Сразу видно, что появилась турбулентная картина течения, диски Маха, акустические возмущения.




Включим схему еще более высокой точности EBR5 с двухуровневыми конструкциями – появляются все детали течения, включая самые мелкие структуры.




Сетка во всех трех случаях одна и та же. Вычислительная стоимость расчета от применения схемы повышенной точности увеличилась менее чем раза в полтора. Чтобы достичь той же точности базовой схемой, понадобилась бы сетка раз так, наверное, в сотни

поподробнее. Соответственно, в сотни раз больше процессорного времени. Выигрыш налицо. А что, так можно повышать точность до бесконечности, и получать такие картинки даже на очень грубой сетке? Увы, нет. Все равно упремся в необходимость разрешать масштаб мелких вихрей, хотя бы по несколько ячеек на вихрь надо обеспечить, как ни крути.

Более подробно про EBR схемы написано тут, если вдруг кому надо:

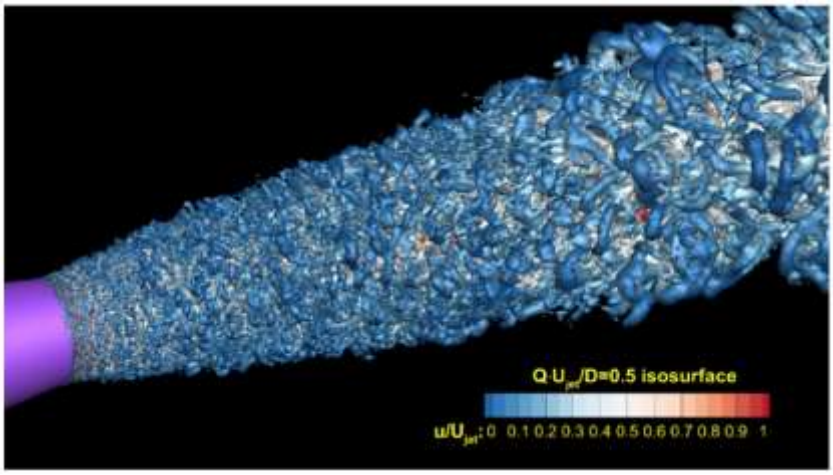
- Bakhvalov Pavel, Abalakin Ilya, Kozubskaya Tatiana. Edge-based reconstruction schemes for unstructured tetrahedral meshes. International Journal for Numerical Methods in Fluids. 2016. Vol.81(6). P. 331–356. https://doi.org/10.1002/fld.4187
- Bakhvalov Pavel, Kozubskaya Tatiana. EBR-WENO scheme for solving gas dynamics problems with discontinuities on unstructured meshes. Computers and Fluids. 2017. Vol. 157, p. 312-324. https://doi.org/10.1016/j.compfluid.2017.09.004

## 1.7 Примеры суперкомпьютерных приложений

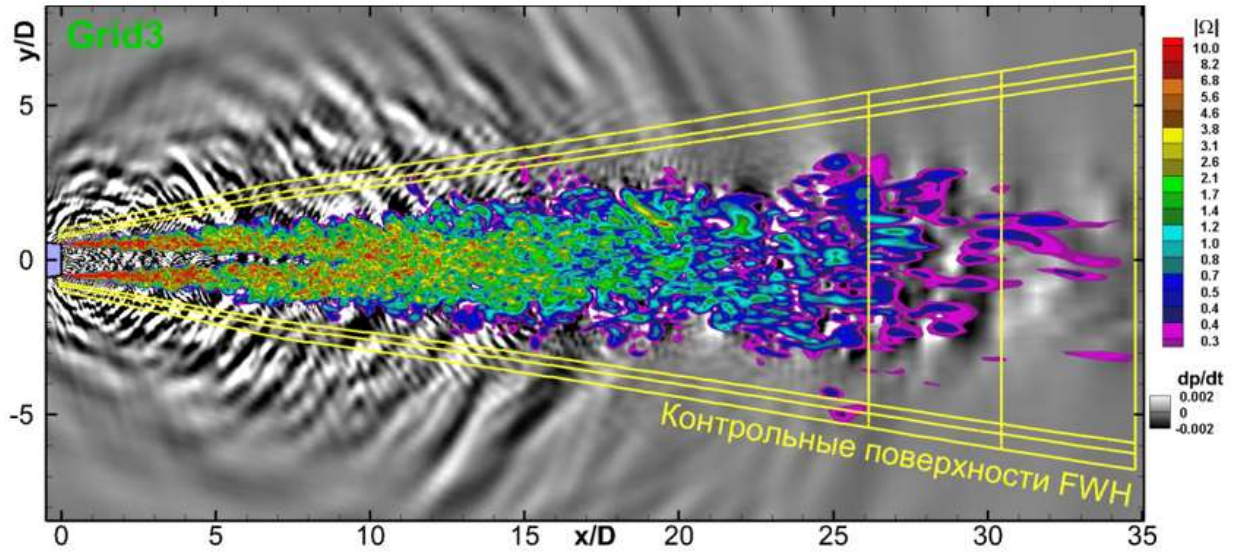
Обычно при организации ресурсоемкого численного исследования последовательность такая: сначала на какой-то модельной конфигурации, для которой есть эталонные данные, настраивается численный метод и вся вычислительная технология. Проверяется точность результатов, исследуется, где можно сэкономить, какое требуется сеточное разрешение, какое разрешение по времени, какой интервал интегрирования по времени необходимы для получения результатов нужной точности. Потом уже выполняются исследования в интересующей заказчика постановке. Далее рассмотрим некоторые примеры.

**Моделирование затопленной ненагретой дозвуковой круглой струи.** Исследуем на модельной задаче потребное сеточное разрешение для данного численного метода. Параметры струи: число Рейнольдса по диаметру сопла $Re_D$=1.1·10$^6$, число Маха $M$=0.9.
Сетки: 1.52 млн (Grid 1), 4.13 млн (Grid 2), 8.87 млн (Grid 3).
Моделирование турбулентности: подход DDES (Detached DES).

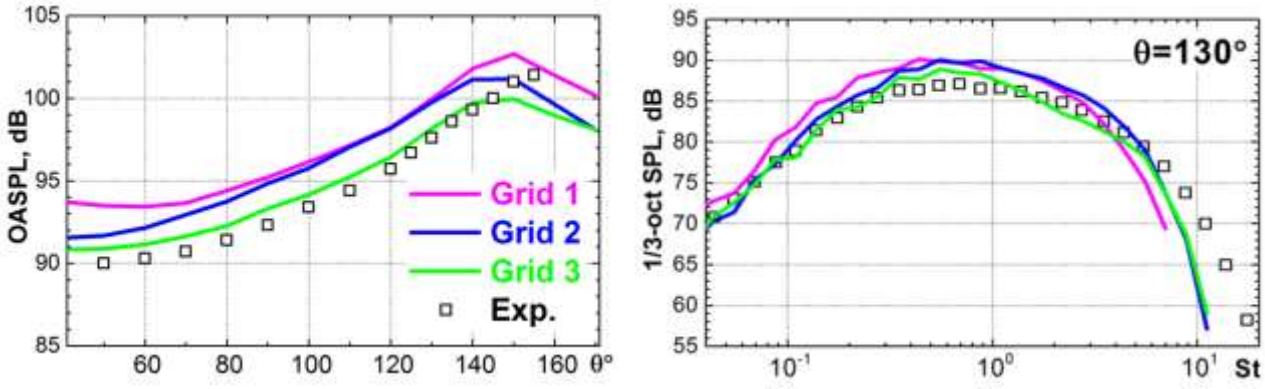
На картинке ниже показаны турбулентные структуры в трехмерном виде (визуализация на основе Q-критерия). Выглядит как-то странно, мягко говоря. Эти червячки показывают структуру вихрей.




На следующей картинке моментальная картинка течения в центральном сечении. Цветом подсвечена турбулентность (по модулю вектора завихренности) на фоне показанного в серых цветах поля производной давления по времени (на котором хорошо просматриваются акустические волны).




На этой картинке еще есть желтые линии. Они показывают позиции контрольных поверхностей. По эволюции во времени физических величин на этих поверхностях с помощью интегрального метода FW–H (Ffowcs Williams and Hawkings) находятся акустические характеристики для удаленного наблюдателя. По результатам обработки данных на контрольных поверхностях получаются диаграммы направленности (в каком направлении и с какой силой шумит моделируемая конфигурация), спектры шума в дальнем поле (дальнем – далеко от моделируемого объекта, для удаленного наблюдателя). Вот, например, диаграмма направленности от этой струи, полученная на разных сетках в сравнении с экспериментом (слева) и спектр шума для какой-то позиции удаленного наблюдателя:



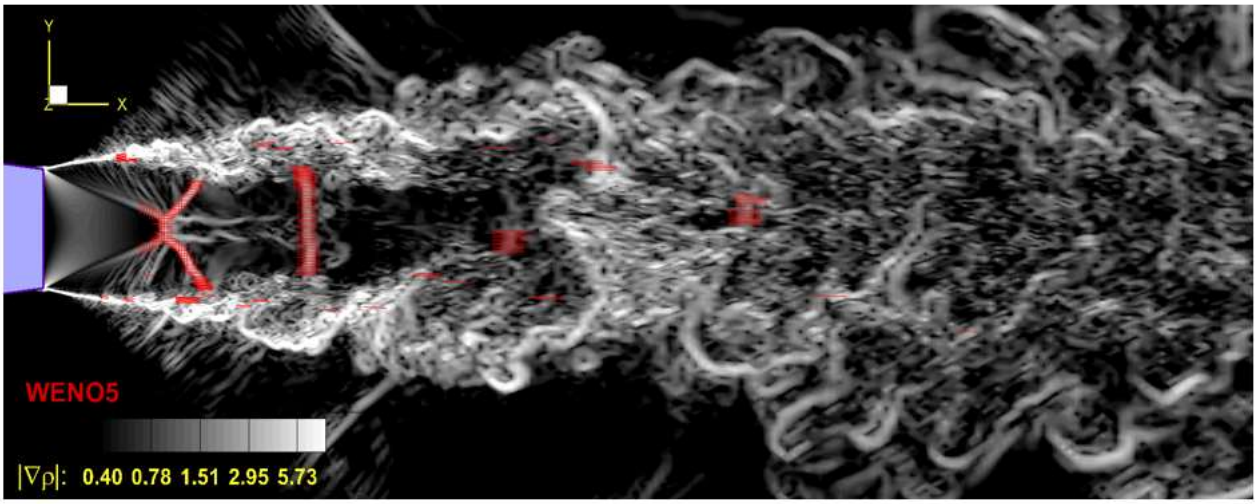

Вот аналогичное численное исследование для другой конфигурации: **горячая недорасширенная круглая струя** (сверхзвуковое течение на выходе из сопла).

Параметры струи: $Re_D=1.27\cdot10^6$, число Маха на выходе из сопла $M=2.18$.

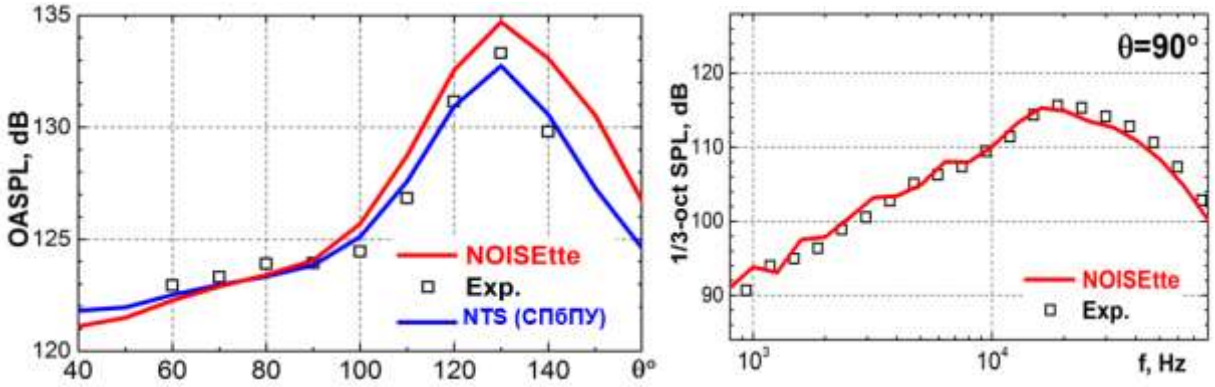
Сетка: 4.55 млн узлов.

Моделирование турбулентности: DDES.

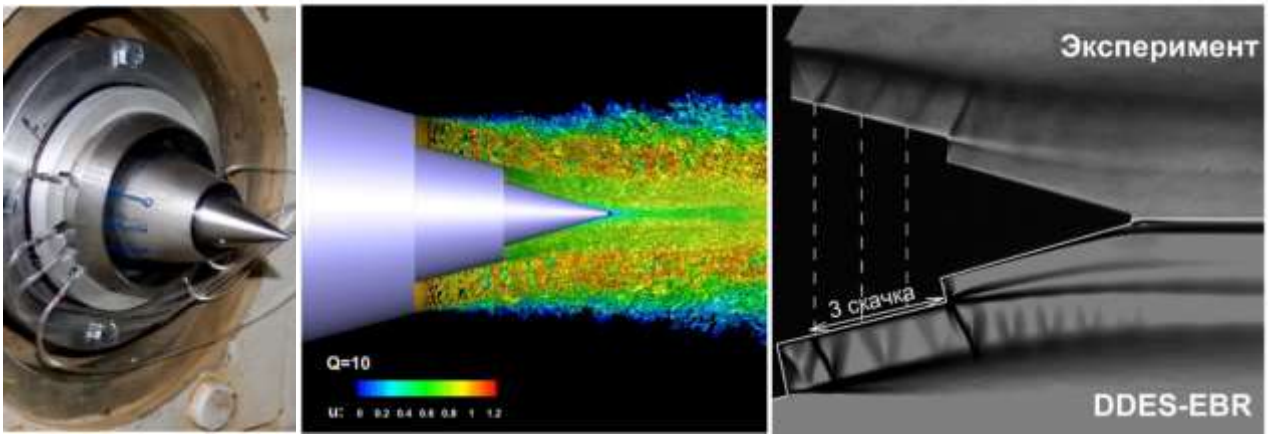
Вот на картинке моментальная картина в центральном сечении на основе модуля градиента плотности (численный аналог шлирен-изображения). Красным цветом отмечены места, где численная схема автоматически переключается в конфигурацию для разрывных решений (где сильные скачки уплотнения).




Получили аналогичные графики – диаграмма направленности шума и спектр шума для какой-то позиции удаленного наблюдателя:

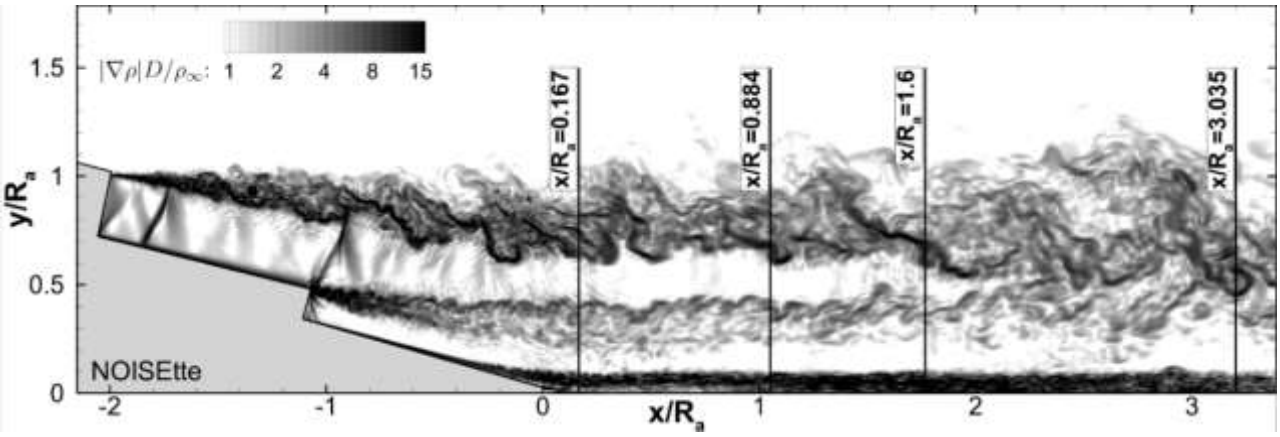



Разобрались с обычными струями с одиночным соплом, все сошлось, переходим к моделированию двухконтурного сопла. Сначала разбираемся с постановками, имеющими эталонные данные, например, эксперимент ИТПМ СО РАН (В. И. Запрягаев):

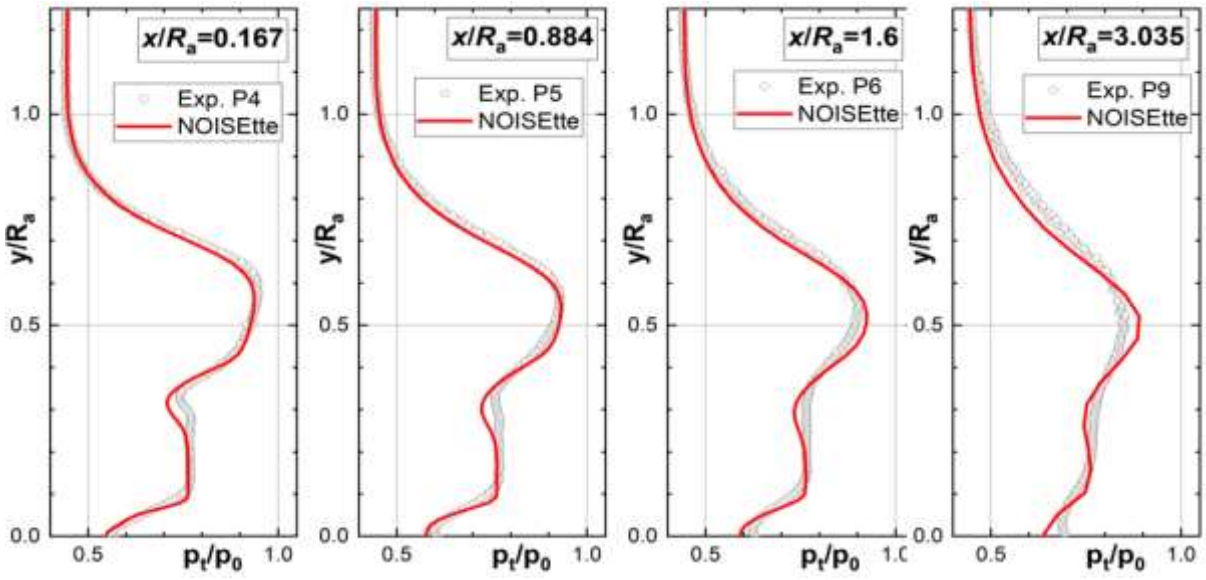



На этой картинке экспериментальная установка (слева), численное решение – моментальная картина турбулентности (по центру, Q-критерий), шлирен-фотография с эксперимента, совмещенная с осредненным численным решением (справа). На вид похоже.

Посмотрим подробнее, построим графики, всякие осредненные профили полного давления (вдоль линий на картинке ниже) на разном удалении от сопла.



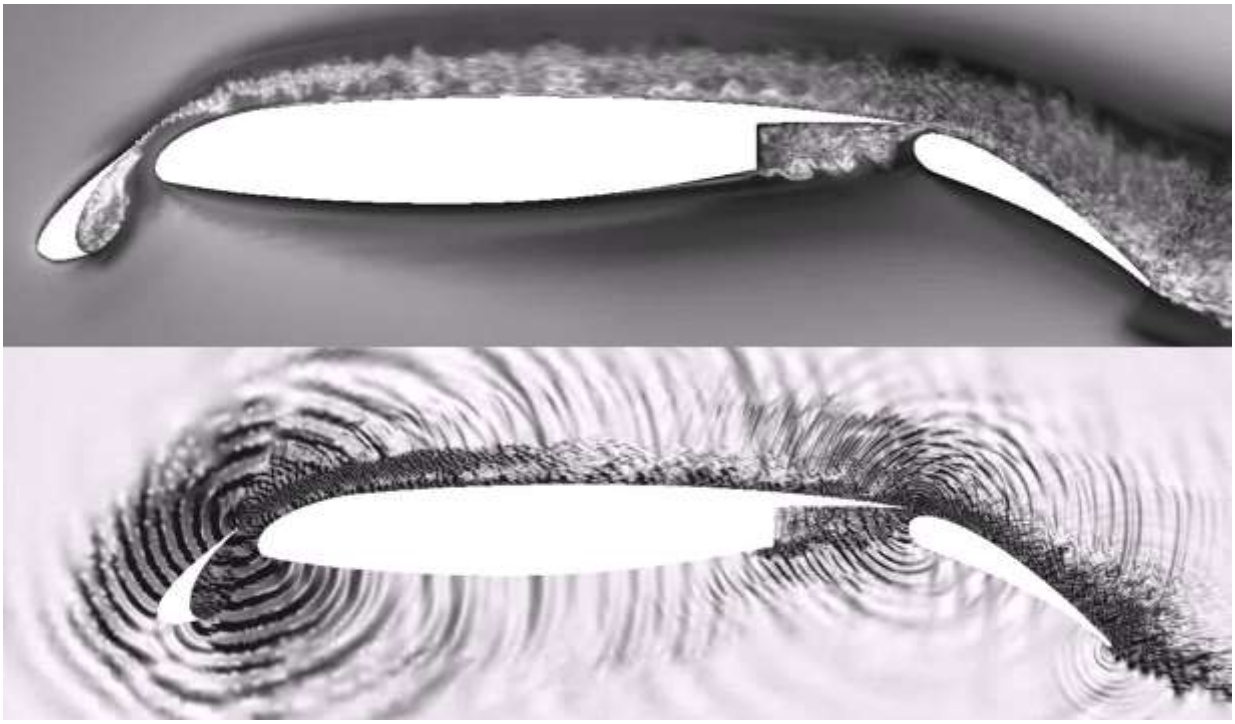

Сравним полученные численные результаты с экспериментом, убедимся, что все в порядке.




Теперь можно переходить к моделированию реальной геометрии в конфигурации заказчика, которая тут, естественно, не может быть показана. Вышеописанное исследование опубликовано тут: С.М. Босняков, А.В. Волков, А.П. Дубень, В.И. Запрягаев, Т.К. Козубская, С.В. Михайлов, А.И. Трошин, В.О. Цветкова. Сравнение двух вихреразрешающих методик повышенной точности на неструктурированных сетках применительно к моделированию струйного течения из двухконтурного сопла // Матем. моделирование, 31(10) (2019) 130-144

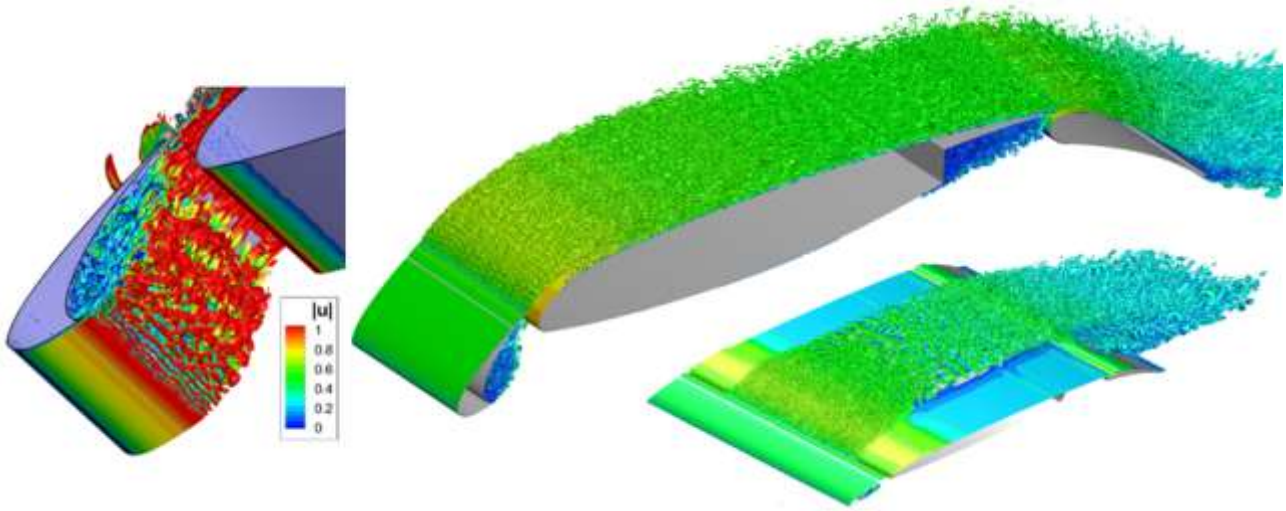
**Моделирование крыла с выпущенной механизацией**. Заказчику нужно предсказывать аэродинамические и акустические характеристики крыла авиалайнера в режиме захода на посадку. Такая же история. Берем наиболее близкую модельную конфигурацию, имеющую референсные (надежные, эталонные) данные, например, конфигурацию 30P30N для прямого крыла с предкрылком и закрылком, для которой есть эксперимент: Pascioni, Kyle & Cattafesta, Louis. 2016. Aeroacoustic Measurements of Leading-Edge Slat Noise. 22nd AIAA/CEAS Aeroacoustics Conference. https://doi.org/10.2514/6.2016-2960/

Делаем серию расчетов в удешевленной за счет периодических по размаху граничных условий постановке. Моделируем сегмент крыла (1/9 хорды), угол атаки 5.5°, Re = $1.711 \cdot 10^6$ (по хорде с убранной механизацией), M = 0.17. Исследуем потребное сеточное разрешение. Адекватные результаты получаются на сетке из 25 – 35 млн узлов. Вот моментальная картинка течения в центральном сечении:

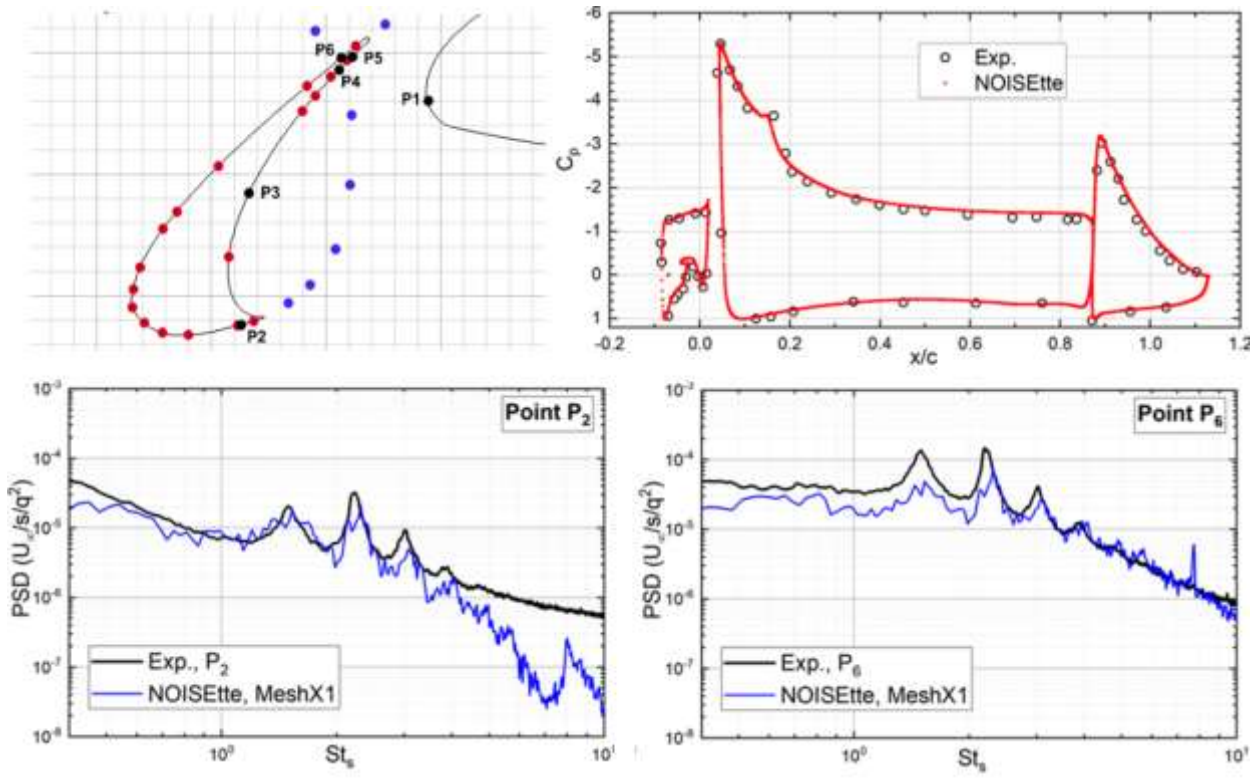


Сверху подсвечена турбулентность, снизу пульсации давления (то есть акустика).

В 3D картинка течения выглядит следующим образом:




Это показана визуализация на основе Q-критерия (справа снизу – случай стреловидного крыла). Но кому нужны эти цветные картинки? Главное, что там с графиками (картинка ниже). Смотрим распределение коэффициента давления по поверхности крыла, сравниваем с экспериментом (сверху справа), смотрим спектры пульсаций турбулентного течения в контрольных точках (расположение точек сверху слева, сами спектры – снизу). От того, насколько хорошо попали в частоту и амплитуду основных пиков зависит, насколько точно получатся акустические характеристики.




Разобравшись с модельной конфигурацией, настроив метод, оптимизировав сеточное разрешение, беремся за геометрию заказчика.

**Моделирование винтов вертолета**. На примере винта вертолета рассказывалась суть алгоритма в разделе **Ошибка! Источник ссылки не найден.**. Там уже были картинки, тут уже и нечего добавить. В общем, принцип тот же – считаем модельные винты, настраиваемся, потом считаем винты заказчика. Несущие винты, рулевые винты. Находим аэродинамические характеристики – сила тяги, момент, акустические нагрузки, диаграммы направленности при разных углах установки лопасти, на разных режимах полета. Вот, например, для рулевого винта в кольце расчетик:

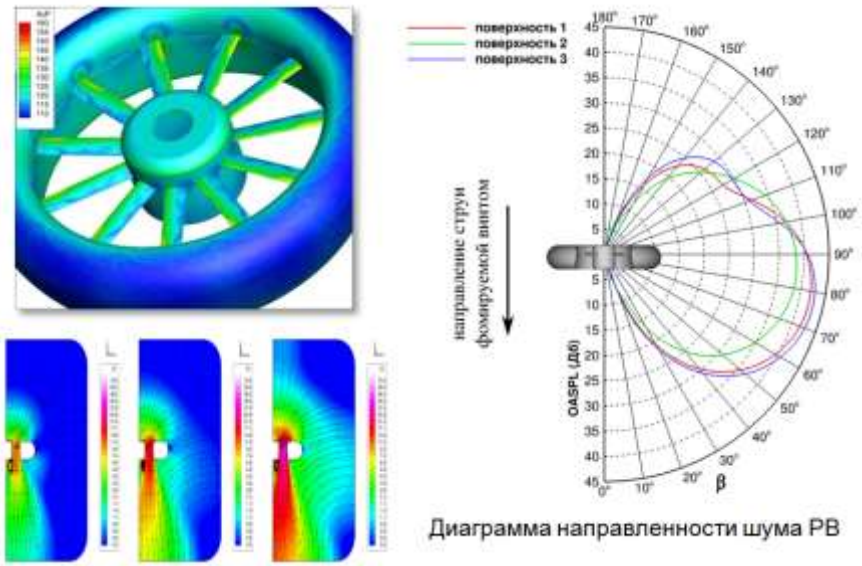
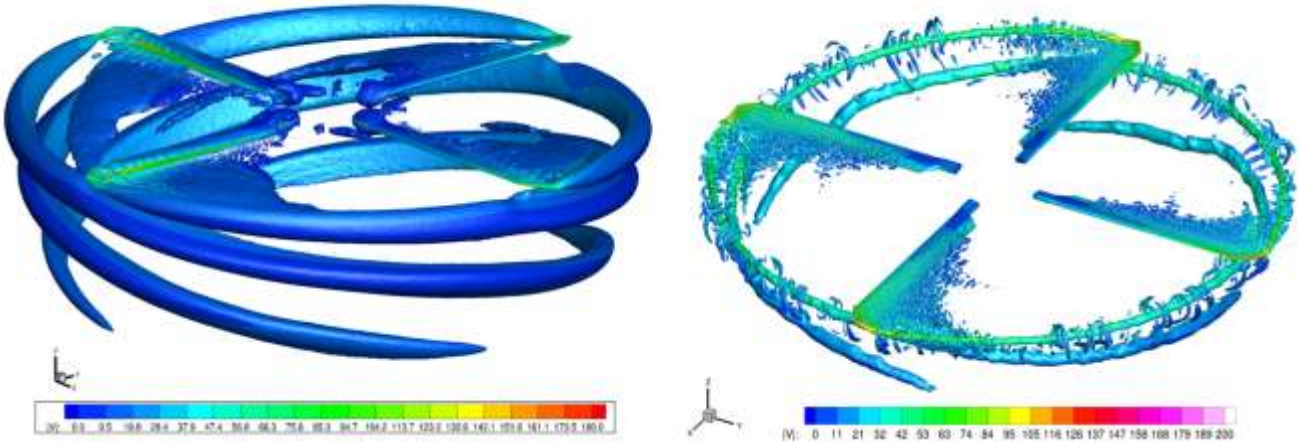


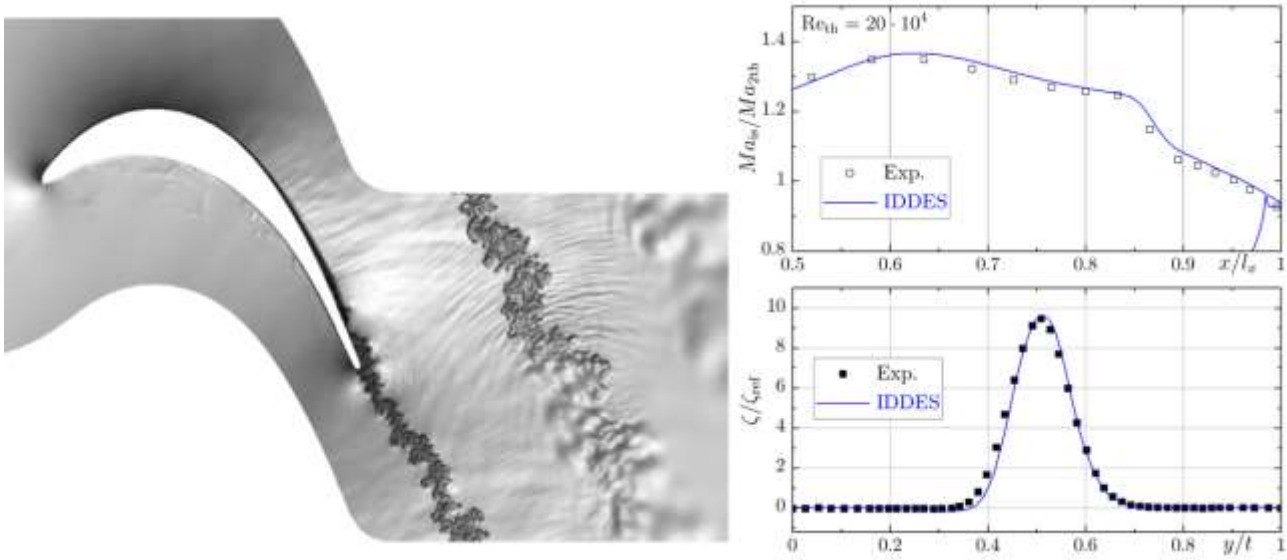

Диаграмма направленности шума РВ

Тут слева сверху – пульсационные нагрузки на поверхности, снизу – картина течения (вертикальная скорость) при различных углах; справа – диаграмма направленности шума.

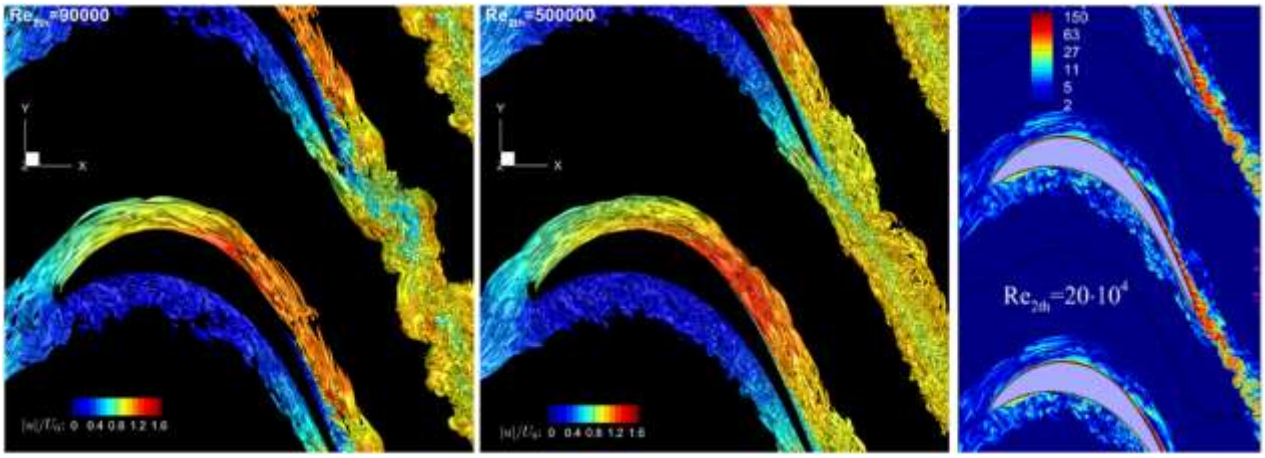
Да, тут вот еще что можно показать – чем отличается RANS расчет от вихреразрешающего. Вот снизу слева – “стационарная” картина течения, полученная RANS подходом, справа – вихреразрешающий расчет. Отличия очевидны.



**Моделирование турбинной решетки.** Это из области задач турбомашиностроения. Целью расчетов является исследование эффекта потерь полного давления в турбинной высоконагруженной решетке при уменьшении числа Рейнольдса. Опять же, берем что-то модельное, хорошо известное, например, турбинную решетку T106C. Настраиваем гибридный метод IDDES, подключаем генератор синтетической турбулентности на входе в расчетную область, считаем, сравниваемся с экспериментом, если требуется, корректируем параметры метода и сеточное разрешение. Вот ниже слева показана моментальная картина течения в межлопаточном канале (шлирен-картинка по градиенту плотности), а справа сравнение с экспериментом – профиль изоэнтропического числа Маха (что бы это ни значило) на поверхности лопатки (сверху) и график потерь полного давления (снизу).




Разобравшись с экспериментальными данными, изучаем течение при разных числах Рейнольдса. Окончательно убедившись, что корректно воспроизводится нужный эффект, обсчитываем геометрию заказчика.

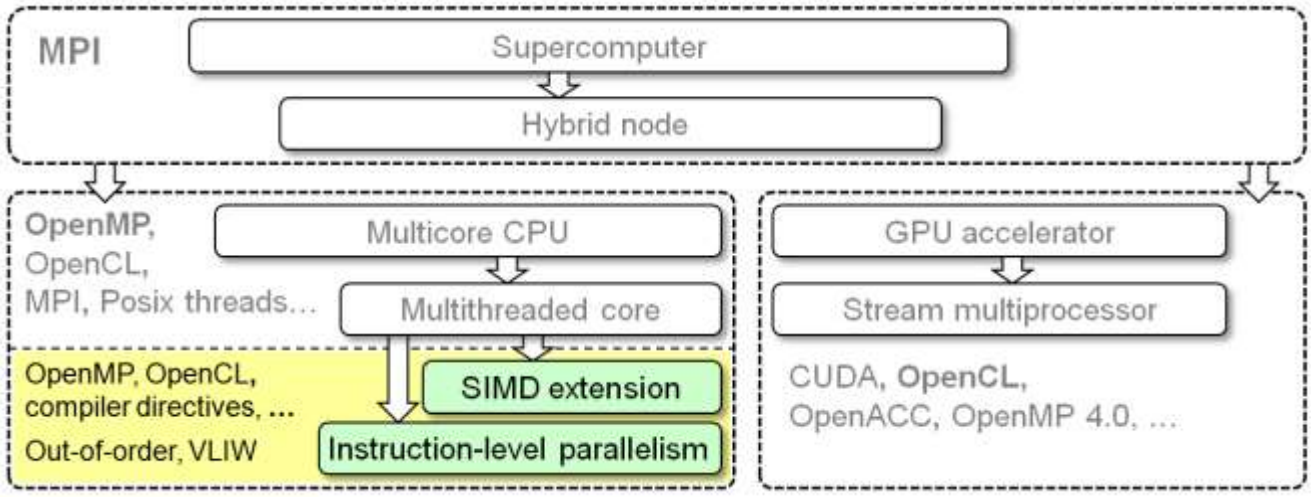



Подробности должны быть тут, если что: А. П. Дубень, Т. К. Козубская, О. В. Маракуева, Д. В. Ворошнин. Численное моделирование течения в турбинной высоконагруженной решетке при низких числах Рейнольдса // Изв. РАН. МЖГ, 2021.

Да и хватит, наверное, с примерами. Картинки разные, а смысл примерно одинаковый. Тем более, эти расчеты быстро устаревают и скоро будут выглядеть совсем по-детски (что, уже выглядят?). Тут просто для примера показаны эти все картинки с расчетов, чтобы было какое-то представление об одном из многих направлений, где применяются параллельные высокопроизводительные вычисления, о которых пойдет речь в следующих главах. Эти расчеты выполнены в нашей научной группе, которая сейчас называется Сектор вычислительной аэродинамики и аэроакустики Института прикладной математики им. М. В. Келдыша РАН. Подробности про эти и другие расчеты есть на сайте http://caa.imamod.ru/, в частности:
http://caa.imamod.ru/index.php/research/computations – разные мультики с расчетов;
http://caa.imamod.ru/noisette – там про расчетный код и есть ссылки на статьи по расчетам.

# 2 Параллелизм процессорного ядра, устройство памяти

В этом разделе мы будем "качать базу". Ну, вы знаете – жим лежа, становая тяга, присяд со штангой. Разберемся с основными понятиями по производительности вычислений, с внутренним параллелизмом ядра.




## 2.1 Производительность вычислений, основные понятия

Разберемся с понятиями, характеризующими вычислительное устройство, процессор или сопроцессор-ускоритель, aka *девайс* (*жарг.* от англ. device), с точки зрения производительности. Обозначение: aka – Also Known As (Георгий Иваныч, он же Гога, он же Гоша, он же Юрий, он же Гора, он же Жора здесь проживает?).

- **IPC** (Instructions Per Cycle) – количество инструкций на такт.
- **CPI (**Cycles Per Instruction) – количество тактов на инструкцию.

Обычно это характеристики выполнения конкретной программы на конкретном вычислительном устройстве. Они показывают, сколько в среднем получается выполнить инструкций (машинных команд) в пересчете на один такт работы процессора, и наоборот, сколько уходит тактов на одну инструкцию. Если взять максимальное теоретически достижимое значение IPC (или минимальное CPI), то это будет характеристика устройства. Ну и для простоты даже можно отбросить "максимальное ...". Если IPC > 1 (CPI < 1), то процессор называют **суперскалярным**. Чтобы получить производительность процессора, надо домножить IPC на тактовую частоту процессора. Получим следующую величину.

- **IPS (**Instructions Per Second) – количество инструкций в секунду.

Но нас, конечно, в первую очередь интересует вычислительная производительность. Значит, нам интересны не все инструкции, а только инструкции, выполняющие арифметические операции с вещественными числами с плавающей точкой. Отсюда следующая величина.

- **FLOPS** (FLoating point Operations Per Second) **–** количество операций с плавающей точкой инструкций в секунду.

Для обозначения самих арифметических операций с плавающей точкой можно использовать (немного жаргонное) обозначение **FLOP** (FLoating point OPeration), aka флоп.

Под флопами обычно имеются в виду **сложение** и **умножение** вещественных чисел. Вычитание – это сложение, только с обратным знаком (К.О.). Деление формально тоже флоп, но оно обычно заметно дороже и делается отдельным юнитом. Поэтому мы всячески

стараемся минимизировать количество делений в коде в критических по производительности местах (не говоря уже о синусах и прочих логарифмах).

Да, вот еще понятие. Юнитом (**unit**, aka execution unit, aka functional unit) или **исполнительным устройством** называется внутренняя составная часть процессора, которая выполняет какую-то машинную команду. У процессора много разных исполнительных устройств: арифметико-логическое устройство ALU (Arithmetic and Logic Unit); LSU (Load-Store Unit), отвечающее за загрузку и выгрузку данных; AGU (Address Generation Unit), отвечающее за вычисление адресов в памяти; разные другие устройства, всякие там сдвиги (Shift), ветвления (Branch). Но нас в первую очередь интересует исполнительное устройство **FPU** (Floating Point Unit), выполняющее арифметические операции с числами с плавающей точкой. Зачастую FPU реализуют операцию совмещенного умножения и сложения **FMA** (Fused Multiply–Add). Такое устройство может называться – FMA unit. Ну или более точно – **FP FMA** (FP – Floating Point). А то мало ли, FMA может быть и для целочисленных значений, ака *интóв* (*жарг.* от типа данных int). Совмещенное умножение–сложение – это операции FMA3 с тремя аргументами, a = a + b×c (b = a + b×c, ...), или FMA4 с четырьмя аргументами, a = b + c×d. Сколько флопов в такой операции? Правильно, два. Одна команда **FMA = 2 FLOP**.

- **FLOP per cycle** – количество флопов на такт.

Эта характеристика равна произведению максимального FP IPC на количество флопов в одной инструкции. А количество флопов в скалярной FMA инструкции – два. В случае векторной инструкции, то есть, когда одна инструкция применяется сразу к нескольким наборам аргументов, надо еще умножить на ширину вектора. Ширина вектора определяется как отношение ширины векторных регистров к разрядности представления числа с плавающей точкой. Обычно используется либо одинарная точность – 32 бита, т. е. 4 байта (**FP32**, *жарг.* флоаты́ – от типа float), либо двойная точность – 64 бита, т. е. 8 байт (**FP64**, *жарг.* даблы́ – от типа double). Для процессора с FMA юнитом шириной 512 бит (AVX-512, например**)** с max IPC = 1 величина FLOP per cycle для даблов будет 512/64×2 = 16. Ну да, есть еще половинная точность, применяемая во всяком ~~мышином~~ машинном обучении (под нее даже приделывают отдельные юниты типа tensor core у NVIDIA). Но это не из нашей оперы.

- **FLOPS per watt** – характеристика энергоэффективности вычислений.

- **TPP** (Theoretical Peak Performance) – теоретическая пиковая производительность, aka Rpeak.

Это характеристика вычислительного устройства, единица измерения – **FLOPS.** Она говорит, сколько устройство может в теории максимально выдавать FLOPS. Эта величина вычисляется как произведение тактовой частоты девайса на его FLOP per cycle (максимальное количество FLOP, которые процессор может делать за один такт). Конечно же, мы знаем, что такое тактовая частота (aka frequency, clock rate, clock speed)? Ну да, это частота, с которой тактовый генератор процессора выдает синхронизирующие импульсы. Посчитаем TPP для какого-нибудь процессора. Пусть процессор имеет тактовую частоту 2.5 ГГц, состоит из 16 ядер, каждое ядро имеет по два векторных FP FMA юнита шириной 512. TPP будет 16 (ядер) × 2.5 (ГГц) × 2 (юнита) × 8 (ширина вектора) × 2 (сложение и умножение в FMA) = 1280 GFLOPS (G – Giga). Или 1.28 **TFLOPS** (T – Tera). Или 0.00128 PFLOPS (P – Peta)...

- **Bandwidth** (aka бэндвис, обозначим для краткости BW) – пропускная способность памяти.

Это характеристика вычислительного устройства, обычно единица измерения – GB/s (ГБ/с). На практике эта характеристика заметно важнее, чем FLOPS. Если кто-то хвастает, что у них кластер на очень много FLOPS, сразу можно спустить на землю вопросом – а как на счет соотношения флопсов к бэндвису?

Теперь посмотрим на производительность с точки зрения алгоритма или программы.

- **Arithmetic Intensity** (обозначим **AI**, не путать с Artificial ~~Idiocy~~ Intelligence) – арифметическая интенсивность, aka lop-per-byte, FLOP/byte, вычислительная плотность.

Это соотношение количества операций и объема данных в памяти, с которыми эти операции делаются. Или даже не объема, а суммарного трафика между процессором и памятью. Рассмотрим примеры. Пусть есть два вектора X, Y размера N. В векторах хранятся даблы, то есть FP64 значения с плавающей точкой (каждое по 8 байт). Какая будет арифметическая интенсивность у поэлементного сложения двух векторов, X[i] = X[i] + Y[i]? Число флопов – N. Данных в памяти – 2×N даблов на чтение, N даблов на запись, итого суммарный трафик 3×8 (байт) × N. AI получается дико низкая – 1/24. У скалярного произведения AI будет 2×N флопов (на самом деле 2N–1, но пренебрежем такими мелочами) на 2×8×N байт зачитки (*жарг.* чтение данных из памяти) итого 1/8. Уже лучше. Такие операции или алгоритмы, когда вычислений мало, а выгребать из памяти надо много данных, называются **memory bound**. Если надо сильно много считать, то – **compute bound**. А как посчитать AI для матрично-векторного произведения? С одной стороны, объем данных будет N×N (матрица) + N (входной вектор) + N (выходной вектор) и все это на 8 байт = 16×N×(N+2). Но, с другой стороны, для каждой строки мы зачитываем весь входной вектор каждый раз заново, и получается, что по суммарному трафику 16×N×(2N+1). Как бы в два раза больше. Получилась оценка снизу и оценка сверху. А что будет по факту – будет зависеть, например, от того, насколько большое это N, насколько эффективно мы будем использовать кэш память, и какого она будет размера. За счет повышения повторной используемости данных в кэш мы снизим суммарный трафик в сторону нижней оценки по объему данных.

- **Computational complexity**, aka вычислительная сложность.

Обычно обозначает асимптотику по числу операций в зависимости от размера входных данных. Например, сложность поэлементного сложения векторов размера N – O(N). Сложность скалярного произведения – тоже O(N). У матрично-векторного произведения с плотной матрицей сложность $O(N^2)$. Аналогично, сложность может быть по потреблению памяти, по суммарному трафику.

- **Computational cost**, aka вычислительная стоимость.

Обычно обозначает количество флопов для решения конкретной задачи. Для поэлементного сложения векторов размера N – N, для скалярного произведения – 2N–1, матрично-векторного произведения с плотной матрицей – N(2N–1). Аналогично, стоимость может быть по потреблению памяти, по суммарному трафику.

Вычислительную стоимость и потребление памяти можно называть общим словом – **ресурсоемкость**.

Теперь попробуем совместить свойства алгоритма и девайса.

- **TBP** (Theoretical Bounded Performance) – теоретически достижимая производительность.

Это некоторая оценка сверху, какой производительности потенциально может достичь конкретный алгоритм на конкретном девайсе. Аббревиатура TBP взята уже просто с потолка,

чтобы как-то обозначить в тексте. В простейшем случае TBP говорит, какой перф (*жарг.* от англ. performance) можно выжать при данных AI, TPP и BW: TBP = MIN(TPP, BW×AI). Для compute bound более значимым фактором будет TPP, для memory bound будет чисто BW. Оценим TBP на процессоре с TPP = 1280 GFLOPS и BW = 128 ГБ/с для операции поэлементного сложения векторов. Получим 128 × 1/24 = 5.3 GFLOPS. То есть **0.4%** от TPP! Неплохо, да? Ну и о чем, спрашивается, в наше время говорит характеристика вычислительной системы в FLOPS? Правильно. Ни о чем. Вот простейший пример, где пол процента от пика – предел мечтаний.

Если мы начнем учитывать другие характеристики, такие как иерархия памяти, размеры кэш, латентность доступа, то оценка сильно усложнится. Одна из методик оценки – Roofline model (см. S. Williams, A. Waterman, D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM. 52 (4): 65–76. PDF доступен по ссылке https://zenodo.org/record/1236156#.XyUKgm0zY-U)

- **Sustained performance**, aka Rmax, фактически достигаемая установившаяся (то есть в среднем на каком-то длительном времени) производительность**.**

Обычно даже при хорошем раскладе она заметно меньше TBP и сильно драматически меньше TPP. Эту производительность мы замеряем на реальных задачах в реальных расчетах, получаем жалкие крохи от TPP, расстраиваемся, возимся с оптимизацией, пытаясь её повысить.

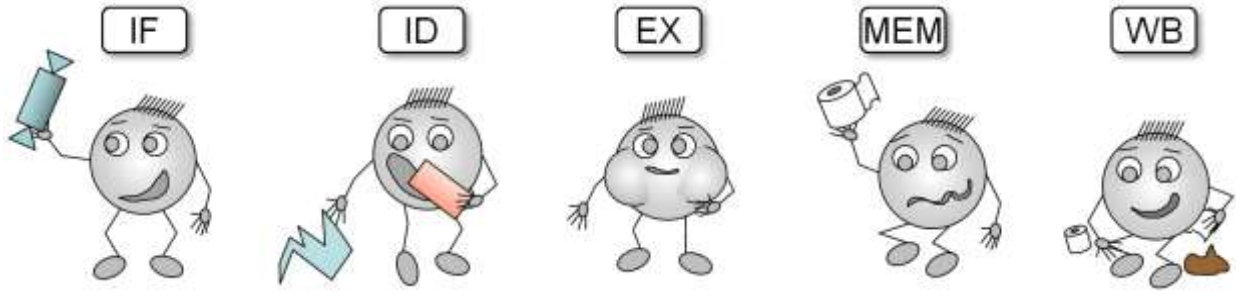
## 2.2 Параллелизм уровня инструкций

Имеется одно процессорное ядро, обрабатывающее один поток команд (инструкций – синоним), которые командуют одним потоком данных. Самый что ни на есть SISD (не) параллелизм.

Параллелизм уровня инструкций позволяет выполнять одновременно множественные **независимые** между собой команды из этого одиночного потока команд. Вся суть поиска этого параллелизма – **борьба за независимость**. Компилятор и процессор стараются так сделать машинный код и упорядочить команды, чтобы как можно больше исполнительных устройств процессора было задействовано одновременно.

### 2.2.1 Конвейерный параллелизм

Посмотрим на что-нибудь простое. Например, в классической RISC (Reduced Instruction Set Computer) архитектуре, помимо прочих интересных особенностей, инструкции обрабатываются в пять этапов: IF (Instruction fetch) – инструкция загружается из памяти; ID (Instruction Decode) – инструкция декодируется, то есть выясняется, какие ей нужны регистры, и что она вообще собирается делать; EX (EXecute) – выполнение инструкции на соответствующем исполнительном устройстве; MEM (MEMory access) – доступ к памяти; WB (WriteBack) – запись результата в регистровый файл.

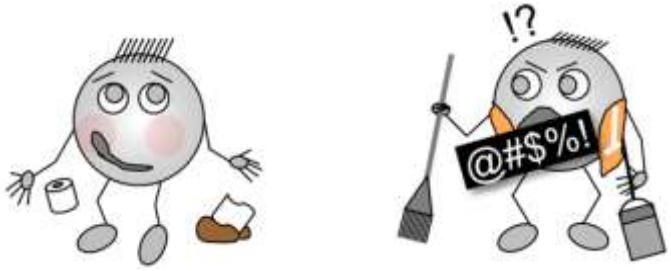


Это что же получается, если пять стадий, то CPI будет как минимум 5? И как с этим жить!? Чтобы получить CPI = 1 используется, например, конвейерный параллелизм. Команды обрабатываются 5-стадийным конвейером (five-stage pipeline). Это базовый классический конвейер RISC архитектуры. Нет, не у всех процессоров именно такие конвейеры, скорее наоборот. Зачастую в процессорах навёрнуты конвейеры на десятки стадий. Но это не важно. Суть данного вида параллелизма в том, что разные исполнительные устройства процессора, работающие на разных стадиях конвейера, могут работать одновременно на одном такте. За счет такого параллельного выполнения в идеале достигается CPI = 1, то есть выполнение по одной команде за один такт.

- **Конвейер** (pipeline) – конвейерный параллелизм при обработке инструкций, требующих множественных тактов процессора, позволяет достигать темпа выдачи в одну инструкцию на такт, CPI = 1.
  Выглядит работа конвейера примерно так:

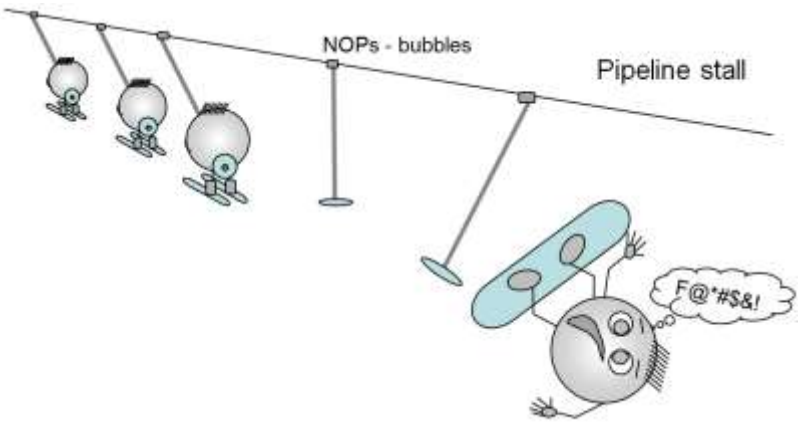
| | | Такты | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Команды | 1 | IF | ID | EX | MEM | WB | | | | |
| | 2 | | IF | ID | EX | MEM | WB | | | |
| | 3 | | | IF | ID | EX | MEM | WB | | |
| | 4 | | | | IF | ID | EX | MEM | WB | |
| | 5 | | | | | IF | ID | EX | MEM | WB |

На такте 5 можем видеть, что все пять стадий выполняются одновременно для 5 команд. Что же нужно для этой идиллии? Во-первых, команд должно быть как минимум 5. Во-вторых, эти команды должны быть **независимы** по данным, то есть, команды не должны использовать на вход результаты работы других команд, находящихся на конвейере. Иначе зависимой команде придется ждать завершения предыдущей команды, а конвейер будет простаивать. Это так называемый конфликт по данным (data hazard). Там бывают RAW, WAR, WAR (Read After Write, ...) хазарды. Вот, например, RAW конфликт (нельзя же убрать то, что еще не нагажено):




Также могут возникать другие неприятности. Например, структурные конфликты (structural hazard), когда разным стадиям конвейера понадобилось одно и то же исполнительное устройство. Или конфликты управления (control hazard), когда на конвейере выполняется, например, команда условного перехода, и не понятно, какие вообще команды выполнять дальше. Из-за этих конфликтов возникают простои конвейера (stall). Из-за хазардов в конвейере возникают пузыри (bubbles), ака NOPы (No-OPeration), когда на каких-то стадиях исполнительные устройства бездействуют в ожидании, пока конфликт разрешится сам собой. Чем больше конфликтов и вызванных ими простоев, тем ниже получается IPC.

Вот, например, все же бывали на горнолыжных курортах? Сноубордисты на бугеле – это типичный hazard (да ладно, сноубордисты, не обижайтесь, это шутка (но в каждой шутке... ;)).




Для борьбы с этими неприятностями используются разные способы оптимизации. С помощью data forwarding снижают простои на конфликтах по данным. Еще более их снижает внеочередное исполнение команд (out-of-order execution), но это уже немного другая история, с которой мы дальше разберемся. Для снижения простоев на конфликтах управления используется предсказание ветвлений (branch prediction), когда специальный юнит-угадыватель пытается предсказать до выполнения команды перехода, на какую ветку будет переход. Также используется спекулятивное исполнение, когда команды из ветки условного перехода наваливаются в конвейер еще до того, как будет известно, что до этой ветки действительно дойдет дело. Что-то делать всяко лучше, чем простаивать.

Также важно отметить, что выполнение стадии EX может занимать множественные такты. Это тоже может приводить к простоям. А сами исполнительные устройства, которые выполняют команду на стадии EX, также могут иметь свой внутренний конвейер. Например, ALU и FPU обычно конвейеризованы. Например, FPU процессора обычно делает сложение где-то за 3 такта, умножение за 5, FMA инструкцию, скажем, за 8 тактов. За счет конвейеризации FPU может делать по одной инструкции за такт. Но для этого надо, чтобы на FPU непрерывно поступали независимые по данным инструкции. Что не всегда происходит на практике.

### 2.2.2 Суперскалярность

Задача конвейера – для инструкций, требующих множественных тактов на выполнение, уменьшить CPI до 1, или, другими словами, увеличить IPC до 1. Достигается это за счет одновременной работы разнородных исполнительных устройств, задействованных на **разных** стадиях выполнения инструкции.

Чтобы сделать IPC > 1, необходимо, чтобы **множественные** инструкции могли находиться на **одной** стадии выполнения. В случае описанного выше конвейера на стадии EX находится только одна инструкция. Но если такой конвейер, например, сдвоить, чтобы на каждой стадии могло находиться по две команды, то получится суперскалярный конвейер, aka dual-issue pipeline. Табличка выполнения команд на таком конвейере приведена ниже. А если еще размножить, то будет multiple-issue pipeline. Но есть нюанс. То, что конвейер сдвоен и может обрабатывать сразу по две команды, не означает, что любые две команды могут одновременно находиться на стадии выполнения. Для этого также необходимо, чтобы исполнительных устройств, выполняющих эти команды, было два. Если в процессоре только одно FPU, то не получится IPC = 2 для арифметики с плавающей точкой. Но зато смогут

выполняться одновременно, например, вещественные и целочисленные арифметические операции, задействующие два разных юнита – FPU и ALU.

| | | Такты | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| Команды | 1 | IF | ID | EX | MEM | WB | | | | |
| | 2 | IF | ID | EX | MEM | WB | | | | |
| | 3 | | IF | ID | EX | MEM | WB | | | |
| | 4 | | IF | ID | EX | MEM | WB | | | |
| | 5 | | | IF | ID | EX | MEM | WB | | |
| | 6 | | | IF | ID | EX | MEM | WB | | |
| | 7 | | | | IF | ID | EX | MEM | WB | |
| | 8 | | | | IF | ID | EX | MEM | WB | |
| | 9 | | | | | IF | ID | EX | MEM | WB |
| | 10 | | | | | IF | ID | EX | MEM | WB |

Но если хочется выполнять заметно больше инструкций за такт, чем по две, то уже одним multiple-issue конвейером не отделаешься. В современном процессоре уже (или пока еще) все несколько сложнее. Можно, например, посмотреть диаграммы устройства разных процессоров на Викичипе. Вот широко использующийся в настоящее время серверный процессор:

https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(server)

В частности, можно посмотреть на диаграмму устройства ядра

https://en.wikichip.org/w/images/e/ee/skylake_server_block_diagram.svg

Если ссылка вдруг будет недоступна на момент чтения, ищем в интернете картинки по запросу "Intel Skylake block diagram" и находим их множество. Или уже не Skylake, а более актуальной микроархитектуры.

Из схемы видно, что ядро декодирует инструкции в 5 "смычков", наваливает инструкции в большой буфер, в котором они переупорядочиваются оптимальным образом, после чего поступают в 8-портовый планировщик (scheduler), который может отправлять на исполнение сразу до 8 инструкций на одном такте.

### 2.2.3 OoO vs VLIW

Когда команды поступают на конвейер, пусть даже суперскалярный multiple-issue pipeline, в исходном строго определенном порядке, то есть это in-order pipeline, то вероятность того, что они окажутся независимыми и смогут эффективно разместиться на конвейере без простоев, мягко говоря, невелика. Чтобы использовать параллелизм уровня команд, даже хотя бы задействовать эффективно конвейер, нужно как-то выделять этот параллелизм в потоке команд, определять независимые по данным команды, которые можно выполнять одновременно. Ключевое слово – **не-за-ви-си-мость** (произносим вслух по слогам). По сути, борьба за повышение производительности суперскалярного ядра – это борьба за независимость (команд друг от друга).

Ответственность за ILP может быть на стороне процессора, тогда это будет аппаратный ILP, а может быть на стороне компилятора.

- **Out-of-Order Execution** (OoOE) – внеочередное исполнение инструкций.

За параллелизм отвечает процессор. Инструкции, прежде чем поступить на исполнение, сначала загоняются (десятками, а то и сотнями) в буфер инструкций, aka scheduler, unified

reservation station (URS). Тут мы дальше углубляться не будем, там уже начинается сложно и не очень важно для разработчика программы. Для более подробной информации можно легко найти и почитать в интернете (и посмотреть видео) о том, как работает Tomasulo algorithm, как устроена URS, как с помощью register renaming устраняются зависимости по данным между инструкциями, и т.д. А тут главное понять суть: у процессора аппаратно реализован поиск параллелизма уровня инструкций и переупорядочивание инструкций для достижения более высокого IPC. Процессор сам расставляет инструкции таким образом, чтобы задействовать одновременно свои множественные исполнительные устройства. Одна инструкция складывает инты, другая умножает даблы, третья вычисляет адрес, четвертая что-то гребет из памяти, и все это – одновременно на одном такте. Идиллия.

- **VLIW – Very Long Instruction Word** – внеочередное исполнение инструкций.

За параллелизм отвечает компилятор. Компилятор выискивает параллелизм, сам упорядочивает инструкции оптимальным, по его мнению, образом и пакует их в широкие команды. Широкая команда состоит из нескольких операций для параллельного выполнения суперскалярным ядром процессора на разных исполнительных устройствах. То есть одна такая комби-команда может загрузить работой множественные юниты. Процессору не надо думать, как распределять команды, все уже распределено и прописано в машинном коде программы. Такая архитектура использовалась, например, в серверных процессорах Intel Itanium. Но мощный OoOE победил, Itanium ушел в закат. VLIW подход также раньше использовался в графических процессорах, в основном у AMD, но там тоже с него слезли. В настоящее время как пример живой VLIW архитектуры можно привести отечественные процессоры Эльбрус. Ядро какого-нибудь процессора Эльбрус-8С может исполнять аж до 25 различных элементарных операций в одном такте. Структура широкой команды (ШК) позволяет разместить до 6 арифметико-логических операций. Соответственно, для вещественной арифметики имеется 6 арифметико-логических устройств с полностью конвейеризированными устройствами умножения и сложения, позволяющих выполнять до 12 FLOP за такт с аргументами двойной точности. Звучит, конечно, заманчиво, но в дикой природе такие редкие экзотические звери почти не встречаются.

Для понимания рассмотрим еще пример из реальной жизни. Вот у процессора есть исполнительные устройства, которые надо по максимуму загружать, чтобы был больше показатель IPC. Вернемся на горнолыжный курорт. Но не на бугель, а на креселку. Ну или телекабину. Все же стояли в очереди на подъемник? Ведь обязательно же перед вами найдется группка, которая хочет непременно ехать вместе и сидеть рядом? Они же еще обязательно все перепутались в очереди и пытаются собраться вместе, стоят, ждут друг друга, блокируют проход, и кресла уезжают полупустыми. Знакомо? Вот он пример зависимости, которая приводит к снижению загрузки подъемного процессора. Если на подъемнике дежурит специально обученный сотрудник (что часто можно наблюдать в часы пиковой нагрузки), который выдергивает из очереди одиночек, ну или парочки, смотря, сколько мест свободно, и пропихивает их вперед вне очереди на пустые места, пока группки тормозят и кучкуются, – это аппаратный out-of-order. VLIW бы получился, если бы сразу при продаже скипаса сказали: "Так, вас, значит, трое, да? Ну вот вам в компанию еще тут есть парочка и еще один одиночка, вот вам один пас на шестерых, так все вместе и катайтесь, а то у нас шестикресельные подъемники".

К преимуществу OoOE можно отнести меньшую требовательность к компилятору и исходному коду. У OoOE заметно выше толерантность к плохому исходному коду. Не так

нужны всякие интринсики (*жарг.* от intrinsic function – встроенные функции компилятора), не нужно заморачиваться, проставлять по коду рестрикты, префетчи (__restrict спецификаторы, обещающие компилятору, что куски памяти, на которые указывают указатели, не пересекаются; __builtin_prefetch – предварительная подкачка данных из памяти). URS всё сама разрулит. Недостатком OoOE является существенное усложнение аппаратной части. Необходим сложный навороченный execution engine с мощным планировщиком с большим буфером команд. Ну и размер этого буфера фиксирован и весьма ограничен.

Преимуществом VLIW является, во-первых, более простая организация управления в процессоре, не нужен большой и сложный scheduler. Соответственно, (потенциально) выше энергоэффективность вычислений. Во-вторых, нет аппаратных ограничений на глубину поиска параллелизма инструкций. Компилятору предоставлен больший потенциал параллелизма. Если в OoOE reorder buffer содержит порядка сотни инструкций, то VLIW-компилятору доступен, по сути, весь исходный код. Ну не прям весь, а тот, который находится в одном файле. Поэтому VLIW любит inline функции. Чем больше кода доступно для анализа, тем больше глубина поиска. Компилятор также является и слабым местом. Он получается сильно сложный, на нем сильно много ответственности. Поэтому компилятор заметно более чувствителен к качеству исходного кода. Простановка рестриктов, префетчей и прочих интринсиков сильно влияет на производительность. Также приходится уделять внимание разрыву зависимостей в исходном коде (например, чтобы разные куски кода не использовали одну и ту же промежуточную переменную, надо плодить структурные блоки и описывать переменные строго по месту использования). Хотя, конечно, на интелах (*жарг.* от процессоры Intel) для автоматической векторизации тоже приходится ставить по коду всякие #pragma ivdep, обещающие независимость указателей. А когда эти обещания не выполняются, начинают происходить чудеса, появляются жэджик-баги (*жагр.* от magic bug – ошибки с нестабильным поведением).

### 2.2.4 Одновременная многопоточность

Вот еще одно важное понятие по теме параллелизма процессорного ядра, правда, уже формально не совсем целиком укладывающееся в SISD категорию.

- **SMT** (Simultaneous MultiThreading) – одновременная многопоточность.

SMT ядро имеет аппаратную поддержку выполнения множественных независимых потоков команд (как бы MIMD параллелизм). По сути это влечет сравнительно небольшие изменения: просто берется чуть побольше регистровый файл, чтобы вмещать данные с нескольких потоков, дублируются регистры состояния, всякие там счетчики команд, чтобы исключить оверхед (*жарг.* от overhead – накладные расходы) на переключении между потоками. Пример SMT – технология Hyper-threading (HT) в процессорах Intel, обеспечивающая аппаратную поддержку выполнения по два потока команд на ядро. Современные процессоры AMD (микроархитектуры Zen, Zen 2, например) также имеют двухпоточную SMT. Процессоры-ускорители Intel Xeon Phi поддерживают по 4 потока на ядро. Процессоры IBM (POWER8,9) поддерживают аж по 8 потоков на ядро. Многопоточность, может, звучит круто, но ведь при этом ресурсы ядра, все исполнительные устройства остаются те же самые. Операционная система может видеть одно многопоточное ядро как несколько виртуальных процессоров, но, по сути, это таки одно ядро. Спрашивается, зачем этот самообман? К чему эти иллюзии?

Ответ прост. Все дело в независимости. Не надо анализировать команды, напрягать планировщик, ходить к гадалке, чтобы понять, что две команды из разных потоков независимы. Поскольку **команды из разных потоков – независимы по определению**. Достаточно посмотреть, к какому потоку относится команда. Поэтому становится проще задействовать ресурсы суперскалярного ядра. Повышается загрузка исполнительных устройств, увеличивается IPC при тех же самых ресурсах. На практике получается 15%, а то и все 30% прироста производительности на двухпоточном ядре. На фаях (*жарг*. от Intel Xeon Phi), например, выигрыш от SMT получается даже в разы. Ну оно и понятно, чем проще ядро, чем слабее возможности OoOE, тем больше выигрыш от SMT.

## 2.3 Векторный параллелизм – SIMD

SIMD расширения позволяют процессорным ядрам выполнять векторные инструкции сразу над множественными наборами аргументов. За счет этого может кратно увеличиваться количество выполняемых операций, и особенно нас интересующих арифметических операций с плавающей точкой – FLOP. Удвоение флопов с FP64 аргументами стало доступно с расширения SSE2 (Streaming SIMD Extension), в котором в 128-битные регистры (XMM регистры) можно было уложить по два дабла и скормить векторному FPU. Расширение AVX (Advanced Vector Extension) уже имело регистры по 256 бит (YMM регистры), в которые можно поместить уже 4 набора дабловых аргументов. Сейчас широко используется расширение AVX-512 с регистрами 512 бит (ZMM регистры), в которые помещается по 8 даблов. Серверные процессоры Intel (Skylake и прочие лэйки) имеют до двух 512-битных FMA юнитов, что дает 32 FLOP на такт (8 ширина × 2 флопа FMA × 2 юнита). Процессоры AMD так за флопами не гонятся, у них (пока) используется расширение AVX2 (256 бит). Архитектура Эльбрус-16С тоже обзавелась векторными регистрами шириной 128 бит, что увеличило максимальное число FLOP на такт до 24 (6 FMA юнитов, вмещающих по два дабла). Совсем зверские векторные расширения имеют процессоры архитектуры NEC SX. FMA юниты ядра векторного процессора NEC SX-Aurora TSUBASA, например, могут работать с наборами из 32 дабловых аргументов. На ядре три таких юнита, что дает пик по 192 флопа на такт (3 юнита шириной по 32 аргумента по 2 флопа FMA).

Векторизация осуществляется обычно встроенными средствами автовекторизации компилятора. Также для этих целей используются директивные средства компилятора, когда программист добавляет в код специальные прагмы. Также могут использоваться интринсики, когда программист в явном виде используются векторные типы данных и встроенные функции. Высокоуровневые директивные средства вроде OpenMP стандарта 4.0 и выше имеют возможности векторизации. Применение открытого вычислительного стандарта OpenCL (в частности, на процессорах Intel) также способствует векторизации.

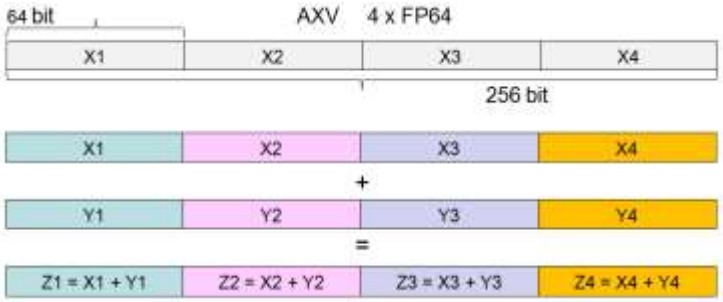
Посмотрим, как примерно это работает. Пусть мы хотим сложить два вектора, для чего у нас есть цикл.

```
for(int i=0; i<N; ++i) Z[i] = X[i] + Y[i];
```

Пообещаем компилятору, что обрабатываемые циклом куски памяти по указателям X,Y,Z не пересекаются, иначе компилятор не решится векторизовать цикл. Для этого добавим прагму (ну или добавим к декларации указателей спецификатор __restrict):

```
#pragma ivdep
for(int i=0; i<N; ++i) Z[i] = X[i] + Y[i];
```

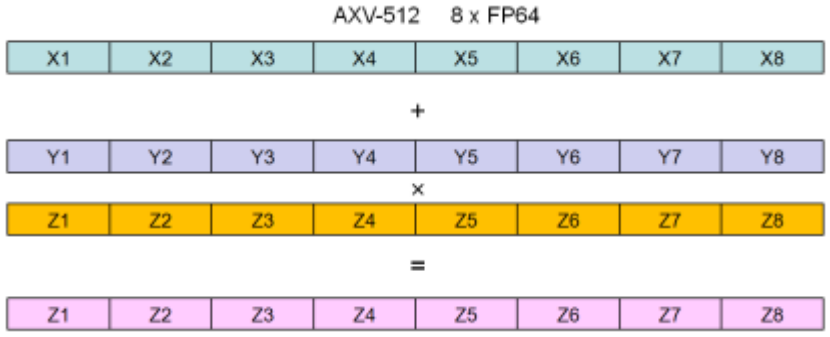
Включим компилятору векторизацию, обязательно проверим по логу компилятора, что цикл успешно векторнулся. Получим векторизованный код, который задействует векторные регистры. Пусть у нас процессор с AVX, тогда мы сможем делать 4 флопа за такт. Изобразить это можно как-то так:




А если у нас поэлементное сложение и умножение векторов, то совсем замечательно, мы задействуем FMA.

```
#pragma ivdep
for(int i=0; i<N; ++i) Z[i] = X[i] + Y[i]*Z[i];
```

Пусть у нас AVX-512. Тогда мы сможем делать 16 флопов за такт, а картинка будет такой:




Звучит прекрасно. Векторное расширение в разы увеличило нам FLOP на такт. Векторный FMA юнит может радостно кушать по много наборов даблов за такт.


Но есть нюанс. Съесть-то он съест, да кто ж ему даст? Ну вот мы векторнули такой цикл, даже посмотрели лог компилятора – там все прекрасно. Даже посмотрели машинный код, нашли там векторные инструкции, значит, всё взаправду. Запускаем на исполнение, замеряем время, вместо 8 раз видим прирост производительности 0.8%. Что за дела? И тут мы вспоминаем, что ранее что-то говорилось про соотношение пиковой производительности и пропускной способности памяти. Мало того, что для векторизации нужно, чтобы данные размещались компактно и шли подряд блоками, (желательно) выровненными в памяти по ширине векторных регистров (для чего надо сильно менять структуру данных программы), так еще, самое главное, эти данные должны как-то успевать поступать из памяти. Вот возьмем какой-нибудь Intel Skylake. В каждом ядре, допустим, по 2 FMA юнита, которые могут за такт работать с 16 наборами даблов. Пусть мы делаем наше сложение с умножением: Z[i] = X[i] +

Y[i]*Z[i]. Пусть в CPU 24 ядра. Итого 384 набора по три дабла, то есть на зачитку идет 1152 даблов, которые могут быть обработаны за один такт. За один такт CPU будет хотеть получить 1152 × 8 байт = 9216 байт данных. Пусть процессор работает на частоте 2ГГц. За секунду он будет желать 18 ТБ данных! А что на это ответит подсистема памяти? Допустим, там 6 каналов DDR4-2666. Это в идеале даст 128 ГБ/с. Это во сколько раз меньше? Закрыв лицо двумя руками, видим сквозь пальцы на калькуляторе число 144. "Абсурд!" – воскликнете вы, и будете правы. Ну и, спрашивается, что нам толку с этих 32 флопов за такт с ядра? Ну это как на каком-нибудь Октоберфесте разливать пиво из пипетки.


Делаем вывод, что векторизация – вещь может и полезная, но весьма ограниченно применимая в реальных приложениях, особенно связанных с обработкой большого объема данных, особенно memory bound алгоритмами. Получить на практике выигрыш в 15% от векторизации на AVX-512 многие сочтут за большое счастье. А вот когда вычисления сильно компактные, большая арифметическая интенсивность, за векторизацию стоит побороться. Например, галёркинцы (*жарг.* – расчетчики, использующие численные схемы на основе метода Галеркина, у которых высокая вычислительная интенсивность и хорошая компактность вычислений) хитро и довольно улыбаются.

## 2.4 Устройство памяти

Начнем с двух ключевых характеристик памяти, от которых напрямую зависит производительность.

- **Bandwidth** (обозначим BW) – пропускная способность памяти.

Эта величина измеряется в байтах в секунду и характеризует максимальную скорость передачи данных.

- **Latency** – латентность доступа, то есть задержки.

Для оперативной памяти эта величина задержки при доступе к данным (сейчас) обычно составляет порядка нескольких десятков наносекунд. Это задержка от момента, когда был инициирован запрос к памяти, до момента, когда начали поступать данные.

Время на чтение N байт данных будет: T = L + N/BW, где L – латентность, N/BW – время на передачу данных после того, как она уже началась. Если N маленькое, доминировать будет латентность, если N большое, то пропускная способность.

### 2.4.1 Иерархия памяти, кэш

Все знают, что есть разные виды памяти, упорядоченные по удаленности от процессора. Чем "дальше" память от процессора, тем (как правило) выше латентность доступа, ниже

пропускная способность, но больше объём. Доступ в оперативную память дорогой и долгий. Выше мы уже видели драму на примере сложения векторов. Поэтому для ускорения доступа к данным используется несколько уровней более быстрой кэш памяти (cache). В настоящее время можно (очень примерно) ориентироваться на величины такого порядка. Наиболее близкая к процессору память, это его регистры, aka регистровый файл (register file). Его размер порядка нескольких килобайт. Ну на AVX-512 векторные регистры здоровые, там и с десяток килобайт выйдет. К регистрам обычно у процессора прямой доступ за 1 такт. Далее идут несколько уровней кэш памяти. Самый ближний кэш L1 имеет размер обычно несколько десятков килобайт, задержку доступа несколько (4–5) тактов, пропускную способность порядка 1 ТБ/с. Кэш L2 имеет задержку порядка десятка тактов (10–15), пропускную способность где-то того же порядка 1 ТБ/с и заметно больший размер в районе несколько сот килобайт – мегабайт на ядро. Кэш L3 обычно уже является общим для множественных ядер процессора. Там задержки доступа уже несколько десятков тактов. Причем эта кэш память некоторым образом распределена между ядрами, и латентность заметно варьируется в зависимости от удаленности ядра от той части кэш памяти, к которой производится доступ. Если запрос в "свою" часть L3, то это будет где-то 40–50 тактов, если в чужую, то и все 70 тактов, а то и больше. Размер L3 обычно порядка десятков мегабайт (по несколько мегабайт на ядро). Пропускная способность порядка нескольких сотен ГБ/с. Уровней кэш может быть и больше. У процессоров IBM POWER8 4 уровня кэш. Последний уровень так и называется – LLC (Last Level Cache). Дальше идет оперативная память. Там латентность доступа уже порядка сотни тактов, суммарная пропускная способность каналов памяти у процессора порядка сотни ГБ/с. Объем памяти, понятное дело, сильно больше, десятки – сотни гигабайт.

Здесь мы не будем углубляться в подробности о там, как работает этот всякий там ассоциативный кэш, какие там алгоритмы. Там все сложно и интересно, на эту тему можно найти много статей в интернете. Нам важны следующие факты.

- Доступ в кэш значительно быстрее, чем в оперативную память.

На то он и нужен. Если данные находятся в кэш, то скорость доступа к ним намного выше, чем к данным в оперативной памяти, как по латентности, так и по пропускной способности (К.О.). Если мы обращаемся к данным, а их в кэш не оказалось, то это получается конфуз, называемый кэш промахом (cache miss). Промахи нам дорого выходят, примерно согласно латентности доступа, указанной выше. Промах в L1 выйдет в десяток тактов, промах в L2 – в несколько десятков, промах в L3 – в сотню-другую тактов.

- Зачитка данных из памяти в кэш происходит кэш линиями.

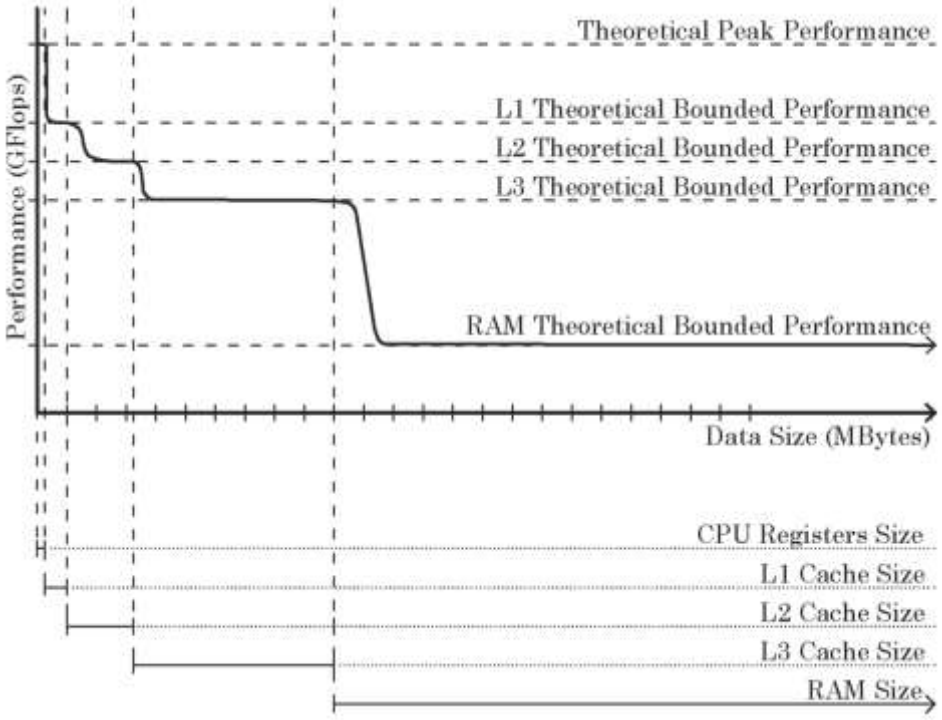
Даже если мы запрашиваем из памяти 1 байт, в кэш поступит вся линия, обычно 64 байта. Эта линия будет лежать в кэше ждать своего часа, пока не заэкспайрится (*жарг.* от expired – истекший срок годности), то есть пока ее не вытеснит более новая кэш линия. Примерно это как-то так: если, например, мы читаем из памяти 2 мегабайта, а кэш 1 мегабайт, то первая считанная кэш линия будет валяться в кэше, пока мы не начнем читать второй мегабайт. Тогда уже будет некуда пихать новую кэш линию, и самая старая линия, к которой дольше всего не было доступа, будет ей затерта. Значит, за что нам нужно бороться? За высокую повторную используемость кэш линий! Если мы считали всего один байт и выкинули кэш линию – это очень грустно и варварски расточительно. Дорогая транзакция с памятью прошла с использованием всего 1.5% данных. Если мы считали откуда попало один дабл (8 байт), это тоже получилось грустно. Идеальный вариант, когда мы считали сразу 8 даблов подряд, использовав всю кэш линию. Тогда транзакция была не зазря. А если мы использовали эти

даблы не по разу, то это совсем прекрасно. Итого: когда мы читаем данные, важно, не какой объем данных мы обработали, а сколько затратили на это кэш линий. Вывод: данные надо размещать максимально компактно, чтобы считывать минимальным числом кэш линий.

- Кэш – вещь аппаратная, для программиста прозрачная.

Работой кэш памяти управляет процессор, программисту напрямую она не видна. Нет какого-то отдельного адресного пространства, связанного с этой памятью. То есть, нельзя, например, сказать в программе – выдели мне эту переменную в кэш памяти. Но можно попросить заранее подгрузить (prefetch) какую-то кэш линию с помощью интринсиков.

Вспоминая, что такое достижимая производительность у алгоритма с учетом пропускной способности памяти (TBP), получим следующую картинку про зависимость производительности от объема обрабатываемых данных:




### 2.4.1 Устройство оперативной памяти

Сейчас обычно для оперативной памяти используется DDR4 SDRAM. Много букв.

RAM – Random Access Memory, память с произвольным доступом. То есть обратиться и получить доступ можно в любое место памяти за примерно одинаковое время. А не как в наше далекое время, когда, чтобы послушать нужный трек, надо было перематывать кассету.

DRAM – Dynamic RAM. Динамическая – потому что содержимое надо периодически освежать, иначе оно протухнет. Раз есть динамическая память, то логично предположить, что есть и статическая. SRAM (Static RAM) – это тоже энергозависимая (volatile) память, в которой всё накопленное добро теряется при отключении электричества. Но её не надо постоянно и упорно регенерировать, чтобы она не забыла хранящуюся в ней информацию. Тогда почему оперативная память не SRAM? Потому что в ячейке памяти DRAM используется в несколько раз меньше элементов, она состоит, в общем-то, из одного конденсатора и транзистора. Такая память кратно дешевле и занимает в несколько раз меньше площади на кристалле. То есть у неё намного выше плотность. Но в ячейке DRAM из конденсатора постепенно утекает заряд, поэтому её надо регулярно подзаряжать. Из-за такой подзарядки память работает медленнее,

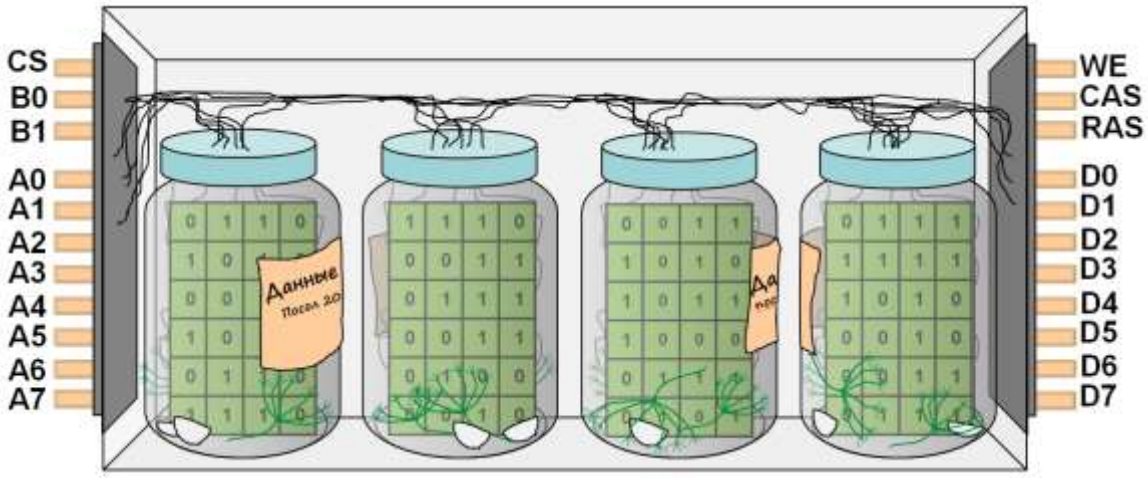
поскольку во время регулярной регенерации ячейка недоступна, приходится ожидать. На это мероприятие может уходить процентов 10 пропускной способности. Теперь понятно, почему кэш память, маленькая, дорогая и быстрая, – это SRAM, а оперативная память, большая, плотная, дешевая, – это DRAM.

SDRAM – Synchronous DRAM. Синхронная – значит, память работает не как попало, а синхронно под управлением тактового сигнала контроллера, и доступ к ней осуществляется потактово.

DDR – Double Data Rate, удвоенная скорость передачи данных. Ну у памяти бывает сильно разная скорость, а удвоенная – это относительно чего? DDR означает, что память работает на удвоенной скорости относительно частоты тактового сигнала, то есть частоты шины памяти. Синхронизация осуществляется по переднему и заднему фронту тактирующего импульса, дважды за период. То есть эффективная частота вдвое выше реальной. В общем, DDR значит, GT/s = 2 × GHz (GT/s – gigatransfers per second). А пропускная способность тогда сколько? Умножаем GT/s на ширину шины, GB/s = 8B × GT/s. DDR4-2666 – это 21.3 ГБ/с, например.

С точки зрения программиста эти особенности работы памяти, single или double, static или dynamic, не особо-то и важны. Просто так из любопытства. Но есть и более существенные детали. Разберемся в общих чертах с этими особенностями и сделаем важные для программиста выводы.

Модуль памяти состоит из набора чипов. Чипы состоят из банков. Банки представляют собой таблички, можно сказать, матрицы, размером по сколько-то тысяч строк и столбцов. В каждой ячейке такой таблички находится ячейка памяти, в которой хранится бит данных.




Чтобы обратиться к этому биту, надо указать его адрес – номер чипа, номер банка, номер строки и номер столбца. Рассмотрим один банк памяти. Адрес строки и столбца передается по общим адресным линиям (на картинке нарисовано A0 – A7, количество ножек взято с потолка). Чтобы память поняла, что происходит, там есть управляющие линии. Сначала на линию RAS (Row Address Strobe) поступает сигнал, который говорит, что на адресные линии поступит значение номера строки. Потом на линию CAS (Column Address Strobe) поступает сигнал, и передается адрес столбца. На считывание адреса строки и столбца тратится какое-то время, составляющее порядка 10–15 нс. Более того, при завершении доступа к строке, её еще надо зачем-то предзаряжать (precharge), чтобы можно было дальше активировать другую строку. На это уходит примерно такое же время. Эти задержки называются тайминги. Из них

складывается латентность. Напрашивается первый вывод: **читать данные из разных строк дольше, чем из одной строки**, т.к. надо передавать адрес строки и делать зарядку.

Но как же получается выдавать данные на каждом такте, если в лучшем случае все равно тратится время на передачу адреса столбца? Для этого используется пакетный режим. Один раз передается адрес столбца, и считывается подряд сразу пакет (burst) из нескольких столбцов. Кроме того, используется interleaving, когда идущие подряд данные распихиваются по разным банкам (то есть адрес X находится в банке X%k, где k – число банков, а % символизирует остаток от деления). Таким образом, все банки в работе, выше пропускная способность последовательной выборки. Отсюда еще вывод: читать данные из памяти эффективнее **подряд**.

Ну оно в общем-то так и получается, чтение из памяти происходит сразу машинным словом, обычно 64 бита, то есть на каждом такте работы памяти получаем 8 байт данных. А точнее, как мы знаем из предыдущего раздела, выборка осуществляется целой кэш линией (обычно в 8 раз больше). В общем, как с точки зрения кэш памяти, так и с точки зрения работы самой памяти, надо бороться за компактность данных (data locality) и unit stride access.

Уточним, что такое unit stride. Пусть мы копируем двухмерный массив. Unit stride:

```
for(int i=0; i<N; ++i)
    for(int j=0; j<M; ++j)
        A[i][j] = B[i][j];
```

Non unit stride, aka доступ "против шерсти" – это так:

```
for(int j=0; j<M; ++j)
    for(int i=0; i<N; ++i)
        A[i][j] = B[i][j];
```

Почему? Вспомним, как считается адрес в Си-шном двухмерном массиве: &A[i][j] – это &A[0][0] + i*M + j. Младший индекс – j. Внутренний цикл по младшему индексу = юнит страйд.

Наконец, DDR4 – четвертое поколение DDR. Ну, сначала была SDR. Потом DDR, работающая, как мы выяснили, на удвоенной скорости относительно частоты шины памяти (по обоим фронтам синхросигнала). Потом DDR2, работающая на учетверенной скорости, то есть еще частота шины в 2 раза выше, чем базовая частота ядра памяти (так производительнее и энергоэффективнее). Потом DDR3, DDR4 – на увосьмерённой скорости.

У какой-нибудь топовой DDR4 частота ядра памяти 400 МГц, частота шины 1.6 ГГц, эффективная скорость получается вдвое выше – 3.2 GT/s (gigatransfers per second – выдач данных в секунду). Еще разок посчитаем пропускную способность – умножаем 3.2 на 8 байт (ширина шины), получаем 25.6 ГБ/с на канал памяти. Память так и обозначается: DDR4-3200 или PC4-25600. В первом случае число 3200 – эффективная частота, равная удвоенной частоте шины, во втором – пропускная способность в МБ/с. Цифра 4 в обоих случаях как бы символизирует четвертое поколение. Каналов памяти у серверного процессора (сейчас) обычно 4–8.

Считать бэндвис у девайса можно следующим образом. Смотрим конфигурацию машинки, находим число каналов и пропускную способность на канал, умножаем одно число на другое. Для этого смотрим спеки (*жарг.* от спецификация) на машинку, или вызываем sudo dmidecode --type memory, или спрашиваем об этом специально обученных судоеров (*жарг.* от sudoer, пользователи с правами на запуск sudo команды). Пусть мы обнаружили, что на машинке 6 каналовDDR4-2666. Значит бэндвис будет 2666 × 8 × 6 = 128 ГБ/с.

Практическую пропускную способность измеряют бенчмарками типа STREAM (см. McCalpin, John D.: "STREAM: Sustainable Memory Bandwidth in High Performance Computers", http://www.cs.virginia.edu/stream/). В STREAM как раз используются операции с большими векторами – копирование, поэлементное сложение, умножение на константу, "триада" – сложение с умножением, a[i] = b[i] + q*c[i].

Более подробно про то, как устроена память, можно поискать в интернете. Там много статей про работу DRAM, про тайминги.

Ну вот, например: https://compress.ru/article.aspx?id=16737

Или даже, что стесняться, тут:

https://en.wikipedia.org/wiki/Memory_timings

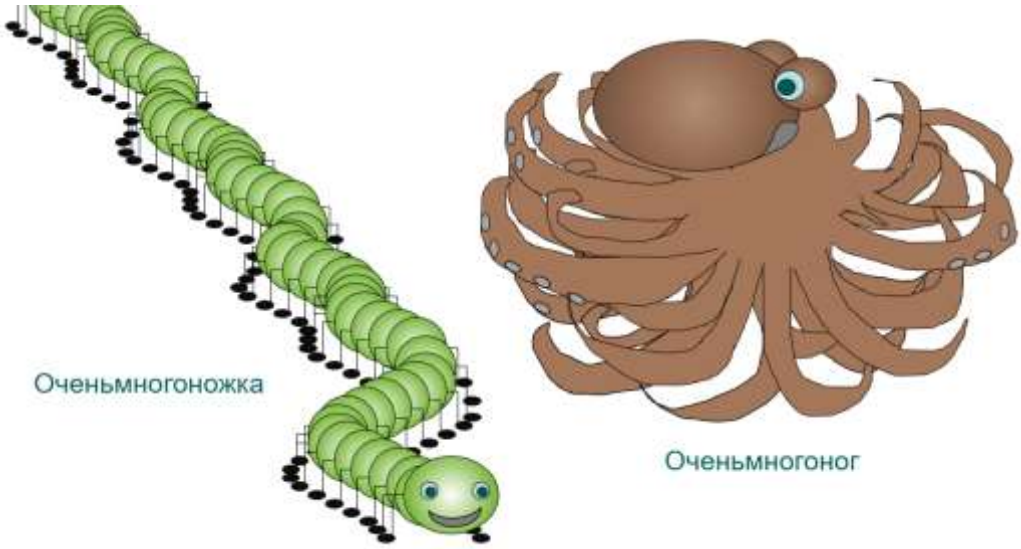
https://en.wikipedia.org/wiki/DDR4_SDRAM

Основной вывод мы уже сделали: по возможности держать данные в памяти компактно и обращаться к ним подряд. Далее сделаем еще ряд печальных выводов.

### 2.4.2 Повышение производительности памяти

Как уже много раз было сказано выше, пиковая производительность растет быстрее пропускной способности памяти. Разрыв между TPP и BW стал настолько велик, что в реальных приложениях приходится зачастую рассчитывать на жалкие проценты от пика. Почему так происходит? Потому что задрать пиковые флопсы – легко. Поставил на ядро два FMA юнита, и TPP стала вдвое больше. Расширил векторные регистры вдвое – еще вдвое больше. Еще один FMA юнит приляпал и сбоку бантик привязал – еще в полтора раза подрос перф. С памятью (пока) так не выходит.

Первая проблема – некуда пихать ножки. Один канал DDR4 – это 288 контактов. Если у процессора 8 каналов DDR4, то сколько ему надо приделать ножек? Взять, к примеру, сокет серверного процессора FCLGA3647. Что это за странное число 3647? Это количество контактов. Эта проблема ограничивает пропускную способность.




Небольшое повышение частоты шины памяти не дает существенного прироста. Проценты, но не разы. Топовый DDR4-3200 – это 25.6 ГБ/с. У прошлого поколения DDR3 была память PC3-17000. Разница всего в полтора раза. Процессоры за это время задрали пиковую производительность как минимум раза в 4. Не вариант. Откуда еще ждать спасения?

Пути, по сути, два:

1) размещать память на борту девайса, тогда на подложке можно протащить кучу контактов и обеспечить во много раз большую ширину шины памяти;

2) использовать высокоскоростные каналы связи между памятью и процессором, работающие на гораздо более высокой частоте, чем шина памяти.

С первым пунктом все понятно. Сейчас, например, используется память многослойной компоновки (3D stack), в которой чипы памяти наслаиваются друг на друга в компактную трехмерную структуру. В настоящее время широко используется память HBM2 (High Bandwidth Memory, 2 поколение). В один стек помещается до 8 чипов. Ширина шины у такого стека 1024 бита (у DDR4 – 64 бита). Один такой стек дает пропускную способность где-то 250 ГБ/с (больше, чем все 8 каналов топовой DDR4 у процессора). Современные девайсы (NVIDIA V100 GPU, векторный процессор NEC SX-Aurora TSUBASA, ARM manycore Fujitsu A64FX, например) имеют по 32 ГБ HBM2, суммарная пропускная способность, соответственно, около 1 ТБ/с. Вроде бы 1 ТБ/с – это уже хорошо, почти на порядок больше. Но есть два нюанса. Первый – на борт много памяти не влезает. Ни геометрически, ни по энергопотреблению (соответственно, тепловыделению), ни по стоимости. 32 ГБ памяти для девайса, который в несколько раз мощнее многоядерного серверного процессора, это маловато. Если идти по этому пути, то нас будет ждать гибридизация памяти, как на девайсах Intel Xeon Phi. Программисту придется заморачиваться, какие данные размещать в какой именно памяти – большой и узкой, или маленькой и широкой. Другой нюанс будет дальше.

Со вторым способом тоже вроде бы все ясно. Для передачи данных используется сильно более высокочастотная шина (более высокая эффективная частота). В набортной памяти, например, такой подход тоже используется. В игровых картах сейчас широко используется графическая память GDDR6 (Graphics Double Data Rate, 6 поколение), которая берет скорее не широтой шины, а частотой. Эффективная частота там сейчас по 14 – 16 ГГц, что раз так в 5 побольше, чем у CPU-шной DDR4. Пропускная способность получается почти как у HBM2.

Но для процессора это было бы даже намного интереснее – за счет многократного увеличения частоты пропускная способность растет, а количество контактов – нет. По такому пути идет (или шла) технология HMC (Hybrid Memory Cube). Там частота шины где-то в районе 10–15 ГГц (у DDR4 – до 1.6 ГГц). Пропускная способность (обещалась) того же порядка, что у HBM2, по несколько сот ГБ/с на "кубик". Другой пример – сейчас IBM разрабатывает интерфейс памяти Open Memory Interface (OMI), который предполагает линки аж по 25 GT/s. У топовой DDR4 это где-то 3.2 GT/s. Предполагается достигать сопоставимой с HBM2 суммарной пропускной способности (см., например, https://www.eetimes.com/ibm-debuts-ddr-alternative). Да и что там какие-то новомодные шины, стандарты, в общем-то, даже у самой обычной системной шины PCI-E 3.0 скорость 8 GT/s – в несколько раз выше, чем у DDR4. Потенциал для повышения пропускной способности за счет частоты – большой. При таком способе память остается внешняя, нет ограничений на её объем, а пропускная способность может получиться как у набортной.

Значит, счастье таки возможно? Но вот он второй нюанс – латентность. Что у набортной памяти, что у внешней латентность только растет. А у такой внешней памяти на быстрых линках задержки только еще вырастут из-за проброса данных через внешний контроллер.

Этот второй нюанс отметим особенно. Печаль в том, что пропускная способность может быть как-то и увеличивается, но латентность не уменьшается. Она только растет! Например, у топовой DDR4 латентность выше, чем было у DDR3 (CAS latency 12.5 нс против 10 нс). У

HBM2 латентность сопоставимая с DDR4 (CAS latency в районе 14 нс). Вот, например, изыскания про латентность:

https://user.eng.umd.edu/~blj/papers/memsys2018-dramsim.pdf

Получается, что мы так и будем огребать за кэш промах сотни тактов. Организация доступа к памяти, устройство структуры данных в программе, это все так и будет критическим местом с точки зрения производительности. Надежд пока не видно. Поэтому так важно понимать, как работает память, дружить с data locality и unit stride.

### 2.4.3 Неоднородный доступ к памяти

Выше была мешанина из аббревиатур, шин, частот и тактов, из которой мы вытащили несколько выводов. Повторим их:

- транзакция с памятью – дорогая штука, обращаясь в память, надо стараться использовать максимум данных из кэш-линии;
- обращаться к данным в памяти лучше подряд, как для снижения числа кэш промахов, так и с точки зрения работы самой памяти – подряд намного быстрее, чем вразнобой;
- по мере развития вычислительной техники производительность растет сильно быстрее, чем пропускная способность памяти;
- латентность доступа не снижается, а только увеличивается.

Осталось разобрать еще одну пакость – это **неоднородность доступа к памяти**. Почему-то скорость доступа к разным адресам в одном адресном пространстве может заметно отличаться. Происходит это явление обычно на многопроцессорной системе с общей памятью, когда разные части памяти и разные процессоры находятся ближе или дальше друг от друга. Эта неприятность называется NUMA (Non-Uniform Memory Access). NUMA-фактор возникает, например, на многопроцессорном сервере. У каждого процессора свой контроллер памяти. Память со всех процессоров собрана в общее адресное пространство. Но когда один процессор лезет в память другого, делает он это не напрямую, а через шину между процессорами, из-за чего латентность доступа становится не 100 нс, а, скажем, все 300 нс, а пропускная способность может снизиться раза в 2. То же самое может происходить и внутри одного многоядерного процессора. Не все ядра процессора равноудалены от контроллера памяти. А контроллеров на процессоре тоже может быть более одного. Из-за этого NUMA-фактор возникает и на одном многоядерном процессоре. Такая же штука с LLC кэш. Кэш тоже поделен между ядрами, и лазить в удаленную от ядра часть кэш наказывается лишними тактами задержки. Поскольку проблема NUMA сильно связана с MIMD параллелизмом с общей памятью, разбираться с этим более подробно мы будем в следующей главе.

## 2.5 Практические примеры

### 2.5.1 Loop unrolling

Рассмотрим простые примерчики, демонстрирующие описанные в этой главе явления. Сделаем, например, такую глупость. Возьмем какое-то большое число, пусть будет N = 330000123. Число неровное, чтобы не делилось нацело на всякие двойки. Это нужно для проверки корректности.

Посчитаем подряд до этого числа операциями с плавающей точкой. Замеряем время. Напечатаем результат. Если не напечатаем, то хитрый компилятор может решить, что результат не используется, и вообще не будет ничего складывать и выкинет код.

```
int N=330000123; // 123 - to check correctness, so that N%2, N%4... != 0
double a = 1.0, result = 0.0, t;
t = omp_get_wtime(); // starting of wall-clock time measurement
result = 0.0;
for(int i=0; i<N; ++i) result += a;
t = omp_get_wtime() - t;  // finishing timing
printf("ADD X1 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));
```

Запускаем, получаем такой результат:

```
ADD X1 res: 3.300001230000E+008 time: 0.311 s GFLOPS: 1.060
```

Число получили правильное. Производительность – 1 GFLOPS. Смотрим характеристики процессора. Intel Core i7-3630QM, базовая частота 2.4 ГГц, максимальная (Intel Turbo Boost) 3.4 ГГц. В память мы не лазаем, сидим на регистрах, сложение делаем с двумя переменными. Почему тогда 1 GFLOPS, а не 2.4 хотя бы? Продолжим эксперименты.

```
result=0;
t = omp_get_wtime(); // starting of wall clock time measurement
{
    double result_tmp[2] = {0,0};
    for(int i=0; i<N/2; ++i){
        result_tmp[0] += a;
        result_tmp[1] += a;
    }
    for(int i=0; i<N%2; ++i) result += a; // finishing remaining iterations
    result += result_tmp[0] + result_tmp[1]; // reduction of result
}
t = omp_get_wtime() - t; // finishing timing
printf("ADD X2 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));
```

Написано тоже самое, но немного по-другому. На итерации цикла уже два сложения. Запускаем, получаем такой результат:

```
ADD X1 res: 3.300001230000E+008 time: 0.310 s GFLOPS: 1.063
ADD X2 res: 3.300001230000E+008 time: 0.155 s GFLOPS: 2.131
```

Видим во второй строке выдачи уже 2.1 GFLOPS. Интересно. А если больше сложений в итерацию насувать? Пробуем.

```
result=0;
t = omp_get_wtime();
{
    double result_tmp[3] = {0,0,0};
    for(int i=0; i<N/3; ++i){
        result_tmp[0] += a;
        result_tmp[1] += a;
        result_tmp[2] += a;
    }
    for(int i=0; i<N%3; ++i) result += a;
    result += result_tmp[0] + result_tmp[1] + result_tmp[2];
}
t = omp_get_wtime() - t;
printf("ADD X3 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));
```

Получаем:

```
ADD X1 res: 3.300001230000E+008 time: 0.310 s GFLOPS: 1.063
ADD X2 res: 3.300001230000E+008 time: 0.155 s GFLOPS: 2.131
ADD X3 res: 3.300001230000E+008 time: 0.103 s GFLOPS: 3.206
```

Уже 3.2 GFLOPS. И что, так можно до бесконечности? Пробуем 4.

```
result=0;
t = omp_get_wtime();
{
    double result_tmp[4] = {0,0,0,0};
    for(int i=0; i<N/4; ++i){
        result_tmp[0] += a;
        result_tmp[1] += a;
        result_tmp[2] += a;
        result_tmp[3] += a;
    }
    for(int i=0; i<N%4; ++i) result += a;
    result += result_tmp[0] + result_tmp[1] + result_tmp[2] + result_tmp[3];
}
t = omp_get_wtime() - t;
printf("ADD X4 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));
```

Получаем:

```
ADD X1 res: 3.300001230000E+008 time: 0.308 s GFLOPS: 1.073
ADD X2 res: 3.300001230000E+008 time: 0.153 s GFLOPS: 2.164
ADD X3 res: 3.300001230000E+008 time: 0.102 s GFLOPS: 3.243
ADD X4 res: 3.300001230000E+008 time: 0.103 s GFLOPS: 3.217
```

Всё, кончилась халява, дальше не растет. X4 получаем тоже самое, что X3 (плюс-минус погрешность измерения). Результат стал примерно соответствовать максимальной частоте процессора. Отметим этот факт и попробуем проделать тот же эксперимент с умножением.

```
double a = 1.0000001;
double result, t;
printf("\n");
t = omp_get_wtime();
result = 1.0;
for(int i=0; i<N; ++i) result *= a;
t = omp_get_wtime() - t;
// note roundoff error may accumulate in last digits
printf("MUL X1 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));
```

Проделаем такую же манипуляцию с циклом. Чтобы получилось покороче, введем еще внутренний циклик по константному диапазону (компилятор его сам размотает).

```
{
      double result_tmp[2]={1,1};
      for(int i=0; i<N/2; ++i){
          // using constant loop, assuming compiler will unroll it
          for(int j=0; j<2; ++j) result_tmp[j] *= a;
      }
      for(int i=0; i<N%2; ++i) result *= a;
      for(int j=0; j<2;   ++j) result *= result_tmp[j];
}
```

Ну и по аналогии со сложением, делаем так до трех умножений. Запускаем, получаем:

```
MUL X1 res: 2.146458698877E+014 time: 0.520 s GFLOPS: 0.634
MUL X2 res: 2.146458698877E+014 time: 0.259 s GFLOPS: 1.275
MUL X3 res: 2.146458698874E+014 time: 0.173 s GFLOPS: 1.913
```

Что за несправедливость? У сложения на X3 уже было 3 гигафлопса, а тут не дотягивает до двух. Вроде бы процессор может делать за такт и умножение, и сложение, и даже их оба сразу. Проявим упорство, увеличим число умножений на внутреннем цикле. Дошли аж до X6.

```
result=1.0;
t = omp_get_wtime();
{
    double result_tmp[6]={1,1,1,1,1,1};
    for(int i=0; i<N/6; ++i){
        for(int j=0; j<6; ++j) result_tmp[j] *= a;
    }
    for(int i=0; i<N%6; ++i) result *= a;
    for(int j=0; j<6;   ++j) result *= result_tmp[j];
}
t = omp_get_wtime() - t;

printf("MUL X6 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));
```

Запускаем, получаем:

```
MUL X1 res: 2.146458698877E+014 time: 0.520 s GFLOPS: 0.634
MUL X2 res: 2.146458698877E+014 time: 0.260 s GFLOPS: 1.269
MUL X3 res: 2.146458698874E+014 time: 0.173 s GFLOPS: 1.903
MUL X4 res: 2.146458698875E+014 time: 0.130 s GFLOPS: 2.534
MUL X5 res: 2.146458698874E+014 time: 0.105 s GFLOPS: 3.143
MUL X6 res: 2.146458698876E+014 time: 0.105 s GFLOPS: 3.135
```

Справедливость восстановлена, на умножении сняли те же 3 с чем-то GFLOPS. Но произошло это не на трех операциях внутреннего цикла, а на 5. А почему это всё так?

Эта, казалось бы, бессмысленная ерунда, которую мы проделали с циклом, называется loop unrolling, aka развертка цикла, размотка цикла, раскрутка цикла. Этой манипуляцией мы рвем зависимости между итерациями. В исходном цикле `for(int i=0; i<N; ++i) result += a;` мы делаем N сложений, но каждое последующее сложение зависит от предыдущего. А мы должны бороться за независимость. Но зачем нам эта независимость? У нас, во-первых, ядро – не какой-то там RISC с 5-ступенчатым конвейером, а мощный out-of-order (Ivy Bridge) с большим буфером команд. Он и так может выдавать CPI = 1. Во-вторых, у ядра всего один FMA юнит, оно всё равно не может делать одновременно два сложения, даже если они независимы. Тогда почему мы наблюдаем, что производительность на сложении растет, пока мы не порвем зависимости на глубину 3, а на умножении – на 5? Дело в том, что FMA юнит сам по себе – конвейеризованный. Это устройство может делать и сложение, и умножение, и даже совмещенное сложение-умножение за один такт, но только в конвейерном режиме. На самом деле на выполнение одного сложения требуется 3 такта. На выполнение умножения – 5 тактов работы FMA юнита. На сложение-умножение еще больше – 8 тактов (ну это только у конкретно этой модели процессора). Поэтому, пока мы не растащили зависимости на глубину конвейера, мы не получали должную производительность.

Но почему это работает, когда мы лениво вписали внутренний цикл вместо того, чтобы плодить строчки? Потому что у внутреннего цикла нет зависимостей между итерациями, т.к.

там обращение к разным позициям массива. А то, что оно цикл – не страшно, компилятор сам легко разматывает циклы по константному диапазону (но иногда может постесняться это делать, если диапазон переменный).

Вообще в данном тесте использовался старенький компилятор с (конечно же) включенной оптимизацией (GCC 4.8, g++ -O3). Новые компиляторы уже не так стесняются, с определенными настройками оптимизации они могут сами и размотать, и векторнуть наш цикл по N. Так что не факт, что у вас этот эксперимент воспроизведется на базовых настройках оптимизации. Ну да ладно, сути это не меняет.

### 2.5.2 SIMD расширение

Мы знаем, что FMA юнит – векторный. У данного процессора (на моем стареньком ноутбуке) SIMD расширение AVX – 256 бит. Оно может вместить по 4 набора дабловых аргументов. Попробуем вместить хотя бы два, посмотрим, что получится. Используем интринсики – `#include <emmintrin.h>`, не забыв выставить компилятору в настройках целевую архитектуру AVX.

```
double a = 1.0, result=0, t;
{
    __m128d result_tmp = _mm_set_pd(0.0,0.0); // 128 bit vector time = 2 FP64 values
    __m128d aa = _mm_set_pd(a,a);
    t = omp_get_wtime();
    for(int i=0; i<N/2; ++i)
         result_tmp = _mm_add_pd(result_tmp, aa); // vector addition intrinsic
    for(int i=0; i<N%2; ++i) result += a;
    result += ((double*)(&result_tmp))[0]+((double*)(&result_tmp))[1]; // reduction
    t = omp_get_wtime() - t;
}
printf("VECADD X1 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));

result=0;
{
    __m128d result_tmp[2] = {_mm_set_pd(0.0,0.0),_mm_set_pd(0.0,0.0)};
    __m128d aa = _mm_set_pd(a,a);
    t = omp_get_wtime();
    for(int i=0; i<N/4; ++i){
        result_tmp[0] = _mm_add_pd(result_tmp[0], aa);
        result_tmp[1] = _mm_add_pd(result_tmp[1], aa);
    }
    for(int i=0; i<N%4; ++i) result += a;
    for(int j=0; j<2;   ++j)
        result += ((double*)(&result_tmp[j]))[0]+((double*)(&result_tmp[j]))[1];
    t = omp_get_wtime() - t;
}
printf("VECADD X2 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n", result, t, N/(t*1E9));
```

Ну и так далее в том же духе для другого количества операций внутри цикла. Получаем:

```
VECADD X1 res: 3.300001230000E+008 time: 0.173 s GFLOPS: 1.909
VECADD X2 res: 3.300001230000E+008 time: 0.104 s GFLOPS: 3.179
VECADD X4 res: 3.300001230000E+008 time: 0.052 s GFLOPS: 6.316
```

Ну вот, когда растащили зависимости на глубину более 3, получили 6 с гаком GFLOPS. Производительность, как и положено, стала вдвое выше за счет векторизации.

### 2.5.3 Пропускная способность памяти

Проделаем примерно такую же бессмысленную вещь, но уже с векторами. Сделаем поэлементное сложение двух векторов.

```
int N=66000123;
double *x=new double[N], *y=new double[N];
double result, t, t1, t4;
for(int i=0; i<N; ++i){ x[i]=i%2; y[i]=i%3; } // initialization with some crap
result = 0.0;
t = omp_get_wtime();
for(int i=0; i<N; ++i) x[i] = x[i] + y[i];
t = omp_get_wtime() - t;
for(int i=0; i<N; ++i) result += x[i]; // control sum
printf("X+Y STRIDE1 T1 X1: %15.12E time: %5.3f s GFLOPS: %5.3f\n",
       result, t, N/(t*1E9));
```

Ну и, наученные уже предыдущим экспериментом, сразу делаем (а вдруг поможет?) размотанные версии вида

```
const int N4 = (N/4)*4;
for(int i=0; i<N4; i+=4){
    x[i  ] = x[i]  +y[i];
    x[i+1] = x[i+1]+y[i+1];
    x[i+2] = x[i+2]+y[i+2];
    x[i+3] = x[i+3]+y[i+3];
}
for(int i=N4; i<N; ++i) x[i] = x[i] + y[i];
```

Запускаем, получаем:

```
X+Y STRIDE1 T1 X1 res: 9.900018400000E+007 time: 0.082 s GFLOPS: 0.800
X+Y STRIDE1 T1 X4 res: 9.900018400000E+007 time: 0.084 s GFLOPS: 0.784
```

Но чуда не происходит. Производительность не меняется. Можно предположить, что компилятор понял, что суммы независимые друг от друга, и сам все раскрутил как надо. Попробуем тогда добавить многопоточное распараллеливание (о котором мы еще не говорили, про это будет в следующих разделах), раз процессор четырехядерный. Эта OpenMP директива поделит N итераций цикла на 4 части между четырьмя потоками, и они будут выполняться одновременно.

```
#pragma omp parallel for num_threads(4)
for(int i=0; i<N; ++i) x[i] = x[i] + y[i];
```

Запускаем, получаем:

```
X+Y STRIDE1 T4 X1 res: 9.900018400000E+007 time: 0.075 s GFLOPS: 0.875
X+Y STRIDE1 T4 X4 res: 9.900018400000E+007 time: 0.075 s GFLOPS: 0.880
```

Да, на четырех ядрах стало чуть быстрее, на целых 10%. Производительность в 3.6 раз ниже, чем было на одном ядре (без векторизации). Почему?

Посмотрим на характеристики памяти: два канала DDR3 PC3-12800, итого суммарно 25.6 ГБ/с. Арифметическая интенсивность у нашего поэлементного сложения векторов 1/24 (N FLOP на 16N чтения и 8N записи). Достижимая производительность будет 25.6/24 = 1.07

GFLOPS. Получилось, что результат на самом деле в районе 80% от максимально достижимой производительности. Вот и ответ.

Ну раз уж взялись за OpenMP, проверим, что оно вообще работает. Заменим сложение на что-то вычислительно тяжелое, например: `for(int i=0; i<N; ++i) x[i] += sin(y[i]);`

Запускаем, получаем:

```
SIN T1 7.151703784123E+007 time: 0.614 s SINPS: 0.108
SIN T4 7.151703784123E+007 time: 0.180 s SINPS: 0.368
```

Ну вот, все работает, ускорение 3.4 раза из 4. Вот что значит memory bound, что такое низкая и высокая арифметическая интенсивность.

### 2.5.4 Паттерн доступа к памяти

Проделаем ту же операцию сложения векторов, но изменим последовательность доступа. Исходно у нас был unit stride access. Внесем смуту. Будем варьировать страйд (*жарг.* интервал доступа) от 1 до 32. Для этого приделываем внешний цикл, перебирающий величину страйда.

```
for(int k=1; k<=32; k*=2){
    for(int i=0; i<N; ++i){x[i]=i%2; y[i]=i%3;}
    result = 0.0;
    t = omp_get_wtime();
    for(int j=0; j<k; ++j){
        for(int i=j; i<N; i+=k) x[i] = x[i] + y[i];
    }
    t = omp_get_wtime() - t;
    for(int i=0; i<N; ++i) result += x[i]; // control sum
    printf("X+Y STRIDE % 2d T1 res: %15.12E time: %5.3f s GFLOPS: %5.3f\n",
           k, result, t, N/(t*1E9));
}
```

Запускаем, получаем:

```
X+Y STRIDE  1 T1 res: 9.900018400000E+007 time: 0.083 s GFLOPS: 0.798
X+Y STRIDE  2 T1 res: 9.900018400000E+007 time: 0.164 s GFLOPS: 0.402
X+Y STRIDE  4 T1 res: 9.900018400000E+007 time: 0.337 s GFLOPS: 0.196
X+Y STRIDE  8 T1 res: 9.900018400000E+007 time: 0.688 s GFLOPS: 0.096
X+Y STRIDE 16 T1 res: 9.900018400000E+007 time: 0.964 s GFLOPS: 0.068
X+Y STRIDE 32 T1 res: 9.900018400000E+007 time: 1.252 s GFLOPS: 0.053
```

Делаем ровно одно и то же. Складываем одни и те же вектора. Только немного в разном порядке. Разница – 15 раз! Теперь немного проясняется, что такое латентность доступа, и зачем надо читать подряд.

## 2.6 Внеклассное чтение

Для дальнейшей прокачки базы можно обратиться к следующей литературе. Если вдруг ссылки на момент чтения уже стали недоступны – ничего страшного. Интернет, хочется верить, все еще будет к тому времени в доступе, и поисковые системы тоже (а если нет, то уж и незачем читать этот материал).

- Лацис А.О. Параллельная обработка данных. Издательство: М.: Академия. 2010. ISBN: 978-5-7695-5951-8. В тему будет почитать главу 1, разделы 1.1 – 1.4, и главу 2. Там про базовые понятия, про то, как вообще устроен компьютер.

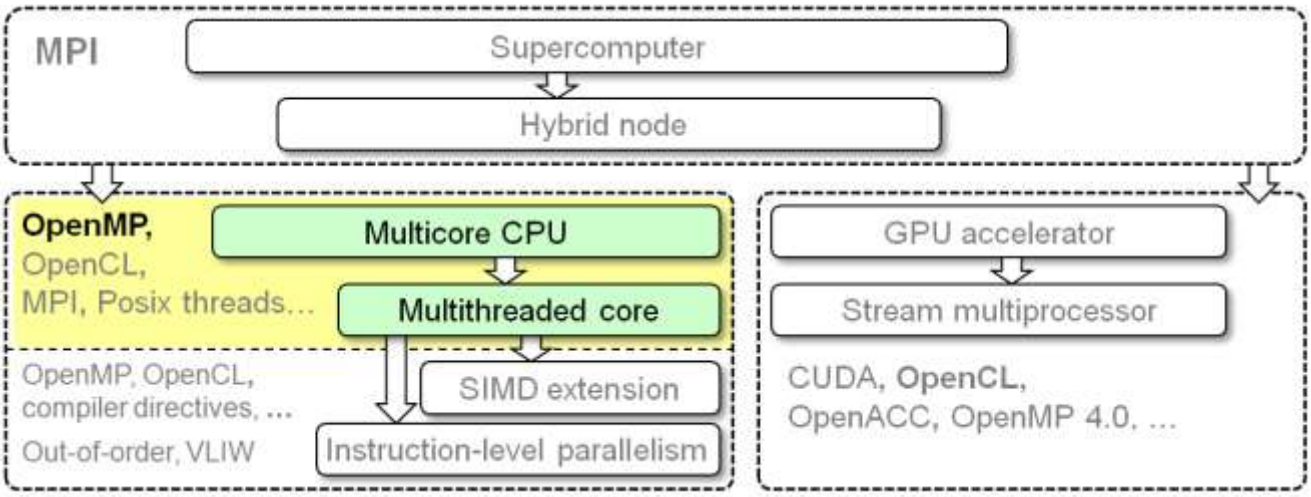
- Сергей Пахомов, Энциклопедия современной памяти. КомпьютерПресс 10’2006. https://compress.ru/article.aspx?id=16737
  Там про устройство памяти кратко, понятно, с картинками. С тех пор уже много новых стандартов памяти вышло, но суть особо не изменилась.
- http://www.ecs.umass.edu/ece/koren/architecture/windlx/main.html
  Тут можно поиграться с конвейером, напихать команд, посмотреть, как они будут исполняться на конвейере. Тут вот тоже про конвейер, про конфликты с FP командами. http://ece-research.unm.edu/jimp/611/slides/chap3_6.html

Есть разные optimization guides, в которых можно понабраться знаний об особенностях работы матчасти.

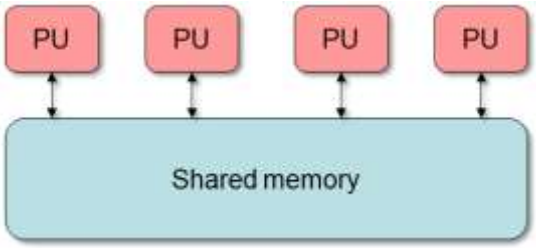
- Bailey, David & Lucas, Robert & Williams, Samuel. (2010). Performance Tuning of Scientific Applications. DOI: 10.1201/b10509. https://www.researchgate.net/publication/262602472 (там доступен для скачивания pdf). Там глава 2 про Roofline model, оценку производительности, разные факторы, влияющие на производительность.
- Agner Fog. Optimizing software in C++: An optimization guide for Windows, Linux, and Mac platforms. Technical University of Denmark. https://www.agner.org/optimize/optimizing_cpp.pdf
- У основных вендоров процессоров есть свои гайды, в которых можно многое почерпнуть про устройство процессоров, все эти регистры, кэш, SIMD расширения...
  Intel® 64 and IA-32 Architectures Optimization Reference Manual.
  https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf
  Аналогично, склад документации у AMD: https://developer.amd.com/resources/developer-guides-manuals/

# 3 MIMD с общей памятью. Многопоточное распараллеливание

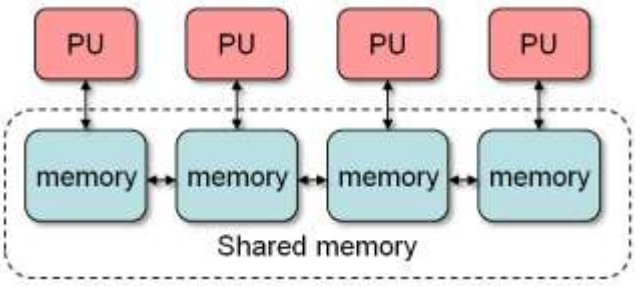
С SISD и SIMD параллелизмом внутри процессорного ядра познакомились, перейдем на уровень выше. SM-MIMD распараллеливание – Shared Memory, Multiple Instruction streams, Multiple Data streams.




Пусть имеется многопроцессорная система с общей памятью, на которой нам нужно что-то посчитать. Такую систему обычно называли SMP (Symmetric multiprocessing).




Почему хочется сказать – называли? Проблема в слове symmetric, подразумевающем, что все должно быть симметричное, одинаковое. Множество одинаковых процессоров подключено к единой общей оперативной памяти, доступ к которой тоже одинаковый для всех процессоров. А вот с этим уже давно проблемы. Доступ к памяти зачастую неоднородный. Память находится на разных контроллерах разных процессоров, доступ между ними происходит через некоторую шину (межпроцессорные линки, например). Значит, хоть адресное пространство памяти и общее, лезть с одного процессора в часть памяти, подключенную к другому процессору, выходит заметно дольше. А это уже называется NUMA (Non-Uniform Memory Access) системой.



Но сути это не меняет. Есть множество процессоров с общей памятью – SM-MIMD машинка. Обычно для таких систем используется многопоточное распараллеливание. Можно использовать и распараллеливание с распределенной памятью, с которым мы еще не разбирались. Иногда (зачастую) это может быть даже более эффективно. Потом станет понятнее почему. А пока, может, даже и непонятно, чем процесс отличается от потока.

Слово многопоточное подразумевает, что есть потоки и их – много. Все же знают, что такое процесс? Выполняется программа, у нее есть поток команд и адресное пространство виртуальной памяти (ну и всякие прочие ресурсы – регистры, стек, счетчики). Это – процесс. А если есть группа процессов, у которой это адресное пространство виртуальной памяти общее, то они – потоки, aka нити, threads. Точнее, на самом деле все наоборот, это процесс может разделяться на множественные потоки, а не процессы кучкуются и становятся потоками. Но это уже несущественные детали. Есть потоки команд, работающие с общим адресным пространством (виртуальной) памяти. Называть мы их будем потоки или нити, считаем эти слова полными синонимами и перемешиваем их в тексте как угодно.

Для многопоточного распараллеливания есть разные средства разработки. Наиболее распространен (сейчас) стандарт OpenMP (Open Multi-Processing), поддерживаемый всеми основными компиляторами C, C++, Fortran. Его мы и будем здесь использовать. Другим вариантом реализации многопоточки может быть стандарт POSIX Threads. Это штука идет в виде библиотеки, она позволяет программисту управлять потоками и их параллельной работой. Есть и всякие разные проприетарные средства и библиотеки, но их даже неохота перечислять, уж больно быстро они, бывает, переходят в разряд музейных реликвий.

## 3.1 Стандарт OpenMP

Предполагается, что у вас уже был курс параллельных вычислений, где рассказывалось про этот стандарт. Если эта информация уже вытеснилась из памяти за ненадобностью, то подгрузим обратно в кэш основные вещи. А если вы не знакомы со стандартом OpenMP, то надо будет еще что-нибудь немного почитать.

OpenMP API (Application Programming Interface) состоит из двух основных частей: директивной и библиотечной. Директивная, это которая использует директивы компилятора – `#pragma omp` или `!$OMP` в Fortran, но тут речь будет про C/C++. Библиотечная часть использует просто вызовы функций.

Спецификации стандартов OpenMP живут тут:

https://www.openmp.org/specifications/

Там два вида полезных документов: 1) спецификация стандарта – большой подробный документ с полным описанием всего; 2) шпаргалка по всему синтаксису на несколько страниц – summary card, aka reference guide. Так много читать не надо, достаточно почитать в спецификации только про используемые в данной главе функции и директивы, а со шпаргалкой можно ознакомиться целиком.

Есть разные версии стандарта, с каждой новой версией стандарт дополняется и усложняется. Нам для данной главы достаточно вообще древнего окаменелого ископаемого стандарта OpenMP 2.0. Хотя, версии 2.0 на сайте уже может не быть, тогда можно смотреть 3.0, ну или другой самый старый стандарт, который там выложен.

Работает OpenMP следующим образом. Сначала надо сказать компилятору, чтобы он включил поддержку OpenMP. Для этого надо выставить соответствующий флаг, который у разных компиляторов может быть разным (GNU g++ -fopenmp; Intel icpc -qopenmp, раньше

было -openmp; IBM xlc -qsmp; Microsoft C++ /openmp). Если OpenMP не включить, директивы будут игнорироваться.

Затем надо подключить заголовок от библиотечной части: `#include <omp.h>`

Понимание хорошо приходит на примерах. С них и начнем. Рассмотрим поэлементное умножение двух векторов:

```
    for(int i=0; i<n; ++i) x[i] = x[i]*y[i];
```

Параллельный вариант может быть таким:

```
omp_set_num_threads(8); // библиотечный вызов
// выставляем число нитей в последующих параллельных областях (например, 8 нитей)
#pragma omp parallel // директива - открытие параллельной области
// для следующего за ней блока
{ // начало параллельной области
    #pragma omp for //директива параллельный цикл, итерации распределяются между нитями
    for(int i=0; i<n; ++i) x[i] = x[i]*y[i];
} // закрытие параллельной области
```

То же самое другими словами:

```
#pragma omp parallel num_threads(8) // указываем напрямую число нитей для этой области
{
    #pragma omp for // директива цикла
    for(int i=0; i<n; ++i) x[i] = x[i]*y[i];
}
```

То же самое другими словами:

```
#pragma omp parallel for num_threads(8) // совмещенная директива параллельного цикла
for(int i=0; i<n; ++i) x[i] = x[i]*y[i];
```

То же самое другими словами:

```
omp_set_num_threads(8);
#pragma omp parallel // директива параллельной области OpenMP
{
    const int nt = omp_get_num_threads(); // библиотечный вызов - узнаем число нитей
    const int tn = omp_get_thread_num(); // еще вызов - узнаем номер данной нити
    const int ibeg = tn*(/*объем работы одной нити*/ n/nt) +
        (/*раздача остатка, если n не делится на nt*/ tn < n%nt ?  tn : n%nt );
    const int iend = ibeg + (n/nt) + (tn < n%nt);
    for(int i=ibeg; i<iend; ++i) x[i] = x[i]*y[i];
}
```

Как видно, одного и того же результата мы добились разными способами: использовали итерационную воркшару (*жарг.* от worksharing construct), распределяющую итерации цикла между нитями; поделили работу вручную, пересчитав диапазон цикла по номеру нити и числу нитей. Без хотя бы одной директивы, а именно, `#pragma omp parallel`, не обойтись.

А как лучше – сначала выставить число нитей по `omp_set_num_threads` или каждой параллельной области приписывать `num_threads`?

Лучше **один раз** выставить число нитей, используя **`omp_set_num_threads`**, и больше его не менять без острой необходимости. Во-первых, зачем писать лишние слова, если число нитей не меняется? Во-вторых, если число нитей таки меняется, то зачем оно это делает?

Смена числа нитей может сильно тормозить, поскольку потоки могут из-за этого порождаться заново. Если где-то понадобилось задействовать меньше нитей, лучше их отмазать от работы в явном виде по условному оператору от номера нити.

Также не надо пробрасывать число нитей через переменные окружения! Лучше в явном виде культурно передать через UI (User Interface) и вызвать `omp_set_num_threads`.

Усложним задачу, распараллелим скалярное произведение двух векторов. Исходный последовательный вариант:

```
double res = 0.0;
for(int i=0; i<n; ++i) res += v1[i]*v2[i];
```

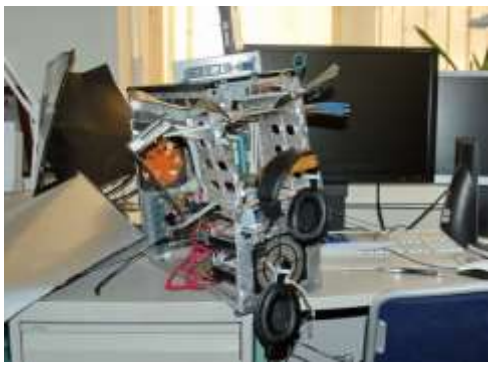
Применим опыт предыдущего примера:

```
double res = 0.0;
#pragma omp parallel
{
    #pragma omp for // директива цикла
    for(int i=0; i<n; ++i) res += x[i]*y[i];
}
```

Это, конечно, неправильный вариант. Раз у нас много потоков, у которых общее адресное пространство, то у них могут быть общие данные, aka shared, и их собственные данные, aka private, живущие, например, в стеке потока (если что, нити – это синоним). Все, что существовало до открытия параллельной области – это (обычно по умолчанию) shared. Все что появилось после открытия, то есть внутри, это private.

В нашем случае накопитель суммы res – это общая переменная (x, y – тоже общие, а счетчик цикла i появился внутри параллельной области, значит он – частный). Суммирование с накоплением в общую переменную – это некорректная операция. Эта операция не атомарная, она состоит из нескольких машинных команд: забрать один аргумент в регистр, забрать другой аргумент в регистр, сложить регистры, записать результат. Если эти команды будут выполняться одновременно с разных потоков, получится конфуз.

Пусть два потока делают общей переменной `x += 1`. Пусть изначально `x = 0`. Какой результат мы бы хотели получить? Наверное, 2? Но возможны варианты: оба потока считали значение 0, прибавили 1, оба записали результат 1, получилось 1; один поток считал значение 0, успел прибавить 1 и записать результат, после чего другой поток считал значение 1, прибавил 1, записал результат, получилось 2. Эта неприятность приводит к неопределенному результату, нестабильному поведению программы, волшебным ошибкам и хаосу. Такое явление называется состоянием гонки, aka race condition, data race. Ловить такие ошибки в коде очень неприятно. Процесс отладки параллельного приложения может быть опасен для психики и приводить к повреждению оборудования рабочего места разработчика.



Скажем OpenMP, что в res мы копим сумму, для чего добавим кляузу (*жарг.* от clause) reduction к директиве parallel:

```
double res = 0.0;
#pragma omp parallel reduction(+:res)
{
    #pragma omp for // директива цикла
    for(int i=0; i<n; ++i) res += x[i]*y[i];
}
```

То же самое другими словами:

```
double res = 0.0;
#pragma omp parallel for reduction(+:res)
for(int i=0; i<n; ++i) res += x[i]*y[i];
```

Другой вариант того же:

```
double res = 0.0;
#pragma omp parallel
{
    double lres = 0.0; // частная частичная сумма данной нити
    #pragma omp for
    for(int i=0; i<n; ++i) lres += x[i]*y[i]; // копим результат в свою переменную
    #pragma omp atomic
    res += lres;
}
```

Тут за устранение гонки отвечает директива atomic, которую можно применять к разным таким бинарным операциям. Можно было бы и сразу прям так написать:

```
double res = 0.0;
#pragma omp parallel for
for(int i=0; i<n; ++i)
    #pragma omp atomic
    res += x[i]*y[i];
```

Но это будет вообще совсем медленно, что лучше использовать последовательный вариант. Еще вариант сборки общего результата:

```
double res = 0.0;
#pragma omp parallel
{
    double lres = 0.0;  // частная частичная сумма данной нити
    #pragma omp for
    for(int i=0; i<n; ++i) lres += x[i]*y[i]; // копим результат в свою переменную

    // соберем общую сумму, избегая гонки
    const int nt = omp_get_num_threads(); // узнаем число нитей
    const int tn = omp_get_thread_num(); // узнаем номер данной нити

    for(int i=0; i<nt; ++i){
        #pragma omp barrier // барьерная синхронизация всех потоков
        if(i == tn) res += lres;
    }
}
```

Тут мы сами выполнили сборку общего результата "вручную". Собирать результат часто вручную приходится, потому что reduction мало что умеет. В плюсах (*жарг.* от C++), например, искать минимум, максимум она не умеет. А так нити культурно по очереди через барьер добавляют свой результат, что устраняет гонку. Если вдруг забыли, барьер – это такая штука, которая гарантирует, что ни одна нить не пройдет ни шагу дальше, пока они все до этого барьера не дойдут.

Можно сделать то же самое и через критическую секцию:

```
double res = 0.0;
#pragma omp parallel
{
    double lres = 0.0;  // частная частичная сумма данной нити
    #pragma omp for
    for(int i=0; i<n; ++i) lres += x[i]*y[i]; // копим результат в свою переменную
    #pragma omp critical
    {
         res += lres;
    }
}
```

Тут за устранение гонки отвечает директива `critical`, которая запускает нити в свой блок строго по одной (но в неопределенном порядке). Если atomic не подходит, используем critical.

Посмотрим еще примеры. Гнездо из двух циклов:

```
int i,j;
#pragma omp parallel for
for(i=0; i<n; ++i)
    for(j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

Кто, увидев это, пришел в ужас? Правильно, тут все нити теребят бедную общую переменную j. Можно записать так:

```
#pragma omp parallel for private(j)
```

Тут мы сказали OpenMP, что переменная j – частная, и вписали ее в private список. OpenMP породит каждой нити по экземпляру переменной j при открытии области. Переменную i добавлять в private список не требуется, переменные циклов, к которым применена директива for, становятся приватными автоматом. Но, конечно, намного правильнее делать так:

```
#pragma omp parallel for
for(int i=0; i<n; ++i)
    for(int j=0; j<m; ++j)
        w[i][j] += v[i][j];
```

Рассмотрим еще пример – матрично-векторное произведение с разреженной матрицей в формате CSR (Compressed Sparse Row):

```
#pragma omp parallel for
for(int i=0; i<n; ++i){
    double sum = 0.0;
    const int jb = ia[i], je = ia[i+1];
    for(int j=jb; j<je; ++j) sum += a[j]*b[ja[j]];
    x[i] = sum;
}
```

С этим форматом еще будем разбираться, а пока важно, что тут есть гнездо циклов (aka nested loops), у которого внутренний цикл с переменным числом итераций. Это значит, что распределяемые итерации внешнего цикла (который по i) – неоднородные. По умолчанию у нас статическая планировка цикла, то есть итерации сразу поделились поровну (или почти поровну, с разницей не более чем на одну итерацию) между потоками еще до входа в цикл. А раз итерации могут иметь разную стоимость, то может возникнуть дисбаланс. Чтобы избежать этой неприятности, используем динамическую планировку:

```
#pragma omp parallel for schedule(dynamic)
for(int i=0; i<n; ++i){
    double sum = 0.0;
    const int jb = ia[i], je = ia[i+1];
    for(int j=jb; j<je; ++j) sum += a[j]*b[ja[j]];
    x[i] = sum;
}
```

Теперь итерации будут распределяться динамически. То есть OpenMP раздаст по одной итерации всем нитям, кто первый доделает, получит следующую, и так далее. Дисбаланс устранен. Но возник оверхед (*жарг.* от overhead – накладные расходы) на динамическую планировку. Если из-за каждой итерации мы будем тратить время, смотреть, кто освободился, кому что выдать посчитать, потратится много времени. Чтобы сократить оверхед на планировку, скажем, чтобы распределялось не по одной итерации, а сразу помногу, сразу пачками, aka chunk.

```
#pragma omp parallel for schedule(dynamic, 100)
for(int i=0; i<n; ++i){
    double sum = 0.0;
    const int jb = ia[i], je = ia[i+1];
    for(int j=jb; j<je; ++j) sum += a[j]*b[ja[j]];
    x[i] = sum;
}
```

Теперь итерации распределяются сразу пачками по 100. Можно считать, что вес накладных расходов в общем времени работы цикла уменьшился, соответственно, раз в 100. Можно взять и 1000, если n достаточно большое. Но если chunk будет слишком большим, то возникнет дисбаланс, потому что если число chunk-ов не будет делиться нацело на число нитей, то дисбаланс уже будет не 1 итерация, а целый chunk.

Подведем итоги. Из директивной части нам попались словечки: #pragma omp… parallel, num_threads, reduction, private, for, schedule, atomic, critical, barrier. Еще туда же добавим master. Из библиотечной части omp_set_num_threads, omp_get_num_threads, omp_get_thread_num. Туда же добавим omp_get_wtime – для замеров времени. Про это всё почитаем подробнее в спецификации OpenMP.

## 3.2 Накладные расходы и неприятности

### 3.2.1 Race condition

Конечно, самой пакостной неприятностью является пересечение по доступу к данным, aka race condition, data race, состояние гонки, или просто гонка. Чтобы началось веселье, всего-то нужно, чтобы более одной нитки полезли в одно и то же место памяти, и чтобы хотя бы одна из них полезла туда на запись. Пусть нитей две. Пусть они обе решили что-то записать в

одну и ту же общую переменную. Если это что-то у них разное, то результат будет непредсказуемый, поскольку неизвестно, кто туда запишет последним. Состояние гонки: кто первый наложит, тому последний всё переложит. Пусть одна нить что-то пишет, другая оттуда же читает. Если нить пишет что-то отличное от того, что там было ранее, то опять результат непредсказуемый.

Примеры. Предположим, был какой-то цикл и две функции. Функция f1 считает какой-то x, f2 берет это x себе на вход.

```
type x;
void f1(type &x);
void f2(const type &x);

for(int i=0; i<n; ++i){
    f1(x);
    f2(x);
}
```

Положим, кто-то решил этот цикл распараллелить, добавив к циклу директиву.

```
#pragma omp parallel for
for(int i=0; i<n; ++i){
```

Что тут неправильно? У нескольких нитей один x. Разве это порядок? Непорядок. Как это исправить? Добавить x в private список, чтобы у каждой нити был свой x?

```
#pragma omp parallel for private(x)
```

Кто скажет, что так правильно, садитесь, два. Правильный вариант – втащить x куда положено. В последовательном коде обычно люди не заморачиваются, описывают переменные где попало и где попало потом используют. Так неправильно. Переменные описываем ровно там, где они используются.

```
#pragma omp parallel for
for(int i=0; i<n; ++i){
  type x;
  f1(x);
  f2(x);
}
```

Бывают в жизни и повеселее примеры. Вот код, вот функции. Ничто не предвещает.

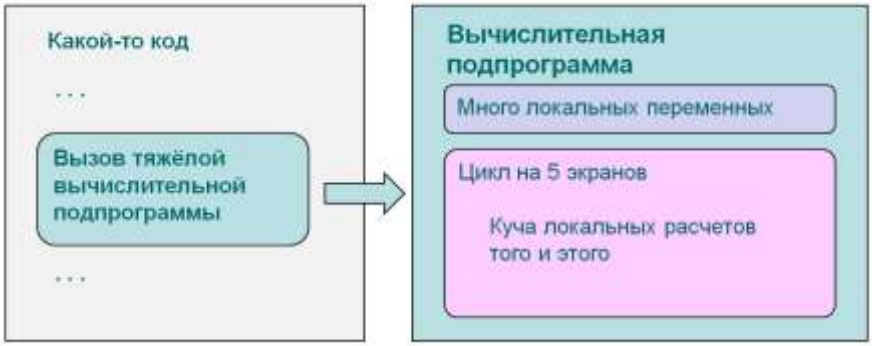
```
for(int i=0; i<n; ++i){
    f1();
    f2();
}
```

Положим, мы решили распараллелить цикл. Добавили директиву. Но кто-то до вас сделал эти функции через *универсальный интерфейс* (*жарг., груб.*). Вместо того чтобы пробросить данные из одной функции в другую культурно через аргументы, кто-то взял и завел где-то там глобальную переменную, через которую эти функции общаются.

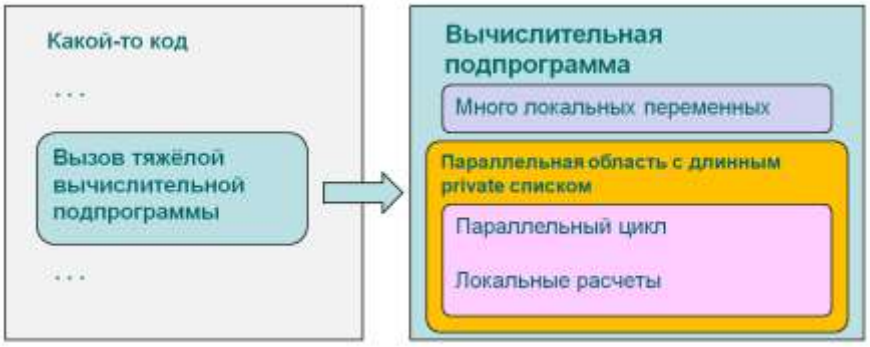
```
type x;
void f1(){
    x = a;
}
void f2(){
    b = x;
}
```

Кажется, какая дикость! Но такое часто встречается (особенно у разработчиков, пересевших на Си с Фортрана). И все. Распараллеливание обломалось. А если внутри этих функций целое дерево вызовов всяких других функций, и где-то там, в глубине, меняется какая-то глобальная переменная, то сколько часов займет ее ловить? Запомним этот вывод: локальные данные, описанные глобально, – это в высшей степени зло с точки зрения многопоточного распараллеливания (да и с любой другой точки зрения). Казалось бы, такие простые вещи. Но сколько из-за этого проблем в прикладных расчетных кодах!

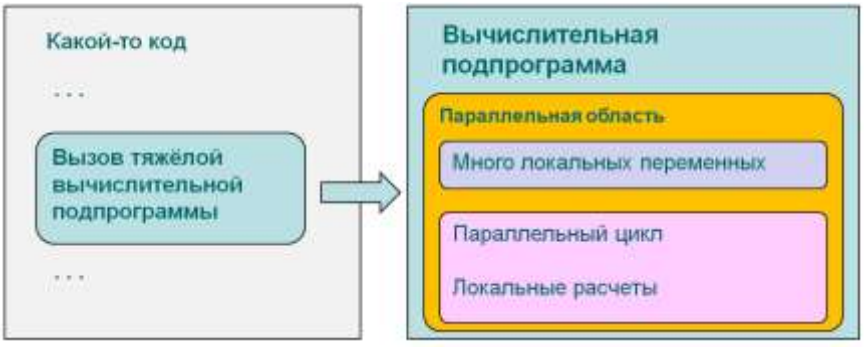
Вот пусть нам надо распараллелить какой-то код. Мы отпрофилировали его, нашли какую-то подпрограмму, на которой тратится много времени. В ней есть цикл, к которому мы можем добавить директиву OpenMP. Но есть беда: в начале подпрограммы описана куча локальных переменных, которые далее используются в цикле.



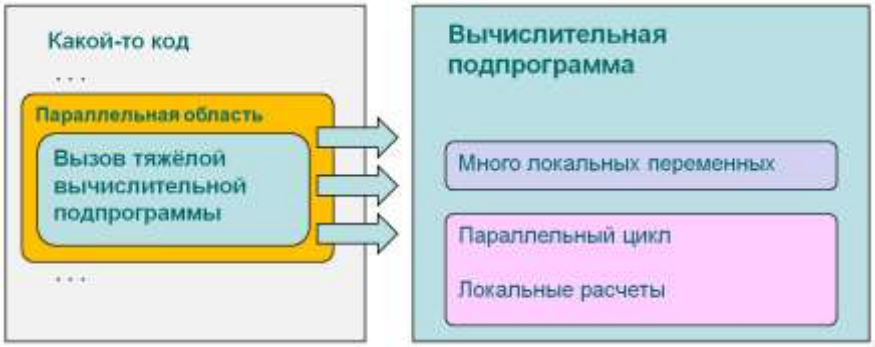

Как мы выше выяснили, работающий, но неправильный вариант:




Правильный вариант, понятное дело,




А если это legacy (*жарг.* – очень старый код, ценное наследие далекого прошлого) код на старом недобром Фортране, где нельзя взять и убрать декларации переменных под блок параллельной области или втащить в цикл? Тогда используем следующий прием:




Ставим параллельную область сверху, вокруг вызова подпрограммы, а в подпрограмме только директиву параллельного цикла. И нет проблем. Стековые переменные подпрограммы сами наплодятся для всех нитей.

А еще отметим, что не всякая запись в одно место приводит к гонке. Пусть мы что-то обрабатываем и выставляем флаг при определенном коде возврата функции.

```
bool found = false;
for(int i=0; i<n; ++i){
    if(f1(i)) found = true;
}
```

Приляпали директиву цикла:

```
bool found = false;
#pragma omp parallel for
for(int i=0; i<n; ++i){
    if(f1(i)) found = true;
}
```

Так можно? Или надо делать редукцию по общей переменной или городить критические секции? Можно оставить и так. Гонки нет, поскольку нити не используют это значение на чтение, а пишут туда только одно и то же – true. Нет гонки – нет проблемы.

### 3.2.2 Дисбаланс

Тут все понятно. Если какой-то нити досталось больше работы, чем остальным, то остальные нити будут простаивать. Надо, чтобы всем было поровну. Пусть $t_i$ – время работы $i$-й нити, затраченное на выполнение своей работы, а n – число нитей. Дисбаланс будем считать так: $d = \mathrm{n} \max_i(t_i) \,/\, \sum_{i=0}^{n} t_i$, то есть как отношение максимального времени работы по всем нитям к среднему времени.

Дисбаланс возникает при статическом распределении работы. Например, если мы статически распараллелили цикл, итерации которого неоднородны, может возникнуть дисбаланс, как уже было в примерчике выше.

Естественно, если мы сами распределили работу, например, для каждой нити составили список объектов, которые она будет обрабатывать, то мы тоже вполне могли ошибиться. Особенно если объекты неоднородные. Если мы сделали на распределении всё, что могли, а дисбаланс не пропал полностью, можно увеличить число нитей. Получится овербукинг, точнее, oversubscription. На одно ядро будет более одного потока, что не очень приятно. Но это может снизить потери на дисбаланс: какие-то нити получили мало работы и завершились, но нитей больше, чем ядер, и все ядра при деле. И вот тут как раз помогает SMT (аппаратная поддержка множественных потоков на одном ядре).

### 3.2.3 Миграция

Нить работает, что-то считает, накопила в кэше много добра, а тут ее операционная система переставляет на другое ядро или, хуже того, на другой сокет (*жарг.* от CPU socket, то есть на другой процессор). Все, что нажито в кэше, надо паковать в коробки, вызывать Газель... Более того, если нить переставилась на другой сокет, то все, что она навыделяла в памяти, станет для нее далеко, доступ с другого сокета будет медленнее. Но это уже следующий раздел.

Чтобы остановить миграцию, надо привязать нить к ядрам, aka affinity, thread binding. У процесса или потока в операционной системе (ОС) есть битовая маска соответствия – affinity. Число значимых битов равно числу виртуальных процессоров (ВП) в системе, то есть числу аппаратно поддерживаемых одновременных потоков, то есть числу ядер, умноженному на число SMT потоков на ядро. Если в системе, например, два 16-ядерных процессора с ядрами 8-way SMT, будет 256 ВП. Получается, у ВП есть s – номер сокета (то есть процессора), c – номер ядра в процессоре, t – номер на SMT ядре. Пусть их S сокетов, в каждом C ядер, на каждом по T потоков. На Linux обычно ВП (и, соответственно, биты в маске) упорядочены следующим образом: номер ВП i = t×S×C + s×C + c. Старший индекс, как ни странно, – номер SMT потока на ядре. Так ОС удобнее размещать потоки, она берет первый свободный ВП и туда ставит поток. При такой нумерации оно само получается, что пока есть свободные ядра, два процесса не окажутся на одном ядре.

Управлять affinity можно по-разному на разных системах. Где-то через переменные окружения, где-то через параметры запуска в системе очередей кластера. Но можно управлять напрямую из программы. Так надежнее. Из программы привязка, то есть affinity, тоже может выставляться по-разному в разных ОС. Поскольку кластеры почти вообще всегда работают под управлением Linux, разберемся с этой ОС. В Linux надо цеплять sched.h и использовать функции sched_setaffinity, sched_getaffinity и функции для работы с масками: CPU_SET – выставить 1 в заданную позицию, CPU_CLR – выставить 0, CPU_ISSET – узнать, что там в заданной позиции. В интернете все написано, как пользоваться. Ну пусть и тут будет пример. Вот мы где-то в программе задали нужную нам маску для удобства в виде массива байтов (чтобы не заморачиваться с установкой битов).

```
#include <unistd.h>

static void SetAffinityMask(unsigned char* mask, unsigned int mask_size){
// converts char array to bit mask and sets system mask
    if(mask_size>sizeof(cpu_set_t)*8){ printf("mask is too big!\n"); return; }
    if(mask == NULL) { printf("mask is null!\n"); return; }

    cpu_set_t mask_SYSTEM;
    memset(&mask_SYSTEM, 0, sizeof(cpu_set_t));

    for(unsigned int i=0; i<mask_size; i++)
        if(mask[i]) CPU_SET(i, &mask_SYSTEM);
    sched_setaffinity(0, sizeof(cpu_set_t), &mask_SYSTEM);
    sched_getaffinity(0, sizeof(cpu_set_t), &mask_SYSTEM);

    for(unsigned int i=0; i<mask_size; i++) // check
      if(CPU_ISSET(i, &mask_SYSTEM) != mask[i]){
          printf("setaffinity failed!");
          return;
      }
}
```

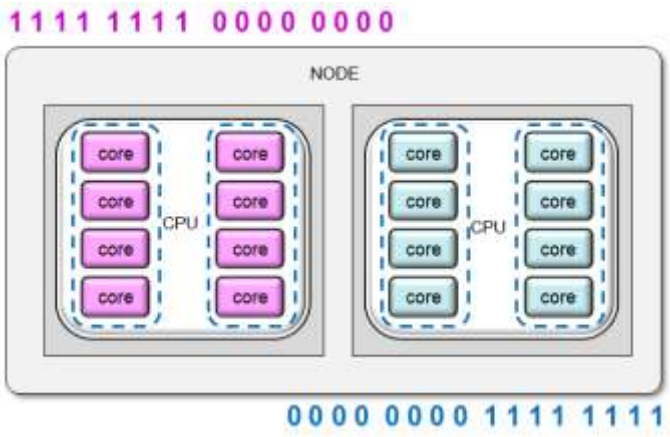
Вместо printf, конечно, должен быть ваш обработчик ошибок. Вызывать SetAffinityMask надо из параллельной области, чтобы не выяснять pid (идентификатор в ОС) каждой нити.

```
omp_set_num_threads(NT); // выставляем число нитей, которое далее будет в программе
#pragma omp parallel
{
    unsigned int mask_size = sizeof(cpu_set_t)*8;
    unsigned char mask[sizeof(cpu_set_t)*8];
    for(int i=0; i<mask_size; i++) mask[i] = 1; // как-то заполняем маску
    SetAffinityMask(mask, mask_size); // выставляем маску для данной нити
}
```

Если далее в программе число нитей не меняется, то affinity как на инициализации прибили, так и держится.

В Windows affinity ставится другими функциями. И ВП там могут быть не так упорядочены. Там надо цеплять windows.h и использовать GetCurrentProcess, GetCurrentThread, GetProcessAffinityMask, SetProcessAffinityMask, SetThreadAffinityMask (см. описание в интернетах).

Affinity можно и не выставлять каждой нити, если она у всех одинаковая. Можно выставить affinity процессу, а нити потом (по идее) должны ее унаследовать. Пусть у нас есть два процесса и два процессора по 8 ядер. Один процесс мы ходим привязать к одному процессору, другой к другому. Маски будут такие (при обычном порядке ВП в Linux):




Если ядра SMT-штные, то маску надо размножить. Пусть на каждом ядре по два потока. Тогда маски будут:

11111111 00000000 11111111 00000000 – только один CPU

00000000 11111111 00000000 11111111 – только другой CPU

Если мы хотим из каждого процесса породить по 8 нитей, и привязать каждую нить к своему ядру, то маски будут такие

10000000 00000000 10000000 00000000

01000000 00000000 01000000 00000000

00100000 00000000 00100000 00000000

... и так далее для всех 8 нитей.

Ну и у другого процесса в том же духе

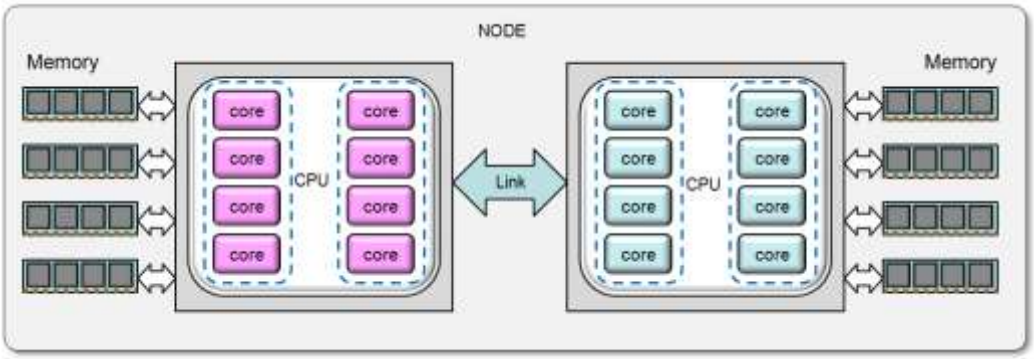
00000000 10000000 00000000 10000000

00000000 01000000 00000000 01000000

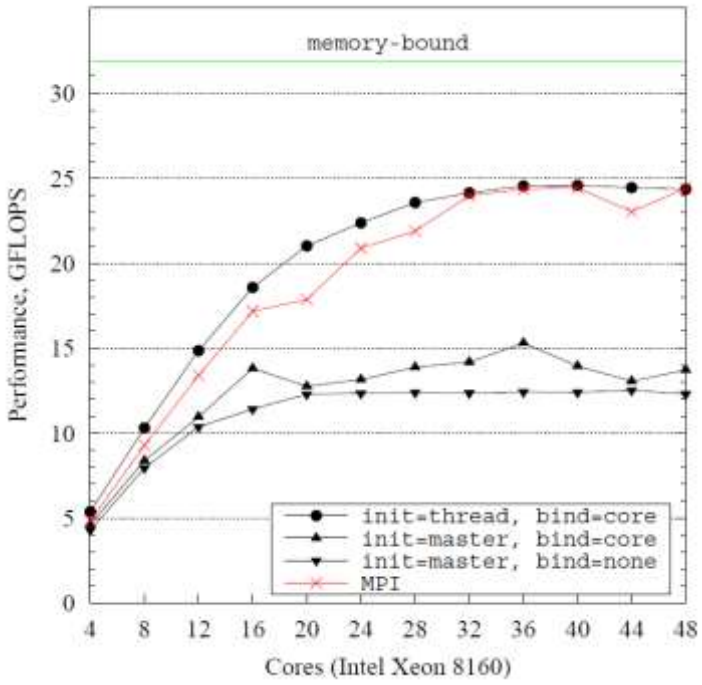
...

### 3.2.4 NUMA фактор

Когда нити на NUMA системе лезут в далекую от себя память, производительность заметно снижается. Дополним предыдущую картинку: у каждого процессора (сокета) свой контроллер памяти, а соединены эти сокеты через CPU link, какую-то шину обычно высокой пропускной способности, но не такой высокой, как суммарный бэндвис каналов памяти CPU.




Нитям с левого проца лазать в данные, размещенные в правом, не комильфо. Латентность вырастет раза в 3, а бэндвис будет ниже раза в полтора–два, например. Что же делать? Первое, конечно, это запретить миграцию установкой правильной привязки (см. предыдущий пункт). Но как проконтролировать, куда именно попали данные в памяти? Чтобы нужные данные оказались близко к нитям, что называют NUMA placement, нужно при выделении памяти выполнять инициализацию данных с той нити, или тех нитей, которые этими данными будут пользоваться. Это делается по так называемому first-touch rule. Когда мы что-то выделяем, данные еще не оказываются физически в памяти. По мере инициализации данных, они будут частями размещаться в памяти по-настоящему. Если нити все время будут работать только со своими данными, то они автоматом окажутся в близкой памяти. А если у нас, например, циклы с динамической планировкой – то ничего не выйдет. Поэтому стараемся использовать статическое распределение работы и сразу инициализировать данные из параллельной области согласно распределению. Вот пример того, как это важно:



Это показана производительность на матрично-векторном произведении с большой разреженной матрицей. Есть узел с двумя процессорами по 24 ядра. Мы пытаемся задействовать все доступные ядра, используя OpenMP. Если мы не выставим affinity и не выполним инициализацию векторов и матрицы согласно распределению работы между нитями, то будет тухляк – графики с треугольничками. Если выставим привязку, и каждая нить сразу проинициализирует свои данные, то получится график с кружочками. Видно, насколько выше производительность. Красный график – это та же операция с MPI распараллеливанием, у которого нет такой проблемы с локальностью данных (у каждого процесса свое адресное пространство).

### 3.2.5 False sharing

Тут мы не будем подробно разбираться. Просто суть явления. Когда нить что-то записывает в память, на кэш-когерентной системе это должно стать доступно всем другим нитям. А, значит, то, что нить понамодифицировала в своем уютном кэшике, должно попасть в оперативную память, чтобы другие нити получали актуальные данные. А если другая нить тоже держит в своем кэше эти данные, то изменения, внесенные одной нитью, должны поступить и в кэш к другой. Это реализует аппаратный механизм кэш-когерентности. Практически все многоядерные процессоры и сервера имеют такой механизм. Это называется ccNUMA система (cache-coherent NUMA). Так вот проброс кэш линии между кэшами – далеко не бесплатная процедурка, приводящая к задержкам. Поэтому надо стараться, чтобы нити не лазали на запись в память очень близко. Если две нити читают и пишут данные в две разные позиции какого-то массива, то это корректно, нет никакой гонки. Но если эти позиции так близко, что попали в одну кэш линию, то это может вызвать тормоза, поскольку кэш линия будет гоняться туда-сюда между кэшами почем зря. Это стоило бы иметь в виду при организации доступа к памяти в многопоточном режиме.

### 3.2.6 Накладные расходы

Что еще снижает производительность? Открытие параллельной области занимает какое-то время. Оно обычно небольшое, поскольку нити один раз порождаются вначале при открытии первой области и так и существуют далее, просто вне параллельных областей они неактивны. Но если открывать области очень часто и много, то будет заметная деградация производительности. Поэтому стараемся не открывать области почем зря, например, так не очень хорошо:

```
#pragma omp parallel for
for(int i=0; i<n; ++i) f1(i);
#pragma omp parallel for
for(int i=0; i<n; ++i) f2(i);
```

лучше так:

```
#pragma omp parallel
{
    #pragma omp for
    for(int i=0; i<n; ++i) f1(i);
    #pragma omp for
    for(int i=0; i<n; ++i) f2(i);
}
```

Конечно, если функции в этих циклах тяжелые, то без разницы. Еще оверхед создают барьеры. Они достаточно быстрые, обычно их незаметно. Но если их много на фоне

небольшой вычислительной нагрузки, то производительность заметно просядет. Поэтому можно ещё немного улучшить – устранить неявный барьер после for, добавив nowait.

```
#pragma omp parallel
{
    #pragma omp for nowait
    for(int i=0; i<n; ++i) f1(i);
    #pragma omp for
    for(int i=0; i<n; ++i) f2(i);
}
```

Еще недешево выходят критические секции и атомарные операции. Высокочастотных (т.е. в которые нити попадают с высокой частотой) критических секций надо избегать, иначе производительность может просесть ниже последовательной версии.

Отдельно стоит сказать про открытие параллельных областей с разным числом нитей. Вот этого тоже надо избегать. Если открыть область на одно число нитей, а потом на другое, то затраты на открытие могут выйти очень сильно больше. Поскольку потоки при этом могут перепорождаться заново. За этим надо следить. Лучше так не делать. Лучше один раз выставить omp_set_num_threads и не трогать. Если для какой-то операции столько нитей не нужно, то лишние нити лучше деактивировать условным оператором от номера нити, но не открывать область на другое число потоков.

### 3.2.7 Почему такое маленькое ускорение?

Вот мы что-то распараллелили и решили промерять ускорение, то есть отношение времени выполнения в последовательном режиме ко времени в параллельном режиме.

Пусть мы все идеально сделали, полностью распараллелили, дисбаланса нет, оверхедов нет, данные даже проинициализировали и потоки привязали к ядрам. Берем 4 ядра, а ускорение почти незаметно. Как так!? А такой пример уже был. Приведем его еще раз.

```
int N=330000123;
double t0=omp_get_wtime();
#pragma omp parallel for num_threads(4)
for(int i=0; i<N; ++i) x[i] = x[i] + y[i];
double t=omp_get_wtime() - t0; // время выполнения, с
```

Запускаем (на каком-то старом ноуте автора с 4-ядерным CPU), замеряем для одной нити и для четырех, получаем:

```
X+Y STRIDE1 T4 X1 res: 9.900018400000E+007 time: 0.075 s GFLOPS: 0.875
X+Y STRIDE1 T4 X4 res: 9.900018400000E+007 time: 0.075 s GFLOPS: 0.880
```

Ускорение 1 раз из 4. То есть его, ускорения, нет. Посмотрим на характеристики памяти: два канала DDR3 PC3-12800, итого суммарно 25.6 ГБ/с. Арифметическая интенсивность у нашего поэлементного сложения векторов 1/24 (N FLOP на 16N чтения и 8N записи). Достижимая производительность будет 25.6/24 = 1.07 GFLOPS. Получилось, что результат на самом деле в районе 80% от максимально достижимой производительности. И не надо возмущенно трясти законом Амдала. Мы просто полностью исчерпали пропускную способность памяти одним потоком. Все параллельное, а ускорения нет. Так бывает. Потому что дело не только в ядрах. Ну ладно, возьмем операцию подороже, например, тригонометрию.

```
#pragma omp parallel for num_threads(4)
for(int i=0; i<N; ++i) x[i] += sin(y[i]);
```

Запускаем, получаем:

```
SIN T1 7.151703784123E+007 time: 0.614 s SINPS: 0.108
SIN T4 7.151703784123E+007 time: 0.180 s SINPS: 0.368
```

Ну вот, все работает, ускорение 3.4 раза из 4. Вывод: оценивать OpenMP ускорение без учета арифметической интенсивности операции и пропускной способности памяти не имеет никакого смысла!

А вот пусть у нас вообще всё идеально распараллелено, нити работают со своими данными, да еще и операция compute bound – очень интенсивная, память нас вообще не держит. Почему опять ускорение не дотягивает? Почему, например, 14 раз на 16 ядрах? Еще одна подстава – частота работы процессора. Когда задействовано одно ядро, процессор может заметно задирать частоту (например, Intel Turbo Boost). Получается, когда программа запущена в последовательном режиме, процессор работает на более высокой частоте. Ну и вот результат. На 16 ядрах процессор работает на 2.8 ГГц, например, а когда на одном – 3.2 ГГц.

А можно получить ускорение относительно последовательного режима больше, чем число ядер? Да запросто. Если использовать SMT, то есть плодить по столько нитей на ядро, сколько оно аппаратно поддерживает. Тогда добавится выигрыш за счет ILP параллелизма и более высокой загрузки исполнительных устройств ядра (больше независимых команд), и за счет снижения потерь на латентность доступа к памяти (пока один поток ждет данные из памяти, другой в это время может что-то считать полезное). Так можно будет обогнать последовательный режим в большее число раз, чем ядер, даже с учетом пониженной частоты.

## 3.3 Устранение зависимостей по данным в сеточном методе

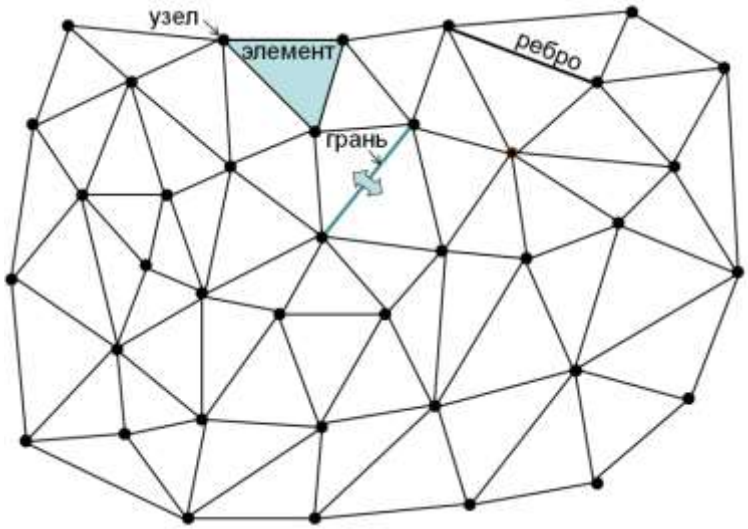
### 3.3.1 Постановка проблемы

То была разминка, теперь хардкор. Рассмотрим проблему гонки и зависимостей на примере какой-то обобщенной операции над сеточными данными. Во введении рассказывалось про устройство CFD алгоритма. Возьмем в качестве примера такой алгоритм.

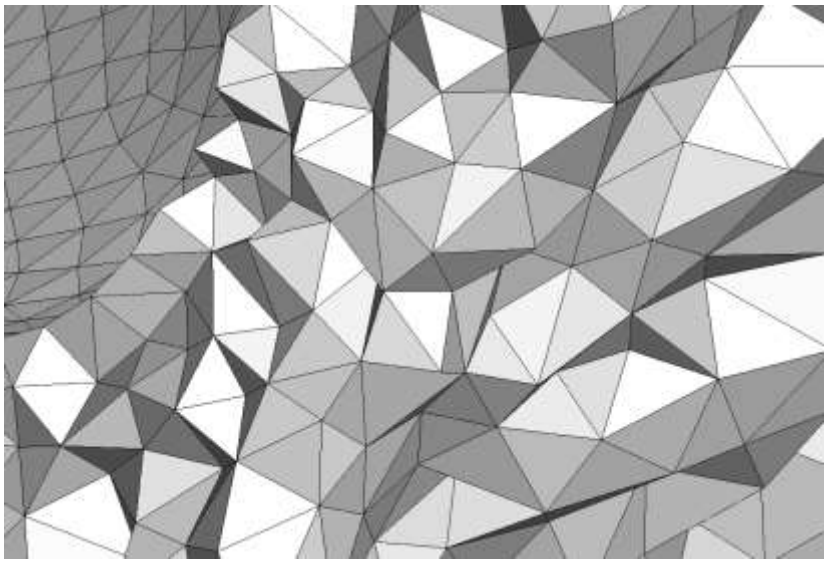
Для дискретизации по пространству используется неструктурированная сетка и явная по времени схема. Кто не знает, что такое сеточные методы и дискретизация, сильно не заморачивайтесь, дальше это все перейдет более простые термины и сведется к графу, наверное, более всем привычному. Но для начала все-таки приведем какое-то минимальное описание того, что происходит, чтобы было понятно, в чем проблема.

Сетка – это разбиение некоторой пространственной расчетной области на сеточные элементы (в двухмерном случае – многоугольники, в трехмерном – многогранники). Сетка задана в виде набора узлов (вершин) и набора элементов. Объединение сеточных элементов полностью заполняет нашу расчетную область, а сами элементы не наползают друг на друга, а стыкуются через общие грани и узлы. Ниже на картинке показан пример двухмерной треугольной сетки. Двухмерной – просто для простоты рисования картинок. В случае трехмерной сетки всё, по сути, то же самое, для наших целей никакой разницы, но изображать сильно сложнее.

У нас есть несколько наборов сеточных объектов – узлов, элементов, ребер и граней этих элементов. На картинке элементы – это треугольнички. А ребра и грани в двухмерном случае – это одно и то же. В трехмерном случае грани станут многоугольниками – гранями многогранных элементов, а ребра останутся ребрами.




В 3D это выглядит как-то так:

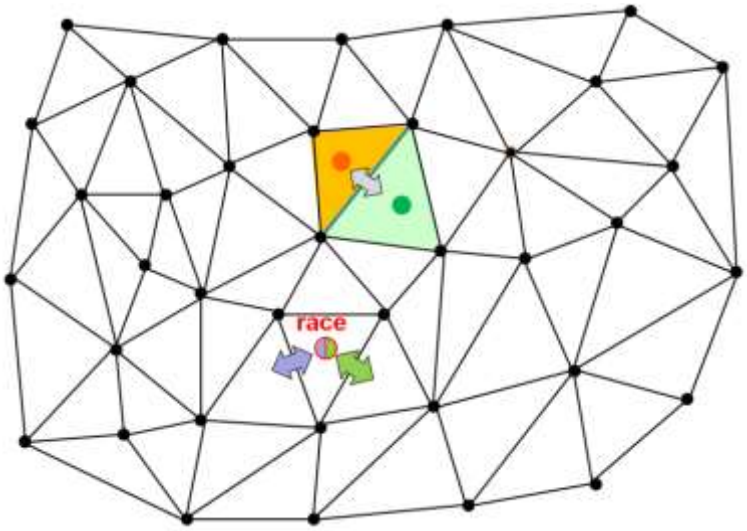


Теперь надо понять, где на сетке заданы сеточные функции, то есть, например, наши переменные, физические или консервативные. Можно задавать переменные в узлах, а можно – в элементах. В первом случае расчетные ячейки, aka контрольные объемы или просто ячейки, будут как-то хитро строиться вокруг узлов, во втором – ячейками будут сами сеточные элементы, что намного проще. Поэтому для простоты пока пусть сеточные функции заданы на элементах. На самом деле и не важно, где, все равно все сведется к графу.

Основной и самой ресурсоемкой операцией нашего простейшего CFD алгоритма является расчет потоков через внутренние грани сетки. Внутренние грани – это грани, разделяющие два сеточных элемента (те грани, что торчат наружу и инцидентны только одному элементу – внешние). Более подробно про алгоритм было в разделе 1.6.

Смысл операции в следующем. В цикле перебираем по одной все внутренние грани сетки. Вот взяли очередную грань, на картинке ниже она цветом выделена с серой стрелочкой. Для этой грани смотрим, какие значения физических переменных в элементах с двух сторон от грани. На картинке ниже – оранжевый и зелененький треугольники. По этим значениям некоторым образом вычисляем, сколько через эту грань протечет моделируемой среды. То

есть, считаем потоки наших переменных. Найдя значения потоков, мы их прибавляем в один из элементов (куда "прибыло") и вычитаем их из другого (откуда "убыло").




Очень упрощенно представим это в виде такого цикла:

```
for(int iface=0; iface<NF/*Number of faces*/; ++iface){ // цикл по граням
    double fluxes[M/*число переменных*/]; // потоки через грань
    // вычисляем потоки через i-ю грань
    CalculateFluxes(iface, fluxes); // большая тяжелая функция
    // далее берем из массива граней номера инцидентных ячеек
    // и добавляем потоки в эти ячейки, но с разным знаком
    for(int j=0; j<M; ++j) Var[Faces[iface].cells[0]][j] -= fluxes[j];
    for(int j=0; j<M; ++j) Var[Faces[iface].cells[1]][j] += fluxes[j];
}
```

где Faces – массив всех граней, то есть массив структур, в которых содержатся данные про грани, в частности, в cells лежат номера инцидентных ячеек. Var – "двухмерный" массив, точнее, блочный вектор, хранящий для каждой ячейки значения переменных (сеточных функций). А двухмерный массив – это как? Тут надо объяснять, как сделан класс блочного вектора, что перегружен operator[], который по первому индексу позиционирует указатель на начало нужного блока в простом векторе... А зачем эти сложности? Давайте в коде ситуацию упростим, чтобы примеры кода были покороче. Пусть число переменных M равно 1, сути это не меняет.

```
for(int iface=0; iface<NF/*Number of faces*/; ++iface){ // цикл по граням
    double fluxe; // поток через грань
    CalculateFlux(iface, fluxes);
    Var[Faces[iface].cells[0]] -= flux;
    Var[Faces[iface].cells[1]] += flux;
}
```

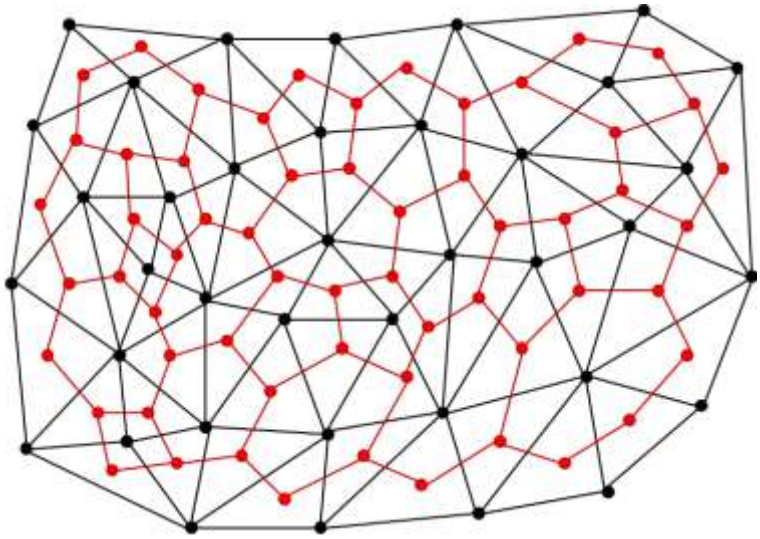
Логично этот цикл распараллелить. Но если мы добавим директиву цикла

```
#pragma omp parallel for
for(int iface=0; iface<NF/*Number of faces*/; ++iface){ // цикл по граням
```

то ничего не будет работать. С легкостью возникнет состояние гонки, когда две нити будут одновременно прибавлять свои потоки в одну ячейку (см. на картинке выше элемент с зеленой и синей стрелочками и подписью “race”).

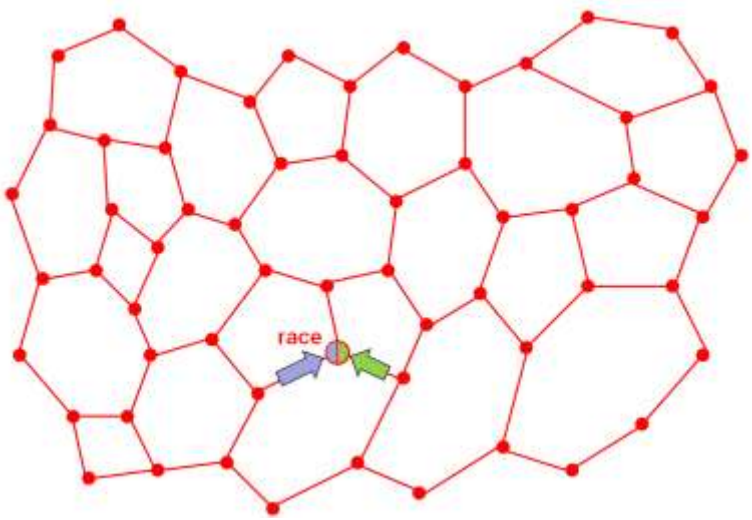
Из-за чего тут проблема. Получается, у нас операция по набору граней, а запись результата происходит в объекты другого набора – ячейки. Обрабатывая грань, результат пишем в инцидентные ей ячейки (в нашем случае пару элементов, содержащих эту грань). На этой записи и происходит конфуз.

Но нам проще дальше говорить в терминах графа, описывающего смежность ячеек (ячейки смежные, если у них есть общая грань). В таком графе связей (или смежности) наших ячеек вершины графа соответствуют ячейкам. Ребрам графа соответствуют грани между ячейками. Если между ячейками i и j есть общая грань, то в графе вершины i и j соединены ребром. Наша сеточка на картинке выше, это, по сути, и есть граф. Но это узловой граф, aka нодальный граф, nodal graph. Его вершины соответствуют узлам. Если ячейки строятся вокруг узлов, то этот граф нам и нужен. Если ячейками являются элементы, то нам нужен граф связей элементов, то есть дуальный граф, aka двойственный граф, dual graph. Вот тут он красным нарисован.



С этим графом, графом связей ячеек, мы и будем дальше иметь дело. Дальше уже неважно, где были заданы переменные – в узлах или элементах. Теперь у нас есть вершины графа, с которыми ассоциированы ячейки и заданные в них наборы переменных, и есть ребра, которым соответствуют грани между ячейками. Если ячейки были бы вокруг узлов, то это черный граф на картинке выше, а в нашем случае, когда ячейки – это элементы сетки, то это красный граф.

Ладно, забыли про сетку. Теперь еще раз в терминах графа. Есть операция по набору ребер графа, то есть цикл по набору ребер, а запись результата для каждого ребра – в две вершины ребра. Мы хотим этот цикл распараллелить, чтобы ребра обрабатывались одновременно множественными нитями. И тут происходит конфуз. Если два ребра, имеющих общую вершину, будут одновременно обрабатываться разными потоками – возникнет состояние гонки на суммировании с накоплением в общую вершину.




С этим надо что-то делать. Далее рассмотрим много разных вариантов. Давайте наш упрощенный код еще немного обобщим: изменим названия, приведем к терминам графа и уберем названия того, что именно считается. Пусть считаются не потоки, а вообще что-то.

```
for(int ie=0; ie<NE/*Number of edges*/; ++ie){ // цикл по ребрам графа
    double r = Calc(ie); // что-то считаем, буква r - result
    X[E[ie].v[0]] += r; // добавляем что-то
    X[E[ie].v[1]] += r; // для простоты даже пусть с одним знаком
}
```

Вот получилась такая простенькая штука.

E – массив ребер графа (E – Edge) размера NE, v в структуре ребра – номера инцидентных ребру узлов (V - Vertex).

X – массив значений какой-то переменной во всех вершинах графа.

Дальше возникнут и другие массивы, будем их комментировать по мере появления.

### 3.3.2 Полное дублирование вычислений

У нас был цикл по ребрам графа с добавлением результата в вершины. Это приводит к гонке. Тогда сделаем цикл не по ребрам, а по вершинам. Будем перебирать все вершины, для каждой вершины перебирать инцидентные ей ребра, делать вычисления, и записывать результат только в эту вершину. Итерации цикла получились независимы по данным, их можно делать в любом порядке, там нет пересечений или гонки. А что есть? Есть – удвоенная стоимость. Каждое ребро будет обрабатываться два раза. А так – всё стало просто:

```
#pragma omp parallel for
for(int iv=0; iv<NV/*Number of vertices*/; ++iv){ // цикл по вершинам графа
    for(int k=0; k<V[iv].Nf; ++k){ // цикл по ребрам из данной вершины
        int ie = V[i].e[k]; // номер данного ребра в массиве всех ребер
        double r = Calc(ie); // что-то считаем
        X[iv] += r; // добавляем это что-то в вершину
    }
}
```

Тут появился массив структур V – массив всех вершин графа, размера NV, свойство e в структуре вершины – номера инцидентных ребер, а Nf – их количество.

Что с этим вариантом не так?

Недостатки: в два раза больше вычислений.

### 3.3.3 Атомики

Критическую секцию даже пробовать не будем, это будет совсем медленно. Просто сделаем суммирование с накоплением атомарным, что исключит гонку.

```
#pragma omp parallel for
for(int ie=0; ie<NE/*Number of edges*/; ++ie){ // цикл по ребрам графа
    double r = Calc(ie); // что-то считаем
    #pragma omp atomic
    X[E[ie].v[0]] += r; // добавляем что-то
    #pragma omp atomic
    X[E[ie].v[1]] += r;
}
```

Если Calc достаточно тяжелая, а число переменных, которые суммируются, не очень большое, то атомики могут и не сильно влиять на производительность.

В случае расчета потоков, если используется явная схема, то переменных будет 5, суммирование в два набора – итого 10 атомарных сложений. Это, может, еще куда ни шло. А если схема неявная, там надо еще будет писать вклад в матрицу Якоби (4 блока 5x5) – еще 100 атомиков. А это уже может быть совсем тухло. А если там не суммирование, а другая операция, к которой атомик неприменим? Синус какой-нибудь, например. Тогда этот вариант не подходит.

Недостатки: накладные расходы на атомики, ограниченная применимость.

### 3.3.4 Разделение на две операции

У нас нет проблемы посчитать, у нас проблема – записать результат. Тогда введем промежуточный массив по всем ребрам (в коде – R). Да, большой массив, а что делать? Сначала посчитаем значения и запишем их в этот массив. У каждого ребра будет свое значение, нет пересечений по данным. А потом просуммируем результаты в вершины. Для этого будем перебирать все вершины, для каждой вершины перебирать инцидентные ей ребра, брать уже посчитанные результаты с этих ребер и добавлять в вершину.

```
#pragma omp parallel
{
    #pragma omp for // первая операция - считаем значения на ребрах
     for(int ie=0; ie<NE/*Number of edges*/; ++ie) // цикл по ребрам графа
         R[ie] = Calc(ie); // что-то считаем, теперь в массив R по всем ребрам

    #pragma omp for // вторая операция - добавляем значения с ребер в вершины
    for(int iv=0; iv<NV/*Number of vertices*/; ++iv){ // цикл по вершинам графа
        for(int k=0; k<V[iv].Nf; ++k){ // цикл по ребрам из данной вершины
            int ie = V[i].e[k]; // номер данного ребра в массиве всех ребер
            X[iv] += R[ie]; // добавляем это что-то в вершину
        }
    }
}
```

Получилось, что тяжелые вычисления не дублируются. Это хорошо.

Недостатки: дополнительный большой массив – расход памяти, лишний memory traffic.

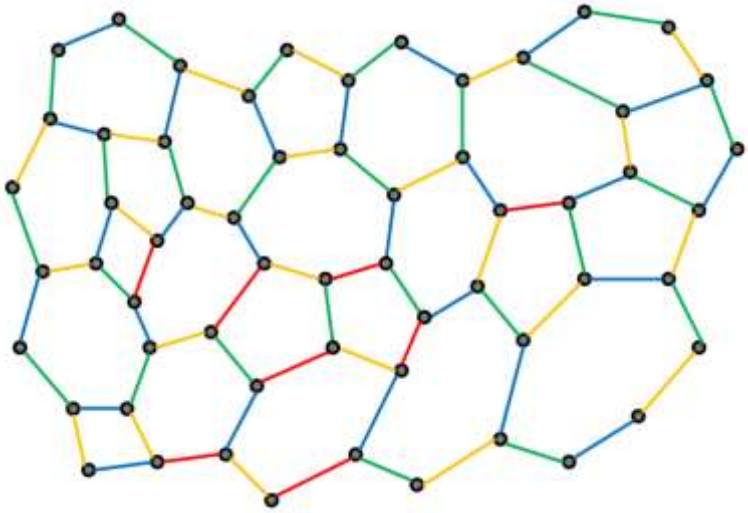
Мы же для простоты положили, что у нас всего одно значение считается. А если их много? Когда число значений большое, то будут большие накладные расходы на повторное чтение с промежуточного массива. Да и в память он может и не влезть.

### 3.3.5 Раскраска графа по ребрам

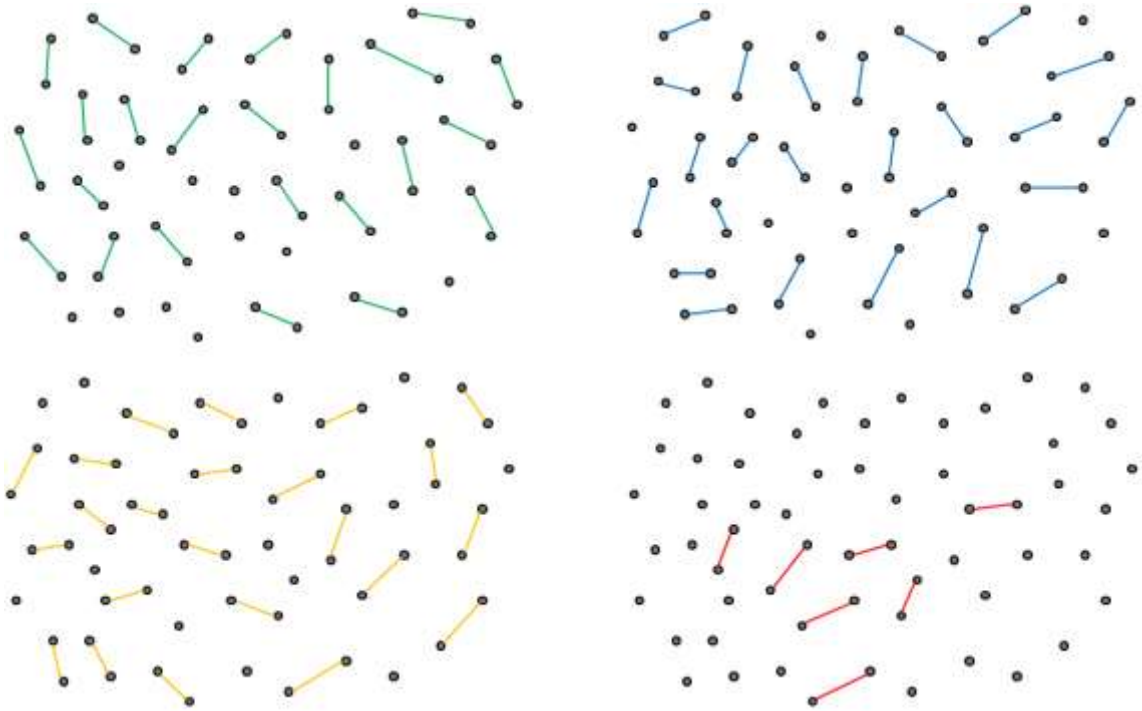
Попробуем покрасить графа. Покрасим ему ребра так, чтобы из одной вершины не торчало двух ребер одного цвета. Число цветов будет не меньше, чем максимальная степень вершины графа, то есть максимальное число ребер, инцидентных одной вершине.

Сделаем это каким-нибудь простеньким жадным алгоритмом (см. в интернете про жадную раскраску и прочие раскраски, если интересно). Там вряд ли получится ровно минимум цветов, обычно на один-два больше. Но для наших целей это вполне приемлемо, несколько лишних цветов почти никак не влияют на производительность.

Покрасили, получилось, как на картинке ниже.



Получилось, что если параллельно обрабатывать не все ребра, а ребра только одного цвета, то не будет пересечений по данным. У ребер одного цвета нет общих вершин, гонка исключена.



Значит, будем обрабатывать ребра по цветам по очереди. Между цветами обязательно барьер, иначе цвета наползут друг на друга, появятся общие вершины, возникнет гонка.

Поменяем нумерацию ребер. Переупорядочим все ребра по цветам. Сначала будут идти все ребра одного цвета, потом другого, ... Все же понимают, как менять нумерацию? Если что, обсудим на занятии.

Пусть получилось NC цветов (C – Color). Занесем диапазоны номеров каждого цвета в массив цветов C размера NC+1. C[i] – позиция начала i-го цвета в массиве ребер. C[NC] равен NE (число ребер).

```
#pragma omp parallel
{
    for(int ic=0; ic<NC/*Number of colors*/; ++ic){ // цикл по цветам
        #pragma omp for
        for(int ie=C[ic]; ie<C[ic+1]; ++ie){ // цикл по ребрам графа данного цвета
            double r = Calc(ie);
            X[E[ie].v[0]] += r;
            X[E[ie].v[1]] += r;
        }
        #pragma omp barrier // для наглядности (в omp for неявный барьер уже есть)
    }
}
```

Теперь получилось, что операция делается не за раз, а за некоторое количество тактов (у каждого цвета свой параллельный цикл) с синхронизацией после каждого такта. Это несколько снижает производительность, но это незаметно, если объем вычислений достаточно большой. Но это не главная проблема.

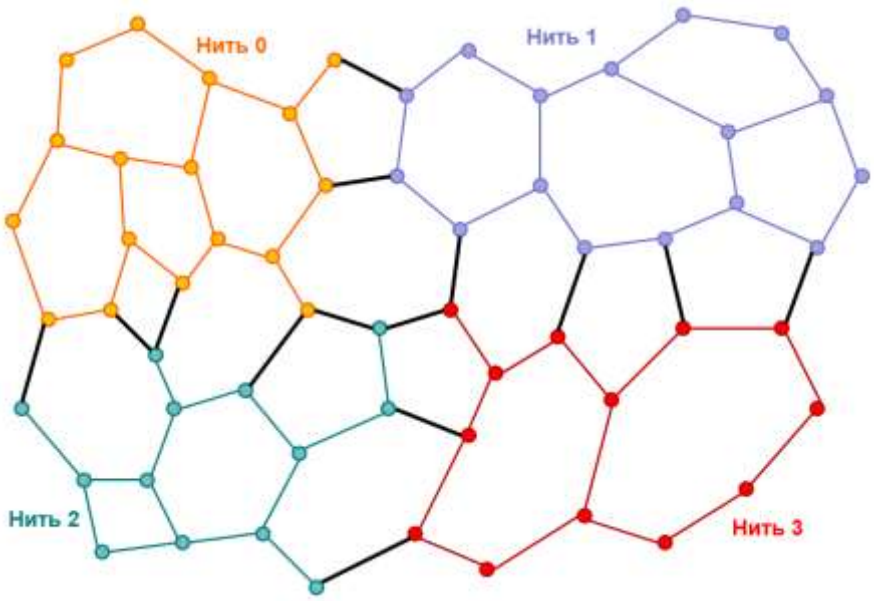
Основная проблема в том, что мы снижаем повторную используемость данных в кэш. Раньше мы могли упорядочить ребра лексикографически, чтобы, обрабатывая ребра подряд, мы часто попадали в одну и ту же вершину (или близкие по номерам вершины) при записи результата. А тут, когда мы адресуемся в инцидентные ребру вершины, мы точно не попадем в одну и ту же вершину. Значит, число кэш-промахов будет заметно выше.

Недостатки: хуже локальность доступа данных.

### 3.3.6 Декомпозиция, последовательная обработка интерфейсных граней

Распределим работу между нитями статически, распределив вершины графа между параллельными потоками (нитями). Сопоставим потокам цвета, чтобы удобнее было рисовать. Получается, надо задать вершинам цвета.

За такую декомпозицию отвечает какой-то внешний алгоритм, какая-то библиотека. Об этом более подробно в разделе 4.3.1. Этот алгоритм так распределит нам вершины графа, чтобы 1) разбиение было сбалансированным (число вершин одного цвета было примерно одинаковым); 2) чтобы количество ребер с вершинами разного цвета, aka edge-cut, было как можно меньше. Такие ребра, у которых вершинки разных цветов, назовем интерфейсными. Они нам и создают состояние гонки. Теперь по раскраске вершин покрасим ребра. Ребра, у которых обе вершины одного цвета, в этот цвет и выкрасим. Интерфейсные ребра будут отдельного цвета (на картинке – черный), которого не было в вершинах. Пример показан на картинке ниже.




Как и в предыдущем случае, переупорядочим все ребра по цветам. У ребер получилось NC цветов. Это NC равно NT+1, где NT – число нитей (T – Threads). Занесем диапазоны номеров ребер для каждого цвета в массив цветов C размера NC+1. В C[k] будет позиция начала k-го цвета в массиве всех ребер E, который мы упорядочили по цветам. Ребра k-го цвета будут в интервале ie = C[k], …, C[k+1]-1. Соответственно, в последнюю позицию C[NC] записываем число ребер NE.

У каждой нити есть свой цвет. Но что делать с интерфейсом? А интерфейс пусть отдельно обработает какая-то одна нить в последовательном режиме.

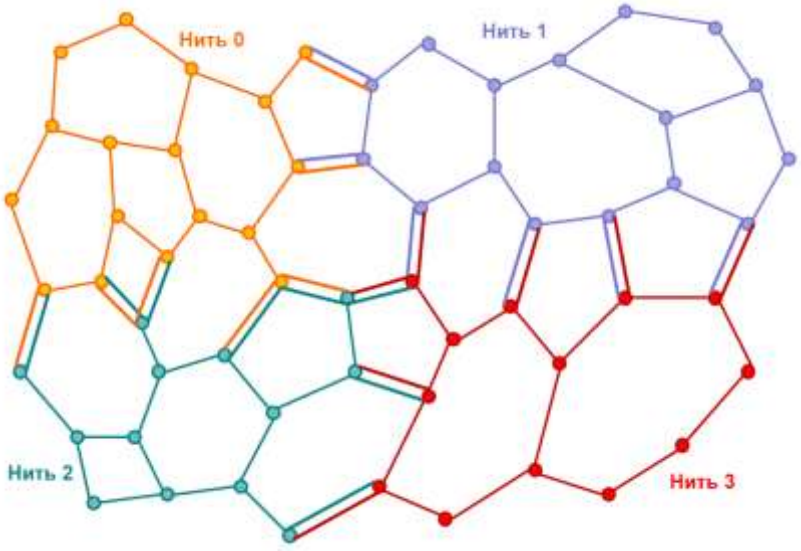
```
#pragma omp parallel
{
    int tn = omp_get_thread_num(); // номер данной нити
    int nt = omp_get_num_threads();   // число нитей
    // теперь тут нет директивы цикла, у каждой нити просто свой диапазон
    for(int ie=C[tn]; ie<C[tn+1]; ++ie){ // цикл по ребрам данной нити
        double r = Calc(ie);
        X[E[ie].v[0]] += r;
        X[E[ie].v[1]] += r;
    }

    #pragma omp barrier // обязательно барьер! иначе интерфейс налезет и будет гонка
    if(tn == nt-1){ // последняя нить займется интерфейсом
        for(int ie=C[nt-1]; ie<C[nt]; ++ie){ // цикл по интерфейсным ребрам
            double r = Calc(ie);
            X[E[ie].v[0]] += r;
            X[E[ie].v[1]] += r;
        }
    }
}
```

Минусы: последовательная обработка интерфейса, деградация с ростом числа нитей.

Если нитей мало, а ребер много, вес интерфейса будет небольшим, и мы получим приемлемое ускорение. Но когда нитей много, последовательная обработка интерфейса всё портит. Попробуем это исправить.

### 3.3.7 Декомпозиция, дублирование вычислений по интерфейсу

Выше у нас уже был вариант с дублированием вычислений по всей сетке. Почему не применить дублирование, но только для интерфейса? Это будет побыстрее, чем последовательная обработка (при числе нитей больше двух). В этом варианте нити перебирают интерфейс, пропускают ребра, если обе инцидентных вершины не принадлежат нити, и выполняют расчет, если одна из вершин им принадлежит. Таким образом, каждое интерфейсное ребро будет обсчитано два раза.




Теперь как это будет в коде. Пусть цвета вершин хранятся в массиве P (Partition). Изменим только интерфейсный блок в коде как-то так:

```
#pragma omp barrier // обязательно барьер! иначе интерфейс налезет и будет гонка
{ // все нити займутся интерфейсом, условие на номер нити убрано
   for(int ie=C[nt-1]; ie<C[nt]; ++ie){ // цикл по интерфейсным ребрам
       // отмечаем, за какие вершины отвечает эта нить
       bool doit[2] = {P[E[ie].v[0]]==tn, P[E[ie].v[1]]==tn};
       if(!doit[0] && !doit[1]) continue; // наших вершин нет - пропускаем
       double r = Calc(ie);
       if(doit[0]) X[E[ie].v[0]] += r;
       if(doit[1]) X[E[ie].v[1]] += r;
   }
}
```

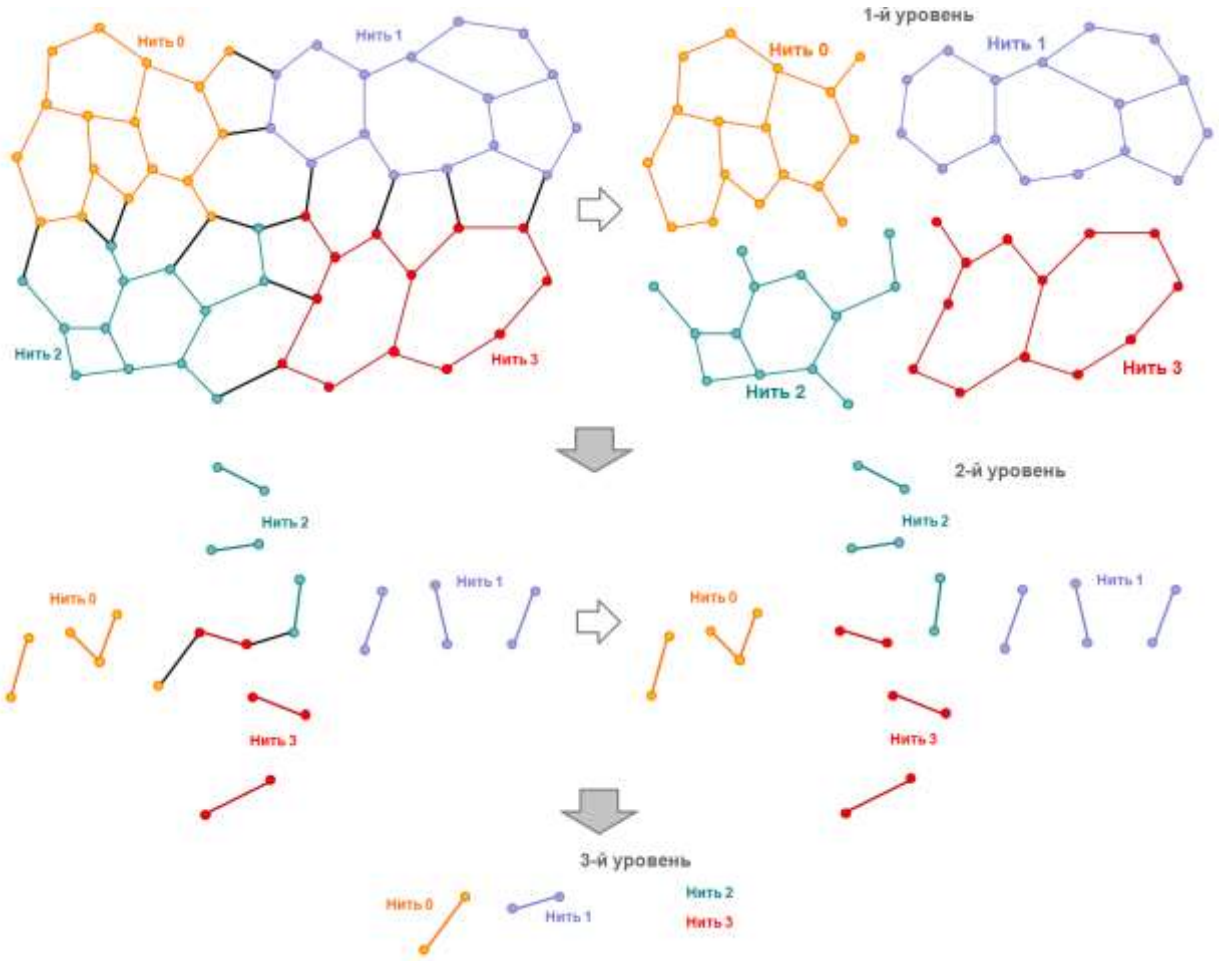
Стало лучше, интерфейс теперь делается параллельно, но...

Минусы: в два раза больше вычислений по интерфейсу, деградация (в меньшей степени) с ростом числа нитей.

### 3.3.8 Многоуровневая декомпозиция

Раз мы умеем выполнять операцию в несколько тактов через барьер, то применим и тут этот подход. Вот мы покрасили, переупорядочили ребра, но в конце массива ребер остался набор интерфейсных ребер. А чем он хуже, чем исходный граф? Да ничем! Берем, выпиливаем из графа интерфейсные ребра, берем инцидентные им вершины, называем это новым графом, применяем ту же процедуру декомпозиции, повторяем то же самое, что делали с исходным графом. Повторяем процедуру, пока интерфейс не усохнет до нескольких ребер, которые можно доделать одним потоком. Получим несколько уровней декомпозиции. Значит, в

параллельной реализации добавится внешний цикл по уровням. Массивы с диапазонами цветов, C, у каждого уровня будут свои. На каждом уровне у нас будет свой набор ребер, упорядоченных по цветам, то есть по нитям.




Для уровней сделаем массив указателей на их массивы диапазонов цветов, назовем его L (Levels), а число получившихся уровней – NL. Получим следующий код.

```
#pragma omp parallel
{
    int tn = omp_get_thread_num(); // номер данной нити
    for(int il=0; il<NL; ++il){ // цикл по уровням декомпозиции интерфейса
        int *C = L[il]; // берем диапазоны цветов данного уровня
        for(int ie=C[tn]; ie<C[tn+1]; ++ie){ // цикл по ребрам этой нити на этом уровне
            double r = Calc(ie);
            X[E[ie].v[0]] += r;
            X[E[ie].v[1]] += r;
        }
        #pragma omp barrier // обязательно барьер! иначе будет гонка
    }
}
```

Минусы: немножко сложненько получается с этой многоуровневой декомпозицией. А так одни плюсы. С числом нитей не деградирует, последовательных частей не имеет, данные компактны в памяти, фиксированное распределение по нитям – легко делать NUMA placement.

## 3.4 Внеклассное чтение

Спецификации стандартов OpenMP живут тут:

- https://www.openmp.org/specifications/

Четвертый стандарт вполне подойдет:

- https://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf – краткая шпаргалка
- https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf – полная спецификация

Ссылки на разные учебные материалы тут:

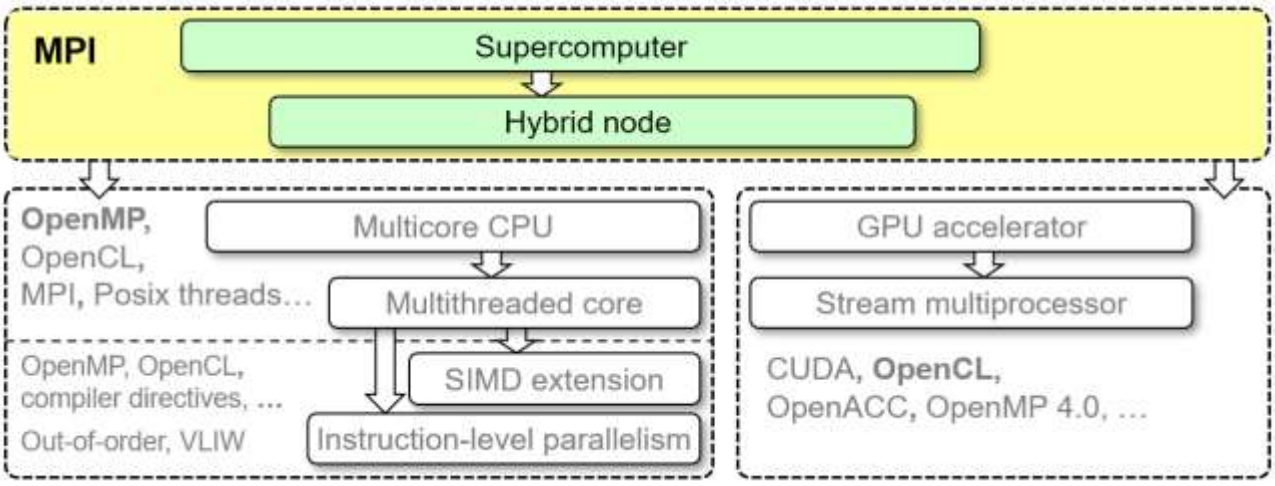
- https://www.openmp.org/resources/tutorials-articles/
  Например: https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf
- https://computing.llnl.gov/tutorials/openMP/
- R.Trobec, B. Slivnik, P. Bulić, B. Robič. Introduction to Parallel Computing From Algorithms to Programming on State-of-the-Art Platforms. 2018. Springer. https://doi.org/10.1007/978-3-319-98833-7 – глава 3.

Ну и наши МГУ-шные учебники

- Антонов А.С. Параллельное программирование с использованием технологии OpenMP: Учебное пособие.-М.: Изд-во МГУ, 2009. - 77 с. ISBN 978-5-211-05702-9. Выложен тут: https://parallel.ru/info/parallel/openmp/
- Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - М.: Издательство Московского университета, 2012.-344 с.- (Серия "Суперкомпьютерное образование"). ISBN 978-5-211-06343-3. Там глава 2 и примеры в главе 3.

# 4 MIMD с распределенной памятью. MPI распараллеливание

Разобрались параллелизмом внутри многопроцессорного узла (сервера), перейдем на уровень выше. DM-MIMD распараллеливание – Distributed Memory, Multiple Instruction streams, Multiple Data streams.




Пусть имеется многопроцессорная система с распределенной памятью, на которой нам нужно что-то посчитать. В такой системе имеется множество вычислительных серверов, которые называют узлами (nodes). А сама такая вычислительная система называется кластерной или просто кластером. Узлы кластера в единое решающее поле объединяет высокопроизводительная сеть, aka interconnect, коммуникационная среда. Или даже несколько сетей: файловая система и система управления обычно сидят на сетке попроще, а самая дорогая и мощная сеть используется для параллельных вычислений. Раз уж упомянули файловую систему, у кластеров это обычно большая параллельная файловая система из нескольких серверов, общая для всех узлов. У кластера имеется управляющий узел (или узлы), aka frontend, морда, головная машина, голова. Этот узел может торчать наружу в сеть Internet для доступа пользователей. С него задания направляются на вычислительные узлы. Ну не будем усложнять. По сути, получается, что просто есть множественные процессоры (на схеме ниже PU – Processing Unit), не имеющие общего адресного пространства оперативной памяти, но соединенные интерконнектом. По этой сети процессы, выполняющиеся на разных узлах, могут обмениваться друг с другом данными.

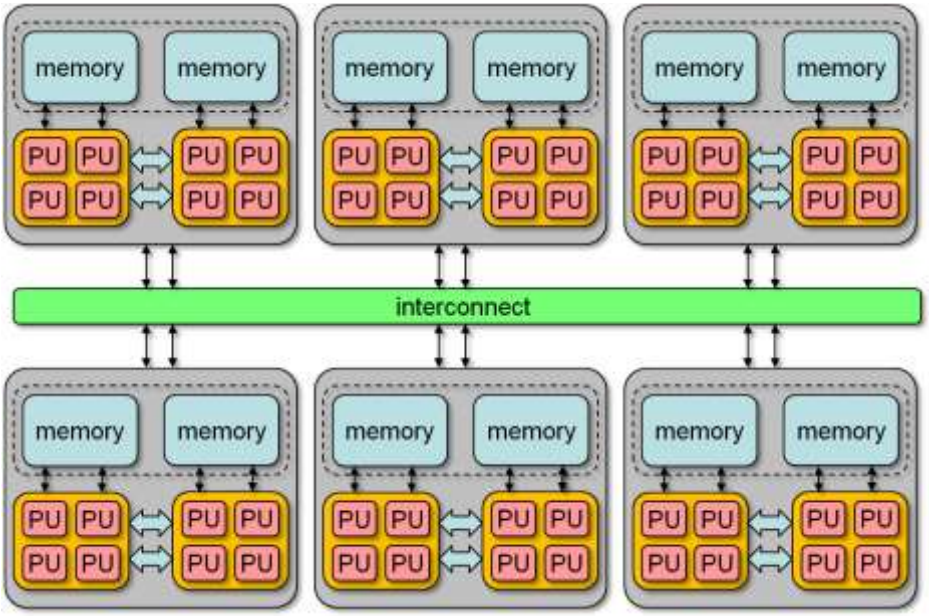
Каждый узел кластера сам по себе является многопроцессорной системой, но с общей памятью: в нем есть множество процессорных ядер, которые можно рассматривать как отдельные процессоры. Да и многоядерных CPU в узлах как правило более одного (сейчас обычно по два). С такими системами мы разбирались в прошлом разделе. Там мы узнали, что если на узле два CPU и более, то это уже NUMA система: у каждого CPU свой контроллер памяти, а общий доступ к ней обеспечивается высокопроизводительной шиной, проложенной между CPU сокетами. Диаграмма такой системы показана на картинке ниже по тексту.

С точки зрения производительности вычислений, к характеристикам узлов, таким как пиковая производительность процессоров, пропускная способность памяти (сейчас порядка сотни ГБ/с на процессор), латентность доступа к памяти (сейчас порядка нескольких десятков нс), добавляются еще:

- **пропускная способность сети** (сейчас широко используется InfiniBand, OmniPath, обычно одна карточка дает по 5–10 ГБ/с;

- **латентность сети** (сейчас обычно на кластерах задержки порядка нескольких мкс).

Еще есть характеристики шины, которая соединяет процессор и сетевой адаптер (PCI-E 3.0 где-то 16 ГБ/с, а OmniPath вообще бывает интегрирован в процессор). Но пока не будем всё усложнять.




## 4.1 MPI: сборка и запуск параллельного приложения

В прошлой главе процессы разделялись на множественные потоки, работающие в общем адресном пространстве памяти. При распараллеливании с распределенной памятью процессы запускаются сразу во множественном количестве. Они не имеют общего адресного пространства памяти. Процессы могут обмениваться данными, передавая сообщения по сети. Обеспечивает запуск и взаимодействие между процессами стандарт MPI (Message Passing Interface). Это, конечно, не единственный, но наиболее распространенный стандарт для распределенных вычислений на кластерах.

В принципе программам ничего не мешает установить связь по сети самостоятельно. Но делать это через инфраструктуру некоторого посредника, который сам все организует, упорядочит, расставит обмены эффективным образом, значительно удобнее. Нам же удобнее отправить письмо или посылку почтой, ну или еще какой курьерской службой, хотя ничего не мешает взять и поехать с посланием самолично и вручить его адресату.

Предполагается, что читатель знаком с MPI. Но мы начнем с того, что освежим в памяти, как это все работает. Если читатель не знаком, то надо будет еще немножко почитать. Не пугайтесь, освоим только самые базовые минимально необходимые вещи, ни во что такое сложное не углубляясь.

Чтобы использовать MPI у себя на компьютере, мы должны установить одну из его реализаций. Под Windows можно установить MS MPI с сайта Microsoft (https://docs.microsoft.com/en-us/message-passing-interface/microsoft-mpi). На Linux – OpenMPI, MPICH, и другие. MPI состоит из библиотеки и системной части, в которой присутствует сервисный процесс (демон), обеспечивающий работу MPI на узле (компьютере), и программа-запускалка (обычно mpirun или mpiexec), которая порождает экземпляры процессов из заданного исполнимого файла и сцепляет их с MPI сервисом. Также с MPI на Linux обычно идут обертки для компиляторов, чтобы вручную не прописывать пути к заголовочным файлам

и библиотеке (всякие mpicxx, mpic++, mpiicpc, см. документацию по конкретной реализации). На Windows можно цепляться напрямую к библиотеке и инклудам (жарг. от include) из вижлов (жарг. от MS Visual Studio).

Итак, на своей системе надо установить MPI. Потом либо прописать компилятору подключение заголовочных файлов библиотеки MPI и прилинковать файл библиотеки, либо использовать обертки MPI. На системе коллективного пользования надо выбрать нужную реализацию MPI из установленных и собрать код согласно документации.

Запуск программы осуществляется через mpirun или mpiexec. Этой программе указывается, сколько нужно породить процессов приложения, из какого исполнимого файла это приложение брать и какие ему передать аргументы командной строки. Например, команда

```
mpiexec -n 8 ./test.px 1 2 3
```

запустит 8 процессов программы test.px с аргументами 1 2 3. Для запуска на системе общего пользования обычно надо использовать команды системы очередей, которые описаны в документации по конкретной системе.

Когда запускается параллельная программа, сразу порождается на одном или множественных узлах (то есть компьютерах) множество экземпляров программы в указанном при запуске количестве. А что с этим фактом делать в самой программе? Для начала надо выполнить инициализацию и узнать, что происходит – сколько экземпляров запущено и какой номер данного процесса в этой группе. Рассмотрим на примере.

```
int main(int argc, char **argv){
    int mpi_res;
    mpi_res = MPI_Init(&argc, &argv); // первым делом подключаемся к MPI
    if(mpi_res!= MPI_SUCCESS) // проверяем код возврата на предмет ошибок
        crash("MPI_Init failed (code %d)\n", mpi_res); // обрабатываем ошибку

    int NumProc;
    mpi_res = MPI_Comm_size(MPI_COMM_WORLD,&NumProc); // узнаем число процессов
    if(mpi_res!= MPI_SUCCESS)
        crash("MPI_Comm_size failed (code %d)\n", mpi_res);

    int MyID;
    mpi_res = MPI_Comm_rank(MPI_COMM_WORLD,&MyID);  // узнаем номер данного процесса
    if(mpi_res!= MPI_SUCCESS)
        crash("MPI_Comm_rank failed (code %d)\n", mpi_res);
```

Ну вот, мы получили, по сути, все необходимые данные для работы параллельного режима – число процессов и номер данного процесса (нумерация с нуля, естественно). Там еще есть функция crash, это просто какой-то наш обработчик авоста в программе, не имеющий отношения к MPI. Можно, конечно, писать printf и exit. Но делать просто exit некорректно, другие MPI процессы могут зависнуть, так и не узнав о гибели товарища. Перед выходом лучше делать MPI_Abort. Вот пример подобной завершалки в случае ошибки:

```
#include <stdarg.h> // vfprintf

// оборачиваем в exit, чтобы компилятор/аналайзер был в курсе, что это точка выхода
#define crash(...) exit(Crash( __VA_ARGS__))

int Crash(const char *fmt, ...){ // обработчик авоста с выдачей сообщения
    va_list ap;
    fprintf(stderr,"\nEpic fail: \n");
```

```
    va_start(ap,fmt);
    vfprintf(stderr,fmt,ap); // печатаем какое-то сообщение пользователю
    va_end(ap);

    fprintf(stderr,"\n");
    fflush(stderr);

    MPI_Abort(MPI_COMM_WORLD,-1); // культурно сворачиваем MPI
    return 0;
}
```

Ну и печатать в стандартный поток теперь надо вдумчиво. Просто printf будет печататься с кучи процессов и перемешиваться в логе в произвольном порядке. Выдачу надо или убирать в лог-файлы по каждому процессу, или оборачивать печать условием, что, например, номер процесса равен нулю. Тогда спамить в общий лог будет кто-то один. Печать в параллельный лог тоже можно переложить по аналогии через `vfprintf`.

MPI установили или подключили уже установленный на системе, сделали инициализацию, собрали программу, напечатали "hello world" с каждого процесса, распечатали их номера, убедились, что они разные и их число соответствует задуманному. Вроде все работает. Теперь разберемся, как процессы будут между собой взаимодействовать.

## 4.2 Обмен данными

Что представляет собой обмен данными, и какие виды обменов бывают? Пусть у одного процесса есть что отправить. Это какие-то данные какого-то размера, лежащие в памяти по какому-то адресу – исходящее сообщение, так сказать, буфер на отправку. Пусть также имеется процесс, который желает эти данные получить. У него есть буфер, куда он планирует получить это сообщение. Первый процесс – отправитель, второй – получатель.

Обмены бывают **точка-точка** и **групповые**. Если у сообщения один отправитель и один получатель – это обмен **точка-точка**, aka point-to-point, peer-to-peer, **p2p**. Если в обмене участвуют более двух процессов, это уже **групповой** обмен данными. Время на один p2p обмен данными будет (да, как и в случае с памятью): $T \geq L + S/B$, где L – латентность сети, S – объем данных, B – bandwidth, то есть пропускная способность сети.

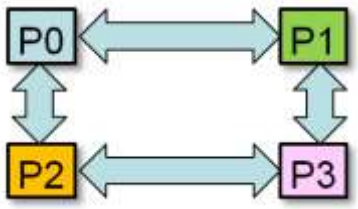
Обмены бывают **блокирующие** и **неблокирующие.** Это также может называться синхронные и асинхронные обмены, соответственно. Блокирующий обмен – это когда процесс вызывает функцию обмена данными (отправки, приема), которая уходит в себя и возвращается только после завершения обмена. Неблокирующий обмен – это когда вызов функции обмена представляет собой всего лишь заявку на отправку или получение данных. Заявка передается в MPI, и функция, приняв к сведению запрос, прямо сразу быстро возвращает управление процессу. Для того чтобы узнать, что обмен завершился, то есть отследить статус посылки, надо вызывать другие функции. Есть функции, которые просто проверяют статус заявки, есть функции, ожидающие завершения обмена по данной заявке. Неблокирующий обмен позволяет процессу заниматься чем-то полезным, пока в фоновом режиме MPI осуществляет обмен данными.

Обмены бывают **двусторонние** и **односторонние**. В двусторонних обменах участвуют два процесса – отправитель и получатель, как было описано выше. Односторонние обмены – это прямой доступ в память, aka RMA (Remote Memory Access), или даже RDMA (D – Direct), если таковой поддерживает матчасть. В односторонних обменах (one-sided communication) один процесс заранее выделяет буфер, aka окно, и говорит его адрес другому процессу. Другой

процесс может передавать в это окно данные или забирать их оттуда без участия первого процесса, не отвлекая того от работы. То есть обмен данными выполняется только одним участником. Если сетевая карта поддерживает режим RDMA (сейчас InfiniBand – поддерживает, OmniPath – нет), то передача данных может происходить без участия процессора на стороне получателя. Сетевая карта запишет данные напрямую в память по нужному адресу. Односторонние обмены мы тут не будем использовать, просто примем к сведению, что такое бывает.

### 4.2.1 Обмены p2p

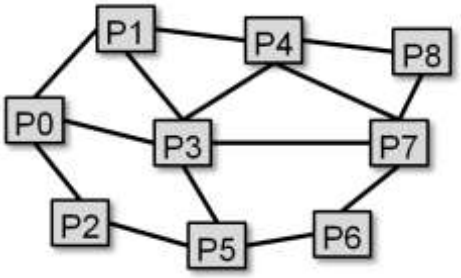
Мы еще не разбирались, что такое декомпозиция. Когда разберемся, станет понятнее. А пока пусть у нас есть 4 процесса, которые хотят обменяться данными, как показано стрелочками:



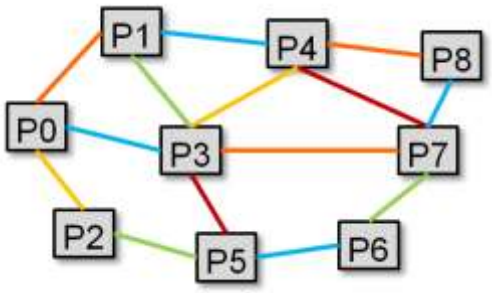

Процесс 0 обменивается чем-то с 1 и 2. Обменивается, значит, что-то отправляет и что-то принимает. Процесс 1 с 0 и 3, процесс 2 с 0 и 3, процесс 3 с 1 и 2. Как им это выполнить обменами точка-точка? Пусть мы используем простые блокирующие функции отправки и получения сообщений (MPI_Send, MPI_Recv). Это будет примерно так, попарно в 4 такта:


Почему получилось 4 такта? У каждого процесса по 2 соседа, то есть по 2 процесса, с которыми он обменивается данными. Каждый обмен состоит из отправки и получения, то есть передачи в одну и в другую сторону. Итого 4.

А если у нас процессы взаимодействуют не пойми как? Например, вот так:



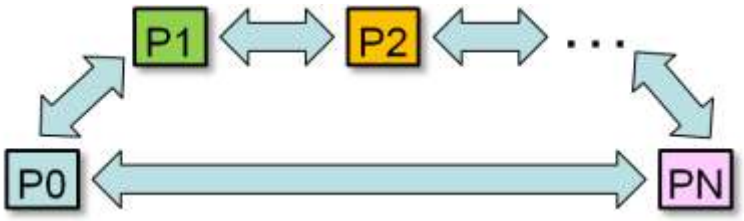

Как тут передать данные p2p обменами? Эта картинка подозрительно похожа на граф. На граф связанности процессов. Вершинам соответствуют процессы, а ребрам – необходимость передачи данных между ними. Ну с графами мы уже наелись в предыдущем разделе. Сразу догадываемся, что надо покрасить ребра и обмениваться по цветам




Сначала обмен по всем ребрам одного цвета, потом другого, потом следующего. Цветов получится не меньше, чем максимальная степень вершины графа (то есть число ребер, торчащих из одной вершины). Тактов блокирующих обменов, соответственно, вдвое больше – отправка и получение. Но неужели нам надо для организации обмена данными строить граф, его красить, городить потактовые обмены? Нет. Мы будем использовать неблокирующие обмены (MPI_Isend, MPI_Irecv), а библиотека MPI сама все построит, покрасит и разберется за нас.

Про MPI_Send, MPI_Recv, MPI_Isend, MPI_Irecv, если не знаете или не помните, как оно работает, необходимо прочитать в документации по MPI.

Рассмотрим простенький пример. Пусть процессам надо обменяться данными по кольцу. То есть обменяться чем-то с процессами, у которых номер на единицу больше и на единицу меньше. Последний процесс обменивается с нулевым процессом.




Пусть они обмениваются просто одним числом. Рассмотрим пример, как это сделать блокирующими обменами. Попробуем его полностью понять ~~и простить~~.

```
double send = MyID + 1000.0; // число, которое будем отправлять соседям
double recvl=0, recvr=0; // сюда будем получать значения от соседа "слева" и "справа"

const int NbL = (MyID>0 ? MyID-1 : NumProc-1); // номер соседа слева
const int NbR = (MyID<NumProc-1 ? MyID+1 : 0); // номер соседа справа

MPI_Status s;
MPI_Comm MCW = MPI_COMM_WORLD; // default communicator

// блокирующий обмен: четные сначала отправляют, потом получают, нечетные - наоборот
if(MyID%2 == 0){ // четные номера
    if(MyID==0 && NumProc%2){ // но есть нюанс с нулевым процессом
        mpi_res = MPI_Recv(&recvl, 1, MPI_DOUBLE, NbL, 10, MCW, &s); // Step 0
        // для краткости тут в тексте забьем на обработку ошибок, но она должна быть!
        mpi_res = MPI_Send(&send,  1, MPI_DOUBLE, NbL, 11, MCW); // Step 1
    }
    mpi_res = MPI_Send(&send,  1, MPI_DOUBLE, NbR, 10, MCW); // Step 0/2
    mpi_res = MPI_Recv(&recvr, 1, MPI_DOUBLE, NbR, 11, MCW, &s); // Step 1/3
    if(!(MyID==0 && NumProc%2)){
         mpi_res = MPI_Send(&send,  1, MPI_DOUBLE, NbL, 12, MCW); // Step 2
         mpi_res = MPI_Recv(&recvl, 1, MPI_DOUBLE, NbL, 13, MCW, &s); // Step 3
    }
}
```

```
else{ // нечетные номера
    mpi_res = MPI_Recv(&recvl, 1, MPI_DOUBLE, NbL, 10, MCW, &s); // Step 0
    mpi_res = MPI_Send(&send,  1, MPI_DOUBLE, NbL, 11, MCW); // Step 1
    mpi_res = MPI_Recv(&recvr, 1, MPI_DOUBLE, NbR, 12, MCW, &s); // Step 2
    mpi_res = MPI_Send(&send,  1, MPI_DOUBLE, NbR, 13, MCW); // Step 3
}
```

Получилось 4 такта, но как-то сложно и мутно. А что будет, если мы перепутаем порядок сендов и рисивов? Будет – deadlock до скончания веков.

Такой же обмен можно реализовать и проще:

```
mpi_res=MPI_Sendrecv(&send, 1, MPI_DOUBLE, NbR, 20, &recvl, 1, MPI_DOUBLE, NbL, 20, MCW,&s);
mpi_res=MPI_Sendrecv(&send, 1, MPI_DOUBLE, NbL, 21, &recvr, 1, MPI_DOUBLE, NbR, 21, MCW,&s);
```

А что, так можно было!? Вот поэтому нужно учить ~~матчасть~~ возможности API.

Ну и рассмотрим третий наиболее универсальный вариант – неблокирующие обмены.

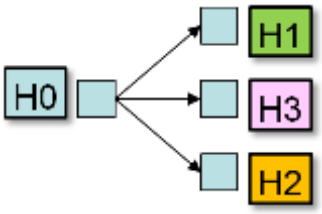
```
MPI_Request req[4];
MPI_Status  sts[4];

// запрашиваем отправку всем соседям
mpi_res = MPI_Isend(&send, 1, MPI_DOUBLE, NbL, 30, MCW, &req[0]);
mpi_res = MPI_Isend(&send, 1, MPI_DOUBLE, NbR, 31, MCW, &req[1]);

// запрашиваем получение от всех соседей
mpi_res = MPI_Irecv(&recvr, 1, MPI_DOUBLE, NbR, 30, MCW, &req[2]);
mpi_res = MPI_Irecv(&recvl, 1, MPI_DOUBLE, NbL, 31, MCW, &req[3]);

// ждем завершение всех обменов
mpi_res = MPI_Waitall(4, req, sts);
```

Этот вид обменов хорошо подходит самому общему случаю с непонятным графом связей между процессами. На этом с p2p вроде бы разобрались.

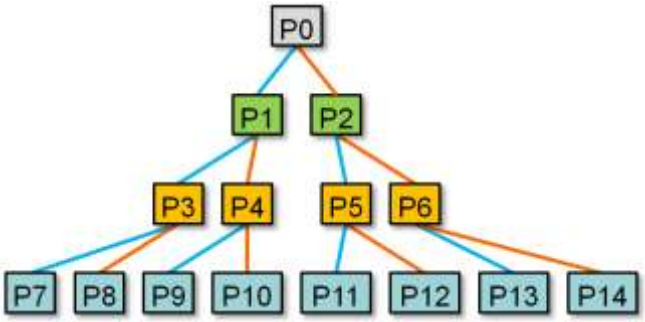
Да, а что это за числа непонятные в аргументах? Какие-то 30, 31 в коде выше. Это тэги, то есть идентификаторы обмена, которые тут выставлены какие-то с потолка, лишь бы разные. Тэги нужны, чтобы обмены не перепутались. Если мы делаем с одним соседом сразу несколько обменов, сообщения могут прийти в разном порядке, возникнет хаос и неразбериха. Поэтому тэги входящих сообщений сопоставляются тэгам с принимающей стороны. Каждому send – свой receive. Если у нас одновременно только один обмен происходит, то можно не заморачиваться и ставить всем сообщениям одинаковое число, например, 1. В этом примере можно так и сделать, все должно работать корректно.

### 4.2.2 Групповые обмены

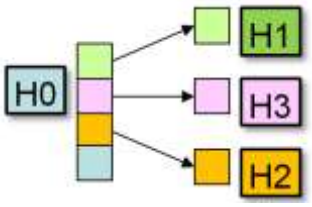
Тут бывают разные обмены. Самый простой – широковещательный обмен, aka broadcast, **MPI_Bcast**. Это когда у одного процесса есть одно сообщение всем процессам в группе. Все процессы в группе (определяемой коммуникатором) должны вызвать MPI_Bcast. Со всеми остальными групповыми обменами то же самое – все в коммуникаторе должны вызвать функцию, но роли у процессов будут зависеть от обстоятельств. В случае Bcast один процесс является отправителем, все остальные (включая его самого) – получатели. Изобразить это можно как-то так:



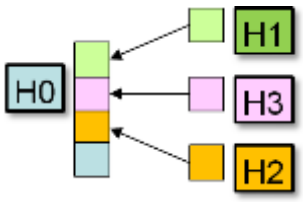

Групповой обмен дороже, чем точка-точка. Мы видим тут из одного процесса много стрелочек, в общем случае не три штуки, а P-1, где P – число процессов в группе. Но выше мы с графом связей процессов разбирались, и выяснили, что количество тактов обменов не получается меньше, чем количество этих стрелочек. Значит, один групповой обмен кушает O(P) тактов? То есть только латентность съест (P-1) × L? Не все так плохо. Сообщение-то всем одинаковое, обмен будет делаться по какому-нибудь двоичному дереву за $O(\log_2(P))$ тактов. Например, пусть у нас 15 процессов. Вот картинка, как может выполняться бродкаст. Получилось не 14 тактов, а 3 уровня по 2 адресата = 6 тактов. Так-то лучше.




Если у отправителя не одно одинаковое сообщение всем, а каждому из получателей своё, то получается Scatter, то есть разброс ~~на вентилятор~~. Выполняется функцией **MPI_Scatter**.

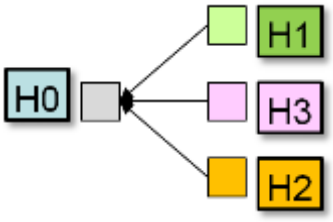



Обратной к операции Scatter является операция Gather – функция **MPI_Gather**. Тут как бы вся группа сдает старосте свои данные.



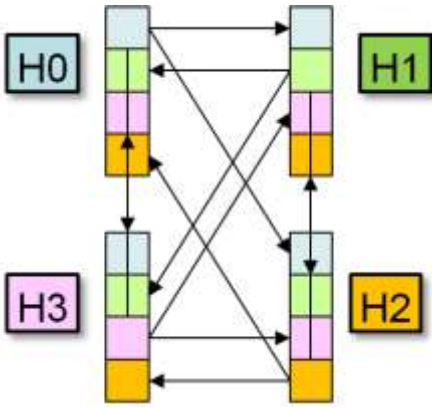

Scatter и Gather, очевидно, подороже, чем Bcast. Данных в P раз больше.

А если взять Gather, то есть все сдают одному процессу свои данные, но для этих данных выполняется редукция, то получится **MPI_Reduce**. Редукция – это когда каждый элемент передаваемых данных идет либо в поиск суммы, либо максимума, либо минимума по всей группе процессов. Это такой же недорогой обмен с константным объемом обмена данными и количеством p2p передач $O(\log_2(P))$, как и Bcast. Выглядит он так:

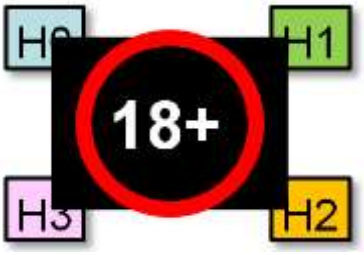



Если скрестить Reduce и Bcast – получится **MPI_Allreduce**. В Reduce результат оказывается только у одного в группе – несправедливость и дискриминация. А в Allreduce результат оказывается у всей группы. Эта штука получается раза в два дороже, чем Reduce.

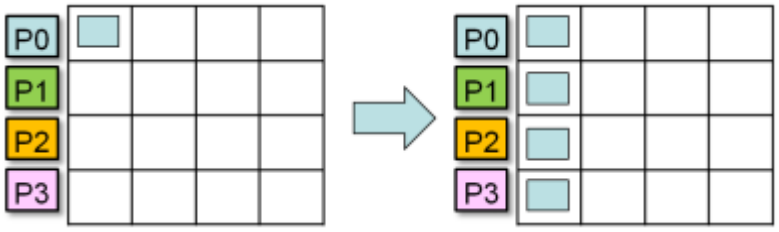
Тоже самое есть, например, и у Gather. Скрестим Gather и Bcast, получится **MPI_Allgather** – результат сборки со всех процессов группы раздается всей группе, и у всех получается одинаковый набор данных. Это уже дорогой обмен.



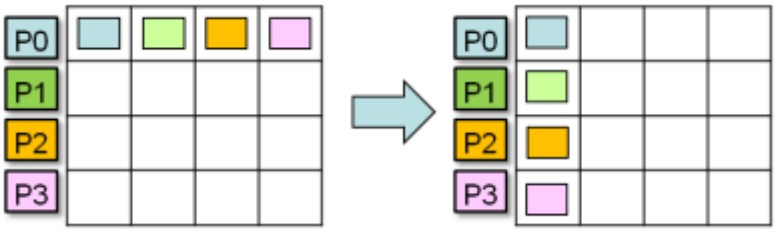

Ну самый дорогой обмен – это **MPI_Alltoall.** Это если взять Allgather, но у каждого процесса не одно сообщение всем, а каждому процессу – свое отдельное. Тогда будет Alltoall.



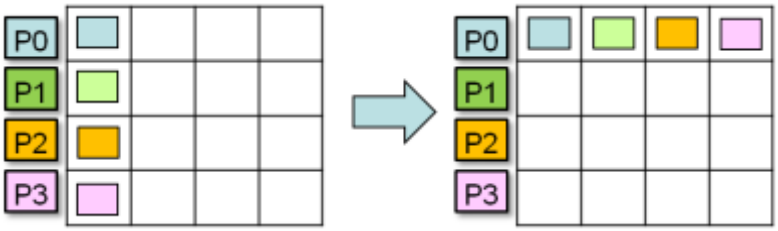

Можно обмены изобразить и по-другому, по аналогии с картинками в документации по MPI (https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf), в виде таблички размера P×P, где P – число процессов. Строки соответствуют процессу, а столбцы – тому, что находится у процесса. Например, Bcast. Была одна клеточка у одного процесса (слева), стало то же самое у всех процессов (справа):



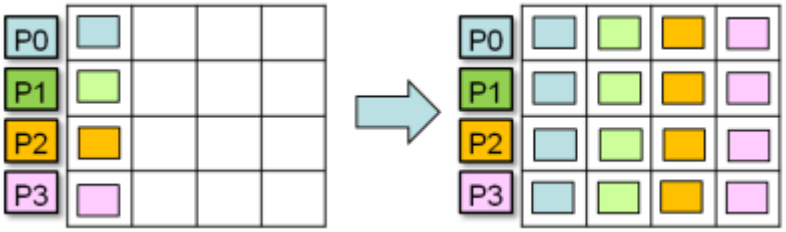
Scatter будет выглядеть так:



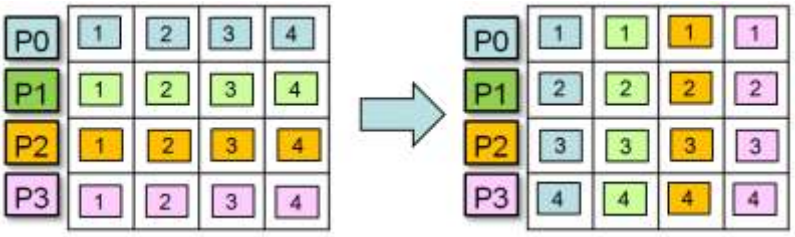

Gather:




Allgather:




А чтобы изобразить Alltoall, то есть полную перестановку, цветов недостаточно, допишем в клеточки цифры.



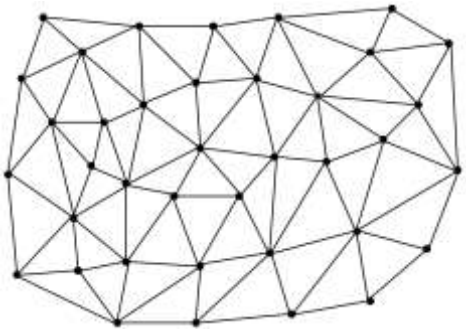

Это были обмены для константного размера сообщений, когда все клеточки одинаковые. Если размеры разные, есть версии обменов и для такого случая (MPI_Scatter**v**, MPI_Gather**v**, ... – надо добавить букву v), но мы не будем уж настолько углубляться.

Рассмотрим примеры, для чего вообще нужны эти обмены. Например, мы распределили элементы вектора между процессами. Главный процесс раздал всем свои части – Scatter. Все что-то посчитали, чтобы собрать результат – Gather. Чтобы результат оказался у всех – Allgather. Чтобы, например, всем найти скалярное произведение от распределенных векторов нужен Allreduce.

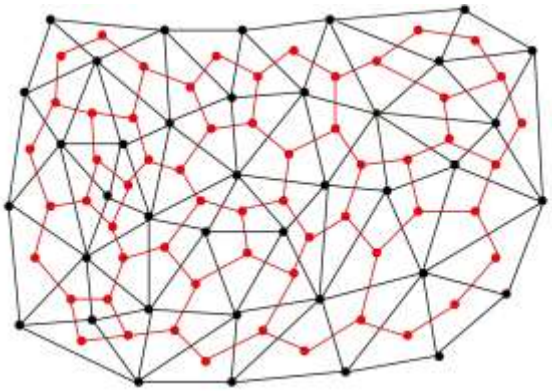
Есть еще важный групповой обменчик, называется барьер – **MPI_Barrier**. По сути, это вырожденный Allreduce – когда собирается ноль значений. Группа вызывает барьер, и пока все процессы не дойдут до этого барьера, никто из процессов в группе не пройдет дальше. Это называется точка синхронизации. А если кто-то забудет вызвать барьер – будет дэдлок (deadlock – взаимная блокировка).

## 4.3 Распределение вычислений сеточного метода между процессами

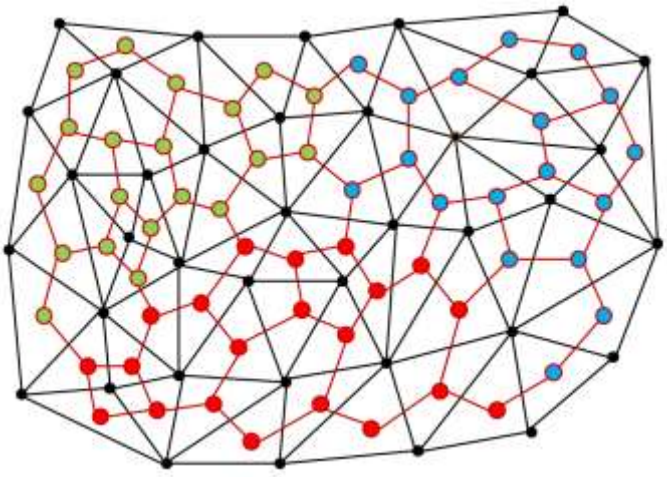
В прошлой главе мы уже разбирались с сеточным методом, было много картинок про сетки, графы, декомпозицию. Тут все примерно то же самое. Возьмем для наглядности такую же сетку из прошлой главы:



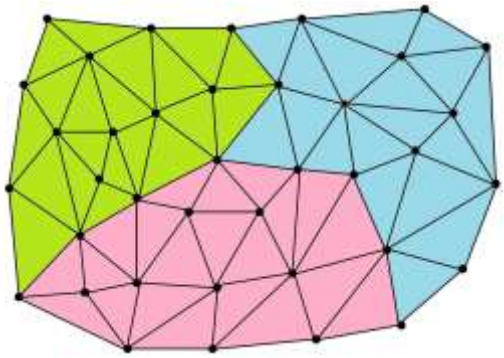
Пусть нам надо распределить работу между MPI процессами. Возьмем опять двойственный граф от этой сетки. Это наш граф связей или, можно сказать, смежности расчетных ячеек.



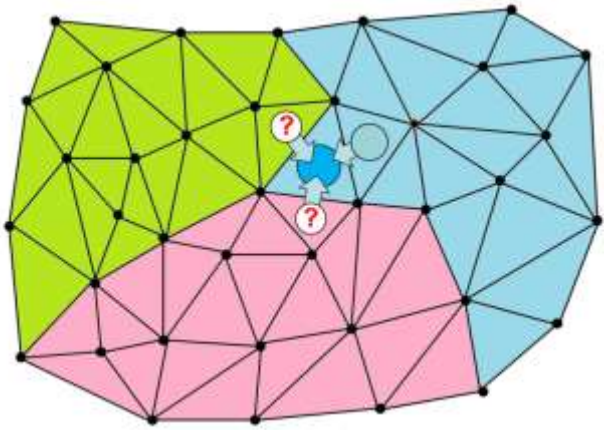
Нам надо распределить вершины графа по наборам, количество которых равно количеству MPI процессов, которое мы хотим использовать в параллельном режиме. То есть нам надо промаркировать вершины, указав каждой вершине, какой подобласти она принадлежит. Пусть у нас три MPI процесса. Для удобства на картинке будем отмечать вершины разных подобластей разными цветами. Вот как-то покрасили графу вершинки в три цвета. Из каких соображений покрасили – пока не важно.



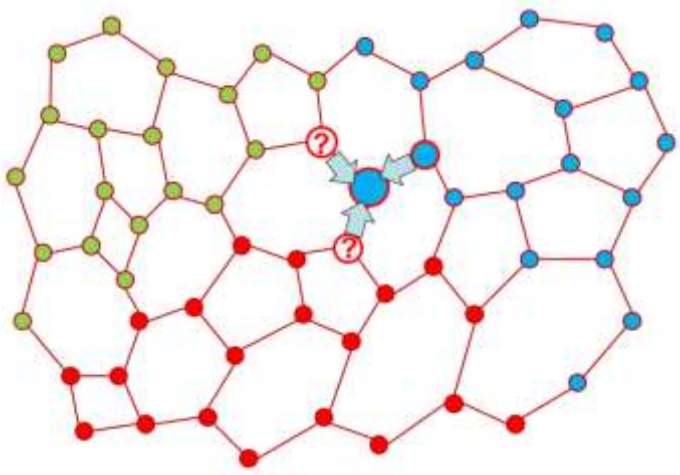
Эта маркировка дает нам распределение вершин по процессам, другими словами, ячеек по подобластям. Покрасим, соответственно, ячейки сетки в три цвета по принадлежности MPI процессам:



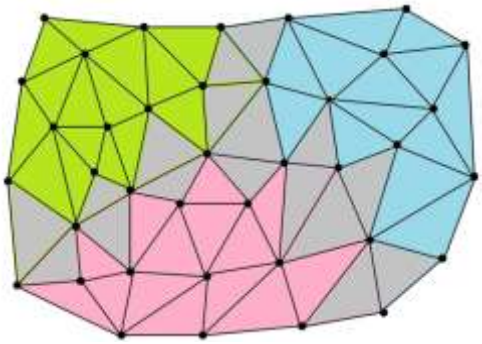
Одному процессу достанется зеленый кусок сетки, другому – розовый, третьему – голубой. Процессы будут вычислять значения сеточных функций в своих ячейках. Из прошлой главы мы помним, что там зачем-то надо считать потоки через грани ячеек. А чтобы их посчитать, надо иметь данные в ячейках по обе стороны от грани. А у нас сетка распределилась. И как посчитать потоки через грани, разделяющие ячейки разных цветов?




А если у нас не конечно-объемный метод, а еще какой-то? Может, конечно-разностный метод, или конечно-элементный, и мы вообще не считаем потоки через грани. Тогда будем иметь в виду формулировку в более общем виде. Есть граф. В вершинах графа есть какие-то данные. Чтобы найти новые данные в вершинах графа, для каждой вершины требуются данные из этой вершины и соседних с ней вершин. По сути, все ровно совершенно то же самое.

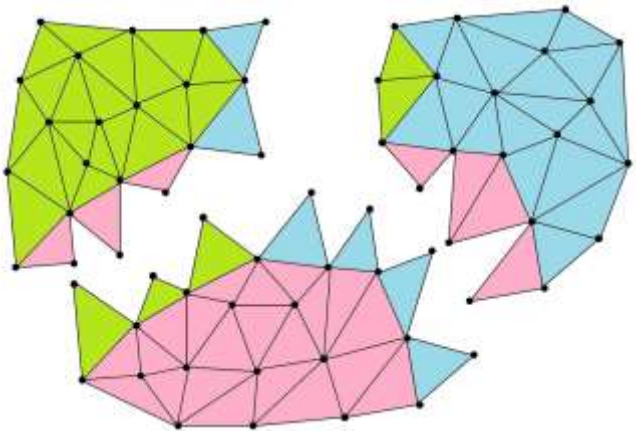


Но вернемся обратно к нашей сеточке, с ней нагляднее. Надо посчитать потоки через грани, разделяющие ячейки разных цветов. Вот тут и понадобятся обмены по интерфейсной зоне. Интерфейсная зона состоит из ячеек, смежных с ячейками из другой подобласти. В данном случае ячейка будет интерфейсной, если она имеет общую грань с ячейкой другого цвета. Выделим интерфейсную зону серым цветом.

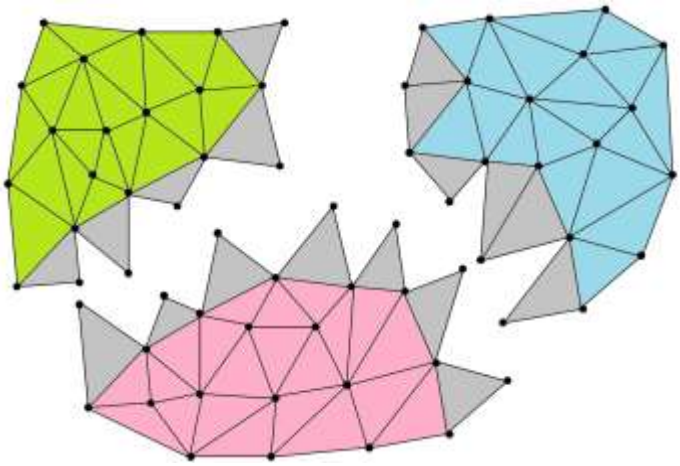


По этим серым ячейкам процессы должны обмениваться данными. Данные из своих ячеек они должны отправить, данные из чужих ячеек, смежных с их ячейками – получить.

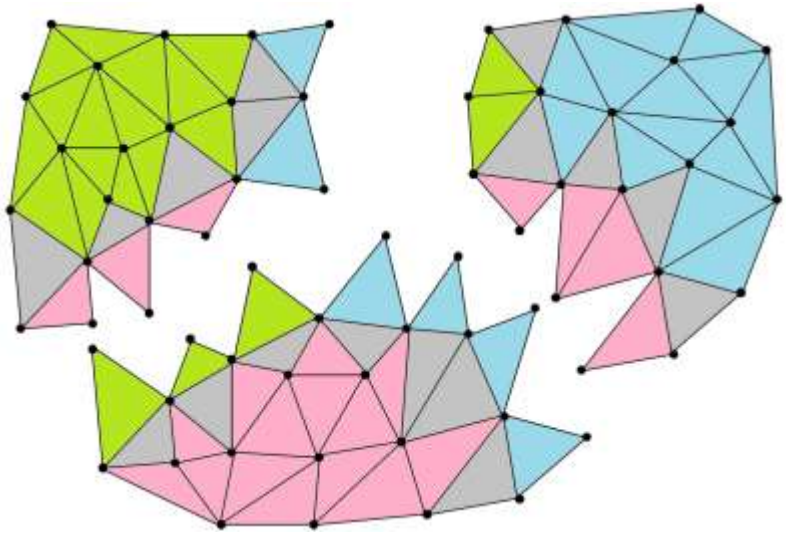
Растащим подобласти процессов для наглядности, чтоб было понятно, что части сетки теперь живут отдельно:



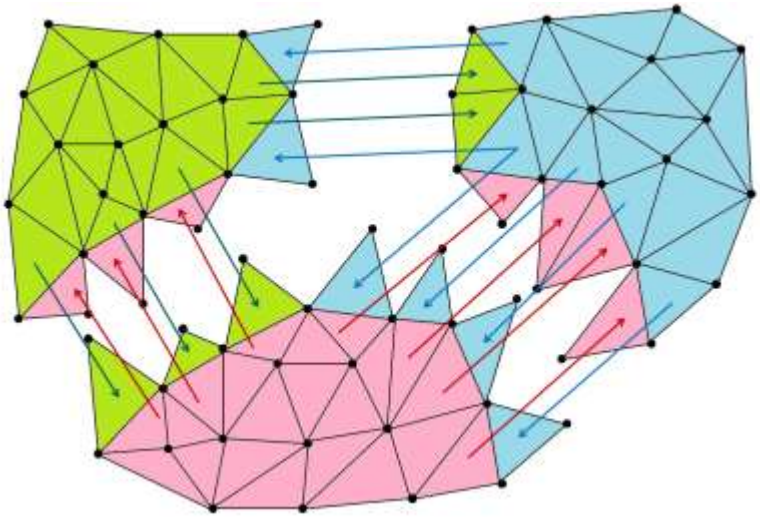
В каждой части оказались ячейки разных цветов. Все правильно, мы растащили собственные ячейки процессов, и приляпали к ним чужие ячейки, которые понадобятся для вычислений в собственных ячейках, лежащих на границе между подобластями. Эти чужие ячейки, добавленные к подобластям процессов из других подобластей, будем называть **гало** (halo). Вот они тут на картинке ниже все отмечены для наглядности серым цветом:



Гало-ячейки процессы обычно не обсчитывают, они запрашивают данные от соседей. А те собственные ячейки процессов, которые граничат с чужими ячейками, назовем **интерфейсными** ячейками. Эти ячейки мы должны соседям отправить. Вот они все отмечены для наглядности серым цветом:



Для окончательной наглядности отметим стрелочками обмен данными, что куда передается:



Чтобы актуализировать данные в гало, надо сделать обмен. Эта операция называется – обновить гало, aka halo update. Этот обмен можно организовать пересылками p2p следующим образом. Когда процессу надо обновить данные в гало, чтобы что-то посчитать, он для каждого соседа сформирует сообщение на отправку (соседи – процессы, имеющие хотя бы одну общую грань между своими ячейками). Процесс соберет все необходимые данные из своих ячеек, смежных с ячейками этого соседа, положит их в буфер на отправку этому соседу. Эту процедуру процесс проделает для всех своих соседей.

Затем процесс инициирует отправку данных всем соседям, обратившись к неблокирующей p2p функции, например, MPI_Isend.

Затем процесс подготовит аналогичные буфера на прием данных в гало от каждого соседа и вызовет неблокирующую функцию приема данных – MPI_Irecv.

Затем процесс должен вызвать функцию, ожидающую завершение обмена данными: сразу для всех send и всех recv вызывается MPI_Waitall.

Когда обмен завершился, процесс раскладывает данные из буферов входящих сообщений в гало ячейки. Теперь значения в гало актуальны, можно заняться вычислениями. Далее еще будем разбираться подробнее, как может быть устроен такой обмен данными. Но сначала надо понять, как устроен граф, по которому будут организовываться обмены.

### 4.3.1 Рациональная декомпозиция сетки, точнее, её графа

Декомпозиция – распределение работы между параллельными процессами. Нам надо поделить сетку на части, и раздать эти части процессам. Чтобы это сделать, нам нужен граф. Да и вообще, мы же программисты, нам проще думать про граф, чем про какую-то сетку. Сформулируем задачу в терминах графа.

Имеется граф с весами в вершинах. Вершины графа соответствуют расчетным ячейкам сетки, ребра графа соответствуют связям ячеек в сетке. Например, между двумя вершинами графа есть ребро, если между соответствующими ячейками сетки есть общая грань. Веса в вершинах – вещественные числа, описывающие вычислительную стоимость обработки ячеек. Если все ячейки одинаковые, то веса не нужны. А если, например, в смешанной сетке из разных видов элементов гексаэдральную ячейку посчитать дороже, чем тетраэдральную, то используем веса.

Задача декомпозиции сетки на P подобластей переходит в задачу распределения вершин графа по P наборам, где P – число параллельных процессов, которое мы хотим использовать в расчете. То есть каждой вершине надо присвоить номер подобласти, к которой она принадлежит. Это надо сделать так, чтобы:

1) дисбаланс был минимальным;

2) объем обмена данными был минимальным;

3) максимальное количество соседей было минимальным.

Разберемся с этими пунктами подробнее. Пусть у нас в сетке N ячеек. Тогда в нашем графе те же N вершин. Мы распределили вершины графа по $P$ подобластям. Раз мы на C++, то пусть нумерация будет с нуля. Пусть $w_i, i = 0, \dots, N-1$ – это веса вершин (w – weight). Пусть $V_p$ – множество вершин графа $p$-й подобласти, $p = 0, \dots, P-1$ (V – vertex).

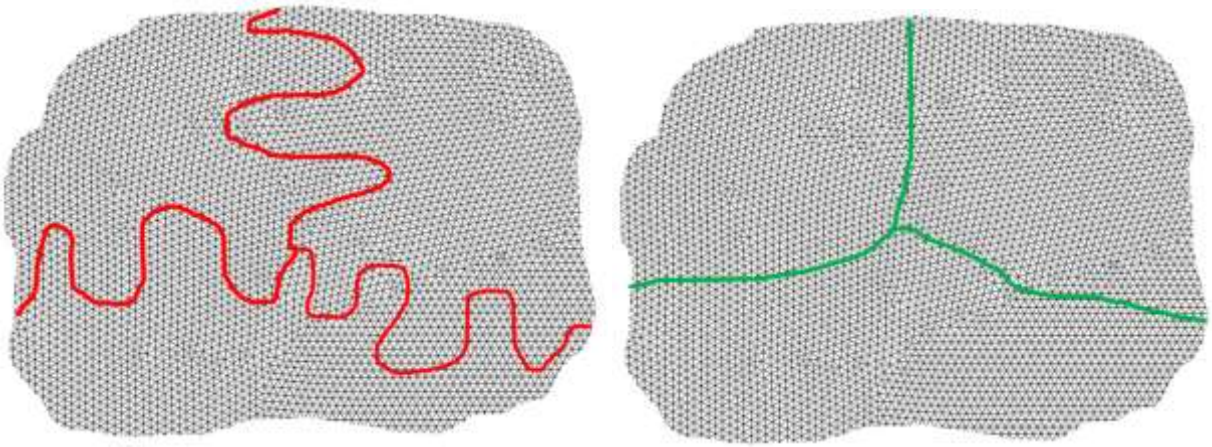
Что значит минимальный дисбаланс? Посчитаем для каждой подобласти суммарный вес вершин: $W_p = \sum_{i \in V_p} w_i$. Найдем среди этих суммарных весов максимальное значение $W_{MAX} = \max\limits_{p=0,\dots,P-1} W_p$. Найдем среднее значение суммарного веса по всем подобластям – поделим суммарный вес графа на количество подобластей, то есть $W_{AVG} = \frac{1}{P}\sum_{p=0}^{P-1} W_p$. Дисбаланс следует из отношения максимального суммарного веса подобласти к среднему весу. Если будем измерять дисбаланс в процентах, то $D = (W_{MAX}/W_{AVG} - 1) \times 100\%$. Если дисбаланс 100%, значит, кому-то досталось вдвое больше работы, чем в среднем. А это нехорошо. Итак, первый критерий – минимизировать $D$.

Дальше про edge cut. Что значит объем обмена данными? Объем обмена данными соответствует количеству "разрезанных" ребер, то есть интерфейсных ребер, у которых вершины оказались в разных подобластях. Обозначим всё множество разрезанных ребер $E_I$ ($E$ от слова Edges, $I$ – от слова Interface). Добавим еще верхний индекс, чтобы обозначить множество интерфейсных ребер $E_I^p$, у которых одна из вершин принадлежит $p$-й подобласти. При декомпозиции можно по-разному действовать, можно минимизировать суммарный обмен

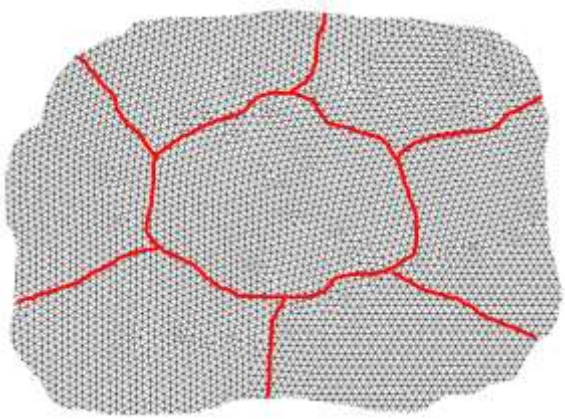
данными, то есть количество ребер в $E_I$, а можно, что более логично, минимизировать максимальный обмен по всем подобластям, то есть минимизировать $\max\limits_{p=0,\dots,P-1} E_I^p$.

Третий пункт – он уже второстепенный. На нем, по сути, минимизируется количество тактов p2p обменов. Соседи $p$-й подобласти – это те подобласти, вершины которых фигурируют в ребрах $E_I^p$, помимо самой подобласти $p$. Чем меньше соседей – тем лучше, меньше будет потерь на латентность сети.

Посмотрим на это с геометрической точки зрения. Пусть имеется расчетная область, покрытая примерно равномерной сеткой. Посмотрим визуально, что такое хорошо, и что такое плохо. Поделим сеточку на 3 кусочка. Слева – плохо, справа – лучше. А почему плохо? Правильно. Слева – явно излишний объем обмена данными.

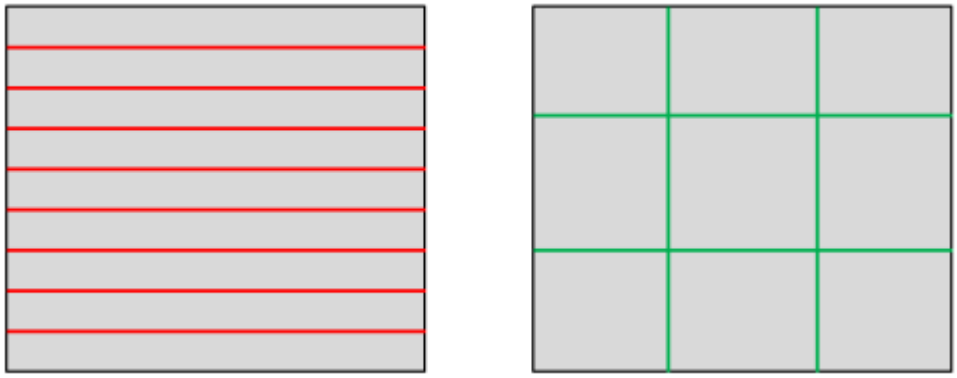


А вот тут чем плохо? Вроде все ровненько порезано.

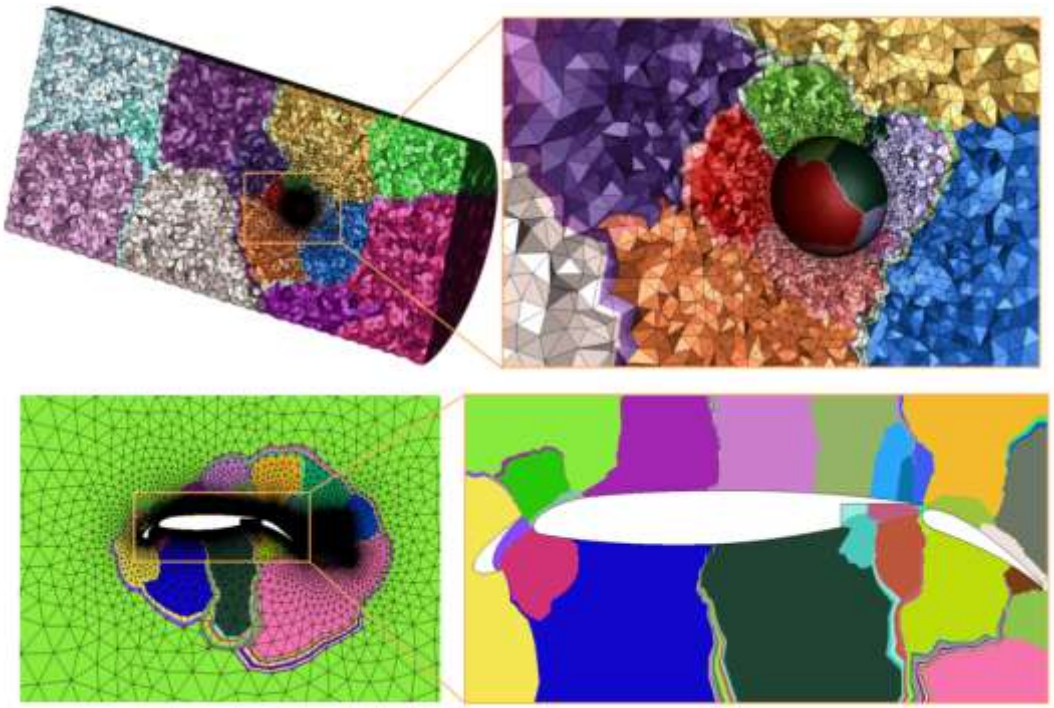


Плохо – слишком много соседей у одного процесса.

Или вот, например, на картинке ниже. Слева – плохо. Справа – лучше. Пусть это квадратик N×N ячеек. Слева максимальный объем обмена у одной подобласти, то есть периметр "забора" к соседям у одного участка, вышел 2N. Справа границы у самого несчастного домена посередине – 4N/3. Но ведь слева всего по 2 соседа, а справа – до 4 соседей. Разве это хорошо? Да, поскольку главный критерий – объем обмена, а число соседей – вторично. Число соседей важно только при прочих равных, а тут неравный объем обмена.



Если мы, например, делим квадратик из N×N ячеек только по одному направлению полосками на P частей, максимально границы у доменов будут по 2N интерфейсных ячеек. Получается, что границы не уменьшаются при росте числа P, а площадь подобласти уменьшается обратно пропорционально P. Из-за этого параллельная эффективность быстро деградирует: вычислений становится меньше, а обменов столько же. А как лучше делить? Ну, например, пусть для простоты P – квадрат какого-то числа P = p×p. Если мы поделим нашу квадратную область не на полоски, а на квадратики, то максимально границы у доменов будут по 4N/p интерфейсных ячеек. То есть, границы уменьшаются с ростом числа подобластей, что так гораздо лучше. На реальных сетках декомпозиция выглядит примерно так:



За декомпозицию отвечают отдельные алгоритмы, которые не рассматриваются в этой главе. Обычно мы подцепляем какую-то внешнюю библиотеку, выдаем ей граф, и она нам его делит на нужное число частей. Про декомпозицию можно почитать, например, тут:

- Якобовский М. В. Введение в параллельные методы решения задач: Учебное пособие. — Издательство Московского университета Москва, 2013. — 328 с. Там глава 7.

  или еще более подробно тут:

  http://keldysh.ru/council/3/D00202403/golovchenko_diss.pdf

  Но чтобы передать библиотеке граф, его надо как-то программно оформить. А как?

### 4.3.2 Представление графа в программе

Пусть у нас есть сетка из N ячеек. Для нее мы заполняем граф, который описывает смежность ячеек. У графа N вершин и E ребер. Как мы будем его хранить в программе? Можно хранить граф в виде портрета разреженной матрицы смежности. Да, портрет матрицы – это множество ее отличных от нуля коэффициентов.

Матрица смежности графа, это матрица N×N, в которой ненулевые позиции соответствуют ребрам между вершинами графа. Коэффициент матрицы $a_{ij}$ в j-м столбце i-й строки равен 1, если между вершинами i и j графа есть ребро, иначе там ноль. Если такое ребро есть, то и $a_{ji}$ тоже неноль (*жарг.* от ненулевое значение). Запишем портрет матрицы, то есть описание, где в матрице ненулевые позиции, в построчено-разреженном формате CSR (Compressed Sparse Row). Для этого нам нужны 2 интовых массива:

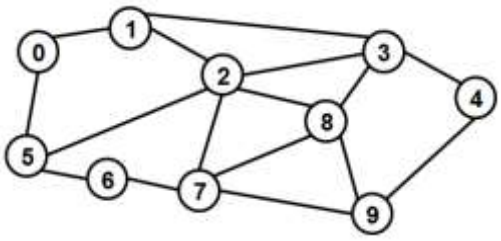
**IA** – размера N+1, aka rowptr, xadj. В нем для каждой строки храним позицию начала списка столбцов данной строки. Последний элемент в массиве, IA[N], хранит общее число ненулей в матрице.

**JA** – размера 2*E, aka colptr, adjncy, хранит номера столбцов по всем строкам подряд. Для определенности номера столбцов каждой строки упорядочены по возрастанию.

Как нам перебрать всех соседей i-й вершины? А вот так, например:

```
for(int col=IA[i]; col<IA[i+1]; ++col){
    j = JA[col]; // номер соседней вершины, т.е. соединенной с вершиной i ребром.
    ...
}
```

Возьмем для примера какой-нибудь граф. Построим для него матрицу смежности и запишем ее портрет. Вот такой граф и его матрица смежности:




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 7 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 8 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

Портрет этой матрицы:

```
int IA[11] = {0, 2, 5, 10, 14, 16, 19, 21, 25, 29, 32};
int JA[32] = {1,5, 0,2,3, 1,3,5,7,8, 1,2,4,8, 3,9, 0,2,6, 5,7, 2,6,8,9, 2,3,7,9, 4,7,8};
```

Пусть нам надо побить графа размера N на P частей. Побить – в смысле поделить. Речь не идет о насилии по признакам принадлежности к социальной группе.

Мы передаем в библиотеку декомпозиции размер N, массивы портрета IA, JA, а также массив весов вершин размера N. Еще мы передаем массив весов подобластей размера P. Сумма элементов этого массива равна 1. Если у нас все процессоры одинаковые, заполняем его

значениями 1.0/P. Если библиотека использует геометрический алгоритм, ей надо передать массив пространственных координат, сопоставленных вершинам графа. На выходе получим массив распределения вершин по подобластям размера N, в котором лежат значения от 0 до P–1.

Как работают такие библиотеки см., например, документацию по библиотеке Metis: http://glaros.dtc.umn.edu/gkhome/views/metis

Итак, мы поделили рациональным образом граф на части, тем самым мы распределили работу между параллельными процессами – для каждой расчетной ячейки определили, какому процессу, точнее, подобласти какого процесса она принадлежит. Этот процесс будет отвечать за обработку этой ячейки. Что же делать дальше? Дальше надо организовать MPI пересылки, чтобы процессы могли обмениваться данными по интерфейсной зоне, отсылать соседям данные по интерфейсным ячейкам и получать от них данные в гало ячейки.

### 4.3.3 Глобальная и локальная нумерация

Была у нас одна простая сетка, а может, и не сетка, а может не простая, а может какой-то еще набор каких-то объектов, для которого был граф, описывающий смежность этих объектов. У этого исходного набора объектов была нумерация. Назовем ее **глобальной** – **global**. Итак, глобально было $N_G$ объектов, пронумерованных от 0 до $N_G$–1 (приписали нижний индекс, буква G – Global). Потом мы поделили этот набор на поднаборы – подобласти. Мы сдали графа библиотеке, которая его побила. На выходе получили массив размера $N_G$, в котором для каждой вершины хранится номер подобласти, к которой она принадлежит. Как-нибудь обозначим этот массив, например, пусть он будет Part – от слова partition.

Каждая подобласть может пройтись по этому массиву и понять, какие вершины (и соответствующие вершинам объекты) ей принадлежат. Собрав все свои объекты в кучу, их можно пронумеровать. У каждой подобласти появилась своя собственная нумерация из собственных объектов – **own**. Это нумерация собственных объектов для процесса-владельца этой подобласти.

Была одна большая глобальная куча из $N_G$ штук чего-то. Кучу поделили на $P$ частей между процессами. Каждому из этих $P$ процессов досталась какая-то своя собственная кучка из $N_O^p$, штук этого чего-то (буква $O$ обозначает – Own), $p = 0, \dots, P-1$. Если сложить кучки, получится опять глобальная куча: $N_G = \sum_{p=0}^{P-1} N_O^p$. Всё так?

Так, да не так. Мы забыли гало. Посмотрим на это с точки зрения какого-то процесса $p$. Если в нашей кучке только наши объекты, откуда мы будем брать чужие объекты, которые нужны для вычислений по объектам нашей кучки? Объекты, которые нужны кому-то еще, кроме владельца (т.е. в графе из вершины, соответствующей этому объекту, торчит ребро в чужую вершину), размножим, что нам жалко, что ли? Добавим к каждой кучке гало-объекты, **halo**, т.е. чужие объекты, которые нужны для вычислений в собственных объектах. Пусть их у каждого процесса $N_H^p$ штук ($H$ – Halo). Получившиеся кучки с досыпанными в них гало назовем **локальными** – **local**. Локальный мир процесса – это свои объекты + гало:

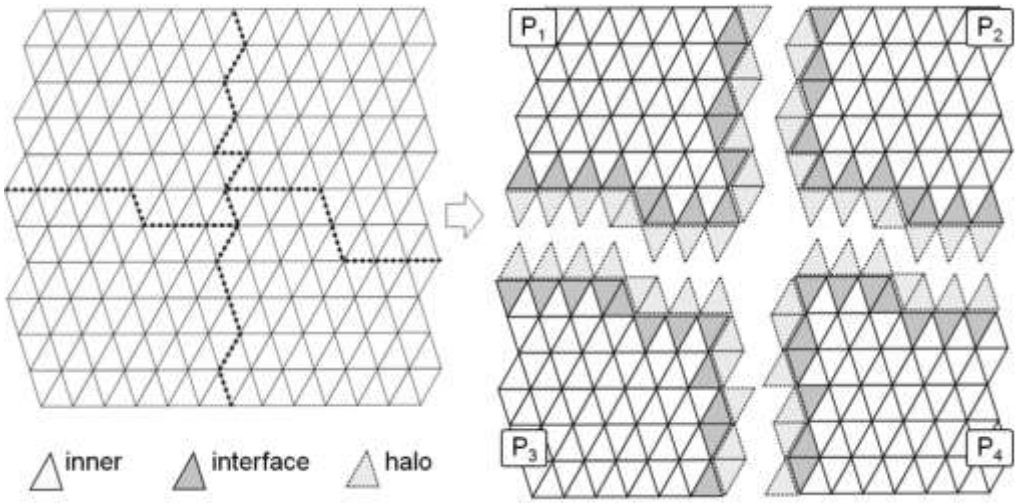
$$N_L^p = N_O^p + N_H^p.$$

С точки зрения одного из процессов, или, другими словами, одной из подобластей, получились такие наборы объектов:

- **Global** – всё имеющееся множество объектов (ячеек сетки);
- **Own** – собственные объекты данного процесса;

- **Halo** – объекты других процессов, т.е. из других подобластей, с которыми у собственных объектов имеются ~~порочные~~ связи;
- **Local** – локальный мир процесса, Local = Own ∪ Halo

Еще раз закрепим обозначения – внутренние, интерфейс, гало. Вот на картинке ниже сеточка, поделенная на 4 подобласти. Понятно, где там интерфейсные ячейки, где гало?




Для связи процесса с внешним миром нужно отображение из его локальной нумерации в глобальную и обратно. Такие отображения строятся для каждой подобласти, они будут у каждого процесса свои. С отображением из Local в Global все просто. Будем хранить его в массиве L2G размера $N_L^p$. По локальному номеру объекта i мы легко определим его глобальный номер, взяв L2G[i].

Отображение из глобальной нумерации в локальную обозначим G2L. Тут уже есть варианты, что использовать. Самый простой способ – для каждой подобласти возьмем массив размера $N_G$. В позициях, которые не входят в локальную подобласть данного процесса, поставим некорректное значение индекса -1, а в остальные позиции впишем соответствующие номера в локальной нумерации процесса. Для глобального номера i берем G2L[i] и узнаем номер объекта в локальной подобласти процесса. Если там оказалось отрицательное значение – объект не принадлежит локальной подобласти.

Если хочется сэкономить память, то альтернатива – ассоциативный контейнер. Взять, например, map, и память будет кушаться пропорционально $N_L^p$. Правда скорость доступа станет не константной, а $O(\log(N_L^p))$. Да и памяти он с большой вероятностью займет (к удивлению) в разы больше, чем забитый минусами глобальный массив. Можно и unordered_map взять, у которого константная скорость доступа. Но эта константная скорость, конечно, будет в разы (на порядки) больше, чем в массив, а памяти он съест еще больше, чем map (одна хэш таблица чего стоит).

А как заполнять эти отображения для данного процесса? Ну, например, возьмем массив Part и глобальную топологию графа связей. Наберем к себе ребер, в которых хотя бы они узел – наш. О том, что он наш, скажет Part. Пусть G2L будет самое простое, то есть массив. Проинициализируем G2L значением -1. Пройдем по всем ребрам, которые мы понабрали, и поставим номерам вершин этих ребер в G2L значения 1. Так мы промаркировали локальную подобласть для данного графа.

Прошлись по G2L, заменили значения 1 на их бегущий счетчик и всё:

```
for(int iG=0, iL=0; iG<NG; ++iG) if(G2L[iG] >= 0) G2L[iG] = iL++;
```

Еще надо заполнить обратное отображение. Например:

```
for(int iG=0, iL=0; iG<NG; ++iG) if(G2L[iG] >= 0) L2G[iL++] = iG;
```

Да, а зачем по отдельности, если можно сразу оба?

```
for(int iG=0, iL=0; iG<NG; ++iG) if(G2L[iG] >= 0){ G2L[iG] = iL; L2G[iL++] = iG; }
```

А если у нас откуда-то уже имеется список (в смысле массив, вектор) глобальных номеров вершин нашей подобласти, то L2G из него следует напрямую. Этот список и будет L2G. Перегнать его в G2L, если нужно, можно так же элементарно. Проинициализируем G2L значением -1, пройдемся по L2G и распихаем номера в G2L:

```
for(int iL=0; iL<NL; ++iL) G2L[L2G[iL]] = iL;
```

Да, мы же оставили локального графа в непотребном виде. Мы набрали ребер, теперь номера вершин подобласти, конечно же, надо перегнать в локальную нумерацию через G2L. Вот теперь получился культурный локальный граф.

Раз тут у нас все равно какая-то своя нумерация, то мы ее можем сделать какой угодно, точнее, какой нам удобно. Чтобы с локальной подобластью было удобно работать, переупорядочим вершины так: пусть сначала идут свои вершины, потом гало. То есть локальный номер любой гало вершины больше локального номера любой своей вершины.

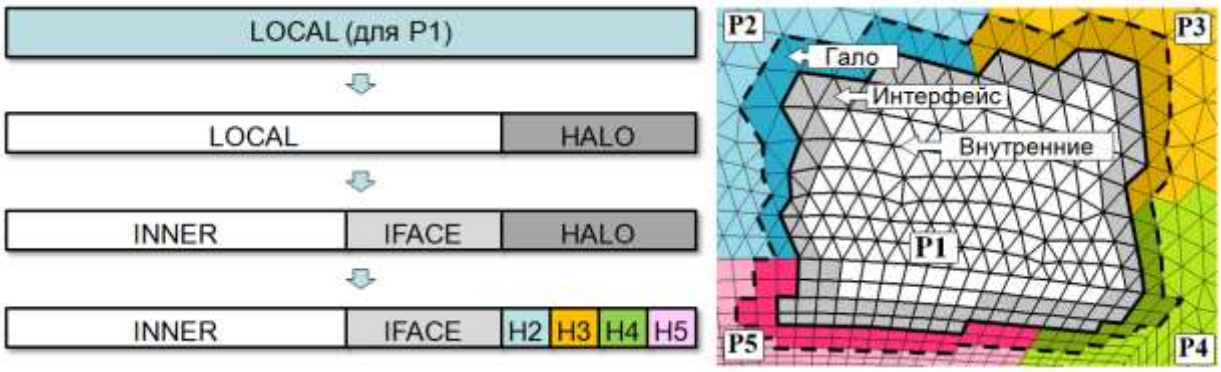
Чтобы стало еще удобнее работать, переупорядочим свои вершины так: сначала пусть будут внутренние вершины (т.е. не связанные ребром с чужими вершинами), потом пусть идет интерфейс (т.е. вершины, связанные ребром с гало вершиной). Теперь все стало удобно. Вершины с номерами от 0 до $N_I^p - 1$ в обменах не участвуют (буква I – Inner). Вершины с номерами от $N_I^p$ до $N_O^p - 1$ нам будет надо передавать соседям. Вершины с номерами $N_O^p$ до $N_L^p - 1$ надо будет получать от соседей. И поскольку мы находимся в локальном мире $p$-го процесса, зачем вообще нам этот верхний индекс $p$? Отбросим его для краткости. Получили в локальном мире следующие диапазоны номеров вершин:

$0, \ldots, N_I - 1$ – собственные внутренние;

$N_I, \ldots, N_O - 1$ – собственные интерфейсные;

$N_O, \ldots, N_L - 1$ – гало.

Для удобства гало упорядочим по соседям, чтобы все вершины каждого соседа шли подряд. Так будет удобнее и быстрее копировать данные из приемных буферов (мы же помним про unit stride access?). Получилось упорядочивание в локальной подобласти, как показано на картинке ниже. Там показан кусочек сетки с подобластью какого-то процесса с номером P1 и кусками подобластей от его соседей (нарисована именно сетка, а не её двойственный граф). Для разнообразия показана интерфейсная зона шириной в два уровня смежности по граням. Это означает, что в интерфейс попадают ячейки, имеющие общую грань с чужой ячейкой, и все ячейки, имеющие с этими ячейками общую грань.




## 4.4 Организация обмена данными

### 4.4.1 Построение схемы обменов

Назовем схемой обменов (чтобы просто как-то это явление называть) набор данных, описывающий, кто с кем и чем именно обменивается. Для каждого процесса (подобласти) надо получить список соседей, то есть номеров процессов, с которыми данный процесс обменивается по интерфейсной зоне. Для каждого соседа данного процесса надо составить список (не в смысле типа контейнера, а в общечеловеческом смысле) вершин графа/ячеек сетки, данные из которых надо этому соседу отправить и список вершин этого соседа, данные из которых надо получить в гало. Чтобы это сделать, надо пройтись по локальному графу, содержащему свои вершины и гало, и для каждого ребра, в котором одна вершина чужая, а одна своя, свою внести в список на отправку владельцу чужой вершины, а чужую вершину внести в список на прием от ее владельца.

Сделать это можно разными способами. Ниже приводится один из вариантов. Предполагается использование C++. Там саморастущие вектора. Если у вас C или Fortran, то все, по сути, абсолютно то же самое, только надо делать по два прохода – сначала посчитать, сколько, где, чего, куда, потом выделить массивы, потом еще раз пройтись и заполнить.

Начнем с простого варианта, а потом посмотрим, что можно улучшить. Чтобы все заполнить за один проход по графу, создаем два комплекта списков номеров вершин – на отправку и прием для вообще всех процессов в группе, которых пусть P штук. Ну пусть процессов даже 10 тысяч, а соседей всего 10. Что нам жалко 9990 лишних пустых векторов сделать? Ну и не обязательно векторов. В чем хранить эти списки на этапе построения – дело вкуса. Можно в векторах, а можно и в сетах, например. В сетах не надо будет потом удалять повторы. Для простоты пусть все на векторах. То есть на просто массивах, правда саморастущих (это для тех, кто на C и Fortran). Итак, сделали вектора SendToProcess и RecvFromProcess из P пустых векторов каждый – для списков номеров вершин на отправку и прием, соответственно.

Заполняем вектора номеров на отправку и прием. Идем по графу, видим, что наша собственная i-я вершина соединена с какой-то чужой j-й, а владельцем j-го номера является p-й процесс. Добавляем i в SendToProcess [p], а j в RecvFromProcess [p].

Получили вектора для соседей, которые лежат разрозненно посреди кучи пустых векторов. Преобразуем это безобразие к культурному виду, сделаем единые общие вектора на отправку и прием сразу по всем соседям.

Создаем пустой вектор для хранения номеров соседей, назовем его Neighbours.

Создаем вектор SendOffset, в котором для каждого соседа будем хранить позицию начала его списка в общем векторе на отправку, помещаем в него один элемент, равный нулю.

Создаем вектор RecvOffset, в котором для каждого соседа будем хранить позицию начала его списка в общем векторе на прием, помещаем в него один элемент, равный нулю.

Создаем пустые вектора Send и Recv для хранения общих списков по всем соседям.

Перебираем в цикле номер процесса в группе p от 0 до P-1. Если видим, что вектора SendToProcess[p], RecvFromProcess[p] для данного процесса не пустые, добавляем его номер в Neighbours.

Надо преобразовать содержимое SendToProcess[p], RecvFromProcess[p] от этого соседа. В том виде, в каком они сейчас, нам не подходит. Чтобы списки на отправку у отправителей были согласованы со списками на прием на стороне получателей, надо, чтобы вершины в них были упорядочены одинаковым образом. Чтобы этого достичь, надо отсортировать вектора SendToProcess[p] и RecvFromProcess[p] по возрастанию глобальных номеров. Глобальных! Сейчас там лежат локальные номера. Глобальные надо взять из отображения L2G.

Если надо, вытащим то, что лежит в SendToProcess[p] и RecvFromProcess[p], в удобное нам для сортировки место. Исходное содержимое можно выкидывать. Перегоняем номера в глобальные через L2G, сортируем, удаляя повторы (!), перегоняем обратно в локальные через G2L.

Если отображение из глобальных номеров в локальные, G2L, недоступно, то можно переложить эти вектора локальных номеров вершин в вектора пар глобальный-локальный номер. Ну или в блочный вектор, в котором номера идут парами – сначала глобальный номер, потом локальный. Глобальные номера заполняем по массиву L2G, который точно доступен. Сортируем это дело лексикографически, чтобы пары были упорядочены по глобальному номеру. Забираем оттуда локальные номера в нужном порядке. Исключая повторы! А то номера вершин туда могли попасть не по разу (если использовали сеты, то повторов не будет).

Эти упорядоченные нужным образом локальные номера без повторов, полученные из SendToProcess[p] и RecvFromProcess[p], добавляем в общие вектора Send и Recv, соответственно. Отмечаем смещения: добавляем в SendOffset размер Send, добавляем в RecvOffset размер Recv. Теперь мы знаем, в каком интервале лежат номера для приема и отправки данному соседу. Переходим к следующему процессу.

Итак, прошли по всем процессам, составили схему обмена данными:

- Neighbours содержит номера всех процессов-соседей, с которыми надо обмениваться;
- для каждого соседа k номер его процесса будет Neighbours[k];
- список вершин на отправку k-му соседу будет лежать в векторе Send в позициях от SendOffset[k] до SendOffset[k+1] – 1.
- список вершин на прием от k-го соседу будет лежать в векторе Recv в позициях от RecvOffset[k] до RecvOffset [k+1] – 1.

SendToProcess и RecvFromProcess более не нужны, выкидываем. То, что мы объединили списки по каждому соседу в общие списки по всем соседям, позволит формировать сообщения на отправку и разгребать входящие сообщения одним циклом по единому списку. Не нужно будет вить гнездо из цикла по соседям и вложенного цикла по его списку. Этот единый цикл можно хорошо ускорить OpenMP распараллеливанием.

Небольшие возможные улучшения. Если нас смущает делать SendToProcess[p] и RecvFromProcess[p] по размеру всей группы, то можно сначала определить соседей и отобразить множество всех процессов в количестве P штук в подмножество наших соседей в

количестве пусть их будет B штук, B≪P, по аналогии с G2L. Сделаем массив P2N (или какой-то другой контейнер), который для номера процесса *p* скажет его позицию в списке соседей Neighbours, а если *p* не наш сосед, то выдаст -1. Тогда у нас будут SendToNeighbour и RecvFromNeighbour размера B, а не P. Но пихать в них надо будет уже не по номеру владельца вершины, а конвертировать владельца в номер соседа через P2N.

### 4.4.2 Обмен данными

По полученной схеме обменов легко сделать обмен, используя MPI_Isend, MPI_Irecv, MPI_Waitall. Делаем буфера на прием, запрашиваем в них прием от соседей – MPI_Irecv. Заполняем каждому соседу буфер на отправку, берем номера из списка Send, по этим номерам берем какие нужно значения из нужных соседу вершин/ячеек и кладем в буфер. Рассылаем сообщения всем соседям – MPI_Isend. Ждем, когда все обмены завершатся – MPI_Waitall. Распаковываем сообщения по Recv списку со всех соседей.

Под каждое сообщение нам нужен буфер, чтобы туда положить данные на отправку каждому соседу, или принять туда входящее сообщение. С буферами поступим та же, как и со списками номеров вершин – выделим единые буфера на отправку и прием. Позицию куска для буфера каждого соседа будем определять по смещениям, которые у нас в SendOffset и RecvOffse (домножая, конечно, на размер пересылаемых данных в одной ячейке).

А как проверить, что обмен работает? Заполняем тестовый вектор (сеточную функцию) по вершинам так: для своих вершин пишем в вектор значения их глобальных номеров. В гало заполняем -1. Делаем обмен. Проверяем, что весь вектор получился заполнен, и значения в гало стали равны глобальникам (что проверяем по L2G).

В общем, ничего не понятно, списки, вектора, буфера. Хотя, что там может быть непонятно? Просто вектора с номерами, взял по списку, отправил, получил. Чтобы стало понятнее, попробуем освоить конкретный пример реализации. Заодно прокачаем навыки разгребания чужих кодов. Вот ниже приведена функция, которая умеет обмениваться данными по одному вектору/массиву/сеточной функции со значениями в вершинах графа/ячеек сетки.

```
template <typename VarType /* тип значений */>
void Update(VarType *V, // Входной массив значений в вершинах/ячейках, который надо обновить
            const tCommScheme &CS/*какая-то структура, описывающая схему обменов*/){

    const int B = CS.GetNumOfNeighbours(); // число соседей
    if(B==0) return; // нет соседей - нет проблем
    // tCommScheme - какая-то структура, замените ее на ваш вариант
    // приведем все к POD типам и неймингу, как было в тексте выше
    int *Send = CS.GetSendList(); // список ячеек на отправку по всем соседям
    int *Recv = CS.GetRecvList(); // список ячеек на прием по всем соседям
    int *SendOffset = CS.GetSendOffset(); // смещения списков по каждому соседу на отправку
    int *RecvOffset = CS.GetRecvOffset(); // смещения списков по каждому соседу на прием
    int *Neighbours = CS.GetListOfNeigbours(); // номера процессов соседей
    MPI_Comm MCW = CS.GetMyComm(); // коммуникатор для данной группы (MPI_COMM_WORLD)

    int sendCount=SendOffset[B]; // размер общего списка на отправку по всем соседям
    int recvCount=RecvOffset[B]; // размер общего списка на прием по всем соседям

    // MPI данные - сделаем статиками, поскольку это высокочастотная функция,
    // чтобы каждый раз не реаллокать (так делать вовсе не обязательно).
    static vector<VarType> SENDBUF, RECVBUF; // буферы на отправку и прием по всем соседям
    static vector<MPI_Request> REQ; // реквесты для неблокирующих обменов
    static vector<MPI_Status>  STS; // статусы для неблокирующих обменов
```

```
    // ресайзим, если надо
    if(2*B > (int)REQ.size()){  REQ.resize(2*B); STS.resize(2*B); }
    if(sendCount>(int)SENDBUF.size()) SENDBUF.resize(sendCount);
    if(recvCount>(int)RECVBUF.size()) RECVBUF.resize(recvCount);

    int nreq=0; // сквозной счетчик реквестов сообщений

    // инициируем получение сообщений
    for(int p=0; p<B; p++){
        int SZ = (RecvOffset[p+1]-RecvOffset[p])*sizeof(VarType); // размер сообщения
        if(SZ<=0) continue; // если нечего слать - пропускаем соседа
        int NB_ID = Neighbours[p]; // узнаем номер процесса данного соседа
        int mpires = MPI_Irecv(&RECVBUF[RecvOffset[p]*sizeof(VarType)], SZ, MPI_CHAR,
                               NB_ID, 0, MCW, &(REQ[nreq]));
        ASSERT(mpires==MPI_SUCCESS, "MPI_Irecv failed");
        //ASSERT - какой-то макрос проверки-авоста, замените на ваш способ проверки
        nreq++;
    }

    // пакуем данные с интерфейса по единому списку сразу по всем соседям
    #pragma omp parallel for // в параллельном режиме с целью ускорения (К.О.)
    for(int i=0; i<sendCount; ++i) SENDBUF[i] = V[Send[i]/*номер ячейки на отправку*/];

    // инициируем отправку сообщений
    for(int p=0; p<B; p++){
        int SZ =(SendOffset[p+1]-SendOffset[p])*sizeof(VarType); // размер сообщения
        if(SZ<=0) continue; // если нечего принимать - пропускаем соседа
        int NB_ID = Neighbours[p]; // узнаем номер процесса данного соседа
        int mpires = MPI_Isend(&SENDBUF[SendOffset[p]*sizeof(VarType)], SZ, MPI_CHAR,
                               NB_ID, 0, MCW, &(REQ[nreq]));
        ASSERT(mpires==MPI_SUCCESS, "MPI_Isend failed");
        nreq++;
    }

    if(nreq>0){ // ждем завершения всех обменов
        int mpires = MPI_Waitall(nreq, &REQ[0], &STS[0]);
        ASSERT(mpires==MPI_SUCCESS, "MPI_Waitall failed");
    }

    // разбираем данные с гало ячеек по единому списку сразу по всем соседям
    #pragma omp parallel for
    for(int i=0; i<recvCount; ++i) V[Recv[i]/*номер ячейки на прием*/] = RECVBUF[i];
}
```

Эту функцию можно немного улучшить. Во-первых, она умеет делать обмен только для одного вектора, а мы знаем, что обмены лучше группировать. Во-вторых, вектора могут быть и блочными, то есть каждой вершине может быть сопоставлен целый набор переменных. Ну и, в-третьих, ожидание сендов и ресивов можно растащить. Далее приводится пример реализации для множественных и к тому же блочных векторов. Да, уточним, что такое блочный вектор. Пусть в каждой вершине графа, в нашем случае можно сказать – ячейке сетки, хранится M значений. Нам эти значения надо обновить для гало. Доступ к j-й переменной i-й ячейки в блочном векторе X будет X[i*M + j]. Чтобы минимизировать количество сообщений, будем группировать обмен, если надо обменяться сразу по нескольким блочным векторам. Поэтому функция берет на вход массив из блочных векторов, а ля вектор векторов. Итак, пробуем разобраться и всё понять.

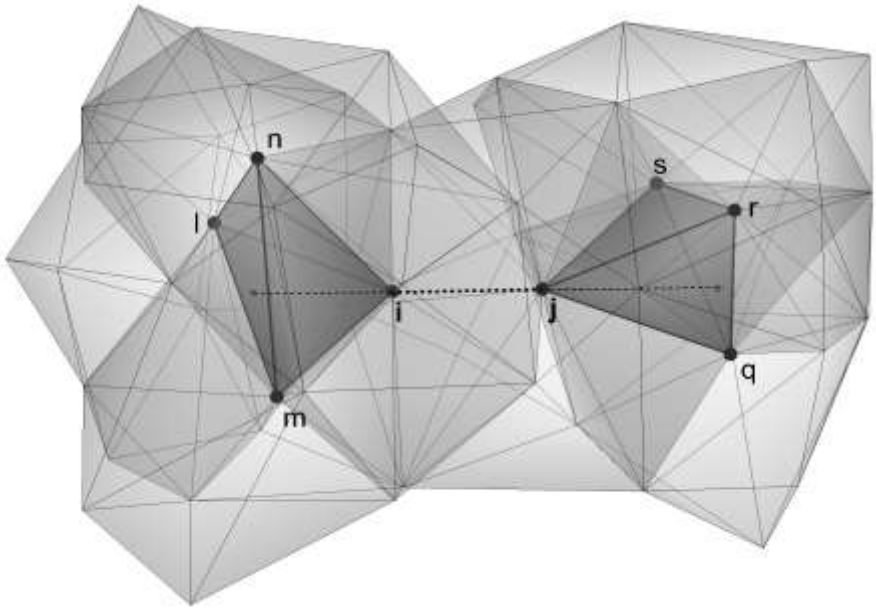
```
template <typename VarType> void Update(
    int nBlocks, // число блочных векторов, которые надо обновить
    VarType **VV, // массив указателей на эти блочные вектора (размера nBlocks)
    const int *VarNums, // массив с числом переменных в блоках каждого блочного вектора
```

```
const tCommScheme &CS /*структура, описывающая схему обменов*/){

const int B = CS.GetNumOfNeighbours(); // число соседей
if(B==0) return; // нет соседей - нет проблем

ASSERT(nBlocks>0, "wrong nBlocks!");
int VAR_NUM = 0; // общее число переменных во всех блоках
for(int iv=0; iv<nBlocks; ++iv){ // считаем сколько переменных во всех блоках
    ASSERT(VarNums[iv]>0, "wrong block size!");
    ASSERT(VV[iv], "NULL pointer!");
    VAR_NUM += VarNums[iv];
}
int *Send = CS.GetSendList(); // список ячеек на отправку по всем соседям
int *Recv = CS.GetRecvList(); // список ячеек на прием по всем соседям
int *SendOffset = CS.GetSendOffset(); // смещения списков по каждому соседу на отправку
int *RecvOffset = CS.GetRecvOffset(); // смещения списков по каждому соседу на прием
int *Neighbours = CS.GetListOfNeigbours(); // номера процессов соседей
MPI_Comm MCW = CS.GetMyComm(); // коммуникатор для данной группы (MPI_COMM_WORLD)

int sendCount=SendOffset[B]; // размер общего списка на отправку по всем соседям
int recvCount=RecvOffset[B]; // размер общего списка на прием по всем соседям
int sendSize = sendCount*VAR_NUM; // размеры общего буфера на отправку
int recvSize = recvCount*VAR_NUM; // размеры общего буфера на прием

static vector<VarType> SENDBUF, RECVBUF; // буферы на отправку и прием по всем соседям
static vector<MPI_Request> SREQ, RREQ; // реквесты для неблокирующих обменов
static vector<MPI_Status>  SSTS, RSTS; // статусы для них же
// ресайзим, если надо
if(B>(int)SREQ.size()){SREQ.resize(B); RREQ.resize(B); SSTS.resize(B); RSTS.resize(B);}
if(sendSize>(int)SENDBUF.size()) SENDBUF.resize(sendSize);
if(recvSize>(int)RECVBUF.size()) RECVBUF.resize(recvSize);

int nrreq=0, nsreq=0; // сквозные счетчики реквестов сообщений

// инициируем получение сообщений
for( int p=0; p<B; p++){
    int SZ = (RecvOffset[p+1]-RecvOffset[p])*VAR_NUM*sizeof(VarType); //размер сообщения
    if(SZ<=0) continue; // если нечего слать - пропускаем соседа
    int NB_ID = Neighbours[p]; // узнаем номер процесса данного соседа
    int mpires = MPI_Irecv(&RECVBUF[RecvOffset[p]*VAR_NUM], SZ, MPI_CHAR,
                           NB_ID, 0, MCW, &(RREQ[nrreq]));
    ASSERT(mpires==MPI_SUCCESS, "MPI_Irecv failed");
    nrreq++;
}

// пакуем исходящие сообщения
#pragma omp parallel for // пакуем в параллельном режиме с целью ускорения (K.O.)
for(int i=0; i<sendCount; ++i){ // пакуем данные с интерфейса по единому списку
    int ic = Send[i]; // номер ячейки на отправку.
    int Ivar=0; // номер переменной в данной ячейке - по всем блокам
    for(int iv=0; iv<nBlocks; ++iv){ // перебираем блоки блочных векторов
        int varnum = VarNums[iv]; // число переменных в блоке в данном блочном векторе
        VarType *v = VV[iv]; // указатель на данный блочный вектор
        for(int ivar=0; ivar<varnum; ++ivar, ++Ivar) // пихаем данный блок в буфер
            SENDBUF[i*VAR_NUM + Ivar] = v[ic*varnum + ivar];
    }
}

// инициируем отправку сообщений
for(int p=0; p<B; p++){
    int SZ = (SendOffset[p+1]-SendOffset[p])*VAR_NUM*sizeof(VarType); //размер сообщения
    if(SZ<=0) continue; // если нечего принимать - пропускаем соседа
    int NB_ID = Neighbours[p]; // узнаем номер процесса данного соседа
```

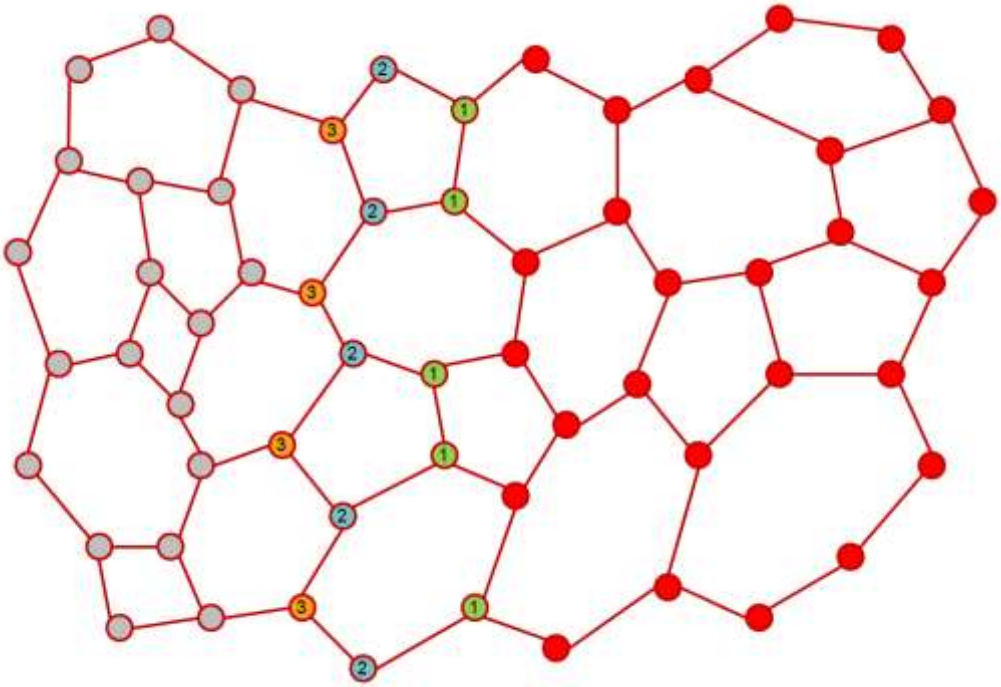
```
            int mpires = MPI_Isend(&SENDBUF[SendOffset[p]*VAR_NUM], SZ, MPI_CHAR,
                                   NB_ID, 0, MCW, &(SREQ[nsreq]));
            ASSERT(mpires==MPI_SUCCESS, "MPI_Isend failed");
            nsreq++;
        }

        if(nrreq>0){ // ждем завершения получения
            int mpires = MPI_Waitall(nrreq, &RREQ[0], &RSTS[0]);
            ASSERT(mpires==MPI_SUCCESS, "MPI_Waitall (recv) failed");
        }

        // распаковка полученных сообщений
        #pragma omp parallel for
        for(int i=0; i<recvCount; ++i){ // распаковываем данные с гало ячеек по единому списку
            int ic = Recv[i]; // номер ячейки на прием
            int Ivar=0;
            for(int iv=0; iv<nBlocks; ++iv){ // перебираем блочные вектора
                int varnum = VarNums[iv]; // число переменных в блоке в данном блочном векторе
                VarType *v = VV[iv]; // указатель на данный блочный вектор
                for(int ivar = 0; ivar<varnum; ++ivar, ++Ivar) // вытаскиваем блок из буфера
                    v[ic*varnum + ivar] = RECVBUF[i*VAR_NUM + Ivar];
            }
        }
        if(nsreq>0){ // ждем завершения отправок
            int mpires = MPI_Waitall(nsreq, &SREQ[0], &SSTS[0]);
            ASSERT(mpires==MPI_SUCCESS, "MPI_Waitall (send) failed");
        }
}
```

### 4.4.3 Широкий шаблон

А если нам надо делать обмен по нескольким уровням смежности, или соседства, или как еще это назвать? Может, у нас схема повышенной точности, у нее широкий разлапистый шаблон, и надо не только в соседние ячейки лезть, а и в смежные с ними, и смежные со смежными с ними. И что тогда? Вот, например, разлапистый шаблон какой-то там численной схемы. Чтобы посчитать поток между ячейками (читай, вершинами графа) i и j, надо сгрести всю окрестную ботву в 2 уровня.



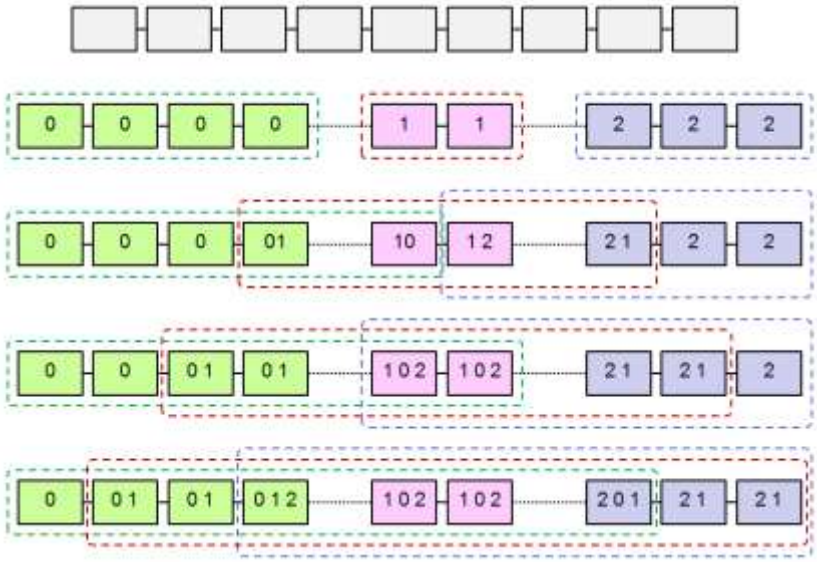
Ну вот уровни показаны на примере того же графа на картинке ниже. Пусть наши вершины – красные. Первый уровень гало – чужие вершины, соединенные ребром с нашими вершинами. Второй уровень – чужие вершины, соединенные ребром с вершинами 1-го уровня гало. Продолжите логический ряд, как говорится.



По базовой схеме обмена по 1 уровню можно пробросить сколько нужно уровней, за соответствующее количество проходов. Впишем в каждую вершину номер процесса (подобласти) владельца. Сделаем проход по всем вершинам локального графа и для каждой вершины добавим все номера процессов (пока там только один номер – владельца), приписанные этой вершине, всем соседним вершинам. Нет, если номер в вершине уже записан, то не добавляем, конечно. Номера – без повторов! Получили у каждой вершины помимо номера владельца, номера всех процессов, которым понадобилась эта вершина на 1 уровне. Еще проход – получили 2 уровень. И т.д.

Но мы же в параллельном режиме. У нас есть только кусок графа, подобласть и гало по 1 уровню. Делаем то же самое, но после каждого прохода делаем обновление гало, обмен с соседями (по 1 уровню) и пересылаем эти списки номеров подобластей в вершинах. Получим интерфейсные вершины, промаркированные до нужного уровня. Дальше надо будет передать помеченные вершины и нужные ребра соседям, чтобы они достроили гало до нужного уровня. Делается это p2p обменом с соседями, в основном по аналогии с предыдущим пунктом.

Посмотрим на одномерном примере. Вот граф, вершины – клеточки, в которых пока пусто. Поделили этого графа между 3 процессами каким-то (пусть не самым оптимальным) образом. Приписали вершинам номера владельцев – 0, 1 или 2. Очертили пунктиром локальный мир подобластей. Пока в нем только собственные ячейки. Сделали проход, вписали в соседние вершины номера. Локальный мир расширился на один уровень. Сделали еще – стало 2 уровня гало. Еще – три уровня. На 3 уровнях смежности гало процессы 0 и 2 даже стали соседями и будут издалека перебрасываться друг в друга информацией.




## 4.5 Эффективность распараллеливания

### 4.5.1 Накладные расходы

Рассмотрим p2p обмен между процессами. У этой операции получились такие этапы:

- разбираемся с буферами – считаем смещения, выделяем память, если надо;
- инициируем прием – MPI_Irecv от всех соседей;
- пакуем сообщения на отправку – выбираем разрозненные данные из массивов по всей подобласти в буфера на отправку соседям;
- инициируем отправку – MPI_Isend всем соседям;
- ждем завершения всех приемов – MPI_Waitall;
- распаковываем входящие сообщения – раскладываем данные из буферов в массивы по всей подобласти;
- ждем завершения всех отправок – MPI_Waitall.

Можно немножко упростить – ждать завершения сразу всех обменов, и отправок, и приемов, тогда будет на один этап меньше. Можно усложнить – ждать завершения приемов поштучно (запрашивая статусы) и распаковывать данные от каждого соседа по мере завершения обмена с ним.

Получаются следующий оверхэд.

- Запаковка и распаковка – это копирование данных, плюс там и косвенная адресация еще, из-за чего копирование не очень быстрое. Тут влияют характеристики подсистемы памяти – пропускная способность и латентность доступа.
- Прием и передача данных MPI. Тут, во-первых, влияют характеристики коммуникационной среды, то есть сети. Это – пропускная способность и латентность сети. Но чтобы данные попали в сеть, они должны попасть на сетевой адаптер. Значит, во-вторых, тут влияют характеристики шины между процессором и сетевым адаптером. В-третьих, данные надо взять-таки из памяти, опять, значит, характеристики памяти влияют.

Не стоит пренебрегать этапами запаковки-распаковки, они тоже едят прилично времени. Так прилично, что приходится применять многопоточное распараллеливание.

Итак, нам надо, чтобы данные лежали в памяти как можно удобнее, чтобы копирование в буферы и из буферов было быстрее; чтобы объем обмена данных был как можно меньше (пропускная способность); чтобы количество обменов данными было как можно меньше (латентность).

### 4.5.2 Параллельное ускорение

Имеется фиксированная задача фиксированного размера. В ней обрабатывается N объектов – ячеек/вершин графа/… Обозначим время “настенных часов” (aka wall-clock time) выполнения задачи на P устройствах (процессорах/ядрах/узлах – нужное подчеркнуть) как $T_P$.

Число N описывает вычислительную стоимость задачи, P – количество используемых вычислительных устройств.

Ускорением в параллельном режиме относительно последовательного режима будет отношение времени выполнения в последовательном режиме ко времени выполнения в параллельном режиме на P устройствах: $S = T_1/T_P$.

Ускорение не обязательно отсчитывать от последовательного режима. Можно отсчитать ускорение относительно какого-то другого числа устройств: $S = T_p/T_P$ – ускорение на P устройствах относительно времени выполнения на p устройствах, $P > p$.

Это ускорение, когда размер задачи фиксирован, а изменяется количество задействованных ресурсов, называют также – speedup, strong speedup, strong scaling.

Параллельной эффективностью по ускорению будет соотношение этого ускорения к соотношению задействованных ресурсов: $E = S/(P/p)\times100\% = (T_p\times p)/(T_P\times P)\times100\%$.

Для ускорения можно построить график, как оно меняется с увеличением числа устройств. К полученному графику обычно пририсовывают “идеальное” линейное ускорение $P/p$ (черная линия на графиках ниже), чтобы было видно, насколько полученный результат от него отклоняется. Выглядят графики ускорения как-то так:

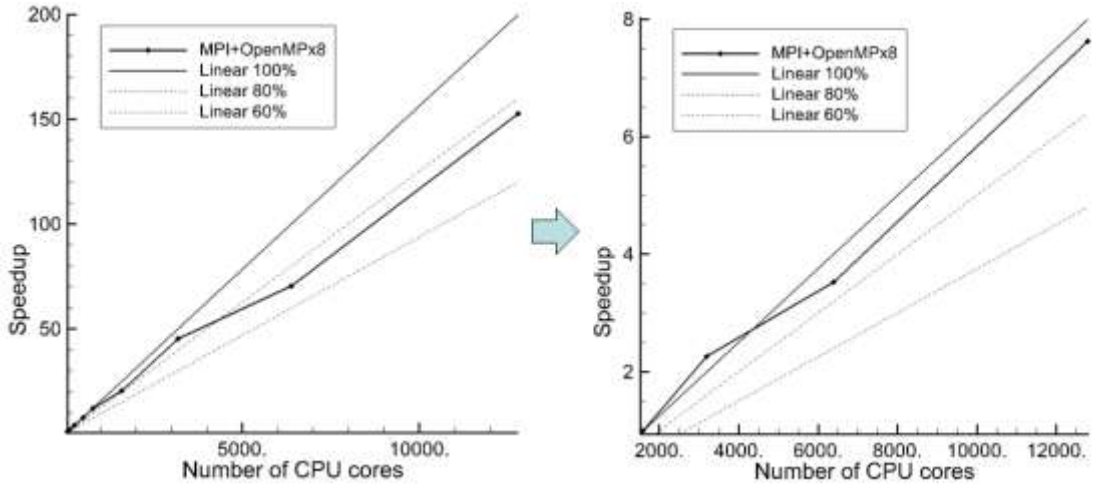

Обычно график ускорения выглядит корявенько, поскольку на системе общего пользования время выполнения может варьироваться по разным причинам (разный разброс процессов по системе, разная загрузка сети другими задачами, и т.д.).

График ускорения обычно доходит до какого-то предела и загибается, параллелизм исчерпывается, и расчет может замедляться. Чем меньше локальный размер задачи, то есть, чем меньше вычислений, тем больше становится вес накладных расходов на обмен данными. Например, пусть у нас сетка – квадратик N×N. Поделили ее по двум осям на p×p частей и раздали P = p×p процессам. Выше уже была такая картинка. Максимальный обмен будет 4N/p. Вычислительная стоимость пропорциональна площади локальных квадратиков, которая равна $(N/p)^2$. Соотношение вычислений и накладных расходов будет $O((N/p)^2/(N/p)) = O(N/p)$. При увеличении p оно уменьшается, пока, наконец, не уменьшится ниже плинтуса. Вывод: чтобы

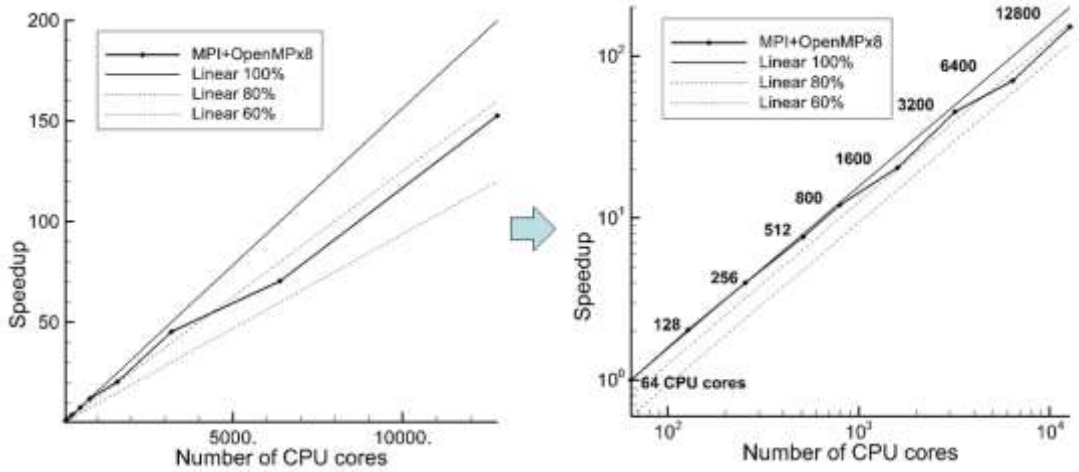
показать хорошее ускорение, можно размер задачи брать как можно больше. Тогда локальная вычислительная нагрузка будет больше, а значит и вклад накладных расходов будет ниже.

Но мы не дадим себя обмануть. Чтобы правильно интерпретировать результаты, надо смотреть не на само ускорение, а на какой локальной нагрузке оно загнулось. Это дает представление о степени параллелизма полученной реализации. То есть информативным результатом, независимо от того, какой мы взяли исходный размер задачи, будет локальная нагрузка на устройство, при которой наступила заметная деградация параллельной эффективности.

Как еще можно схитрить, чтобы график ускорения выглядел лучше? Можно взять за исходное число устройств большее значение. Тогда в baseline уже будет заложен нехилый overhead на обмены. Вот, например, взяли исходное число ядер в несколько раз больше, и график стал выглядеть намного лучше:




Можно тот же график цинично переложить в логарифмическую шкалу, тогда он будет выглядеть намного ближе к идеальному ускорению:




Но мы не дадим себя обмануть. Можно брать разные размеры задачи, разное начальное число устройств, по-разному это нарисовать, но мы можем, например, смотреть на параллельную эффективность удвоения числа устройств при какой-то конкретной локальной нагрузке на устройство. Тогда все будет понятно.

На графике ускорения также может наблюдаться сверхлинейное ускорение. Оно может возникать по разным причинам – хорошим и плохим. Получаемая производительность

устройств может расти при уменьшении локального объема работы, приходящегося на устройства, за счет уменьшения числа кэш промахов. Чем меньше данных, тем больше попаданий в кэш. Этот бонус будем называть кэш-эффектом. Это хорошая сверхлинейность. Обычно это несильный эффект, отклонение от линейного ускорения небольшое. Если график явно в грубой форме идет нелинейно, то это может быть из-за нелинейной сложности вычислений по локальным подобластям. Значит, последовательный алгоритм сделан через “универсальный интерфейс”. А это – плохо.

### 4.5.3 Масштабирование

Другой вариант оценки эффективности распараллеливания – масштабирование, aka scaling, weak speedup, weak scaling. В этом случае размер задачи N не фиксирован, а меняется пропорционально количеству задействованных ресурсов: $N = n \times P$, где n – локальный размер задачи, приходящийся на одно устройство. Вот этот размер как раз фиксирован.

График масштабирования будет выглядеть как-то так:


По вертикальной оси может быть время выполнения задачи. Тут уже “идеальная” линия – горизонтальная. В идеале при увеличении числа устройств и пропорциональном увеличении размера задачи время должно оставаться неизменным. Но с увеличением числа устройств может расти, в частности, communication overhead, поэтому время выполнения растет. Опять же, чтобы график выглядел лучше, над взять побольше локальную нагрузку n. Чем больше вычислений, тем меньше вес оверхеда в общем времени выполнения. Но мы не дадим себя обмануть, смотрим, на какой именно локальной нагрузке происходит это масштабирование, и как оно будет меняться при уменьшении этой локальной нагрузки.

Эффективностью масштабирования будет отношение времени на базовом числе устройств ко времени выполнения на большем числе устройств: $E = T_p/T_P \times 100\%$

## 4.6 Снижение накладных расходов на обмен данными

Рассмотрим для примера что-нибудь простое. Возьмем операции линейной алгебры над векторами. Пусть у нас есть граф связей ячеек (или других каких объектов) расчетной области. Пусть этих ячеек – N. В ячейках задана какая-то переменная. Из этих переменных во всех ячейках получим вектор размера N, в котором число значений равно числу ячеек, то есть числу вершин в графе.

Вершины мы распределили, значит, распределили и вектор. У этого графа есть матрица смежности. Пусть у нас есть какая-то разреженная матрица того же размера $N \times N$, портрет которой совпадает с портретом матрицы смежности графа везде, кроме главной диагонали.

Пусть на главной диагонали тоже могут быть “ненули”. Матрицу распределили построчно между параллельными процессами согласно декомпозиции.

У нас есть распределенные вектора и матрицы. Будем делать с этими объектами какие-то простенькие операции. По умолчанию выполнять вычисления каждый процесс будет **только для собственных позиций**, об остальных значениях позаботятся другие процессы.

Операцию матрично-векторного произведения **b** = **Ax** с разреженной матрицей **A** обозначим **SpMV(b, A, x)** (Sparse Matrix-Vector Product).

Если в i-й строке матрицы попался неноль в j-м столбце, то нам надо считать из входного вектора значение из j-й позиции. А если эта позиция – не наша? Тогда это гало. А чтобы что-то взять из гало, это что-то надо обновить. Тут нам и понадобится p2p обмен для обновления гало. Пусть это обновление гало делает функция Update.

Возьмем еще такую операцию с векторами **x** = a**x**+b**y**, где **x**, **y** – вектора, a, b – скалярные значения. Просто линейная комбинацию двух векторов. Обозначим эту операцию **axpby(x, y,** a, b**)**. Для axpby обмен не требуется, потому что каждое значение выходного вектора получается из значений в той же позиции входных векторов, нет зависимости по данным от соседних позиций.

Назовем эти базовые вычислительные операции кернелами (kernel). Этих простейших операций будет достаточно для объяснения сути происходящего.

### 4.6.1 Группировка обменов

Если нужно обновить гало у нескольких массивов, надо выполнять обмен сразу для всех этих массивов, а не обновлять каждый массив по очереди. Тогда количество MPI сообщений будет кратно меньше, а значит, меньше потерь на латентность сети и на запаковки-распаковки.

Пусть нам надо выполнить несколько SpMV:

```
SpMV(b, A, x); // b = Ax
SpMV(c, A, y); // c = Ay
```

Чтобы эти операции отработали корректно в распределенном режиме, надо добавить Update-ы:

```
Update(x);
SpMV(b, A, x); // b = Ax
Update(y);
SpMV(c, A, y); // c = Ay
```

Как мы уже догадались, это будет, выражаясь тактично, suboptimal. Лучше как-то научить Update обновлять сразу несколько векторов:

```
Update(x,y);
SpMV(b, A, x); // b = Ax
SpMV(c, A, y); // c = Ay
```

### 4.6.2 Перекрытие вычислений на разных процессах

Усложним пример, добавим туда еще axpby и еще SpMV:

```
SpMV(b, A, x); // b = Ax
SpMV(c, A, y); // c = Ay
axpby(x, y, 1, 2); // x = x + 2*y
axpby(y, x, 3, 2); // y = 3*y + 2*x
SpMV(d, A, x); // d = Ax
SpMV(e, A, y); // e = Ay
```

Чтобы это работало в распределенном режиме, надо добавить обмены. Добавим каждому SpMV соответствующий Update (для axpby обмен не требуется):

```
Update(x);
SpMV(b, A, x); // b = Ax
Update(y);
SpMV(c, A, y); // c = Ay
axpby(x, y, 1, 2); // x = x + 2*y
axpby(y, x, 3, 2); // y = 3*y + 2*x
Update(x);
SpMV(d, A, x); // d = Ax
Update(y);
SpMV(e, A, y); // e = Ay
```

А зачем так много? Update же умеет сразу по несколько векторов брать. Тогда так:

```
Update(x, y);
SpMV(b, A, x); // b = Ax
SpMV(c, A, y); // c = Ay
axpby(x, y, 1, 2); // x = x + 2*y
axpby(y, x, 3, 2); // y = 3*y + 2*x
Update(x, y);
SpMV(d, A, x); // d = Ax
SpMV(e, A, y); // e = Ay
```

Попробуем еще улучшить алгоритм. Добавим этим вычислительным функциям, кернелам, еще аргумент, указывающий зону действия. Пусть это перечисление (enum) с вариантами _OWN, _LOCAL, _INNER, _IFACE, _HALO. Эти обозначения нам уже знакомы из предыдущих разделов.

```
Update(x, y);
SpMV(b, A, x, _OWN); // b = Ax
SpMV(c, A, y, _OWN); // c = Ay
axpby(x, y, 1, 2, _OWN); // x = x + 2*y
axpby(y, x, 3, 2, _OWN); // y = 3*y + 2*x
Update(x, y);
SpMV(d, A, x, _OWN); // d = Ax
SpMV(e, A, y, _OWN); // e = Ay
```

Наверное, уже понятно, что раз у axpby для каждой позиции выходного вектора нет зависимости по данным от соседних позиций, то можно схитрить:

```
Update(x, y);
SpMV(b, A, x, _OWN); // b = Ax
SpMV(c, A, y, _OWN); // c = Ay
axpby(x, y, 1, 2, _LOCAL); // x = x + 2*y
axpby(y, x, 3, 2, _LOCAL); // y = 3*y + 2*x
SpMV(d, A, x, _OWN); // d = Ax
SpMV(e, A, y, _OWN); // e = Ay
```

Расширили для axpby зону действия на _LOCAL, теперь эта операция обрабатывает и собственные позиции, и гало. А раз вектора **x** и **y** уже были обновлены для SpMV ранее и с тех пор не менялись, обмен больше не требуется. Вместо обменов процесс просто сам взял и посчитал значения в гало.

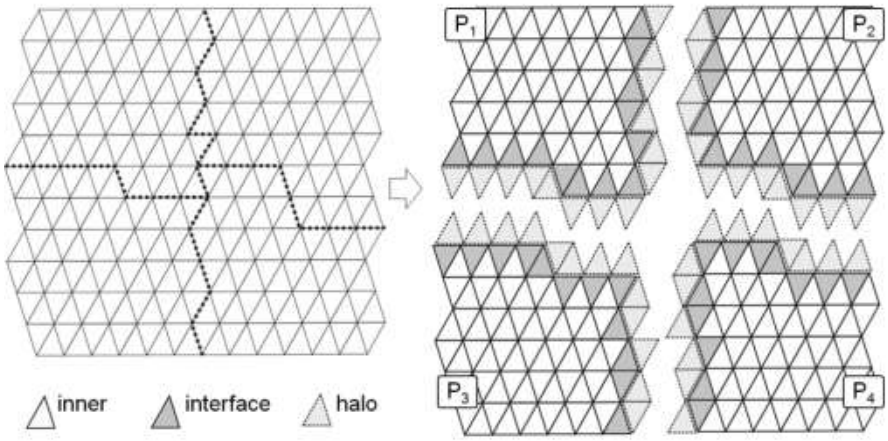
Получилось некоторое перекрытие вычислений: гало позиции посчитали более одного процесса. Ну и ничего страшного, зато не пришлось делать обмен, что гораздо дороже. А вот если бы вместо очень(-очень) дешевой операции axpby было что-то очень вычислительно интенсивное, очень дорогое, то мы бы подумали, что лучше – считать или качать по сети.

Итого: если что-то можно задешево посчитать в гало самостоятельно без обменов – используем эту возможность.

### 4.6.3 Перекрытие обменов и вычислений – режим overlap

Раз уж мы ввели область действия, поступим еще хитрее и реализуем режим сокрытия обменов за вычислениями (или вычислений за обменами, смотря, что из них дольше), aka communication and computation **overlap,** communication hiding.

У нас ведь есть зона внутренних ячеек – _INNER, не имеющая связей с чужими ячейками. Еще раз посмотрим на картинку:




У нас вектора хранят значения в ячейках. Каждому треугольничку на картинке соответствует позиция с каким-то значением в векторе. В операции SpMV с нашей матрицей мы можем найти значения выходного вектора в белых ячейках, не выполняя Update, поскольку в строках матрицы, соответствующих этим ячейкам, нет ненулей в столбцах, соответствующих чужим ячейкам. Рассмотрим одиночную операцию SpMV:

```
Update(x);
SpMV(b, A, x, _OWN); // b = Ax
```

Запишем эквивалентно, но немного по-другому:

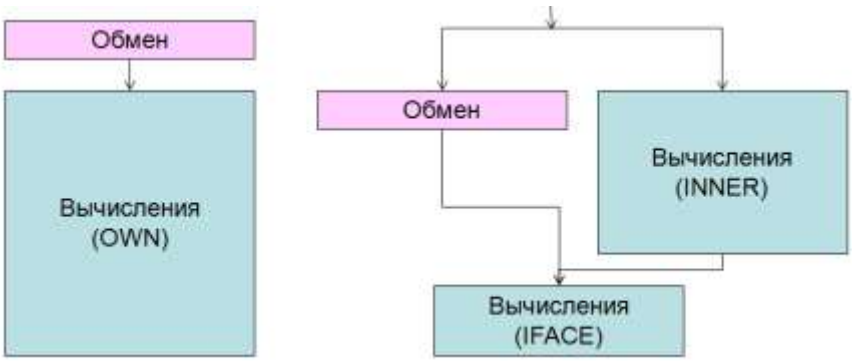
```
Update(x);
SpMV(b, A, x, _INNER); // b = Ax – inner cells only
SpMV(b, A, x, _IFACE); // b = Ax – interface cells only
```

Посчитали SpMV сначала только для внутренних позиций (белые треугольнички), потом для интерфейсных позиций (темно-серые треугольнички). В итоге получилось все то же самое.

Теперь прокачаем операцию Update. В ней внутри, как мы уже знаем, есть инициализация обменов – MPI_Isend, MPI_Irecv, и ожидание завершения обменов – MPI_Waitall. Растащим инициализацию и ожидание на две функции: UpdateInit, UpdateWait. Запишем теперь так:

```
UpdateInit(x);
SpMV(b, A, x, _INNER); // b = Ax – inner cells only
UpdateWait(x);
SpMV(b, A, x, _IFACE); // b = Ax – interface cells only
```

Теперь библиотека MPI будет работать в фоновом режиме, а мы в это время будем делать SpMV для внутренних позиций (коих обычно подавляющее большинство). Тем самым мы скрыли обмен за вычислениями.

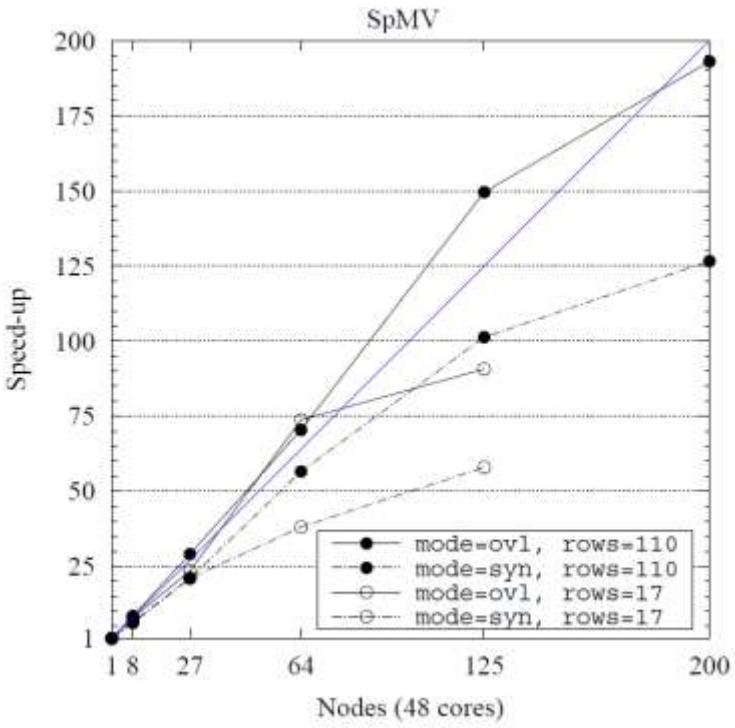



Но тут у нас оверлапнулась только MPI часть, а запаковка и распаковка – нет. Можно сделать и по-другому. Вернемся обратно к единой функции Update, но используем многопоточный режим. Разделим нити по ролям. Пусть одна нить обменивается данными, а другая вычисляет. Так оверлапнется весь обмен – и MPI часть и запаковка-распаковка.

```
#pragma omp parallel
{
    if(omp_get_thread_num() == 0) Update(x);
    else SpMV(b, A, x, _INNER); // b = Ax – inner cells only
}
SpMV(b, A, x, _IFACE); // b = Ax – interface cells only
```

В случае гибридного MPI+OpenMP распараллеливания, если мы хотим делать в многопоточном режиме и обмены, и вычисления (то есть много нитей сразу вычисляют), то надо будет использовать вложенные параллельные области, либо использовать OpenMP teams, либо еще как-то выкручиваться. Но в такую глубь тут мы не пойдем.

Да, обычно обмены на CPU скрываются не полностью, какой-то довесок остается торчать, поскольку MPI библиотека в фоне будет на обменах подъедать и ресурсы CPU, и пропускную способность памяти. Но все равно можно достичь существенного сокращения оверхеда. Вот пример, что у нас с коллегами получалось на таком SpMV:



Тут SpMV для двух разных размеров сетки – 17 миллионов и 110 миллионов ячеек. Пунктиром – исходный синхронный вариант выполнения обменов (syn) и вычислений: сначала обмены, потом вычисления. Сплошной линией – асинхронный режим overlap (ovl), когда обмены идут одновременно с вычислениями. Выигрыш налицо. Да, 200 – это кластерных узлов, в которых по два 24-ядерных CPU, суммарно это 9600 ядер. Тут кстати вон видно и сверхлинейное ускорение (aka superlinear speedup) проявляется на кэш-эффекте.

### 4.6.4 Двухуровневая декомпозиция

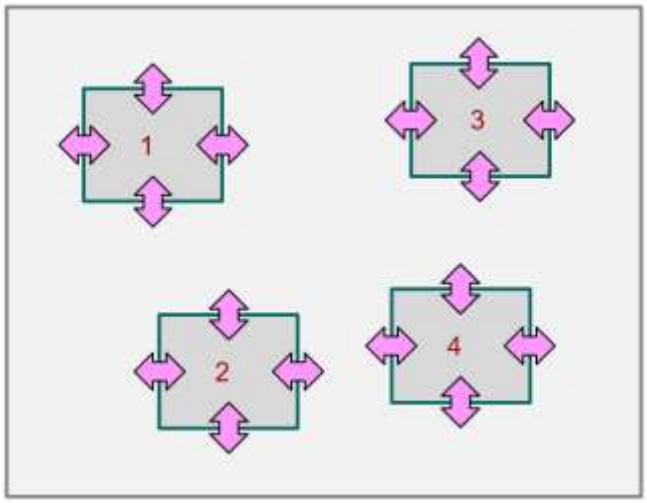
Да, что-то длинная глава получается. Вы, бедненькие, наверное, уже устали это читать. Сочувствую. А представьте, как я устал это писать! Потерпите, осталось чуть-чуть.

Если мы хотим задействовать многоядерные узлы кластера только средствами MPI, то нам надо будет наплодить столько MPI процессов на каждый узел, сколько на нем есть ядер. То есть у нас будут множественные MPI процессы на каждом узле кластера.

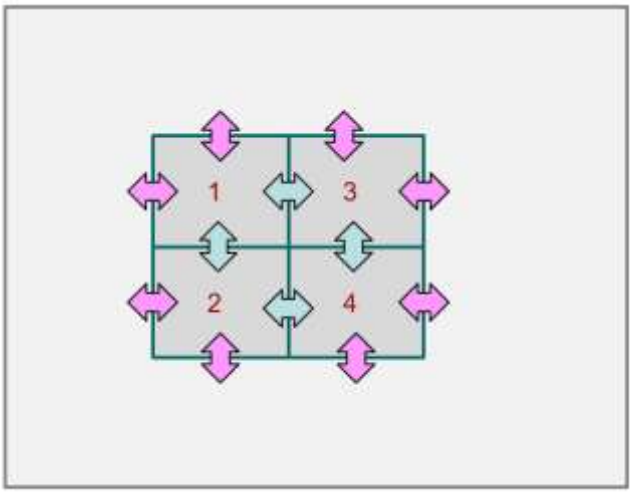
Если мы используем комбинированное распараллеливание MPI+OpenMP, то мы можем разместить 1 процесс на узел, а далее его размножить на нужное число потоков. Но, если у нас на кластере узлы многопроцессорные, то, чтобы не нарываться на NUMA фактор, будет эффективнее разместить по одному MPI процессу на многоядерный процессор. Опять-таки, получается более одного MPI процесса на узел.

А чем отличаются процессы на одном узле от процессов на разных узлах? Правильно, скоростью обмена. Процессы на одном узле общаются копированием данных в оперативной памяти, что очень сильно быстрее, чем гонять трафик по сети. Получается, что у нас есть какие-то подгруппы процессов, намного теснее связанные друг с другом. Скорость обмена между ними несопоставимо выше. Грех не использовать эту особенность.

Посмотрим на картинки. Вот пусть у нас есть кластер с 4-процессорными узлами. Ну или мы просто так запускаем какую-то большую группу процессов, и размещаем их по 4 процесса на узел (то есть задействуем в 4 раза меньше узлов кластера, чем запускаем процессов). Пусть есть такая вот какая-то расчетная область (серый контур), мы ее поделили на подобласти, и вот 4 процесса оказались на одном узле кластера – P1, P2, P3, P4. Их подобласти отмечены зеленым контуром. Поскольку при декомпозиции мы не использовали информацию о том, что процессы группируются по 4, их подобласти получились как попало. То есть, они не оказались соседями, и по всей своей границе они будут обмениваться по сети с процессами на других узлах. Этот факт символизируют розовые стрелочки.



Модифицируем процедуру декомпозиции, сделаем все то же самое, но в 2 этапа, сначала поделим графа на подобласти узлов кластера, а потом каждую подобласть еще разделим между процессами внутри одного узла. Тогда они гарантированно будут проживать компактно, окажутся соседями, и обмен по сети будет только по внешней границе их общего надела. А между процессами в этой подгруппе обмен станет копированием в оперативной памяти узла, что отмечено голубыми стрелочками.




Сокращение сетевого трафика налицо. Да, нам же нужно узнать, какие именно процессы оказались в таких подгруппах вместе на одном узле кластера, то есть внутри параллельной системы с общей памятью. Для этого можно использовать

MPI_Comm_split_type(…, MPI_COMM_TYPE_SHARED, …).

Если вдруг MPI на системе такой допотопный, что не поддерживает MPI_COMM_TYPE_SHARED, можно выяснить имя узла (для чего можно использовать MPI_Get_processor_name или просто popen("hostname","r") на Linux) и сгруппировать процессы с одинаковым именем обычным MPI_Comm_split.

### 4.6.5 Другие прочие лайфхаки

Как узнать, что у нас есть дисбаланс? Ставим перед вычислительной нагрузкой барьер, чтобы все начали одновременно. Ставим после вычислительной нагрузки барьер и таймируем его. Если время выполнения барьера после нагрузки стало сильно больше времени выполнения пустого барьера, то все это результат дисбаланса. Как узнать, кто именно тормозит? Тот, у кого барьер сработал быстро, тот и тормоз. Остальные его ждут, у них барьер будет долгим.

Как проверить однородность узлов кластера, aka решающего поля? Вдруг какой-то узел перегрелся и понизил частоту процессора? Или еще по какой причине впал в тормоза и будет нам задерживать весь расчет. Используем тот же метод. Замеряем тестовую вычислительную нагрузку и барьер. Если барьер подзастрял, значит есть проблемы. Выдаем hostname с процессов, у которых барьер быстрый, сообщаем имена виновных админам, чтобы эти узлы спихнули в drain.

А как проверить, что распараллеливание у нас корректно, что мы не потеряли нигде обмен, ничего не запортили? Параллельная версия должна давать такой же результат, как и последовательная. Значит, результат надо сравнить. Сравнивать большие массивы данных, огромные вектора чисел – неудобно. Можно сравнивать какие-то контрольные значения. Ну

контрольная сумма по байтам или еще какая хэш функция нам не подойдут, поскольку результат должен быть одинаковым, но с точностью до погрешности округления. Ведь от перемены мест слагаемых при конечной длине мантиссы сумма-то меняется! То есть просто от изменения порядка арифметических операций при распараллеливании может изменяться результат, но (обычно) на очень малую величину. Тогда будем брать какие-то осмысленные контрольные значения, например, сумма всех элементов вектора, L2 норма вектора, минимум и максимум по всем элементам вектора. Можно просто сравнивать такие контрольные значения, что позволит быстро обнаружить ошибки. Эти скалярные значения можно просто печатать в лог, и не нужно будет городить большие файлы данных для сравнения.

А как узнать, не воткнули ли мы лишний обмен? Легко. Убираем обмен и смотрим на контрольные значения. Если они изменились, значит, обмен был зачем-то нужен. Если нет, то это, конечно, ничего не гарантирует, но дает повод задуматься, откуда и зачем обмен там взялся. Вполне возможно, что он там зазря.

А еще, если у нас в алгоритме нет групповых обменов, только обмены точка-точка, то такая штуковина может разболтаться, пойти вразнос и начать тормозить. Чтобы программа работала стабильно, туда можно добавить барьеров, чтобы хотя бы раз в какое-то время выполнялась общая синхронизация.

## 4.7 Заключение

Итак, чтобы сделать эффективное DM MIMD распараллеливание, надо:

- рационально распределить работу – балансировка загрузки, минимизация обменов;
- переупорядочить локальные данные: inner, interface, halo;
- эффективно организовать обмены – группировка обменов, перекрытие вычислений, сокрытие обменов за вычислениями, двухуровневая декомпозиция;
- проверить корректность распараллеливания;
- замерять параллельную эффективность, чтобы определять оптимальное количество ресурсов в зависимости от размера задачи.

## 4.8 Внеклассное чтение

Спецификации стандартов MPI живут тут:

- https://www.mpi-forum.org/docs/

Например, 3.0 (хотя для этой главы и древний стандарт 1.3 подойдет):

- https://www.mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf

Местные учебные материалы ВМК

- А.С.Антонов Параллельное программирование с использованием технологии MPI: Учебное пособие. -М.: Изд-во МГУ, 2004.-71 с. Правда осторожно, там Fortan! https://parallel.ru/tech/tech_dev/MPI/
- Антонов А.С. Технологии параллельного программирования MPI и OpenMP: Учеб. пособие. Предисл.: В.А.Садовничий. - М.: Издательство Московского университета, 2012.-344 с.- (Серия "Суперкомпьютерное образование"). ISBN 978-5-211-06343-3. Там глава 1 и примеры в главе 3.
  https://parallel.ru/tech/tech_dev/MPI&OpenMP
  https://parallel.ru/tech/tech_dev/MPI&OpenMP/examples/ – примеры

Западные учебные материалы

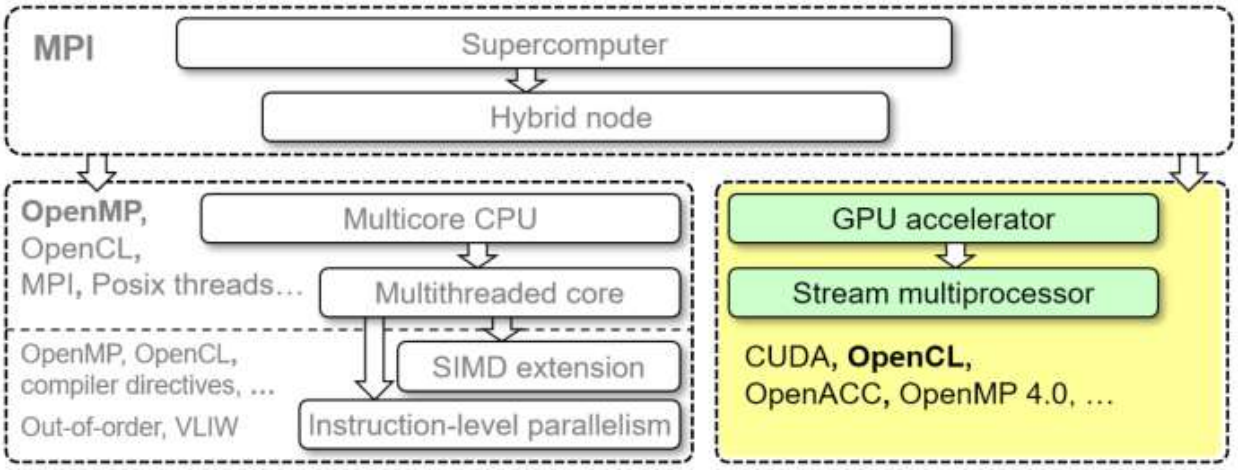
- R.Trobec, B. Slivnik, P. Bulić, B. Robič. Introduction to Parallel Computing From Algorithms to Programming on State-of-the-Art Platforms. 2018. Springer. https://doi.org/10.1007/978-3-319-98833-7 – глава 4.
- S. K. Prasad, A. Gupta, A. L. Rosenberg, A. Sussman, C. C. Weems. Topics in Parallel and Distributed Computing: Introducing Concurrency in Undergraduate Courses. Elsevier. 2016. https://doi.org/10.1016/C2015-0-00519-6 – глава 6.
- https://extremecomputingtraining.anl.gov/archive/ – большой архив с лекциями по параллельным вычислениям. Там видео лекций, выложены презентации типа такого https://extremecomputingtraining.anl.gov/files/2014/01/B.Gropp1_.00hybrid-2014.pdf
- https://www.mcs.anl.gov/~thakur/sc17-mpi-tutorial/slides.pdf

Про декомпозицию

- Якобовский М. В. Введение в параллельные методы решения задач: Учебное пособие. — Издательство Московского университета Москва, 2013. — 328 с. глава 7.
- Головченко Е. Н. Декомпозиция расчетных сеток для решения задач механики сплошных сред на высокопроизводительных вычислительных. http://keldysh.ru/council/3/D00202403/golovchenko_diss.pdf

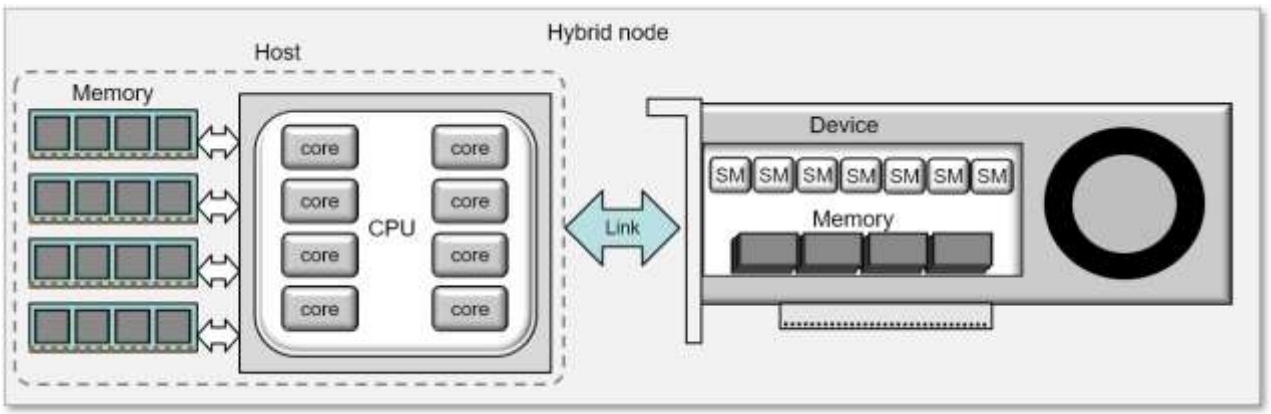
# 5 Вычисления на GPU, потоковая обработка

В предыдущих главах мы поднялись по стороне центральных процессоров с нижнего уровня, от параллелизма процессорного ядра, до верхнего уровня – уровня кластерной системы и распараллеливания с распределенной памятью. Теперь с высоты верхнего уровня, оглядев окрестности, порадовавшись пейзажу и наделав селфиков, спускаемся вниз по другому склону, по стороне ускорителей. Одним из наиболее распространенных видов таких ускорителей являются графические процессоры – GPU (Graphics Processing Unit). В этой главе мы только очень (очень-очень) поверхностно попробуем понять принципы работы этих устройств и как на них что-то посчитать.




Пусть наша вычислительная система теперь гибридная. На узлах установлены центральные процессоры и массивно-параллельные ускорители-сопроцессоры. Массивно-параллельные – означает, что в них есть много, очень много исполнительных устройств, которые могут что-то вычислять одновременно. Пусть имеют место следующие факты:

- на узле имеется один или несколько многоядерных центральных процессоров – CPU;
- у этих процессоров имеется общая оперативная память;
- на узле имеется один или несколько ускорителей;
- ускорители не имеют общей памяти с CPU, у ускорителей своя собственная набортная память высокой пропускной способности;
- ускорители соединены с процессорами какой-то внутренней шиной узла, по которой они могут обмениваться данными.



У ускорителей вовсе необязательно своя отдельная память. Отдельная память только у дискретных GPU, у интегрированных GPU память с CPU общая. Но это не так важно. Программная модель, в которой у CPU и GPU отдельная память, вполне подходит для случая, когда память у них общая. Ну они просто будут обмениваться данными не по шине, а копированием в оперативной памяти. Это ничему не противоречит.

В соответствии с терминологией OpenCL, сторону нашей гибридной модели, где живет CPU со своей оперативной памятью, будем называть хост, aka host. Вычисления на хосте – это вычисления на CPU.

Ускорители могут быть разных видов. В основном это графические процессоры, aka графон (*жарг.*), GPU. Эти устройства будем называть – device, aka девайс.

На данный момент для вычислений общего назначения широко используются GPU NVIDIA, менее широко GPU AMD. Производительность у них сопоставимая, и у тех, и у других сейчас по 4 модуля HBM2 на борту, выдающие бэндвис под 1 ТБ/с. NVIDIA обычно дороже, AMD обычно немного быстрее. И те и другие поддерживают открытый вычислительный стандарт OpenCL (Open Computing Language). NVIDIA развивает свой самобытный проприетарный фреймворк CUDA, который сейчас наиболее распространен для вычислений на графоне. Другие производители GPU, как Intel, например, обычно поддерживают OpenCL для вычислений общего назначения, но тоже развивает свою поделку типа Intel oneAPI на DPC++.

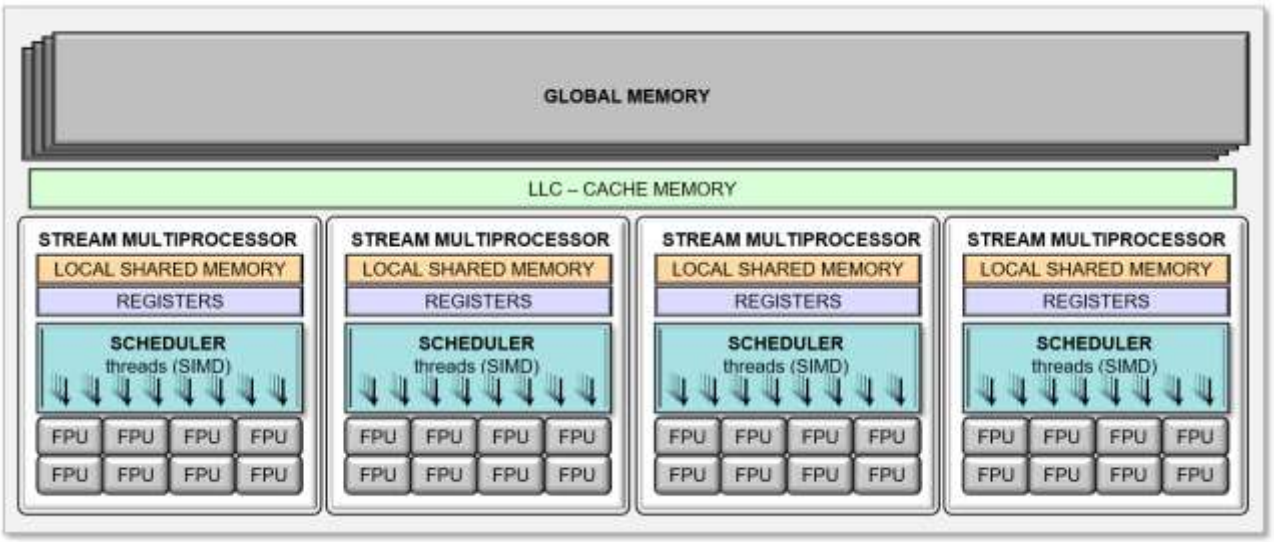
В этом курсе будет рассматриваться OpenCL, поскольку им можно считать почти на всем. Между OpenCL и CUDA сильно много общего, одну реализацию можно перегнать в другую с минимальными усилиями или даже (почти) автоматически конвертировать одно в другое. Поэтому, если вы знаете CUDA, считайте, что вы знаете OpenCL, и наоборот. Из отличий можно отметить, что

- OpenCL - это обобщенная платформонезависимая модель вычислений;
- компиляция kernel кода происходит в рантайме (*жарг.* во время выполнения – runtime) хостовой программы, что позволяет широко использовать препроцессор и делать более эффективный код (ветвления выносятся на уровень препроцессора, переменные делаются константами и т.д.).
- OpenCL имеет векторные типы данных;
- peer-to-peer передача данных между девайсами обычно поддерживается только в виде расширений от конкретных производителей.

## 5.1 Устройство GPU в совсем общих чертах

Мы уже примерно знаем, как устроен CPU. Попробуем осознать отличия GPU от CPU. Скажите что-нибудь расхожее, что GPU – это как CPU, только намного больше ядер, но ядра эти попроще, ну и спасибо что пришли, увидимся на пересдаче.

Да, есть термин CUDA core, например. Про GPU говорят, что там столько-то тысяч ядер. Эти ядра не имеют никакого отношения к ядрам CPU. Это совсем другая сущность. И чтобы эти сущности не путать, мы тут вообще не будем использовать слово ядра применительно к GPU. А из чего тогда состоит GPU? Вот посмотрим на такую упрощенную картинку.



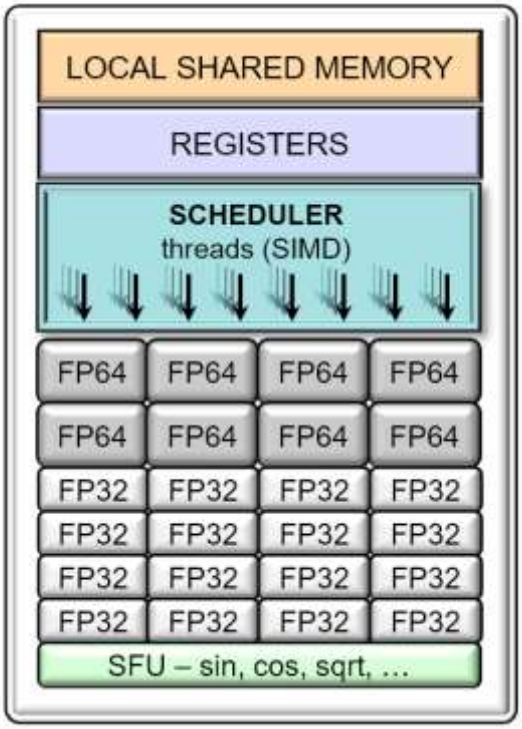

Разберемся что тут понарисовано. Имеется набортная оперативная память – global memory. Ее там (сейчас) может быть несколько десятков гигабайт. Такая же в общем-то SDRAM память, как у CPU, только с очень широкой шиной, не 64 бита, а 4096 бит, например (у 4 модулей HBM2). Или не с такой уж широкой шиной, но с эффективной частотой в разы выше, чем у CPU (GDDR). Но латентность доступа обычно даже побольше, чем на CPU. Память нарисована в виде плиточек, наваленных одна на другую, что как бы символизирует, что сейчас часто используется многослойная 3D-stack память. Рядом нарисована кэш память. Тоже примерно все как на CPU.

Дальше нарисованы потоковые мультипроцессоры – SM (Stream Multiprocessor). Ну это больше в терминологии NVIDIA. В терминологии OpenCL и AMD – Compute Unit (CU). Будем считать синонимами, SM = CU. Обычно на GPU этих SM намного больше, чем на картинке, но не будем усложнять. Сколько-то SM. Глобальная память у них – общая. Аналогом процессорного ядра в какой-то степени можно считать SM. SM, как и ядро CPU, выполняет поток (потоки) команд. Этим занимается планировщик, aka scheduler. На нем нарисовано много стрелочек, много наборов по много стрелочек. Стрелочки символизируют некоторую сущность, которую кто как ни называет - нити, потоки, threads, work item, рабочие единицы… Разберемся с этим далее. Внутри SM много исполнительных устройств для арифметики с плавающей точкой – FPU.

А какой тут вообще параллелизм? Каждый SM выполняет свой поток команд, есть общая память, значит, это – Shared Memory MIMD? MIMD да не MIMD. В чем отличия. Память общая, но не предполагается кэш когерентности, SM-ки обычно не могут напрямую взаимодействовать между собой, не могут иметь общие переменные, не могут синхронизироваться.

А что внутри SM и чем это отличается от ядра CPU? В CPU тоже есть какой-то scheduler, тоже есть SMT – аппаратная поддержка выполнения множественных потоков. Вот на картинке есть планировщик, в нем куча стрелочек. Стрелочки – нити, не имеют ничего общего с нитями на CPU. Это чистый SIMD. У всех стрелочек на картинке – один поток команд, но множественные потоки данных. Поэтому и множественные стрелочки. Выполняются как бы много экземпляров одной и той же программы. На исполнение поступает одна команда, но сразу с множественными наборами аргументов. Эти потоки хранят свои данные в общем регистровом файле. Там есть локальная общая память, через которую потоки могут

обмениваться данными между собой. Но что значит обмениваться, если поток команд как бы один? Ну это значит, что на следующей команде можно воспользоваться данными с предыдущей команды, посчитанными другим потоком.




У scheduler-а нарисовано много групп стрелочек. Это символизирует, что этот планировщик SM может оперировать сразу кучей групп потоков, по аналогии с SMT в CPU ядре. Если одна группа потоков ждет данные с памяти, SM может ее задвинуть ждать долгие сотни тактов поступления данных из глобальной памяти и поставить на исполнение другую группу, у которой все данные уже имеются. Это дает мощный механизм сокрытия латентности. SM может держать много групп потоков, пряча тем самым накладные расходы на доступ к памяти за вычислениями.

На SM есть множественные FMA FPU. На картинке FPU для двойной точности (double precision, 64 bit) – FP64, для одинарной (single precision, 32 bit) – FP32. Там есть и целочисленные ALU, и всякие прочие LSU (Load-Store Unit), но не будем усложнять картинку. Что мы видим: сингловых FPU сильно больше, чем дабловых. В GP GPU обычно соотношение 1:2. В игровых карточках может быть 1:16, 1:32, даже 1:64. Но, как ни странно, это обычно не сильно влияет на производительность в двойной точности. А почему, кстати? Не знаете? Тогда, GOTO глава 2. Также еще там есть SFU, которое делает всякие там синусы, косинусы и прочие корни. Но FMA юнитов много, а SFU совсем мало. Значит, математические функции эти всякие – тормозят. Стараемся в коде от них избавиться. Ой, а кто делает деление? А деление обычно никто не делает. Оно программно сводится к умножениям-сложениям, вследствие чего дико тормозит. Гораздо более дико, чем на CPU. Вывод: деление на GPU – это злобный враг производительности. Минимизируем операции деления, вводим промежуточные переменные, расставляем скобки, но выпиливаем лишние деления любой ценой.

Получилось много слов, но пока еще как-то мутно и непонятно, как эта штука работает. Хаос, мешанина и сумбур. Что мы смогли пока понять: на SM одна команда задействует сразу много FPU. Это прямо-таки SIMD параллелизм. Значит GPU – это какой-то смешанный MIMD-о-SIMD? Ну и не совсем MIMD, а какой-то недоMIMD. НедоМИМДоСИМД. Что это за конструкция такая? А эта конструкция – потоковая обработка. Поймем, что это такое, и сразу все встанет на свои места.

А что обычно называют ядрами применительно к GPU? Ядром обычно называют отдельное исполнительное устройство, точнее, комплект юнитов, например, FPU + ALU. Если на SM-ке по 32 FPU и 32 целочисленных ALU – говорят, что там 32 ядра. Но, конечно, никакие это не ядра в терминологии CPU. Все эти “ядра” могут выполнять одновременно только **одну**, ровно одну и ту же команду. А ядро CPU – это, по сути, отдельный процессор, со своим потоком команд, кэш памятью, набором исполнительных устройств.

## 5.2 Потоковая обработка на OpenCL

Потоковая обработка, aka stream processing, что это вообще такое? Вся суть состоит в следующем. Пусть есть цикл с большим количеством итераций (сотни тысяч, миллионы), в котором выполняется какая-то вычислительная нагрузка, какое-то неизвестно что:

```
for(int i=0; i<N; i++){
    WhateverWorkload(i);
}
```

Пусть WhateverWorkload одинакова для всех итераций, то есть она делает что-то одно и тоже, но с разными наборами данных.

Пусть итерации этого цикла независимы по данным между собой. То есть, итерации цикла не используют на вход никакие результаты других итераций этого же цикла.

Пусть итерации этого цикла можно выполнить в любом порядке, и это ничего не испортит.

Назовем WhateverWorkload – элементарным заданием. Имеется множество однородных (то есть одинаковых, выполняющихся одним и тем же программным кодом) независимых по данным между собой элементарных заданий, которые можно выполнить в произвольном порядке. Вуаля! Это – потоковая обработка. Даем девайсу множество элементарных заданий, и этот девайс сам разбирается, как и в каком порядке эти задания выполнить на множестве своих исполнительных устройств.

Такие функции типа WhateverWorkload будем называть kernel, что в переводе означает… ядро! Что? Опять? Ядро!? И опять в каком-то совершенно другом смысле! “@#$%^&!!!” (*неценз., груб., разг.*) – скажете вы, и будете правы. Ладно, забыли слово ядро, будем называть такие функции – кернелами. Итак, вся суть. Был цикл:

```
for(int i=0; i<N; i++) WhateverWorkload(i);
```

Выкинули цикл, то, что было внутри цикла, назвали кернелом:

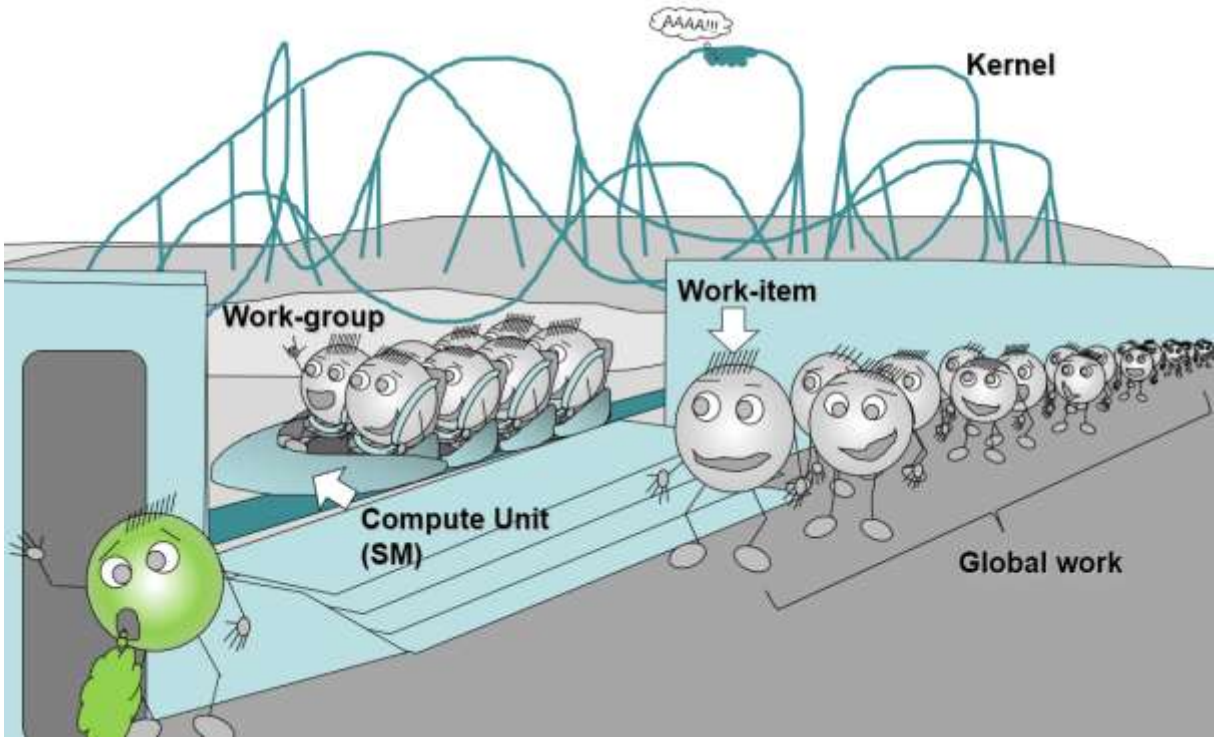
```
__kernel tSomeType WhateverWorkload(…);
```

Скармливаем эту штуку девайсу, указав, в каком диапазоне меняется счетчик i. Ну и все в общем-то. Разве сложно? Вроде бы нет. Но мы то знаем, что ничего никогда не бывает просто…

Нижеследующее эпическое полотно метафорически символизирует суть потоковой обработки на графическом процессоре. Рассмотрим его подробнее.

У нас есть большая очередь условных человечков (ну не будем придираться к художественным способностям автора), которых нужно покатать на аттракционе. Это глобальная очередь элементарных заданий, которые называются worki-item в OpenCL терминологии, рабочие единицы, так сказать. Эти задания нужно выкатать на каруселька, то есть для каждого из них выполнить **kernel** программу. Каждому человечку, то есть work-item, присвоен свой глобальный номер в этой большой очереди.

У аттракциона есть вагонетки. В каждой вагонетке одновременно катается какое-то количество человечков. Эта вагонетка – compute unit, на GPU compute unit-ом будет потоковый мультипроцессор, SM.




Группка человечков, которая катается на вагонетке, то есть одновременно обрабатывается одним SM, это локальная рабочая группа – **work-group**. Внутри рабочей группы у каждого человечка тоже есть свой локальный номер. Рабочая группа разделяет ресурсы одного SM – регистровый файл, общую локальную память, исполнительные устройства. Внутри рабочей группы человечки могут общаться, обмениваться информацией, болтать, делать селфики, ~~затошнить друг другу одежду~~. Размер рабочей группы не обязательно должен быть равен количеству “нитей”, которые одновременно исполняет scheduler SM. Если размер больше, то рабочая группа будет выполняться по частям за несколько подходов. Если меньше, то часть ресурсов SM будет простаивать.

Человечки в вагонетке едут все вместе по рельсам ровно по одной и той же траектории. Вообще все человечки в очереди поедут по одной и той же траектории по одним и тем же рельсам. Эти рельсы – **kernel** функция.

Нет, на самом деле, конечно, поток команд у work-item может различаться. Например, в кернеле может быть условный оператор, который у части айтемов (*жарг.* от work-item) в рабочей группе пойдет по одной ветке, у другой части – по другой. Такое допустимо. Но. Compute unit, который SM, не умеет такое исполнить. Поэтому он сначала возьмет одну ветку, выполнит ее, в это время айтемы, которые пошли по другой ветке, будут простаивать (или фиктивно выполнять то же самое и выкидывать результат). Потом будет выполнена другая ветка. Выполнение кернела – сериализовалось, то есть стало последовательным по разным вариантам. Это, конечно же, неэффективно. Тоже самое будет, если в кернеле есть цикл, число итераций которого различается для айтемов внутри группы. Айтемы, у которых меньше итераций, просто будут простаивать. Значит, будут простаивать ресурсы SM, что тоже снижает эффективность.

Теперь открываем документацию от стандарта OpenCL. Можно взять для начала что-то самое старенькое, простенькое, например, версию 1.2 или даже 1.1.

https://www.khronos.org/opencl/ – сайт OpenCL

https://www.khronos.org/registry/OpenCL/ – страница со спецификациями стандартов.

Там есть (как у OpenMP) полная спецификация
https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf
и краткая сводка
https://www.khronos.org/registry/OpenCL//sdk/1.2/docs/OpenCL-1.2-refcard.pdf

Открываем спецификацию и читаем вводную часть – разделы 1, 2, 3. Там всего-то страничек 10–20.

Итак, что мы там узнали. OpenCL, получается, состоит из двух частей: API на стороне хоста (то есть в программе на CPU) и исходный код кернелов для выполнения на девайсах. К программе на CPU мы просто подключаем заголовочный файл и линкуем библиотеку OpenCL. Нам становятся доступны вызовы функций, которые организуют и управляют вычислениями на девайсах. Кернел программы, то есть исходный код того, что будет выполняться на девайсах, мы пишем по стандарту OpenCL, в котором за основу взят просто язык C (или C++), дополненный некоторыми интринсиками, спецификаторами, типами данных. Кернел код мы кормим компилятору OpenCL на рантайме хостовой программы API-шным вызовом. Мы можем, например, хранить кернел-код в файлах на диске и зачитывать при запуске программы. Вот, собственно, и все. Вспомним, какие там нам попались основные сущности.

**Platform** – платформа, к которой относятся устройства. Например, у меня на ноуте сейчас две платформы – Intel и NVIDIA.

**Device** – вычислительное устройство конкретной платформы. Например, у меня два девайса платформы Intel – процессор и интегрированная на нем видеокарта (а могут и как две отдельных платформы числиться, зависит от дров), и одно устройство платформы NVIDIA – дискретная видеокарта.

**Context** – пока вообще не заморачиваемся, что это такое, восприняли его как данность, создали и используем.

**Command Queue** – очередь команд, их можно насоздавать несколько для девайса. В очереди можно ставить запуск кернелов и операции копирования с девайса на хост и обратно.

**Program** – исходный код программы на OpenCL, в которой живут кернелы.

**Kernel** – кернел функция, которую можно запускать на девайсе.

**Kernel Argument** – аргументы кернела.

**Buffer** – выделенная область памяти на устройстве. На хосте мы делаем new или malloc. На девайсе – создаем буфера. Важно, что это не указатель. Это handle. Просто айдишник выделенного буфера в системе. Адресная арифметика к буферу неприменима. Массив указателей – неосуществим.

**Work-item** – рабочая единица, aka элементарное задание, нить, поток, …

**Work-group** – локальная рабочая группа из work-item, которые скучкованы на одном compute unit.

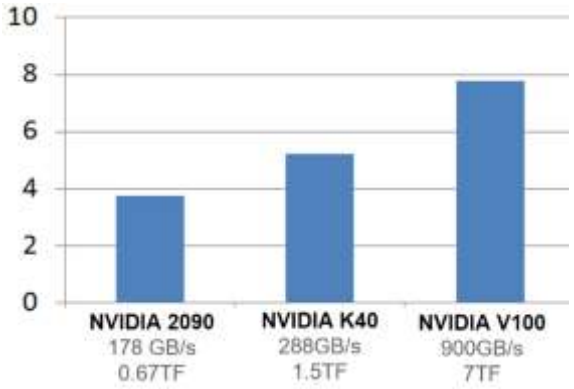
У OpenCL обычно немного длинная и сложная инициализационная часть, которая поначалу всех отпугивает. Но в ней нет никаких проблем, ее один раз сделать или передрать откуда-нибудь, и все. Она сидит себе в функции, есть не просит, никого не трогает. Как это работает:

- Запрашиваем доступные платформы: clGetPlatformIDs
- Перебираем платформы, выбираем нужную: clGetPlatformInfo
- Запрашиваем девайсы платформы: clGetDeviceIDs
- Перебираем девайсы, выбираем нужный: clGetDeviceInfo
- Создаем контекст на выбранном девайсе: clCreateContext

- Создаем очереди exec и comm под контекстом: clCreateCommandQueue
- Загружаем исходный kernel код под контекст: clCreateProgramWithSource
- Компилируем kernel код: clBuildProgram
- Смотрим лог, правим ошибки в коде, пробуем еще: clGetProgramBuildInfo
- Достаем из программы кернелы: clCreateKernel
- Создаем нужные буфера: clCreateBuffer
- Загружаем нужные данные в буфера на девайс: clEnqueueWriteBuffer
- Задаем аргументы нужного кернела: clSetKernelArg
- Задаем размер рабочей группы (число заданий) и выбираем оптимальный размер локальной группы.
- Запускаем кернел в очередь: clEnqueueNDRangeKernel
- Ждем завершения выполнения кернела, если надо: clFinish
- Выгружаем посчитанные данные с девайса: clEnqueueReadBuffer
- Копирование данные из буфера в буфер, если надо: clEnqueueCopyBuffer
- Освобождаем буфера: clReleaseMemObject

Конечно, стало еще более непонятно. Что вообще происходит!? Какие буфера, какие платформы, что за дичь? На CPU взял цикл и все посчитал, а тут что за хаос? Дальше будем разбирать примеры, хаос немного упорядочится.

## 5.3 Доступ к памяти на GPU



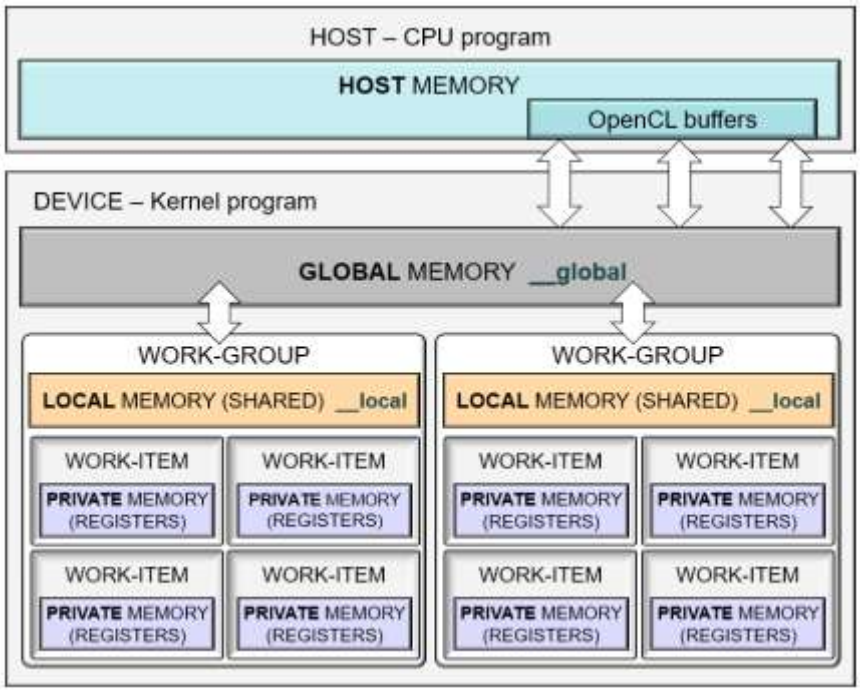

Тут все те же проблемы. Производительность растет быстрее, чем пропускная способность памяти. На картинке показано, что происходит с соотношением производительности и бэндвиса. А латентность доступа остается дико высокой, транзакция выходит сотни тактов. В общем, с памятью надо обращаться очень бережно.

По сравнению с CPU есть некоторые отличия. Наверное, надо начать с того, что приходится иметь дело аж с четырьмя видами памяти, то есть с разными адресными пространствами. В программе на стороне девайса, в которой мы делаем кернелы на производном от C (C++) языке OpenCL, у адресных пространств есть спецификаторы. То есть, описывая переменную, мы не только указываем ее тип, но и говорим, к какой памяти она должна существовать. Описывая указатели, мы, естественно, должны указать, к какой памяти он относится.

- Память хоста – **host memory**. Это просто память в программе для CPU. В программе стороны девайса, то есть в кернелах, эта память недоступна и никак не видна.
- Глобальная память девайса – **global memory**. Между памятью хоста и глобальной памятью девайса данные перемещаются с помощью функций копирования. На OpenCL со стороны хоста эта память представлена лишь “ручкой” (handle), то есть идентификатором, порядковым номером, который был присвоен буферу при выделении. Это тип cl_mem. Сам адрес в памяти девайса в программе со стороны хоста недоступен. В программе по стандарту OpenCL на стороне девайса у этого

адресного пространства спецификатор __global. Например, указатель на буфер из даблов на стороне девайса будет **__global** double *.

- Локальная общая память – **local memory**. У GPU это та маленькая память (обычно несколько десятков килобайт), которая находится на SM. Эта память общая для локальной рабочей группы, поскольку work-item-ы (“нити”) локальной группы выполняются на одном SM. Эта память обычно такая же быстрая, как регистровый файл. В этой памяти так называемые нити могут иметь общие переменные. На стороне хоста эта память никак не представлена. На стороне девайса это спецификатор **__local**. Например, __local int i; – общая переменная для всей локальной группы.
- Память в частной собственности – **private memory**. Это память нити, aka work-item-a. На стороне хоста эта память никак не представлена. На стороне девайса это просто отсутствие спецификатора. Если мы не указали __local или __global, то переменная относится к частной памяти нити.




Первое, что надо помнить, локальной памяти и регистровой памяти – очень мало. Ее надо экономить. Минимизируем количество переменных, ограничиваем их область существования, делаем структурные блоки, и т.д. Когда мы делаем в кернеле какую-то переменную, надо понимать, что мы делаем по экземпляру каждой нити локальной рабочей группы. Например, если у нас размер локальных групп 64, то, написав “int i;”, мы израсходовали сразу 256 байт.

Но самое главное и основное – это доступ к глобальной памяти девайса. Транзакция с памятью, то есть что-то оттуда взять или записать, выходит очень дорого из-за высокой латентности. Зачитать какую-то произвольную кэш линию может выйти в сотни тактов. А кэш линии на девайсе обычно пошире, чем у CPU. На девайсе кэш линия обычно 128 байт. Мы же помним из главы 2, что чтение из памяти происходит кэш линиями? Что даже если мы взяли из памяти один маленький байт, из памяти будет считано и помещено в кэш все 128 байт. Поэтому, если мы считали кэш линию, то мы должны ее максимально полно использовать. И максимально как можно быстрее, т.к. кэша по сравнению с CPU очень (очень) мало, линия быстро заэкспайрится. Если с транзакции пришла кэш линия 128 байт, а мы взяли оттуда,

скажем, всего одно значение двойной точности 8 байт, то это супер-мега ~~@*(неценз. груб.)~~ как неэффективно. Это как погнать авиалайнер, в котором всего один пассажир. Поэтому, как и в случае CPU, стараемся размещать данные в памяти компактно, чтобы максимально использовать данные в кэш линиях. Кроме того, как и на CPU, гораздо (гораздо) эффективнее выбирать данные подряд. Тогда мы сможем использовать мощный бэндвис девайса, который (сейчас) в разы больше, чем у CPU.

Первое, про что надо знать и понимать, это – слияние доступа, aka coalescing. Рассмотрим простейший пример, уже знакомую нам операцию axpby (то есть линейную комбинацию двух векторов: **x** = a**x**+b**y**, где **x**, **y** – вектора, a, b – скалярные значения). Исходный последовательный код на CPU:

```
void axpby(int n, double *x, double *y, double a, double b){
    for(int i=0; i<n; ++i) x[i]=a*x[i]+b*y[i];
}
```

Кернел на OpenCL будет выглядеть так:

```
__kernel void  knlAXPBY(int n, __global double* x, __global double *y,
                        double a, double b){
    const int i = get_global_id(0);
    if(i>=n) return; // Intel CPU OpenCL бывает плохо от return, приходится делать i=n
    x[i] = a*x[i] + b*y[i];
}
```

Что изменилось? Добавились спецификаторы: `__kernel` говорит о том, что это кернел функция, которую можно вызывать с хоста; `__global` говорит, что это глобальный буфер. Еще пропал цикл. Теперь вместо цикла у нас множество заданий. Этими заданиями стало тело цикла. А номер итерации i – это теперь номер задания, который мы берем интринсиком `get_global_id(0)`.

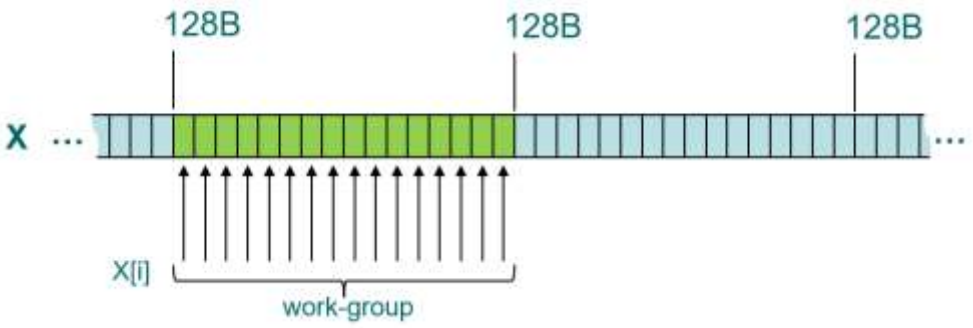
Так что же такое coalescing? А вот что. Пусть размер рабочей группы мы поставили 16. Это значит, что n заданий, где n – размер векторов, будут поступать на исполнение группами по 16. Что происходило на CPU, когда мы обращались к x[i]? Происходило просто считывание кэш линии, содержащей 8 байт по адресу x+i (ну или двух кэш линий, если данные не выровнены). А что будет на GPU, когда в кернеле мы обращаемся к x[i]? Будут считываться сразу 16 значений для всей рабочей группы. Там же у нас SIMD режим внутри SM. Значит, в этом кернеле сначала считываются сразу 16 значений из x, потом сразу 16 значений из y, начиная с позиции первого ворк-айтема в группе.

Получается, что 16 потоков одновременно обратились к памяти, запросив 16*8 = 128 байт данных. Все данные попали в одну кеш линию. Значит все 16 потоков получили свои данные за одну транзакцию с памятью. Это и есть – **coalescing**, слияние доступа, когда много потоков получили свои данные за одну транзакцию с памятью. Запросы от потоков рабочей группы как бы слиплись в один. Выборка идет подряд, с юнит страйдом, да еще и вся кэш линия используется. Что может быть прекраснее?

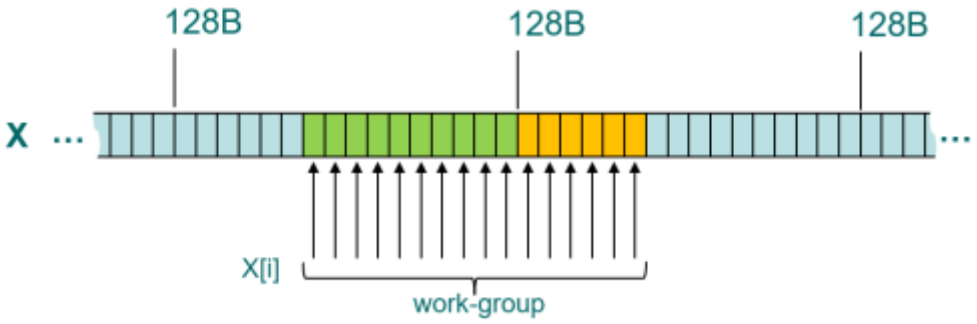
Вспомним еще одно понятие – **выравнивание**, aka alignment. Тут у нас рабочие группы были из 16 нитей, начало буфера обычно выровнено по кэш линии (то есть адрес в памяти кратен размеру кэш линии). Значит, каждая рабочая группа считывает свой блок из 16 даблов, начало которого тоже выровнено по ширине кэш линии. То есть, каждая рабочая группа делает зачитку из буфера по адресу, кратному ширине кэш линии. Это позволяет получить все данные

рабочей группы за минимальное число транзакций. Если рабочая группа начнет зачитку тех же 128 байт, но с адреса некратного ширине кэш линии, то есть доступ будет не выровненный, то ей уже по-любому понадобится 2 кэш линии.

Вот рабочая группа считывает свой набор дабловых x[i], выровненный доступ, используется вся кэш линия:




Вот то же самое, но доступ не выровненный, на тот же комплект иксов понадобилось 2 кэш линии. В общем-то не так страшно, следующая группа с какой-то вероятностью подберет часть кэш линии, и кэш линия заюзается (*жарг.* от used – использована) полностью, но все равно будет немного suboptimal.




Ну и небольшой нюанс. Операция чтения происходит не прям для всей рабочей группы одновременно, а по частям. Количество нитей, которые обращаются к памяти одновременно, определяется возможностями SM. Например, на GPU NVIDIA обычно (сейчас) чтение происходит блоками по 16 нитей, так называемыми (полу)варпами (warp) в терминологии NVIDIA. Но сути это не меняет. Если бы мы сделали размер рабочей группы 64, то данные бы считывались просто за 4 подхода по 16 нитей, все равно получилось бы, что все кэш линии заполнены, и каждая кэш линия приносит данные для запросов от 16 нитей. Полный coalescing.

Coalescing – это то, на что следует обращать внимание в первую очередь. Data locality и unit stride – наше все, как и на CPU. Но немного по-другому. Как мы выяснили, за каждым x[i] стоит массовый групповой запрос от всей рабочей группы. Имеем это в виду.

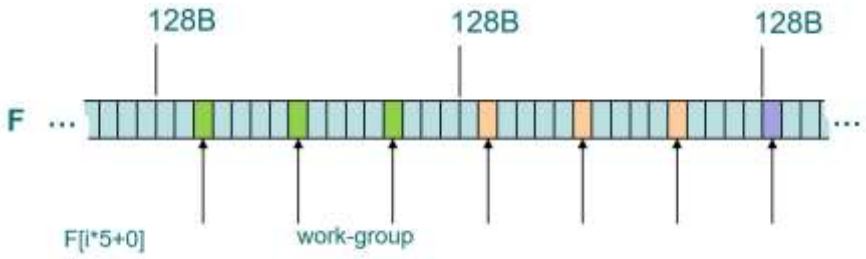
Но с векторами все просто. А у нас в алгоритме в ячейках заданы много сеточных функций. Приходится иметь дело с блочными векторами. В главе 2 разбирали подходы AOS и SOA. Посмотрим, как с этим обстоят дела на GPU. Вспомним наш обобщенный сеточный метод из предыдущих глав. Пусть у нас в ячейках заданы по 5 переменных – плотность R, три компоненты скорости U, V, W, и давление P. Как нам это добро хранить? Рассмотрим примерчик. Исходно на CPU была какая-то такая последовательная операция (считаем полную энергию в ячейках, хотя, не важно, что именно):

```
    for(int i=0; i<N; ++i){ // F - блочный массив с блоками по 5 переменных
        double R = F[5*i + 0], U = F[5*i + 1], W = F[5*i + 2],
               V = F[5*i + 3], P = F[5*i + 4];
        E[i] = P/(gamma - 1.0) + (U*U + V*V + W*W)*R/2.0; // считаем в массив энергии
    }
```

Тут у нас используется подход AOS – array of structures. Ну структура не в явном виде как тип данных, а просто как блок в блочном векторе. На CPU блок переменных читается компактно с юнит страйдом, значения переменных идут подряд, все хорошо. Тут еще смущают тормозные операции деления и вычитание ненужное. Оптимизирующий компилятор, возможно, это разрулит, но не будем рисковать, предварительно посчитаем или впишем макросом константное значение от `1.0/(gamma - 1.0)` в `inv_gamma_m1`. Поправим код и затащим этот цикл в кернел на GPU:

```
__kernel void CalcE(int N, __global double *F, __global double *E){
    const int i = get_global_id(0);
    if(i >= N) return; // проверяем размер, а то число заданий обычно должно быть
    // кратно размеру рабочей группы, а N - вовсе не обязано.
    double R = F[5*i + 0], U = F[5*i + 1], W = F[5*i + 2],
           V = F[5*i + 3], P = F[5*i + 4];
    E[i] = P*inv_gamma_m1 + 0.5*(U*U + V*V + W*W)*R;
}
```

На GPU у нас уже не получается юнит страйд. Как мы зачитываем массив F? Вот так:




Во-первых, каждая переменная считывается сразу для всей группы. Пусть размер рабочей группы 16. Значит, сначала читаются сразу 16 штук R с шагом 5 (то есть 40 байт), потом 16 штук U, и так далее. То есть используется что-то в районе 20% от кэш-линии на каждой транзакции. Во-вторых, у нас не получится читать выровнено, потому что 5 даблов, 40 байт, это ни к селу, ни к городу, не кратно ширине кэш линии. Выглядит как-то suboptimal.

Ну тут данных мало, если повезет, то кэш линии не заэкспайрятся, и будет реюз (*жарг.* reuse – повторное использование) на следующей переменной. Тогда производительность просядет не сильно. Но если данных побольше, то в кэш может и не влезть, уж больно он там маленький. Вывод: лучше использовать подход SOA и “транспонировать” этот блочный массив F из N×5 в 5×N:

```
__kernel void CalcE(int N, __global double *F, __global double *E){
    const int i = get_global_id(0); if(i >= N) return;
    double R = F[i], U = F[i + N], W = F[i + 2*N],
           V = F[i + 3*N], P = F[i + 4*N];
    E[i] = P*inv_gamma_m1 + 0.5*(U*U + V*V + W*W)*R;
}
```

Теперь доступ получился с юнит страйдом, кэш линии хорошо заполнены на каждом запросе по каждой переменной. А если еще сделать каждой переменной padding до кратного кэш линии размера, то доступ будет еще и выровненный. Посчитаем где-то на хосте:

```
int align_size = 128/sizeof(double);
int Npad = (N%align_size==0 ? N : N + align_size - N%align_size);
```

Приляпали довесок, чтобы размер делился на ширину кэш линии – 128 байт. Выделяем буфер F по размеру Npad вместо N. Добавим Npad в аргументы, чтобы в кернеле не делать лишних вычислений, и пропатчим доступ к F:

```
__kernel void CalcE(int N, int Npad, __global double *F, __global double *E){
    const int i = get_global_id(0);
    if(i >= N) return;
    double R = F[i], U = F[i + Npad], W = F[i + 2*Npad],
           V = F[i + 3*Npad], P = F[i + 4*Npad];
    E[i] = P*inv_gamma_m1 + 0.5*(U*U + V*V + W*W)*R;
}
```

Теперь, если буфер выровнен (то есть начинается с кратной кэш линии позиции), а обычно считается, что OpenCL буфер выровнен, то все 5 кусочков переменных будут выровнены и доступ к ним тоже будет выровнен, потому что размер блока данных рабочей группы кратен кэш линии (мы же, конечно, сделаем размер рабочей группы кратным 16?).

Получилось, что доступ выровненный, плотненький, целыми кэш-линиями. Идиллия. Но можно еще улучшить, если транспонировать не весь блочный массив целиком, а нарезать его на куски по размеру рабочей группы и их транспонировать. То есть не весь блочный массив N×5 транспонируем в 5×N, а “нарезаем” его маленькие на части по W×5, где W – размер рабочей группы, и транспонируем эти части в 5×W. Тогда кэш линии рабочей группы еще и будут идти подряд, за счет чего уменьшатся потери на латентность.

Вообще на старых устройствах типа NVIDIA 2050 (Fermi) эти манипуляции с транспонированием из AOS в SOA давали заметный прирост производительности, у нас получалось до полутора раз. На Кеплере (NVIDIA K40) уже процентов 20 – 30. На следующих поколениях выигрыш еще уменьшился, на новых моделях уже почти незаметно. Но суть устройства доступа к памяти рабочей группой все равно лучше иметь в виду.

А если мы не хотим ничего транспонировать? Хотим, чтобы формат данных был одинаковым с CPU версией, чтобы можно было копировать буфера без всякой конвертации из одного формата хранения в другой. Вернемся к исходному блочному вектору `F[5*i + j]` вида AOS. Зачитаем весь блок данных рабочей группы из глобальной памяти к себе в уютную локальную память самым удобным для чтения образом. А потом будем использовать эти данные из локальной памяти уже удобным для вычислений образом.

```
define WGS 32 // work-group size - используем какой-то фиксированный размер
// если надо менять, припечатываем к коду нужное значение перед компиляцией
__kernel void CalcE(int N, __global double *F, __global double *E){
    const int iloc = get_local_id(0); //номер внутри группы
    __local double f[5*WGS]; // локальная копия данных группы из блочного массива F
    {
        int grp_offset = get_group_id(0)*WGS; //смещение группы в общем массиве
        __global double* gptr = &F[5*grp_offset];
        int grp_block_size = (grp_offset + WGS <= N) ? //проверяем, чтобы не вылететь
                             5*WGS : 5*(N - grp_offset);
        for(int I=iloc; I<grp_block_size; I+=WGS) // читаем с юнит-страйдом
            f[I] = gptr[I]; // как ни странно +=WGS! ведь читает сразу вся группа
        barrier(CLK_LOCAL_MEM_FENCE); // барьер группы
    }
    const int i = get_global_id(0);
    if(i >= N) return;
    double R = f[5*iloc + 0], U = f[5*iloc + 1], W = f[5*iloc + 2],
           V = f[5*iloc + 3], P = f[5*iloc + 4];
    E[i] = P*inv_gamma_m1 + 0.5*(U*U + V*V + W*W)*R;
}
```

## 5.4 Простейшие примеры с исходным кодом

В интернетах можно найти полно примеров.

Вот тут, например: https://developer.nvidia.com/opencl

### 5.4.1 AXPBY

Тут рассмотрим что-то самое простейшее – уже известную нам операцию axpby, то есть **x** = a**x**+b**y**, где **x**, **y** – вектора, a, b – скалярные значения. Пример приводится целиком, включая ~~убогий~~ минималистичный код инициализации OpenCL.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <CL/cl.h> // OpenCL

int main(int /*argc*/, char** /*argv*/){

  // Входные параметры
  const cl_int devID = 0; // Номер нужного девайса
  const char *platformName = "NVIDIA CUDA"; //Нужная платформа // "Intel(R) OpenCL" // "AMD"

  // OpenCL переменные
  cl_context clContext; // OpenCL контекст
  cl_command_queue clQueue; // OpenCL очередь команд
  cl_program clProgram; // OpenCL программа
  cl_int clErr; // код возврата из OpenCL функций

  {// Инициализация OpenCL
    printf("OpenCL initialization\n");

    //PLATFORM
    // узнаем число платформ в системе
    cl_uint platformCount=0;
    clErr = clGetPlatformIDs( 0, 0, &platformCount);
    if(clErr != CL_SUCCESS){ printf("clGetPlatformIDs error %d\n", clErr); exit(1); }
    if(platformCount <= 0){ printf("No platforms found\n"); exit(1); }
    printf("clGetPlatformIDs: %d platforms\n", platformCount);

    // запрашиваем список платформ
    cl_platform_id *platformList = new cl_platform_id[platformCount];
    clErr = clGetPlatformIDs(platformCount, platformList, 0);
    if(clErr != CL_SUCCESS){ printf("clGetPlatformIDs error %d\n", clErr); exit(1); }

    // ищем нужную платформу
    #define STR_SIZE 1024 // размерчик буфера для названия платформ
    char nameBuf[STR_SIZE]; // буфер для названия платформы
    cl_int platform_id=0;
    for(cl_uint i=0; i<platformCount; i++){
      clErr = clGetPlatformInfo(platformList[i], CL_PLATFORM_NAME, STR_SIZE, nameBuf, 0);
      if(clErr != CL_SUCCESS){ printf("clGetPlatformInfo error %d\n", clErr); exit(1); }
      printf("  Platform %d: %s\n", i, nameBuf);
      if(!strcmp(platformName, nameBuf)) platform_id=i; // found
    }
    if(platform_id<0){ printf("Can't find platform\n"); exit(1); }
    printf("Platform %d selected\n",platform_id);

    // DEVICE
    // узнаем число девайсов у выбранной платформы
    int deviceCount = 0;
    clErr = clGetDeviceIDs(platformList[platform_id], CL_DEVICE_TYPE_ALL,
                           0, NULL,(cl_uint *) &deviceCount);
```

```
if(clErr != CL_SUCCESS){ printf("clGetDeviceIDs error %d\n", clErr); exit(1); }
printf("%d devices found\n", deviceCount);
if(devID >= deviceCount){ printf("Wrong device selected: %d!\n", devID); exit(1); }

// запрашиваем список девайсов у выбранной платформы
cl_device_id *deviceList = new cl_device_id[deviceCount]; // list of devices
clErr = clGetDeviceIDs(platformList[platform_id], CL_DEVICE_TYPE_ALL,
            (cl_uint)deviceCount, deviceList, NULL);
if(clErr != CL_SUCCESS){ printf("clGetDeviceIDs error %d\n", clErr); exit(1); }
delete[] platformList; // больше не нужно

// печатаем девайсы платформы
for(int i=0; i<deviceCount; i++){
  clErr = clGetDeviceInfo(deviceList[i], CL_DEVICE_NAME, STR_SIZE, nameBuf, 0);
  if(clErr != CL_SUCCESS){ printf("clGetDeviceInfo error %d\n", clErr); exit(1); }
  printf("  Device %d: %s \n", i, nameBuf);
}

// CONTEXT
clContext = clCreateContext( NULL, 1, &deviceList[devID], 0, 0, &clErr);
if(clErr != CL_SUCCESS){ printf("clCreateContext error %d\n",clErr ); exit(1); }

// COMMAND QUEUE
clQueue = clCreateCommandQueue(clContext, deviceList[devID], 0, &clErr);
if(clErr != CL_SUCCESS){ printf("clCreateCommandQueue %d\n",clErr ); exit(1); }

// PROGRAM
const char *cPathAndName="kernel.cl";  // файл с исходным кодом кернелов
printf("Loading program from %s\n", cPathAndName);

// сюда можно напихать каких нужно дефайнов, чтобы они подставились в программу
const char *cDefines=" /* add your defines */ ";

char * cSourceCL = NULL; // буфер для исходного кода
{ // читаем файл
  FILE *f=fopen(cPathAndName, "rb");
  if(!f){ printf("Can't open program file %s!\n", cPathAndName); exit(1); }
  fseek(f, 0, SEEK_END); // считаем размер
  size_t fileSize = ftell(f);
  rewind(f);
  int codeSize = fileSize + strlen(cDefines); // считаем общий размер: код + дефайны
  cSourceCL = new char [codeSize + 1/*zero-terminated*/]; // выделяем буфер
  memcpy(cSourceCL,cDefines,strlen(cDefines)); // подставляем дефайны
  size_t nd = fread(cSourceCL+strlen(cDefines),1,fileSize,f); // читаем
  if(nd != fileSize){ printf("Failed to read program %s!\n", cPathAndName); exit(1); }
  cSourceCL[codeSize]=0; // заканчиваем строку нулем!
}
if(cSourceCL == NULL){printf("Can't get program from %s!\n", cPathAndName); exit(1); }

// сдаем исходники в OpenCL
size_t szKernelLength = strlen(cSourceCL);
clProgram = clCreateProgramWithSource(clContext, 1, (const char **) &cSourceCL,
                                      &szKernelLength, &clErr);
if(clErr != CL_SUCCESS){printf("clCreateProgramWithSource error %d\n",clErr ); exit(1);}

// компилим кернел-программу
printf("clBuildProgram... ");
clErr = clBuildProgram(clProgram, 0, NULL, "-cl-mad-enable", NULL, NULL);
printf("done\n");

// запрашиваем размер лога компиляции
int LOG_S=0;
clErr = clGetProgramBuildInfo(clProgram, deviceList[devID], CL_PROGRAM_BUILD_LOG,
                              0, NULL, (size_t*)&LOG_S);
```

```
  if(clErr != CL_SUCCESS){ printf("clGetProgramBuildInfo error %d\n", clErr);exit(1); }
  if(LOG_S>8){ // если там не пусто - печатаем лог
    char *programLog= new char[LOG_S];
    clErr = clGetProgramBuildInfo(clProgram, deviceList[devID], CL_PROGRAM_BUILD_LOG,
                                  LOG_S, programLog, 0);
    if(clErr != CL_SUCCESS){ printf("clGetProgramBuildInfo error %d\n", clErr);exit(1); }
    printf("%s\n", programLog);
    delete[] programLog;
  }
  if(clErr != CL_SUCCESS){ printf("Compilation failed with error: %d\n",clErr); exit(1); }
  delete [] cSourceCL;
}

// KERNELS
printf("Creating kernels\n");
cl_kernel knlAXPBY; // x=ax+y
knlAXPBY = clCreateKernel(clProgram, "knlAXPBY", &clErr);
if(clErr != CL_SUCCESS){ printf("clCreateKernel knlAXPBY error: %d\n",clErr); exit(1); }
printf("   knlAXPBY\n");

const int N=1000123; // размер тестовых векторов

// BUFFERS
printf("Creating opencl buffers\n");
cl_mem clX = clCreateBuffer(clContext, CL_MEM_READ_WRITE, N*sizeof(double),NULL, &clErr);
if(clErr != CL_SUCCESS){ printf("clCreateBuffer clX error %d\n",clErr); exit(1); }
cl_mem clY = clCreateBuffer(clContext, CL_MEM_READ_WRITE, N*sizeof(double),NULL, &clErr);
if(clErr != CL_SUCCESS){ printf("clCreateBuffer clY error %d\n",clErr); exit(1); }
printf("Init done\n");

// TEST EXECUTION
// данные на стороне CPU
double *X = new double[N];
double *Y = new double[N];
const double a = 1.234, b = 3.456;

// заполняем вектора какой-то ерундой
for(int i=0; i<N; i++){
  X[i]=(double)(i%123)*(i%456);
  Y[i]=(double)(i%248)*(i%134);
}

// копируем вектора на девайс
clErr = clEnqueueWriteBuffer(clQueue, clX, CL_TRUE, 0, N*sizeof(double), X, 0,NULL,NULL);
if(clErr != CL_SUCCESS){ printf("clEnqueueWriteBuffer clX error %d\n", clErr); exit(1); }
clErr = clEnqueueWriteBuffer(clQueue, clY, CL_TRUE, 0, N*sizeof(double), Y, 0,NULL,NULL);
if(clErr != CL_SUCCESS){ printf("clEnqueueWriteBuffer clY error %d\n", clErr); exit(1); }

// выставляем параметры запуска
size_t lws = 128; // размер рабочей группы
size_t gws = N; // общее число заданий
if(gws%lws>0) gws += lws-gws%lws; // делаем кратное lws
// выставляем аргументы кернелу
clSetKernelArg(knlAXPBY, 0, sizeof(int), &N);
clSetKernelArg(knlAXPBY, 1, sizeof(cl_mem), &clX);
clSetKernelArg(knlAXPBY, 2, sizeof(cl_mem), &clY);
clSetKernelArg(knlAXPBY, 3, sizeof(double), &a);
clSetKernelArg(knlAXPBY, 4, sizeof(double), &b);
// отправляем на исполнение
clErr= clEnqueueNDRangeKernel(clQueue, knlAXPBY, 1, NULL, &gws, &lws, 0, NULL, NULL);
if(clErr != CL_SUCCESS){ printf("clEnqueueNDRangeKernel error %d\n",clErr); exit(1); }
clFinish(clQueue); // ждем завершения
```

```
  // делаем ту же операцию на хосте
  for(int i=0; i<N; ++i){
    X[i] = a*X[i] + b*Y[i];
  }

  // забираем результат с девайса
  double *R = new double[N];
  clErr = clEnqueueReadBuffer(clQueue, clX, CL_TRUE, 0, N*sizeof(double), R, 0, NULL, NULL);
  if(clErr != CL_SUCCESS){ printf("clEnqueueReadBuffer clX  error %d\n", clErr); exit(1); }

  // сравниваем с хостом, находим норму отличий
  double sum=0;
  for(int i=0; i<N; i++){
    sum += fabs(R[i]-X[i]);
  }
  printf("Test execution done\n  Error = %g\n", sum/N);

  delete [] R;  delete [] X;  delete [] Y;
  clErr = clReleaseMemObject(clX);
  if(clErr != CL_SUCCESS){ printf("clReleaseMemObject clX error %d\n", clErr); exit(1); }
  clErr = clReleaseMemObject(clY);
  if(clErr != CL_SUCCESS){ printf("clReleaseMemObject clY error %d\n", clErr); exit(1); }
}
```

Код кернел-файла kernel.cl:

```
#pragma OPENCL EXTENSION cl_khr_fp64 : enable // включаем поддержку даблов
// y = ax+y
__kernel void knlAXPBY(int n, __global double* x, __global double *y, double a, double b){
    const int gid = get_global_id(0);
    if(gid>=n) return;
    x[gid] = a*x[gid] + b*y[gid];
}
```

### 5.4.2 Сумма и другие редукции по вектору

С кернелом axpby все просто. Рассмотрим что-то, казалось бы, не менее простое – поиск суммы вектора. Последовательный вариант:

```
    double *sum = 0.0;
    for(int i=0; i<N; ++i) sum += X[i];
```

Всего 2 строчки! Теперь параллельный вариант с OpenMP:

```
    double sum = 0.0;
    #pragma omp parallel for reduction(+:sum)
    for(int i=0; i<N; ++i sum += X[i];
```

Уже целых 3 строчки. Параллельный вариант с MPI+OpenMP:

```
    double lsum = 0.0, gsum;
    #pragma omp parallel for reduction(+:lsum)
    for(int i=0; i<N; ++i) lsum += X[i];
    MPI_Allreduce(&lsum, &gsum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Получилось 4 строчки. А как же это будет выглядеть на OpenCL? На CUDA не особо лучше. В потоковую обработку этот цикл поиска суммы не вписывается, так как у итераций цикла есть зависимость. Каждая последующая итерация зависит от предыдущей. В MPI+OpenMP варианте выше мы сначала находим частичную сумму MPI процесса, потом ее собираем в общую сумму вызовом `MPI_Allreduce`. В OpenMP варианте делается примерно то же самое –

находится частичная сумма каждой нити, потом эти частичные суммы складываются, но от нас это скрыто за директивой. Может, на GPU тоже можно через частичные суммы? Попробуем. Запускаем число заданий, равное размеру вектора. Получим:

```
__kernel void knlSUM(int n, __global double* x, __global double *res){
    const int gid = get_global_id(0); if(gid>=n) return;
    double lsum = x[gid];
    ...
```

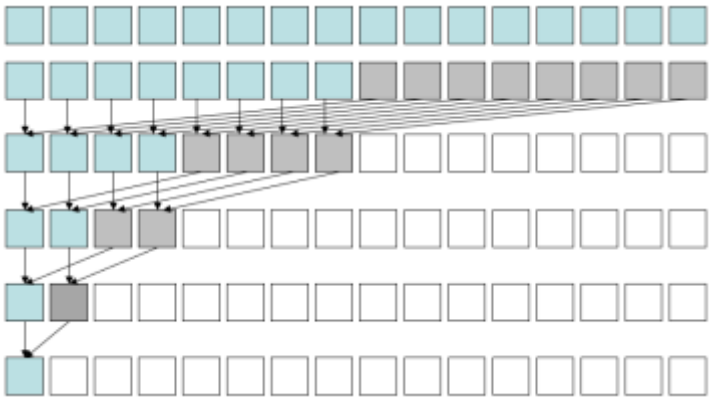
Э… Локальная сумма нити просто равна элементу вектора. Ну и что дальше с этим делать? Наверное, попробуем найти локальную сумму по рабочей группе. У рабочей группы же есть общая память. Пусть размер рабочей группы для простоты фиксирован – 256. Тогда сборка будет выглядеть так:

```
#define REDUCTION_LWS 256 // пусть размер рабочей группы - фиксирован
__kernel void knlSUM(int n, __global double* x, __global double *res){
    const int gid = get_global_id(0); // глобальный номер work-item
    const int lid = get_local_id(0); // номер внутри рабочей группы
    __local double sdata[REDUCTION_LWS]; // сюда будем собирать сумму группы.
    sdata[lid] = gid < N ? x[gid] : 0.0;
    barrier(CLK_LOCAL_MEM_FENCE); // барьер, чтобы sdata весь был заполнен

    // делаем сборку сдваиванием
    if(lid <128) sdata[lid] += sdata[lid +128]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 64) sdata[lid] += sdata[lid + 64]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 32) sdata[lid] += sdata[lid + 32]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 16) sdata[lid] += sdata[lid + 16]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  8) sdata[lid] += sdata[lid +  8]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  4) sdata[lid] += sdata[lid +  4]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  2) sdata[lid] += sdata[lid +  2]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  1) sdata[lid] += sdata[lid +  1]; barrier(CLK_LOCAL_MEM_FENCE);

    // записываем результат группы
    if(lid == 0) res[get_group_id(0)] = sdata[0];
}
```

Ну ладно, сборку можно сделать и циклом (цикл может внести немного оверхеда):

```
    for(int s=REDUCTION_LWS/2; s>0; s>>=1/*деление на 2 сдвигом*/){
        if(lid<s) sdata[lid] += sdata[lid + s];
        barrier(CLK_LOCAL_MEM_FENCE);
    }
```

Если пока не очень понятно, что происходит в этом цикле, то вот картинка:



На выходе получили вектор частичных сумм res, размер которого в 256 раз меньше. Тут мы предполагали, что число work-item больше или равно размеру вектора. Можно повторить вызов этого кернела теперь уже с вектором res на вход, и он еще в 256 раз уменьшится. И так пока, наконец, не найдем сумму. А можем забить, выкачать res на хост и там его просуммировать. Все равно наверняка результат суммы нужен на хосте. Скачать и досуммировать на хосте может выйти быстрее, чем гонять еще раз вызов кернела. Так и поступим. Но, во-первых, res может быть великоват. А, во-вторых, сборка сдваиванием портит нам производительность. На каждом шаге остается работать вдвое меньше work-item-ов, а значит на этапе сборки суммы по рабочей группе большая часть айтемов простаивает. Нехорошо. Сделаем общее число work-item-ов в несколько раз меньше, чем N. Тогда в несколько раз уменьшится вектор res, и вес сборки локальной суммы тоже уменьшится в несколько раз. Тогда придется в кернеле развернуть цикл по частичной сумме каждого айтема. Вместо

```
sdata[lid] = gid < N ? x[gid] : 0.0;
```

будет

```
sdata[lid]=0.0; // зануляем свою позицию
const int gsz = get_global_size(0); // общее число нитей
for(int i=gid; i<n; i+=gsz) sdata[lid] += x[i]; // находим сумму данной нити
```

Теперь мы можем выставить по своему усмотрению число заданий, чтобы уменьшить размер вектора частичных сумм и снизить накладные на сборку локальных сумм. Но общее число work-item сделать слишком маленьким тоже неэффективно. Девайсу нужно достаточно много заданий, чтобы получился хороший коэффициент загрузки ресурсов мультипроцессора. Так что количество сумм, которые делает один work-item, можно варьировать в зависимости от характеристик устройства и размера вектора. Мы берем число 8 (сумм на один айтем) и не паримся. Итак, что у нас получилось. На хосте:

```
cl_kernel knlSUM;
knlSUM   = clCreateKernel(clProgram, "knlSUM", &clErr); // создаем кернел
if(clErr != CL_SUCCESS){ printf("clCreateKernel knlSUM error: %d\n",clErr); exit(1); }

#define REDUCTION_LWS 256 // размер рабочей группы для суммы
#define REDUCTION_ITEM 8  // сколько элементов будет суммировать один work-item (нить)

// размер буфера частичных сумм рабочих групп
int REDUCTION_BUFSIZE = ((N/REDUCTION_ITEM)/REDUCTION_LWS) +
                        ((N/REDUCTION_ITEM)%REDUCTION_LWS>0);

// буфер под частичные суммы рабочих групп
cl_mem clSum = clCreateBuffer(clContext, CL_MEM_READ_WRITE,
                              REDUCTION_BUFSIZE*sizeof(double), NULL, &clErr);
if(clErr != CL_SUCCESS){ printf("clCreateBuffer clSum error %d\n",clErr); exit(1); }

// выставляем параметры запуска
size_t lws = REDUCTION_LWS; // размер рабочей группы
size_t gws = (N/REDUCTION_ITEM); // общее число заданий
if(gws%lws>0) gws += lws-gws%lws; // делаем кратное lws

// выставляем аргументы кернелу
clSetKernelArg(knlSUM, 0, sizeof(int), &N);
clSetKernelArg(knlSUM, 1, sizeof(cl_mem), &clX);
clSetKernelArg(knlSUM, 2, sizeof(cl_mem), &clSum);
```

```
    // отправляем на исполнение
    clErr= clEnqueueNDRangeKernel(clQueue, knlSUM, 1, NULL, &gws, &lws, 0, NULL, NULL);
    if(clErr != CL_SUCCESS){ printf("clEnqueueNDRangeKernel error %d\n",clErr); exit(1); }
    clFinish(clQueue); // ждем завершения

    // забираем результат с девайса
    double *Sum = new double[REDUCTION_BUFSIZE];
    clErr = clEnqueueReadBuffer(clQueue, clSum, CL_TRUE, 0,
                                REDUCTION_BUFSIZE*sizeof(double), Sum, 0, NULL, NULL);
    if(clErr != CL_SUCCESS){printf("clEnqueueReadBuffer clSum error %d\n", clErr); exit(1);}
    clFinish(clQueue);

    // досуммируем результат на хосте, там осталось где-то 0.05% от общего объема работы
    double lsum = 0.0;
    #pragma omp parallel for reduction(+:lsum)
    for(int i=0; i<REDUCTION_BUFSIZE; ++i) lsum += Sum[i];

    // досуммируем по всей MPI группе, если у нас имеется MPI распараллеливание
    double gsum;
    MPI_Allreduce(&lsum, &gsum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

А теперь кернел:

```
#define REDUCTION_LWS 256 //reduction workgroup size - fixed!
__kernel void knlSUM(int n, __global double* x, __global double *res){
    const int gid = get_global_id(0); // глобальный номер задания
    const int lid = get_local_id(0); // номер внутри рабочей группы
    const int gsz = get_global_size(0); // общее число нитей
    __local double sdata[REDUCTION_LWS]; // сюда будем собирать сумму группы
    sdata[lid]=0.0; // зануляем свою позицию
    for(int i=gid; i<n; i+=gsz) sdata[lid] += x[i]; // находим сумму данной нити
    barrier(CLK_LOCAL_MEM_FENCE); // барьер, чтобы вся группа нашла свою локальную сумму
    // делаем сборку сдваиванием
    if(lid <128) sdata[lid] += sdata[lid +128]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 64) sdata[lid] += sdata[lid + 64]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 32) sdata[lid] += sdata[lid + 32]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid < 16) sdata[lid] += sdata[lid + 16]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  8) sdata[lid] += sdata[lid +  8]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  4) sdata[lid] += sdata[lid +  4]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  2) sdata[lid] += sdata[lid +  2]; barrier(CLK_LOCAL_MEM_FENCE);
    if(lid <  1) sdata[lid] += sdata[lid +  1]; barrier(CLK_LOCAL_MEM_FENCE);
    // записываем результат группы
    if(lid == 0) res[get_group_id(0)] = sdata[0];
}
```

И эта вся жесть вместо одной строчки `for(int i=0; i<N; ++i) sum += X[i];`
Весело, не правда ли?

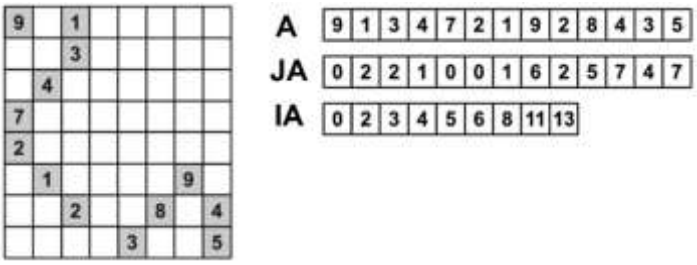
### 5.4.3 SpMV

SpMV – Sparse Matrix-Vector Product. Эта операция уже неоднократно встречалась в предыдущих главах. Пусть матрица N×N хранится в уже знакомом по предыдущим главам формате CSR в трех массивах:

**IA** – целочисленный, размера N+1, в котором для каждой строки храним позицию начала списка номеров столбцов с ненулевыми коэффициентами данной строки, а последний элемент в массиве, IA[N], хранит общее число ненулей в матрице;

**JA** – целочисленный, размера IA[N], хранит номера столбцов с ненулевыми коэффициентами по всем строкам подряд (ну то есть упорядоченно по строкам);

**A** – вещественный, размера IA[N], хранит значения ненулевых коэффициентов по всем строкам подряд.




Последовательная версия SpMV **x** = **Ay** для CPU может быть такой:

```
for(int i=0; i<N; ++i){
    double sum = 0.0; // компайлер не знает о зависимостях между указателями A, x и y
    // не будем его смущать, будем суммировать в локальную переменную
    for(int _j = IA[i]; _j<IA[i+1]; ++_j){
        // _j – с подчеркиванием, чтобы подчеркнуть, что это не номер столбца
        sum += /*A[i][j] -->*/A[_j] * x[JA[_j]/*<-- вот он всамделишний j */];
    }
    y[i] = sum;
}
```

Параллельная версия по OpenMP, например:

```
#pragma omp parallel for schedule(dynamic/*неоднородные итерации*/, 500)
for(int i=0; i<N; ++i){
    double sum = 0.0;
    for(int _j = IA[i]; _j<IA[i+1]; ++_j) sum += A[_j]*x[JA[_j]];
    y[i] = sum;
}
```

Параллельная версия с комбинированным MPI+OpenMP распараллеливанием:

```
UpdateVec(x); // обновляем значения в гало входного вектора
#pragma omp parallel for schedule(dynamic, 500)
for(int i=0; i<N_own; ++i){ // неизвестные переупорядочены по наборам – свои, гало
    double sum = 0.0;
    for(int _j = IA[i]; _j<IA[i+1]; ++_j) sum += A[_j]*x[JA[_j]];
    y[i] = sum;
}
```

Версия OpenCL кернела – заменяем переменную цикла на айдишник айтема:

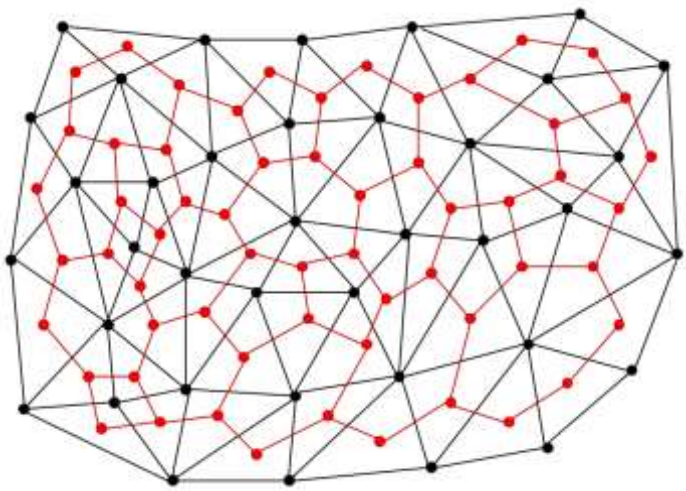
```
__kernel void knlSPMV_CSR(int N,
    __global double* A, __global int* IA, __global int* JA,
    __global double* x, __global double* y){
    const int i = get_global_id(0); if(i >= N) return;
    double sum = 0.0;
    for(int _j=IA[i]; _j<IA[i+1]; ++_j) sum += A[_j]*x[JA[_j]];
    y[i] = sum;
}
```

Как видим, тут нет такой жести, как в редукции. Кернел несложный, эквивалентен исходной CPU версии. Потому что итерации исходного (внешнего) цикла по строкам матрицы независимы между собой. Это, конечно, будет работать, но есть нюансы.

Во-первых, у нас тут цикл переменного диапазона внутри кернела, а это плохо: неоднородность загрузки айтемов рабочей группы приведет к простоям исполнительных устройств, а переменный цикл не развернется компилятором также эффективно, как цикл по константе.

Во-вторых, у нас тут косвенная адресация к элементам входного вектора. А это значит – неэффективное (неполное) использование кэш линий и просто кэш промахи. Но хосту от этого ничуть не лучше, так что и хост, и девайс – в равных условиях.

Какое-то единое решение этих проблем на все случаи жизни сложно придумать. Многое зависит от конкретной структуры матрицы. Рассмотрим простые примеры. Пусть у нас матрица пришла из CFD приложений из какого-то конечно-объемного метода на треугольной или тетраэдральной сетке. Пусть ячейками являются сеточные элементы. Портрет матрицы соответствует портрету матрицы смежности графа связей сеточных элементов по общим граням, дополненному главной диагональю. Ну вот эта картинка уже в предыдущих главах появлялась сотни тысяч раз. Граф, по которому портрет матрицы, это который красный.



Какую мы тут видим особенность? Ну на этой картинке у большинства вершин в графе степень 3. И максимальная степень по всему графу тоже 3. Потому что у треугольника 3 ребра. В трехмерном случае будут тетраэдры, значит, 4 грани. Только у тех ячеек, которые на краю сетки, получается меньше соседей. 4 грани – это по сколько ненулей в строках? Правильно – 5: 4 соседа и главная диагональ.

Для такой задачи лучше подойдет другой формат – ELLPACK. Этот формат предполагает, что у всех строк равное число ненулевых коэффициентов. Раз оно равное, то мы просто берем CSR, выкидываем из него массив IA, который больше не нужен, а позицию строки в векторах JA и A находим, просто умножая номер строки i на константу, например, m, равную числу ненулей в строке. Получим такой кернел:

```
__kernel void knlSPMV_ELL(int N, int m, __global double* A,
    __global int* JA, __global double* x, __global double* y){
    const int i = get_global_id(0); if(i >= N) return;
    JA += i*m; A += i*m; // позиционируемся на нужную строку
    double sum = 0.0;
```

```
    for(int _j=0; _j<m; ++_j) sum += A[_j]*x[JA[_j]];
    y[i] = sum;
}
```

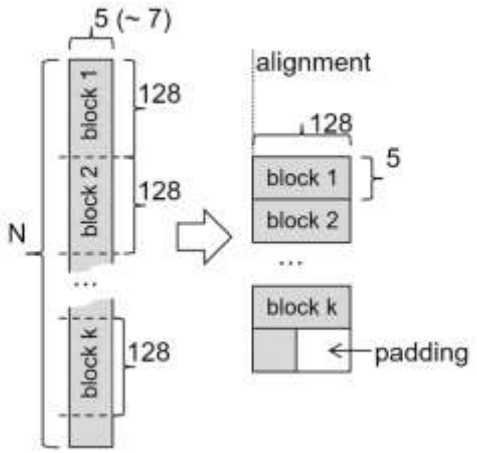
Но что делать с граничными ячейками, у которых нет столько ненулей? Нет ненулей, доберем нулями, делов-то. В каких-то редких строках в ненулевых коэффициентах окажутся нули. Хорошо, в A мы положим 0, а что в JA? В JA поставим, например, номер столбца последнего ненулевого коэффициента в строке, чтобы не надо было читать никаких лишних позиций из входного вектора. Но опять получается неоптимально, мы-то знаем, например, что решаем задачу на тетраэдральной сетке, а цикл в кернеле остался по переменному диапазону. Решим эту проблему:

```
__kernel void knlSPMV_ELL5(int N, __global double* A,
    __global int* JA, __global double* x, __global double* y){
    const int i = get_global_id(0); if(i >= N) return;
    JA += i*5; A += i*5; // позиционируемся на нужную строку
    double sum = 0.0;
    for(int _j=0; _j<5/*константа*/; ++_j) sum += A[_j]*x[JA[_j]];
    y[i] = sum;
}
```

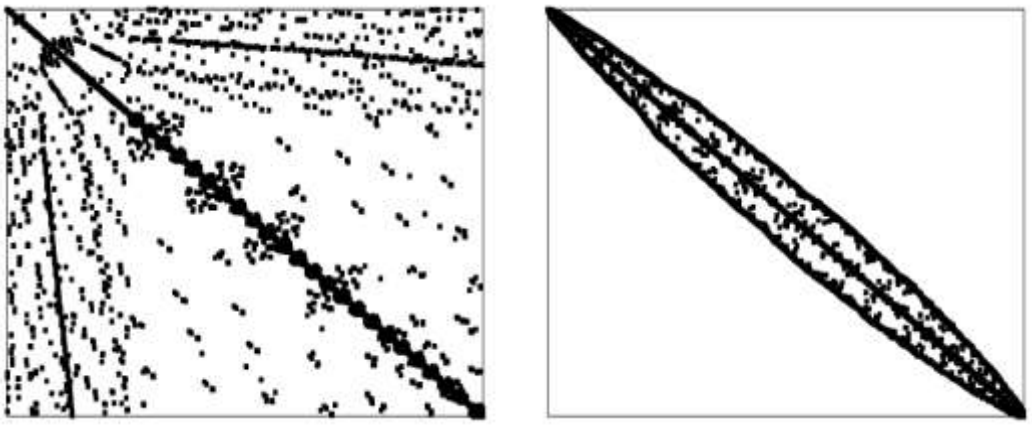
А если у нас сетка смешанная, и могут быть элементы от 4 граней (тетраэдры) до 6 граней (гексаэдры)? Тогда будет зависеть от соотношения их количества. Если сетка преимущественно гексаэдральная, используем ELLPACK по 7 ненулей на строку и не заморачиваемся. Если сетка преимущественно тетраэдральная, или, например, разных типов элементов примерно поровну, то матрицу можно расслоить. Переупорядочим ячейки по числу граней. Сначала тетры, потом 4-угольные пирамиды, потом треугольные призмы, потом гексы. Тогда матрица получится из трех слоев с 5, 6 и 7 ненулями. Сделаем 3 кернела для этих значений. Можем делать SpMV в 3 захода: вызовем knlSPMV_ELL5 для первой части, knlSPMV_ELL6 для второй, knlSPMV_ELL7 для третьей, например. Такой способ будет эффективным, если у нас сетка состоит из блоков из разных типов элементов, например, – тетраэдральная зона сетки, где-то гексаэральная зона, в переходных зонах призмы с пирамидами. Тогда такая перенумерация ячеек не испортит локальность (компактность) данных. А если все элементы вперемешку навалены зачем-то (что редко бывает на практике), то так расслоить матрицу может быть не очень выгодно, т.к. доступ ко входному вектору будет сильно болтаться в памяти и требовать больше кэш линий. Тогда может оказаться выгоднее использовать просто ELLPACK по максимальному числу ненулей и западдить (жарг. от padding) нулями все строки, где ненулей меньше.

Да, раз речь зашла о локальности данных, подумаем теперь на эту тему. Пусть у нас простейший ELLPACK для тетраэдральной сетки (неизвестные определены в сеточных элементах), строки по 5 ненулей. Соседние айтемы будут зачитывать свои коэффициенты матрицы со страйдом 5. Одна нить полезет в A[i*5], другая в A[(i+1)*5], и так полуварпу из 16 нитей понадобится на один коэффициент выгрести 640 байт вместо 128. Ну да, там есть кэш, не все считанное непосильным трудом пропадет, но существенная часть кэш линий может заэкспайриться. Поэтому читать может быть заметно эффективнее с юнит-страйдом. Для этого можем “транспонировать” массивы JA, A. Сейчас там N блоков по 5 значений, а можем сделать 5 блоков по N значений. А чтобы доступ каждой рабочей группы был еще и выровнен, можем транспонировать блоки данных рабочих групп и выровнять их по ширине кэш линии. Тогда читать будем целыми кэш линиями, и эти линии будут идти подряд. На картинке ниже

показан пример для рабочей группы из 128 айтемов. Если не хотим перетасовывать данные в матрице, можем просто использовать зачитку через локальную общую память (выше был пример).

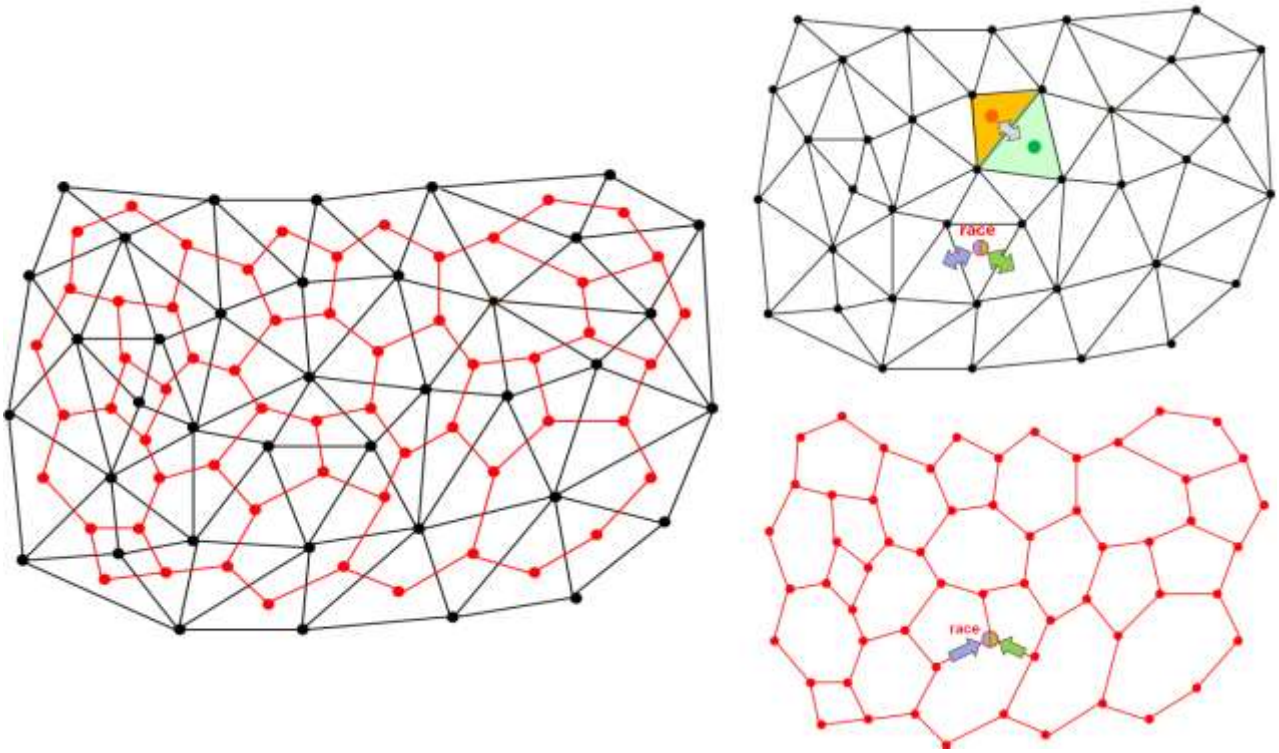



Коэффициенты читаются более-менее локально. В худшем случае сколько-то кэш линий потеряется, производительность просядет, но не критично. А вот доступ к входному вектору при плохой нумерации может быть сильно некомпактным, в худшем случае за каждый икс итый придется платить целой кэш линией. Потери на латентность будут совсем недетские. Обычно в расчетах сеточным методом нумерация неизвестных следует из нумерации сеточных объектов, а эта нумерация приходит из сеточного генератора, который обычно делает культурную приемлемую нумерацию, достаточно локальную. Но чтобы быть уверенными, можно на препроцессе или инициализации перенумеровать неизвестные, чтобы разброс позиций в строке был как можно меньше. А что значит, как можно меньше? Значит – привести матрицу к ленточному виду с как можно меньшей шириной ленты. Такие вещи обычно делаются какой-нибудь простейшей версией алгоритма Катила – Макки. Суть у алгоритма простая, он делается в O(10) строк кода. Там проблема может быть только в выборе периферийной вершины, а как раз с этим можно и не заморачиваться, для сеточного метода берем любую вершину с минимальной степенью и все, дальше тривиально. Портрет матрицы, если он был сильно взъерошенный из-за плохой нумерации, превращается в что-то более удобное. Слева – было, справа – стало. Более подробно об этом в разделе 7.1.1.



### 5.4.4 Пример про сеточный метод

Наконец, вернемся к нашему сеточному примеру. В главе 3 в разделе 3.3 рассматривались способы устранения зависимостей при OpenMP распараллеливании обобщенной операции сеточного метода. Там мы считали потоки через грани между ячейками, и возникала гонка при записи вклада этих потоков в ячейки. Какие из представленных там способов подходят для потоковой обработки? Вот такая была картинка, помните?

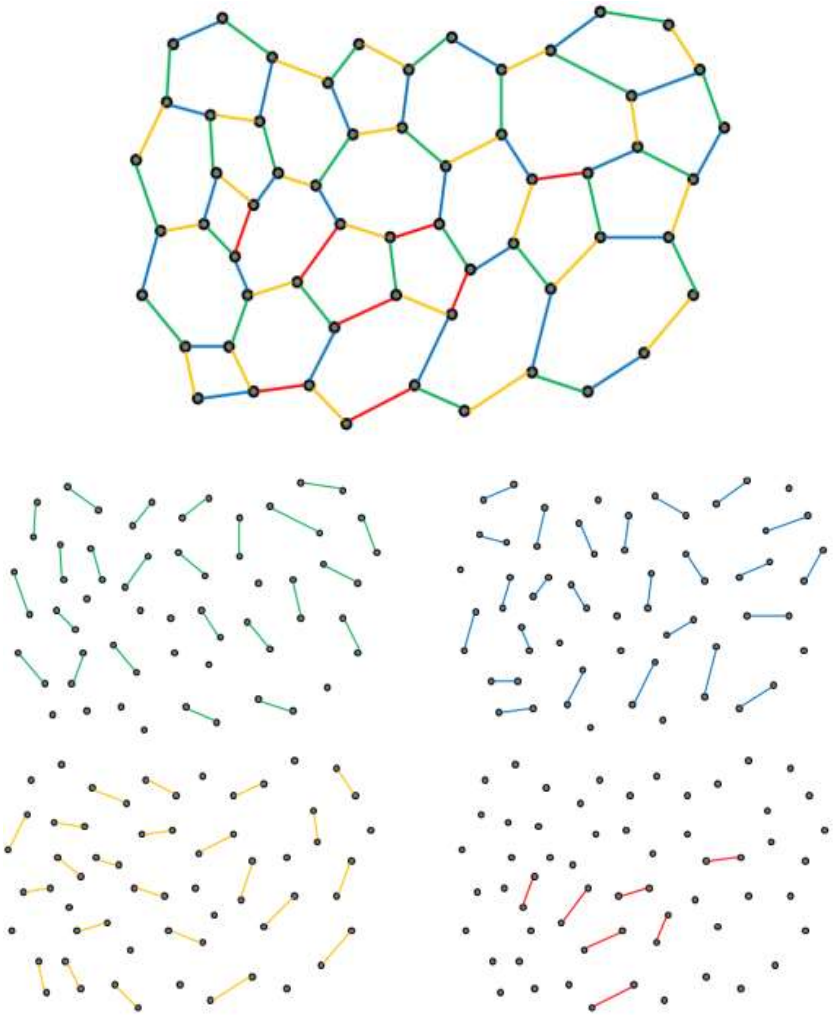



Полное дублирование вычислений потоков – подходит. Например, каждый work-item обрабатывает одну ячейку (вершину, если в терминах графа), перебирает ее грани (ребра), считает потоки, записывает к себе результат. Минусы – удвоенная вычислительная стоимость, неоднородность работы айтемов, т.к. число граней может быть разное.

Атомики – может и подойдут, если повезет, что девайс их поддерживает, и хватит сил с ними возиться. Но проблемы как минимум те же, что и в случае OpenMP – тормозит.

Разделение на два этапа с промежуточным массивом – хорошо подходит. Первый кернел по набору граней находит потоки через грани, второй по набору ячеек суммирует эти потоки. Все идеальненько получается. Только может быть (очень) жалко памяти на промежуточный массив, на GPU памяти мало.

Раскраска графа, разделение на такты по цветам – подходит. Покрасили ребра, вызываем кернел на каждый цвет (см. картинку ниже). Минусы – сильно портится coalescing, тут мы сразу исключаем, что будем обращаться к одной и той же ячейке, что заметно снижает используемость кэш линий. Но в остальном работать будет хорошо, только еще небольшой оверхедик на множественный запуск кернела по цветам. Доступ к памяти может быть заметно менее эффективен, чем в предыдущем случае.



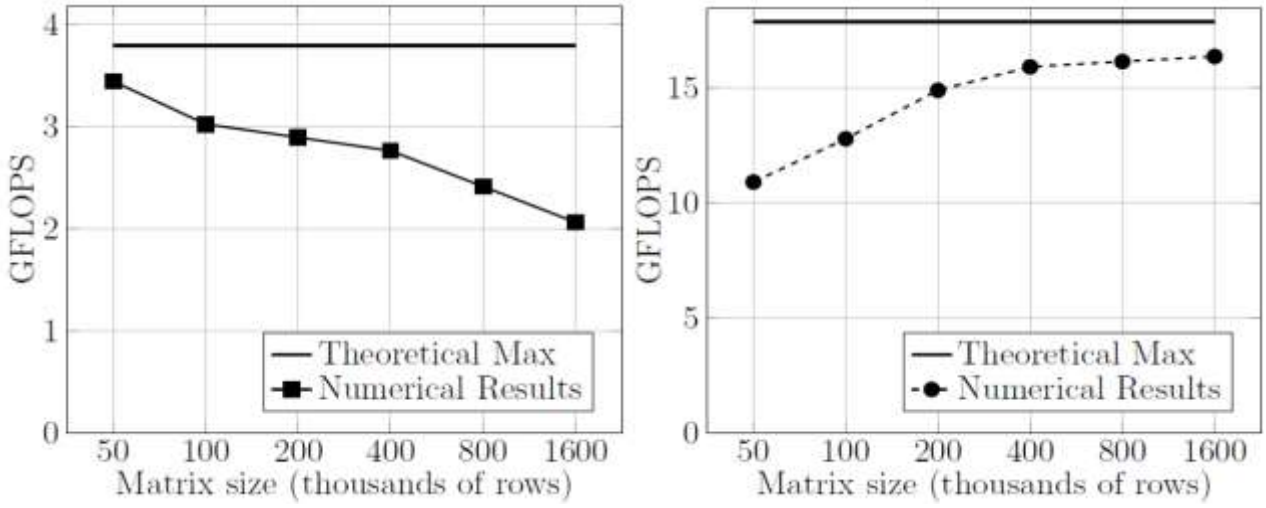
Декомпозиция. Все, что связано с последовательной обработкой интерфейса – не подходит. Последовательно – это слишком медленно. Многоуровневая декомпозиция может сработать, если поделить сетку на микродомены work-item-ов, и сбалансировать их по числу граней, чтобы у рабочей группы работа максимально совпадала, тогда простои будут небольшими. Ну и запускать кернел тоже надо будет несколько раз по уровням интерфейса. И доступ к памяти будет еще сильнее некомпактный, поскольку каждый айтем будет лазить в вообще в свою подобласть. Да и в реализации это, конечно, будет сильно посложнее.

## 5.5 Ускорение, масштабирование

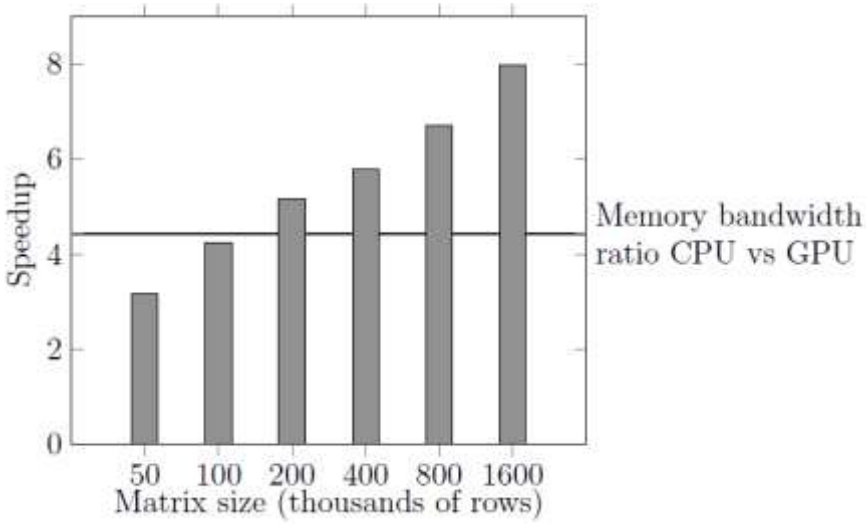
На GPU есть некоторые особенности. Рассмотрим их на примере операции SpMV, которая более-менее удобно ложится на потоковую обработку, но требует обмена данными для обновления гало. Получается репрезентативно и очень похоже на паттерн вычислений сеточного метода.

Первое, что обычно меряют и гордо показывают окружающим, это ускорение на девайсе относительно процессора. Зачастую показывают ускорение относительно одного ядра, потому что так число получается больше. Но информативнее, конечно, сравнивать производительность на целом устройстве – многоядерный CPU против GPU. Сразу отметим, что ускорение само по себе вообще не информативно. Чем хуже мы сделаем код для CPU, чем больше мы сможем получить ускорение на GPU. А в идеале еще код на CPU компилировать, выключив оптимизацию и включив отладку. Тогда ускорение будет совсем замечательное.

Поэтому, конечно, ускорение должно подкрепляться оценками фактически достигаемой производительности и достижимой с учетом пропускной способности памяти. Посмотрим, что происходит с ускорением GPU vs CPU на нашем простейшем SpMV с форматом ELLPACK по 5 ненулей на строку с матрицей из сеточного метода на тетраэдральной сетке (с определением переменных на сеточных элементах). Вот тестик на каком-то древнем ископаемом гибридном узле, 6-ядерный CPU Intel Xeon E5649 (32 GB/s, 60 GFLOPS) и GPU NVIDIA M2090 (156 GB/s – ECC on, 670 GFLOPS). Оценим сверху достижимую производительность с учетом пропускной способности памяти. На строку у нас 5 умножений и 4 сложения, итого 9N флопов, где N – число строк в матрице. Так как мы оцениваем сверху, будем считать память по объему, а не по трафику. С трафиком все непонятно, там мало того, что во входной вектор мы лазаем по нескольку раз в каждый элемент, еще и неизвестно в каком порядке, а там кэш промахи и потери на латентность. То есть пусть у нас идеальный вариант и полная повторная используемость всех значений из входного вектора в кэш. Получается, читаем матрицу в количестве 5N даблов (по 8 байт) и 5N интов (по 4 байта) – коэффициенты и номера столбцов, соответственно, плюс N даблов на входной вектор и N на выходной, итого суммарно 76N байт. Получили арифметическую интенсивность < 0.12. Значит, на SpMV достаточно большого размера, чтобы оно не влезало в кэш, получим 32 GB/s × 0.12 = 3.84 GFLOPS на CPU. На GPU будет 18.7 GFLOPS – меньше 3% от пика (грустно вздыхает, вытирает слезу). Построим графики достигаемых FLOPS в зависимости от размера N. Что же мы видим? На CPU чем меньше N, тем больше перф. Оно и понятно, разброс позиций в строке по входному вектору уменьшается, меньше кэш промахов. А у GPU – все наоборот. Кэш там никакусенький (0.8 МБ на сотни “нитей” против 12 МБ на 12 потоков у CPU) и почти никакой роли не играет. А на маленькой матрице – слишком маленький объем работы, девайс не успевает “раскочегариться”. Поэтому с увеличением размера N производительность растет и выходит на какой-то стабильный уровень.

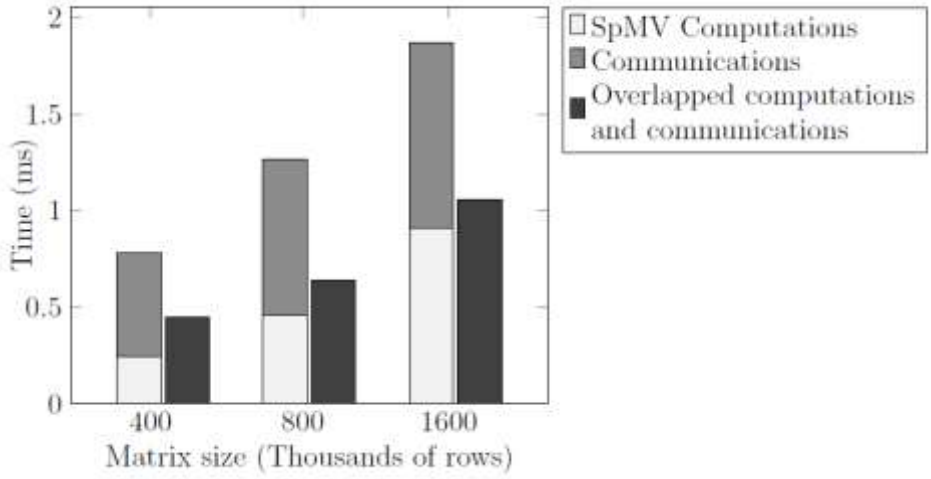



Итак, CPU с увеличением N замедляется, а GPU ускоряется. Какой из этого можно сделать вывод? Правильно. Ускорение GPU vs CPU надо мерять на больших матрицах, тогда оно будет больше и можно увереннее радовать окружающих результатами. Проверим на практике:



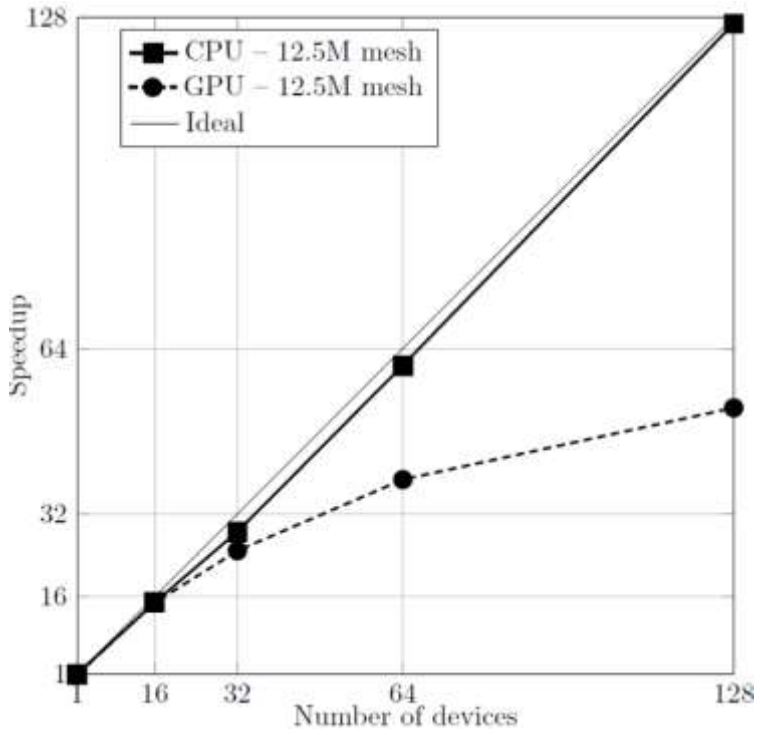

Ну вот, ускорение на достаточно большой матрице под 8 раз против 6-ядерного CPU, то есть, получается, эквивалент 48 ядер. А на маленькой матрице всего 3 раза. Но мы же его не будем никому показывать?

Посмотрим, как работает SpMV на multi-GPU. Наплодим MPI процессов, добавим halo update в режиме overlap. Замеряем время на вычисления и обмены в синхронном режиме, а потом замерим общее время в режиме overlap, когда обмен данными и вычисления выполняются одновременно. Вот что получилось:



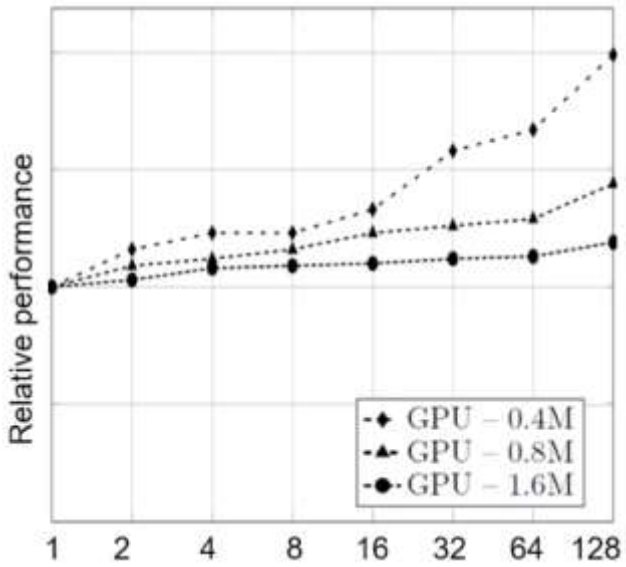

Видим, что сокрытие обменов в режиме overlap работает эффективно. Только кернел настолько дешевый, а девайс считает так быстро, что обмены занимают больше времени. Получается, что это вычисления скрываются за обменами. Но не суть. Меньшее скрылось за большим и стало почти в 2 раза быстрее. Красота. Раз у нас все так хорошо оверлапнулось, наверное, у нас будет очень хорошее параллельное ускорение? А вот и нет.

Как видно на картинке ниже, параллельное ускорение на 128 GPU совсем издохло, да и на 64 уже было не особо, а на 128 CPU ускорение прям живее всех живых. Почему? Потому что с увеличением числа процессоров уменьшается локальный размер данных и, соответственно, уменьшается число кэш промахов. Действительно, сетка 12.5 млн ячеек, на 128 процессорах – это будет порядка 100 тыс. на многоядерный CPU.




Если посмотрим на самый первый график, где GFLOPS от N, то увидим, что проц на таком размере как минимум в полтора раза быстрее, чем он же на большой матрице. Это компенсирует накладные расходы на обмен данными и ускорение остается линейным, идеальным. А GPU наоборот выдает все меньше производительности с уменьшением объема работы. Поэтому график параллельного ускорения ведет себя совсем не так, как на CPU. Конечно, чем дороже кернел, чем больше арифметическая интенсивность, тем в меньшей степени эти эффекты проявляются. Поэтому на дешевом SpMV это видно лучше всего.

А что же с масштабируемостью на GPU? Это когда мы увеличиваем N пропорционально числу девайсов. С масштабированием – все хорошо. Чем больше объем работы, тем ближе к идеалу график, как и положено, как и в случае CPU. Вот графики для загрузки по 0.4, 0.8 и 1.6 миллиона строк на девайс:



## 5.6 Внеклассное чтение

Спецификации стандартов OpenCL живут тут:

- https://www.khronos.org/opencl/
- https://www.khronos.org/registry/OpenCL/

Стандарт 1.2 вполне подойдет:

- https://www.khronos.org/registry/OpenCL//sdk/1.2/docs/OpenCL-1.2-refcard.pdf – шпаргалка
- https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf – полная спецификация, в ней читаем хотя бы раздел 3.

Ссылки на разные optimization guide:

- https://www.cs.cmu.edu/afs/cs/academic/class/15668-s11/www/cuda-doc/OpenCL_Best_Practices_Guide.pdf
- https://developer.amd.com/wordpress/media/2013/12/AMD_OpenCL_Programming_Optimization_Guide.pdf

Учебнички

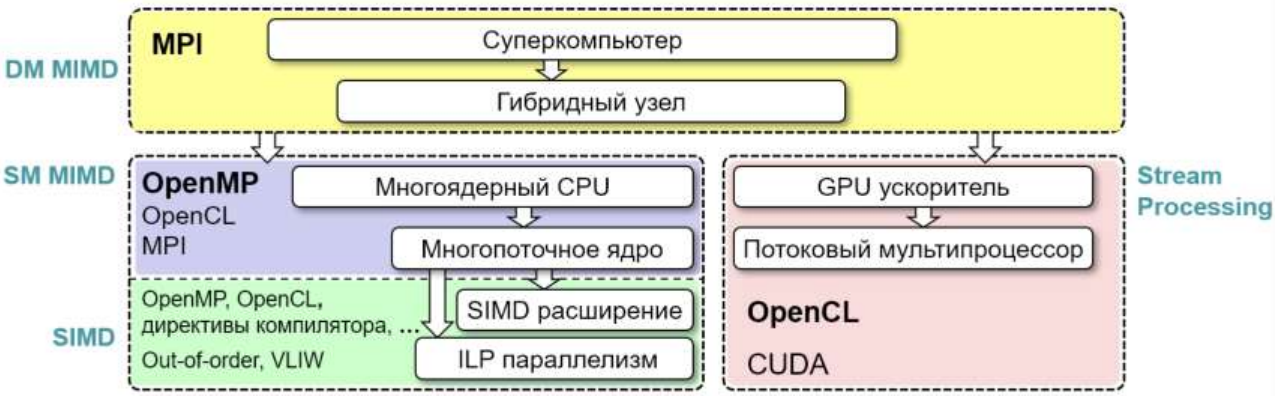
- R.Trobec, B. Slivnik, P. Bulić, B. Robič. Introduction to Parallel Computing From Algorithms to Programming on State-of-the-Art Platforms. 2018. Springer. – глава 5. https://doi.org/10.1007/978-3-319-98833-7

Статьи, материалы из которых использованы в этой главе:

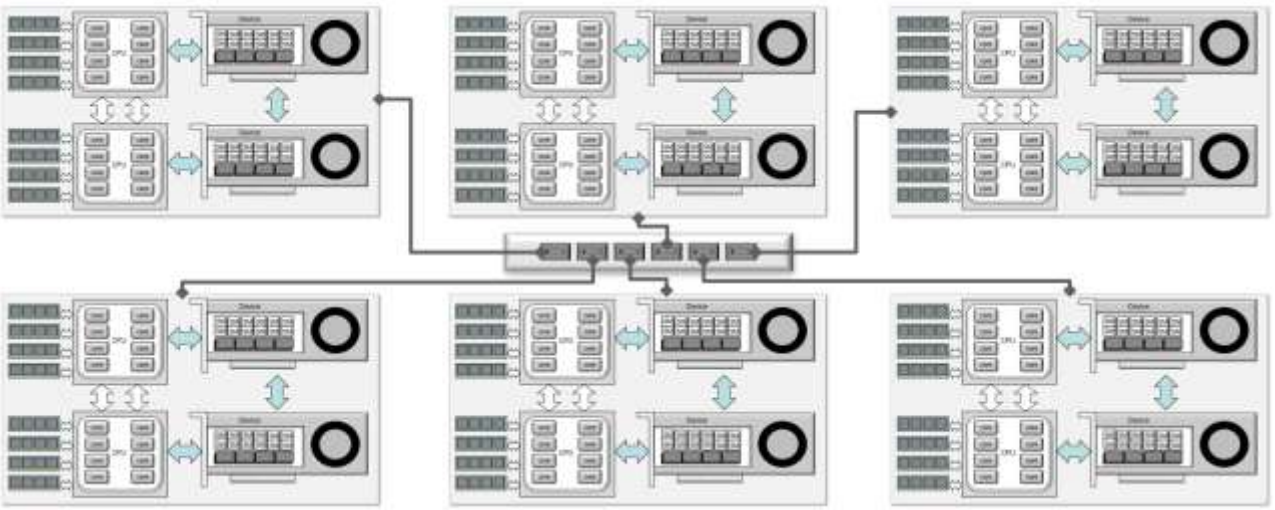
- X. Alvarez-Farre, A. Gorobets F. X. Trias. A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers. Computers & Fluids. Volume 214, 2021, 104768. https://doi.org/10.1016/j.compfluid.2020.104768
- A.Gorobets, S.Soukov, P.Bogdanov. Multilevel parallelization for simulating turbulent flows on most kinds of hybrid supercomputers. Computers & Fluids. Volume 173, Pages 171–177. 2018.https://doi.org/10.1016/j.compfluid.2018.03.011
- X.Alvarez, A.Gorobets, F.X.Trias, R.Borrell, and G.Oyarzun. HPC$^2$ - a fully portable algebra-dominant framework for heterogeneous computing. Application to CFD. Computers & Fluids. Volume 173. Pages 285–292. 2018. https://doi.org/10.1016/j.compfluid.2018.01.034
- S. A. Soukov, A. V. Gorobets. Heterogeneous Computing in Resource-Intensive CFD Simulations. Doklady Mathematic. 2018. Volume. 98, No. 2, Pages 1–3. https://doi.org/10.1134/S1064562418060194

# 6 Гетерогенные вычисления на GPU и CPU

В предыдущих главах мы разобрались с процессорами и ускорителями, с распределенными вычислениями на кластерной системе. Теперь осталось просто это все объединить, задействовать все уровни параллелизма. Тут пойдет речь в основном об организации обмена данными, распределении вычислений, балансировке загрузки. Много общего с главой 4.

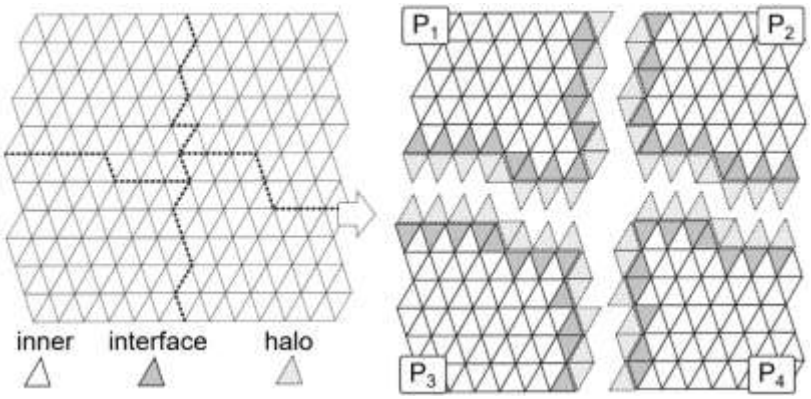



Имеется гибридная кластерная система. На узлах установлены центральные процессоры и массивно-параллельные ускорители-сопроцессоры. Ускорители, aka девайсы, не имеют общей памяти с CPU, у них своя набортная память высокой пропускной способности. CPU и девайсы общаются друг с другом по внутренней системной шине узла.



Для распределения работы между узлами кластерной системы и между девайсами внутри узла используется декомпозиция. Набор обрабатываемых объектов распределяется между вычислительными устройствами. Для этого строится граф, описывающий связи между этими объектами (например, расчетными ячейками сеточного метода), вершины которого распределяются по подобластям с минимизацией числа “разрезанных” ребер, то есть ребер, у которых вершины попали в разные подобласти.

Вспомним, что было в предыдущих сериях. Вот у нас уже была такая картинка в главе 4. Делим треугольную сеточку на 4 подобласти, чтобы 4 разных устройства, не имеющих общей оперативной памяти, могли проделать параллельные вычисления:




Сеточные ячейки (а они же – вершины графа, держим графа в уме) с точки зрения каждой подобласти получились таких видов:

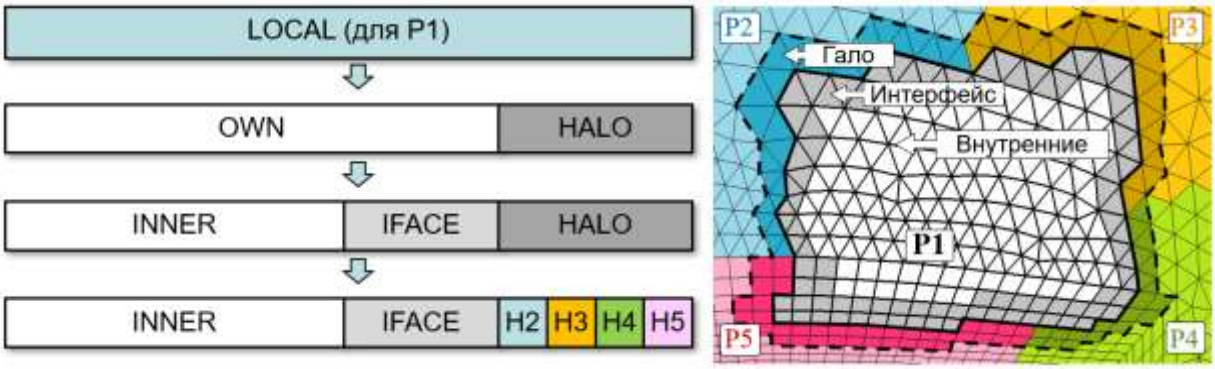
**собственные** (own) – которые попали в подобласть;

**гало** (halo) – чужие, которые имеют связи с собственными (общая грань, ребро в графе);

**внутренние** (inner) **–** собственные, не имеющие связей с гало;

**интерфейсные** (interface, iface) – свои, имеющие связи с гало.

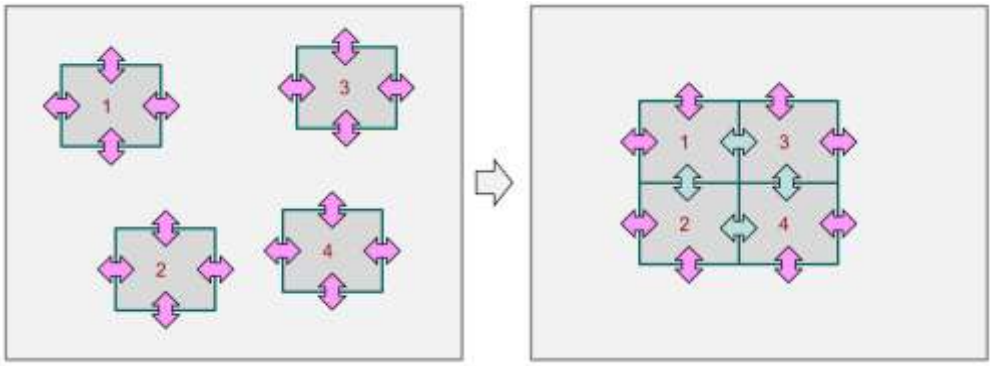
**Локальная** (local) область процесса или девайса – это объединение его собственных ячеек и гало. Ячейки локальной области сразу на инициализации переупорядочиваются по принадлежности вышеперечисленным наборам, чтобы было проще организовать обмены и вычисления: сначала идут внутренние ячейки (или не ячейки, а какие-то еще объекты), потом интерфейсные, потом идет гало, упорядоченное по владельцам:




Чтобы можно было реализовать распределенные вычисления, понадобится в нужные моменты выполнять обмен данными, чтобы обновлять гало. Посчитанные данные по своим интерфейсным ячейкам надо отправлять соседям, а данные в гало ячейках, которые вычисляют соседи, надо получать от соседей. Вспомнили? Если нет, то отматываемся назад на главу 4. Осталось освежить в памяти, подгрузить в кэш еще пару вещей.

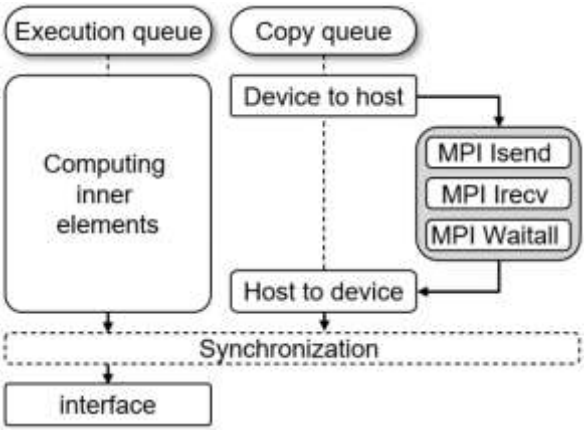
На кластерной системе у нас возникает некоторая иерархия в декомпозиции. В простейшем случае это означает, что процессы и их девайсы могут находиться на одном узле, а могут находиться на разных узлах. В первом случае обмен данными будет происходить по внутренней системной шине узла – intra-node обмены, во втором случае добавляется передача данных по сети, то есть inter-node обмены. Конечно же, гонять трафик по сети выходит дороже (в смысле временных затрат). Поэтому декомпозиция должна учитывать, как расположены процессы или девайсы, и группировать подобласти, находящиеся на одном узле. Декомпозиция с учетом иерархии на кластерной системе получается такой: на первом уровне разделяем всю сетку (или всего графа) между узлами кластерной системы, потом, на втором

уровне, разделяем подобласти узлов на под-подобласти девайсов на этом узле (а если на одном девайсе оказалось несколько процессов, то делим дальше на под-под-подобласти). За счет такой иерархической декомпозиции мы уменьшаем объем сетевого траффика, поскольку обмен будет проходить только по внешней границе областей первого уровня, а не по вообще всем границам подобластей. Символизирует это явление следующая картинка:




Внутри какой-то большой расчетной области у нас 4 подобласти девайсов, находящихся на одном узле кластера. Если мы будем делить расчетную область на подобласти в один уровень сразу по числу всех девайсов на всех узлах, подобласти получатся как попало, и все границы подобластей девайсов будут участвовать в inter-node обменах (что символизируют розовые стрелочки). Если мы побьем область в два уровня, то подобласти наших 4 девайсов слипнутся, и часть границ станет для узла внутренней, с намного более быстрыми intra-node обменами (голубые стрелочки).

Ну и, наконец, последнее, что надо вспомнить – это сокрытие обменов за вычислениями, так называемый режим overlap, когда мы выполняем обмен данными для актуализации гало во время вычислений по внутренним объектам, не связанным с гало. Поскольку мы на ините все культурно упорядочили, разделили на внутренние, интерфейс, гало, теперь нам очень удобно поступить так: будем сначала обрабатывать только внутренние ячейки (вершины нашего графа или еще какие ассоциированные с ними объекты), в это время в неблокирующем режиме в фоне будет идти обмен данными для актуализации гало. Когда и обмены, и вычисления по внутренностям завершатся, запускаем вычисления по интерфейсной зоне, для которой необходимы актуальные данные в гало. Тем самым обмены спрятались на фоне вычислений (ну или наоборот вычисления за обменами, если обмены идут дольше вычислений).



## 6.1 Способы использования гибридной системы

Сначала разберемся, что будет пониматься под гетерогенными вычислениями. Гетерогенная вычислительная система, это система, состоящая из разных вычислительных устройств. Просто разных. Если в системе есть несколько CPU, но разных моделей, это тоже гетерогенная система. Поэтому обычно систему, в которой используются и CPU, и GPU, называют – гибридной, что как бы символизирует, что устройства не просто разные, а принципиально разные. **Гибридный** кластер – кластер, в котором на узлах установлены центральные процессоры и вычислительные ускорители-сопроцессоры. Ну, то есть, гибридная система – это подвид гетерогенной.

А с кодами почему-то терминология сложилась по-другому. С точки зрения программы обычно программу для гибридной системы называют **гетерогенной**. Тут вот в чем дело. Вероятно, это из-за того, что раньше, в доисторические времена, задолго до нашей эры, для распараллеливания в основном использовался стандарт MPI и параллельная модель с распределенной памятью. Тогда процессоры были одноядерные, а по земле еще ходили мамонты. Потом под системы с многоядерными процессорами начали широко применять комбинированное MPI+OpenMP распараллеливание, сочетающее модели с общей и распределенной памятью. Такое распараллеливание широко называли гибридным, не догадываясь, какие настоящие гибриды ждут впереди. Ну и теперь, чтобы не путаться, что именно имеется в виду, про распараллеливание, в котором используются и процессоры, и ускорители, стали говорить – гетерогенное. Также некоторые пытаются использовать термин co-execution, типа соисполнение, чтобы подчеркнуть, что CPU и GPU совместно используются для вычислений. Не будем тут дальше заморачиваться с терминологией, просто все, что связанно с совместными вычислениями на процессорах и ускорителях будем называть гетерогенщиной, а саму матчасть – гибридной вычислительной системой.

Можно выделить четыре режима вычислений на гибридной системе:

- **CPU -only**. Это очевидный вариант, тут не о чем и говорить, это просто программа для CPU (главы 1–4). Такое использование гибридной системы следует считать циничным варварством и вандализмом, допустимым только в случае простоя системы. В ином случае в практике администрирования такой системы должны иметься специально заготовленные тряпки, отмоченные в растворе с высоким содержанием карбамида.
- **GPU-only**, или device-only – это когда все ресурсоемкие вычисления выполняются на ускорителе, а процессор занимается только управлением, то есть выполняет алгоритм на верхнем уровне, пускает кернелы и организует обмен данными. Пример. В рамках этого курса есть практическое задание (см. раздел 8) по реализации итерационного решателя (солвера) СЛАУ, который использует 3 алгебраических кернела – SpMV (матрично-векторное произведение с разреженной матрицей), dot (скалярное произведение), axpby (линейная комбинация двух векторов). Оно подойдет для наглядности. Если сам алгоритм этого солвера записан на стороне CPU, а кернелы выполняются только на GPU – это пример GPU-only режима. CPU крутит решатель, при этом оперирует только со скалярными значениями, а тяжелые вектора и матрицы живут на девайсе.
- **Частичное портирование.** Это случай, когда выборочно портированы на девайс (то есть реализованы кернелы для выполнения на ускорителе) наиболее ресурсоемкие операции алгоритма. А более неудобные вещи, или даже просто несовместимые с потоковой обработкой, оставлены на CPU. В нашем примере с решателем СЛАУ частичным портом было бы, например, реализовать кернелы для SpMV и axpby, но оставить на CPU dot, не

разобравшись, как делать редукцию (и каждый раз выкачивать на хост вектора, чтобы сделать им dot). Обычно в CFD кодах первым делом портируют решатель СЛАУ, поскольку он зачастую ест более половины общего времени. Да, конечно, при таком подходе задействуются в ресурсоемких вычислениях и CPU и GPU. Но нет, это нельзя назвать полноценным гетерогенным режимом. Это, по сути, просто недоделанный GPU-only подход. Хотя бы потому, что CPU и GPU не работают одновременно. Ну и понятно, что, например, ускорив в 10 раз какой-нибудь солвер, который ест половину времени, общее ускорение ограничено значением 2. А ускориться в 2 раза на девайсе за миллион денег с 10-кратным бэндвисом и производительностью – это циничное кощунство и прочее святотатство.

- **Гетерогенный режим**, aka co-execution. В этом случае все ресурсоемкие операции алгоритма полностью портированы на девайс и существуют в реализациях, исполнимых на CPU и GPU. Действительно, когда мы портируем CPU код на GPU, в результате мы как минимум получаем два варианта кода: существовавший изначально, который может задействовать CPU, и GPU-only вариант. А раз так, то почему бы не приложить совсем небольшие (относительно самого портирования) усилия, чтобы можно было использовать CPU и GPU одновременно? С этим будем разбираться далее в этой главе.

## 6.2 Способы реализации гетерогенных вычислений

Итак, у нас есть две версии программы – для CPU и для ускорителей. А в случае OpenCL даже достаточно одной версии, поскольку задействовать можно и CPU, если соответствующие драйверы установлены на системе. Причем зачастую OpenCL реализация делает OpenMP по скорости на CPU (но еще более зачастую CPU-шный драйвер сильно глючный, поэтому мы пока не используем OpenCL на CPU). Что мы можем сделать с этими версиями, чтобы одновременно задействовать и CPU, и ускорители?

Тут пока все то же самое, как в главе 4: нужно распределить работу путем декомпозиции графа связей обрабатываемых объектов. Все ровно точно также, только декомпозиция выполняется для распределения работы между устройствами гибридного узла – CPU и девайсами. Поскольку у устройств разная производительность, надо просто подобрать веса подобластям так, чтобы распределение работы было пропорционально фактической производительности.

С распределением работы путем декомпозиции все достаточно очевидно. Что с этим делать дальше?

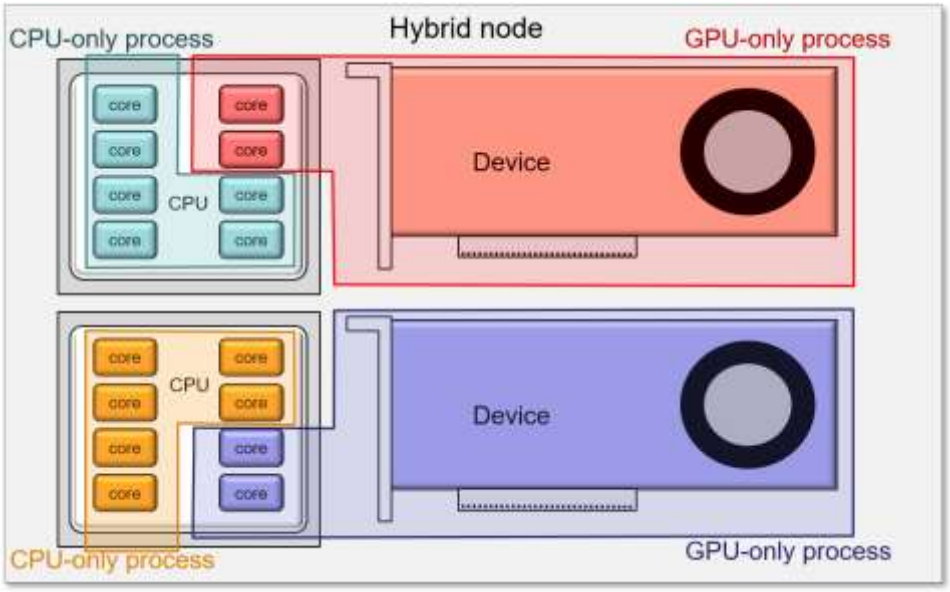
### 6.2.1 Разделение на уровне MPI. Подход 1 процесс – 1 девайс

Это простой и удобный способ. Если у нас реализована двухуровневая декомпозиция (1 уровень распределяет работу между гибридными узлам кластера, 2 уровень – между устройствами на узле), реализована возможность варьировать веса подобластей, реализованы CPU-only и GPU-only версии, то задача почти решена. Будем запускать, например, по одному процессу на вычислительное устройство – CPU (не ядро, а весь процессор) или ускоритель. Надо просто добавить процессам роли. В зависимости от номера MPI процесса в подгруппе внутри узла, процесс будет превращаться либо в CPU-only, либо в GPU-only.

Дальше надо (эмпирически) определить, сколько ядер надо выдать GPU-only процессам для управления заданиями и обменом данными. Эти процессы не выполняют ресурсоемких вычислений на CPU, они вполне могут обойтись и одним ядром CPU. Но если обмен данными

обрабатывается в многопоточном режиме, то может быть выгодно дать GPU-only процессам не по одному, а по несколько ядер, чтобы обмены работали быстрее, ну а все остальные ядра, конечно, надо отдать CPU-only процессам, которые выполняют ресурсоемкие вычисления. Тут все зависит от того, сколько CPU ядер и сколько девайсов на узле. Если ядер мало, а девайсов много, то GPU процессам придется обойтись одним ядром. Правда в этом случае, когда девайсов много, и гетерогенный режим не будет иметь смысла. На фоне большого количества мощных девайсов вклад от CPU будет незаметен, проще сразу все ресурсы CPU отдать под обработку обменов между девайсами.

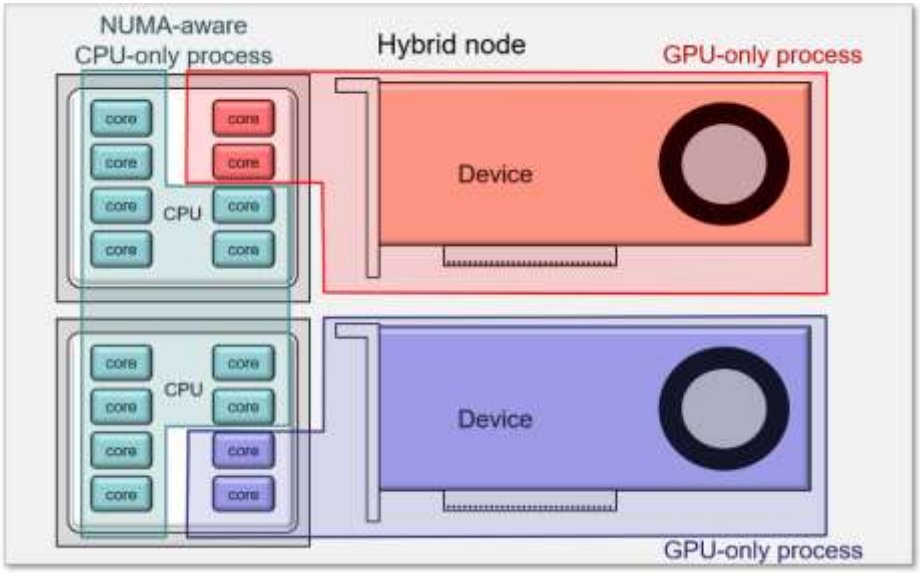
Рассмотрим пример. Пусть у нас есть гибридный узел, на котором два многоядерных CPU и два ускорителя GPU – итого 4 устройства. Запускаем по 4 процесса на узел, то есть в 4 раза большее число процессов, чем количество узлов кластера, которые хотим задействовать. Группируем процессы по подгруппам внутри узла (используя, например, MPI_Comm_split_type с MPI_COMM_TYPE_SHARED). Первые два процесса в подгруппах берут CPU-only роль, вторые два – GPU-only роль. Привязываем CPU процессы к процессорам (выставляем affinity), чтобы они не метались между CPU сокетами и не вносили NUMA фактор. В качестве общественной нагрузки поровну привязываем к CPU сокетами и GPU процессы, чтобы они тоже не мигрировали и не огребали NUMA потерь при копировании буферов. В общем все процессы обязательно куда надо привязываем. Получается такое распределение ресурсов:




Обратите внимание, что GPU-only процесс хоть и вычисляет на GPU, но выполняется тоже на CPU, ему тоже нужны ядра для выполнения программы, вычислительного алгоритма на верхнем уровне и организации обмена данными.

Если многопоточное распараллеливание CPU-only версии умеет работать с NUMA, то есть там есть привязка и NUMA-placement (данные размещаются в “ближней” памяти), то в CPU-only роли может быть один процесс на все CPU, как на картинке ниже.

Ну и все, гетерогенный режим готов. Правда есть еще один нюанс. На верхнем уровне алгоритмы CPU и GPU версий должны совпадать. Верхний уровень, это та программа, которая выполняется на CPU в GPU-only версии. На этом уровне алгоритм должен соответствовать CPU версии, иначе возникнет рассинхрон и процессы встанут в deadlock.

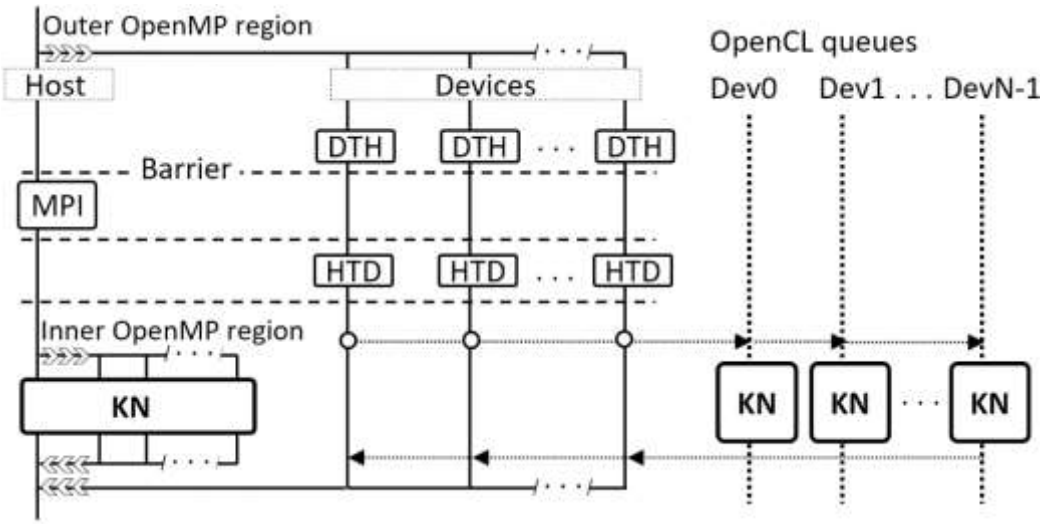



Возьмем наш пример с солвером из практического задания (оно дальше в разделе 8, но на лекциях уже было ранее). Сам алгоритм солвера должен быть записан одинаково, вызовы операций SpMV, dot, axpby и обработка скалярных величин и всей логики работы должны быть идентичны. Внутри кернелов, конечно, реализации отличаются для CPU и GPU версий.

### 6.2.2 Разделение на уровне OpenMP. Подход 1 процесс – 1 гибридный узел

Декомпозицию второго уровня можно втащить в MPI процесс. Это более сложный и менее удобный способ. Он может иметь смысл, если кернелы имеют очень низкую арифметическую интенсивность, и приходится бороться за каждый процент.

Если код умеет работать с множественными подобластями, то почему бы и не распределять работу по множественным устройствам внутри одного процесса? OpenCL это позволяет. Под каждую подобласть каждого девайса создадим очереди и будем отправлять в них кернелы – в цикле по подобластям, или из OpenMP нитей, ответственных за девайс. Тут может быть масса способов реализации, но все они, конечно, посложнее, чем предыдущий вариант. Рассмотрим пример. Пусть есть некоторый кернел, требующий обновление гало. Это может-быть какое-нибудь SpMV или знакомая по всем предыдущим главам операция расчета потоков через грани контрольных объемов. Вот, например, схемка выполнения обновления гало и кернела с использованием вложенной параллельной области (региона) OpenMP:

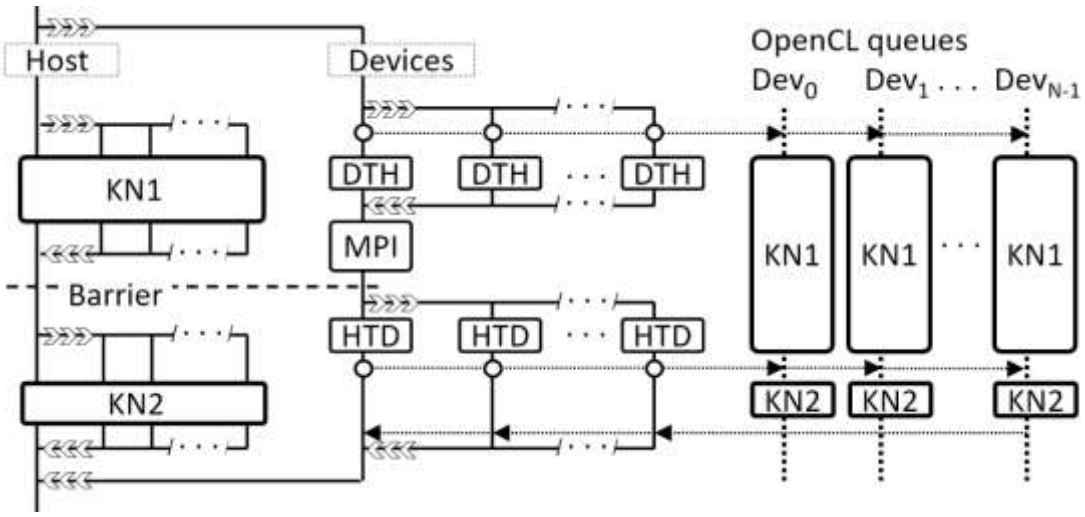


Разберемся, что тут понарисовано. Внешняя область OpenMP порождает нити в количестве на одну больше, чем планируется задействовать девайсов с помощью OpenCL. Количество подобластей второго уровня – такое же. Одна подобласть идет всем ядрам CPU, остальные подобласти идут на свои девайсы. Балансировка подобластей второго уровня – согласно фактической производительности CPU и ускорителей.

Главная нить будет отвечать за вычисления на CPU (host) для одной подобласти, отданной всем процессорным ядрам, доступным для вычислений данному процессу. Остальные нити – нити девайсов, будут обрабатывать подобласти ускорителей.

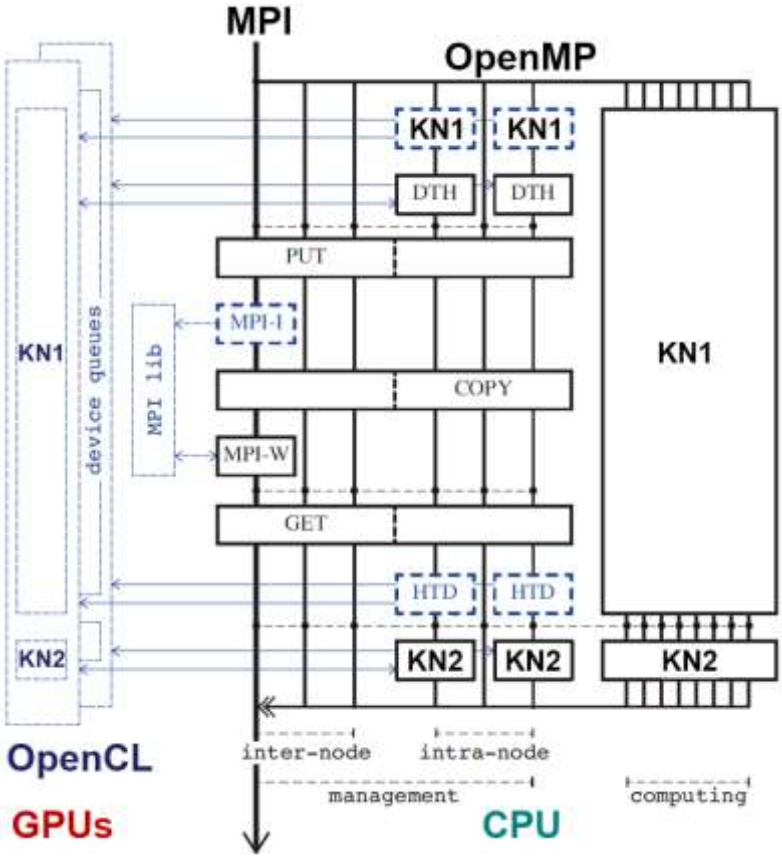
На этой картинке выше сначала выполняется операция обновления гало, потом вычисления. Итак, нити девайсов запускают обмены данными с девайсами – DTH (Device To Host), выкачивают оттуда данные по интерфейсным ячейкам. Затем, имея данные со всех подобластей процесса, главная нить выполняет обмен данными с соседними процессами с других узлов кластера средствами MPI. Затем полученные данные в гало раздаются в подобласти девайсов – копирование с хоста на девайс HTD (Host To Device). Теперь данные в гало обновлены, выполняется запуск кернела – KN. Кернелы запускаются на всех девайсах, а на CPU главная нить открывает вложенную параллельную область, которая задействует доступные для вычислений ядра, и выполняет объем работы, доставшийся CPU.

Это было выполнение кернела с синхронным вариантом обмена данными. А мы знаем, что лучше делать overlap. Разделим выполнение кернела на два этапа: KN1 – вызов кернела только для внутренних ячеек, KN2 – вызов только для интерфейсных ячеек, для которого и требуется обновление гало. Схема выполнения гетерогенных вычислений изменится:




Теперь у верхней области OpenMP две нити: одна отвечает за вычисления на CPU, другая за все обмены и вычисления на всех девайсах. CPU-шная нить разворачивает вложенную область и считает первый этап кернела. Управляющая нить раскрывает вложенную область (что вовсе не обязательно, можно оставить так, как было на предыдущей картинке) по одной нити на девайс. Каждая нить девайса запускает на исполнение на своем девайсе первый этап кернела (это символизирует вот там кружок и пунктирная стрелочка). Девайсы начинают считать в фоновом режиме свою работу. В это время нити девайсов, как и в предыдущем случае, выкачивают интерфейс с девайсов. Затем вложенная область девайсов закрывается, выполняется MPI обмен данными. Все это время и CPU, и девайсы заняты вычислениями. Когда MPI обмен завершается, нити девайсов закачивают в свои девайсы полученные данные для гало. Когда обмен завершен, выполняется второй этап кернела KN2 – для интерфейса.

С вложенными областями OpenMP могут возникать неудобства. В частности, могут быть проблемы с привязкой потоков вложенной области к ядрам. Зачастую нити начинают не слушаться и разбегаются по не тем ядрам. Поэтому удобнее хотя бы вычислительные нити держать на верхнем регионе OpenMP. А можно сделать статическое распределение нитей по ролям и делать синхронизацию по подгруппам (teams). Также можно использовать вложенную область для обменов и управления, чтобы не заморачиваться с подгруппами. Это проще и удобнее. Версия для одного уровня OpenMP показана на картинке ниже.




Тут нити делятся на 3 группы: 1) группа обменов между узлами (inter-node); 2) группа обменов внутри узла и вычислений на ускорителях (intra-node); 3) группа вычислений на CPU (computing) – самая большая. Первые две группы могут плодиться из двух нитей вложенным регионом. Третья группа строго верхний регион. На этой схеме почти все то же самое, как в предыдущей версии, только показано более подробно. Операция put – это когда мы выгребли интерфейс с девайсов, эти данные надо расставить по буферам для MPI сообщений (на предыдущей картинке она, конечно, тоже незримо присутствует, но не была показана для простоты, чтобы сразу так не пугать людей). То есть, put – это перестановка данных с внутриузловых буферов обмена данными с девайсами в буфера сообщений на отправку соседним MPI процессам. Название выбрано от балды (*англ*. put – положить), чтобы просто как-то обозначить. MPI-I – это неблокирующая инициализационная часть обменов (MPI_Isend, MPI_Irecv). MPI-W – это ожидание завершения MPI обменов (MPI_Waitall). Операция copy – перестановка данных из “исходящих” внутриузловых буферов, в которых лежат данные по интерфейсу с девайсов, во “входящие” буфера для гало на девайсах. Эта операция обеспечивает внутриузловой обмен данными по внутренним границам между подобластями девайсов. Интерфейс от одних девайсов перекладываем (копируем) в гало других девайсов.

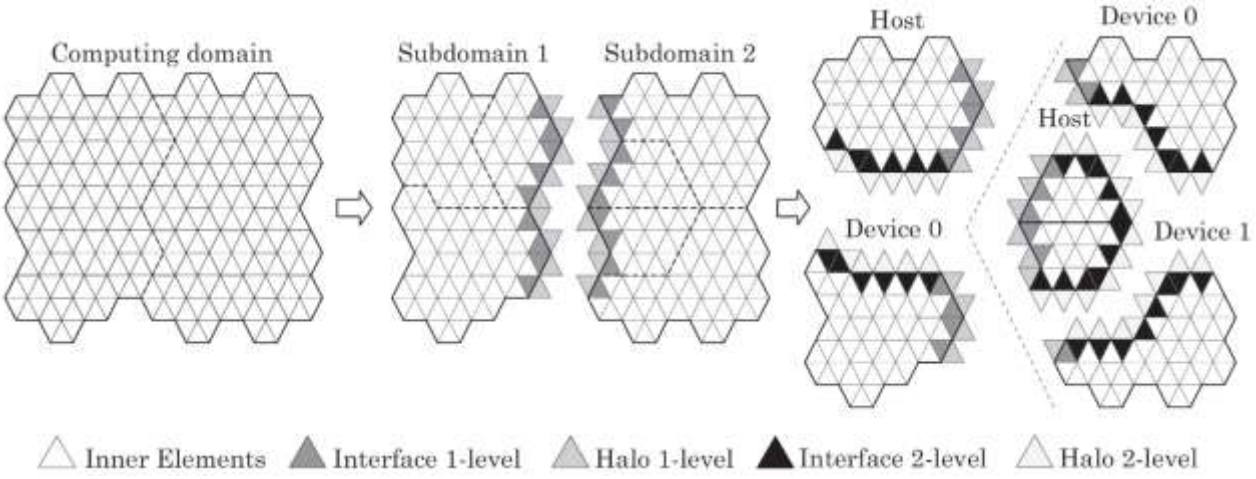
Когда MPI обмены завершились, после MPI-W, выполняется операция get – аналогично put, но в обратную сторону. Get растаскивает входящие MPI сообщения по гало-буферам девайсов на хосте.

Put, get, copy выполняются в многопоточном режиме с целью ускорения, поскольку там небыстрая выборка по косвенной адресации, всякая там перестановка. С OpenMP эти операции легко в несколько раз ускоряются, что достаточно важно и оказывается существенным для общего времени выполнения обмена данными. Оптимальное количество нитей определяется эмпирически. В общем, наверное, пока не очень все понятно с этими DTH, put, copy, get, HTD? Тогда читаем следующий раздел с поясняющими картинками.

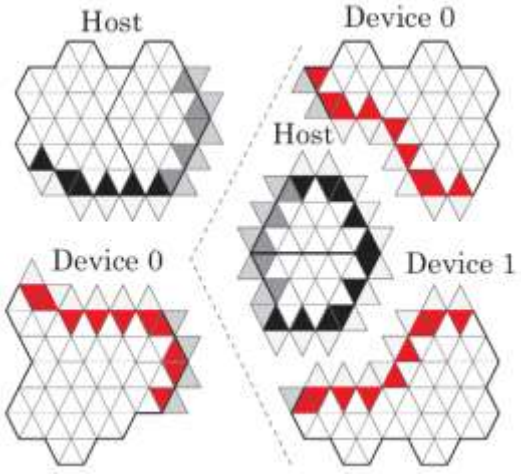
## 6.3 Организация обмена данными

### 6.3.1 Этапы обмена данными

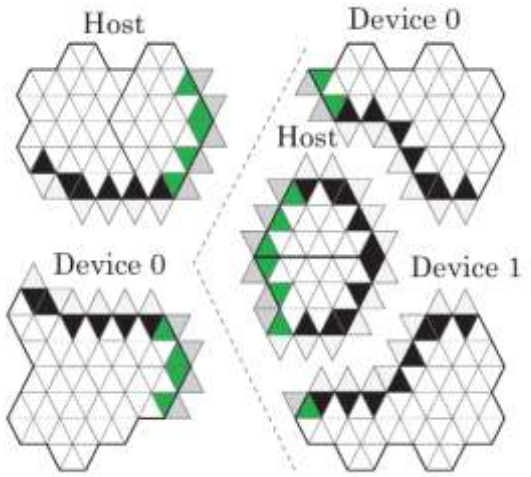
В предыдущем разделе фигурировали всякие этапы обмена данными – DTH, put, MPI-I, copy, MPI-W, get, HTD. Чтобы стало понятнее, что там происходит, посмотрим картинки. Пусть у нас есть какая-то сетка (слева). Поделим ее на две подобласти первого уровня – между двумя гибридными узлами (по середине). Потом поделим каждую подобласть на подобласти вычислительных устройств на узлах (справа). Пусть на одном узле 2 подобласти – CPU (host) и один девайс, на втором узле 3 подобласти – CPU и два девайса.



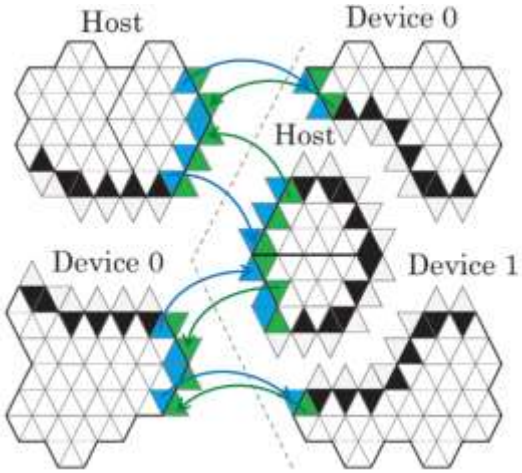

На операции DTH мы выкачиваем с девайсов их интерфейс, вот он красненький:



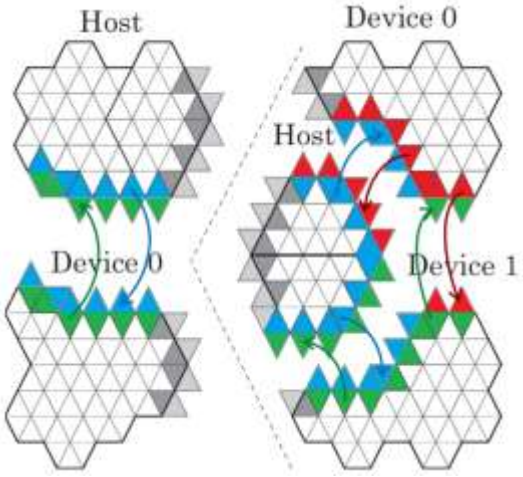
Далее пакуем внешний интерфейс (те ячейки, которые смежные с ячейками с другого узла) в сообщения на отправку – put. Это вот эти ячейки зеленые:



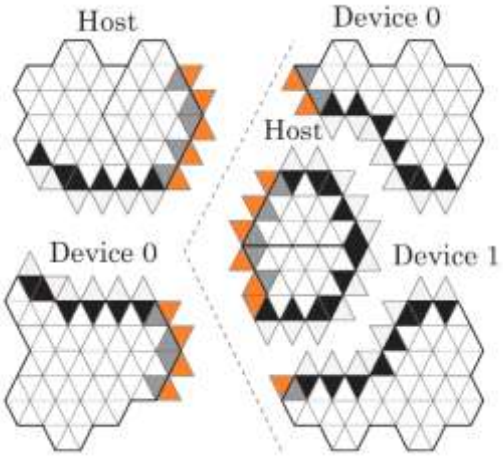

Инициируем обмен данными – MPI-I. В фоновом режиме начинают работать MPI_Isend, MPI_Irecv. Происходит inter-node перестановка интерфейс-гало по коммуникационной сети:



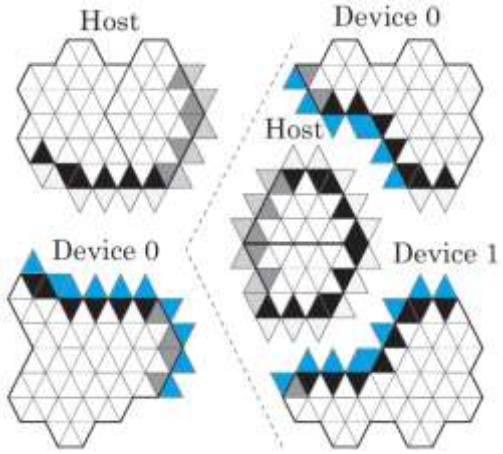

Пока MPI выполняет в фоновом режиме обмен, сделаем intra-node перестановку интерфейс-гало для границ между подобластями внутри одного узла. Это операция copy. Отметим, что одна ячейка может одновременно входить и во внешний и во внутренний интерфейс (если у нее оказались соседние ячейки и из внутренней, и из внешней подобласти).



Дожидаемся завершения MPI обменов – MPI-W. Когда обмены завершились, растаскиваем данные из входящих MPI сообщений по подобластям, операция get, в вот эти оранжевые ячейки на прием:




Мы сделали перестановку по внутреннему интерфейсу-гало на copy и разложили входящие сообщения на get. У подобласти хоста (CPU) данные были сразу помещены куда надо, а у девайсов они все еще в промежуточных буферах на хосте. Делаем им копирование на девайсы – HTD. Заливаем гало на девайсы. Гало девайсов – это вот:




Ну и все. Обмен завершен. Всего-то 6 этапов.

### 6.3.2 Overlap – сокрытие обменов за вычислениями

С этим оверлапом мы с 4-й главы разбираемся, уже, надеюсь, разобрались. Осталось обсудить еще небольшой нюанс для гетерогенного режима, ну и как-то доразобраться со временем выполнения кернела с обновлением гало.

Обозначим время буквой $t$.

$t_{HD}$ – время intra-node обменов между хостом и девайсами (включая все эти перестановки на хосте, которые copy).

$t_{MPI}$ – время inter-node MPI обменов между узлами (включая запаковку и распаковку, которые put и get).

Пусть $t_{Dev}^{k}$ – затраты времени в пересчете на одну ячейку у $k$-го девайса (величина, обратно пропорциональная производительности). Девайс – пока не важно, CPU это или GPU, просто вычислительное устройство. Пусть $N_{OWN}^{k}$ – число собственных ячеек в подобласти $k$-

го девайса, $N^k_{INNER}$ – число внутренних ячеек в ней, $n^k_{IFACE}$ – число интерфейсных ячеек (маленькой буквой $n$, что как бы символизирует, что интерфейсных ячеек сильно меньше, чем внутренних).

Для кернела с обновлением гало в синхронном режиме (сначала обмен, потом вычисления) получим такое время:

$$t_{SYNC} = t_{HD} + t_{MPI} + \max_k(N^k_{OWN} t^k_{Dev}).$$

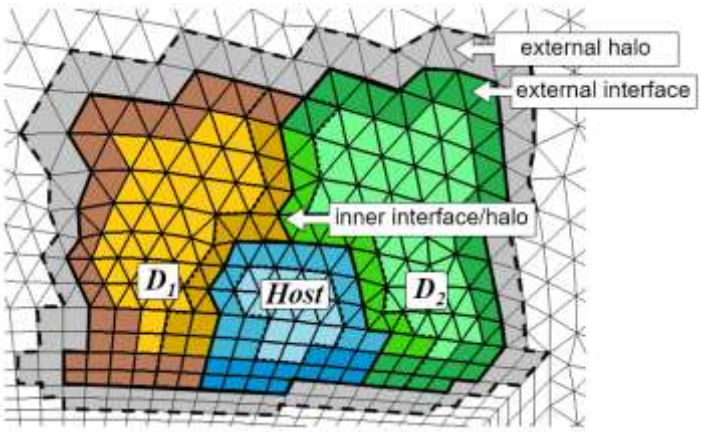
То есть просто получилась сумма времен для обменов внутри узла, обменов по MPI и вычислений (время вычислений – произведение времени на ячейку на число ячеек, ваш К.О.).

В режиме overlap обмены и вычисления внутренних узлов выполняются одновременно. Получится такая формула:

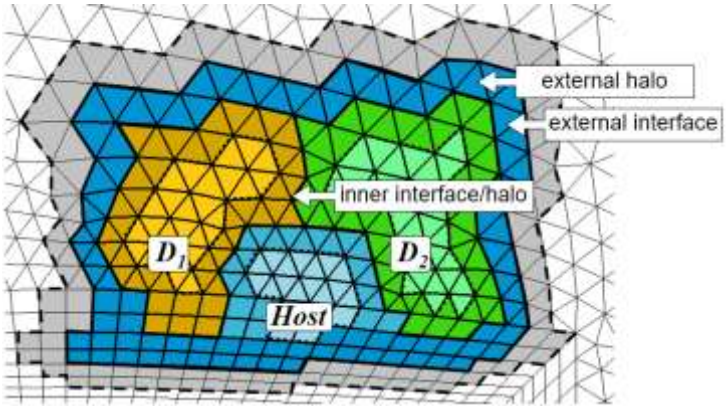
$$t_{OVER} = \max\left(t_{HD} + t_{MPI},\ \max_k(N^k_{INNER} t^k_{Dev})\right) \ + \ \max_k(n^k_{IFACE} t^k_{Dev}).$$

Как видим, сумма схлопнулась в максимум из времени обменов и времени вычислений внутренних ячеек, то есть одно слагаемое спряталось за другим.

Попробуем еще схитрить и замутить двойной оверлап. Посмотрим картинку. Вот какая-то подобласть гибридного узла, в ней 3 подобласти 2-го уровня – подобласть хоста, то есть CPU, и подобласти двух девайсов. Внешние границы (external) – дорогие, там несколько этапов обменов, сначала выкачать с девайсов, потом передать по MPI, потом обратно в девайсы закачать. Внутренние границы – дешевле, там нет MPI обменов по сети между узлами. Вот на картинке выше две большие подобласти девайсов, и маленькая синяя подобласть CPU. Предполагаем, что CPU слабее девайсов, поэтому ему досталось меньше работы. И у девайсов, и у CPU есть внешние границы, то есть все подобласти участвуют в inter-node обменах.




Сделаем финт. Припишем весь внешний интерфейс CPU. Пусть все интерфейсные ячейки подобласти всего узла обрабатываются на процессоре. Получилась вот такая прелесть:



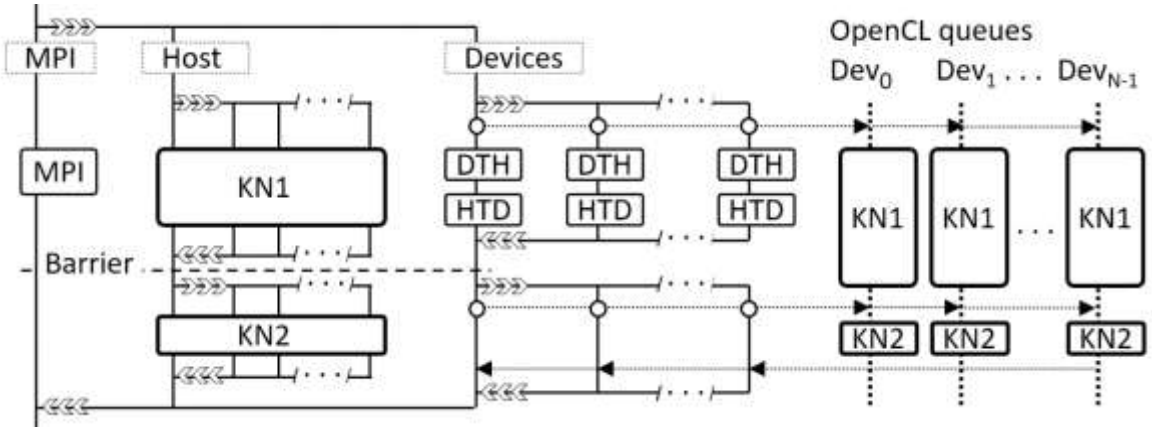
Что мы тут видим? Правильно, у подобластей девайсов вообще нет внешнего интерфейса. Прослойка из CPU-шных ячеек изолировала девайсы от внешнего мира. То есть данные с девайсов вообще не участвуют в MPI обменах. А что это значит? Значит – обмен с девайсами можно оверлапнуть с MPI обменом. Получим такую формулу:

$$t_{2xOVER} = \max\left(\max\left(t_{HD}, t_{MPI}\right), \max_k(N_{INNER}^k t_{Dev}^k)\right) + n_{IFACE}^{CPU} t_{CPU}$$

В максимум схлопнулась еще одна сумма – теперь host-device обмены и MPI обмены скрылись друг за другом. Но есть подстава. А именно последний член в сумме. Это время вычислений именно на CPU именно его интерфейса, которое стало сильно побольше, чем в предыдущем случае, и его никуда не спрятать. Там раньше был максимум по всем девайсам, но теперь с очевидностью максимум получается именно на CPU, значит, взятие максимума по девайсам можно выкинуть из формулы. Из-за того, что вырос вклад вычислений по интерфейсу, такую схему обменов имеет смысл использовать только если, во-первых, времени вычислений внутренней части не хватает, чтобы скрыть обмены, и эти обмены сильно выпирают из-за вычислений внутренних ячеек, и, во-вторых, кернел должен быть достаточно дешевым (с точки зрения времени вычислений), а CPU достаточно мощным, чтобы выигрыш от сокрытия обменов не был перекрыт увеличением времени вычислений интерфейса на CPU. Другими словами, то, что мы скрываем, должно быть больше, чем дополнительный оверхед на интерфейс, который получаем: $\min\left(t_{HD}, t_{MPI}\right) > n_{IFACE}^{CPU} t_{CPU}$.

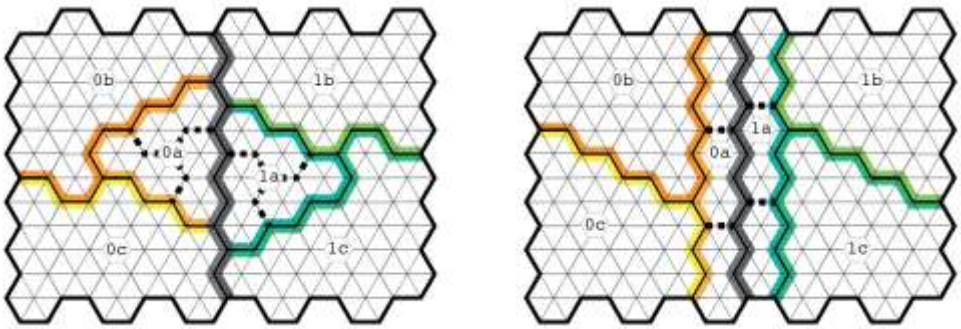
Честно говоря, обычно на практике мы такой двойной оверлап (пока) не используем.

Диаграммка для двойного оверлапа будет такой:

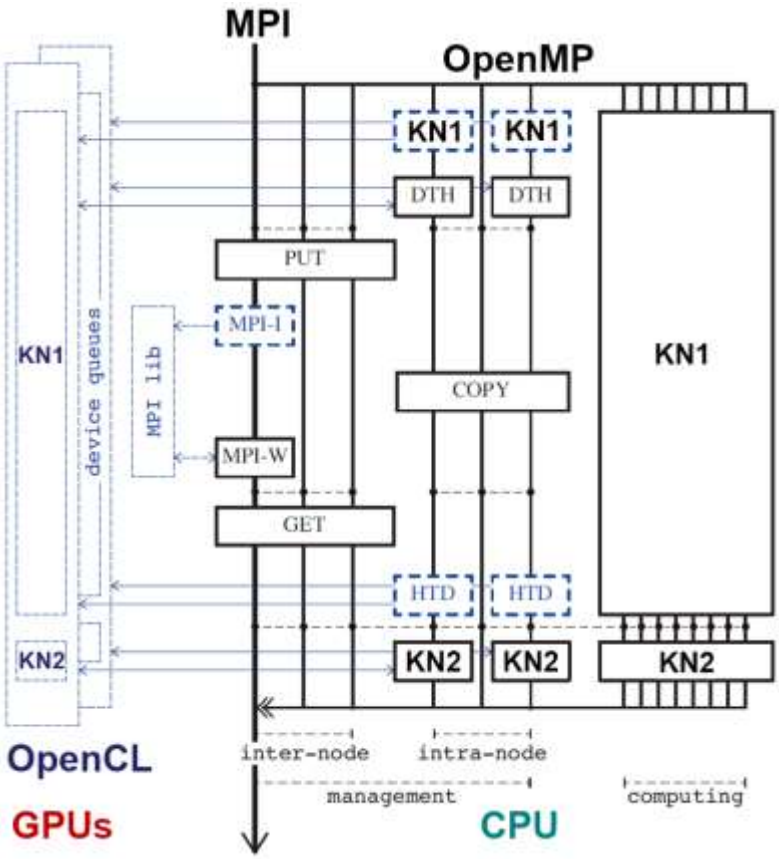



Что, непонятно ничегошеньки? А потому что на лекции надо ходить.

Закрепляем пройденное. Вот пример – сеточка треугольная, два гибридных узла, на каждом хост (a) с тремя ядрами под вычисления и два ускорителя (b и c). Какой вариант декомпозиции просто для оверлапа, а какой подойдет для двойного оверлапа?



А для версии с одноуровневым OpenMP регионом оверлап превращается в элегантный двойной оверлап просто отключением синхронизации между inter-node и intra-node подгруппами нитей.




## 6.4 Производительность гетерогенного режима вычислений

А когда вообще имеет смысл заморачиваться с одновременным использованием процессоров и ускорителей на гибридной системе? Начнем с более общего вопроса: а когда вообще имеет смысл заморачиваться? Смысл тратить существенные усилия имеется, когда эти усилия дают какой-то существенный выигрыш, какие-то преимущества. А какие преимущества дает использование CPU? Преимущество – сокращение временных затрат на расчет за счет повышения производительности вычислений на гибридном узле. Значит, чтобы сокращение времени было существенным, вклад производительности CPU в суммарную производительность узла должен быть существенным.

Рассмотрим пример подходящей для гетерогенщины системы – суперкомпьютер Ломоносов-2 МГУ. Там гибридные узлы с одним 14-ядерным Intel Xeon E5-2697 v3 (Haswell) и одним GPU NVIDIA K40 (Kepler). У процессора пропускная способность памяти в районе 70 GB/s, у девайса – около 290 GB/s. То есть можно выгадать где-то четверть по скорости. Четверть – это сколько? Это процентов 25. Ну за 25% можно и повозиться.

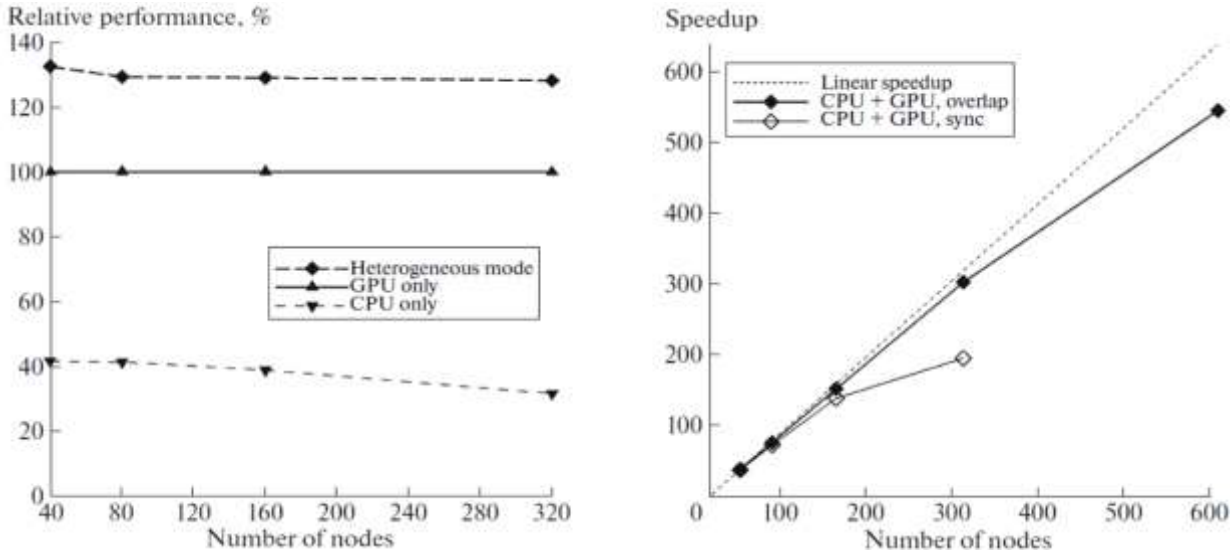
Результаты такого “повозиться” представлены, например, тут:

https://doi.org/10.1016/j.compfluid.2018.03.011

https://doi.org/10.1134/S1064562418060194

Там был экспериментальный CFD код Тапир для моделирования течения сжимаемого газа на неструктурированной смешанной сетке (полиномиальная реконструкция на линейных полиномах). Вот картинка для теста на сетке из 500 млн ячеек. Слева показана относительная

производительность, за 100% взята производительность GPU-only режима. В результате продемонстрирован выигрыш около 30% за счет использования CPU в вычислениях совместно с GPU. Что дает режим overlap, показано на графике параллельного ускорения справа. Синхронная схема обменов быстро загнулась, после 160 узлов уже наступила деградация, оверлап же легко натянулся на все доступные узлы в количестве 640 штук.

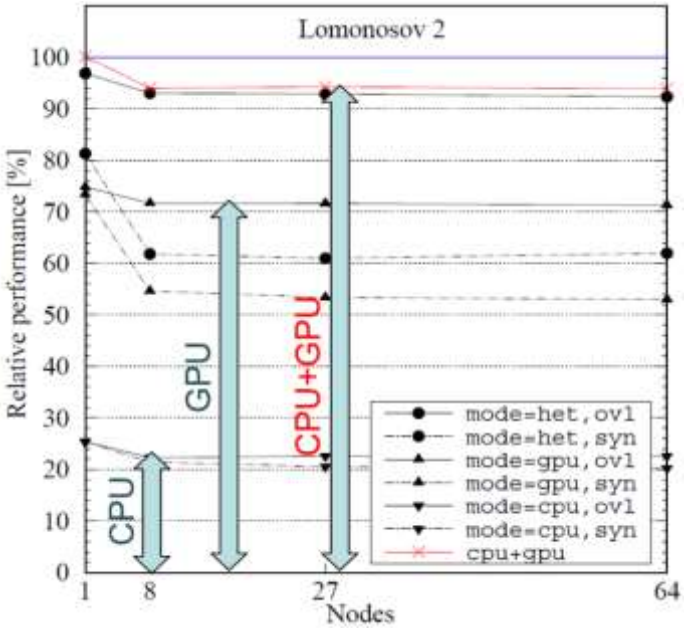



Или вот еще пример. CFD код HPC[2] для моделирования несжимаемых течений, тоже на неструктурированных сетках. Подробности тут:

https://doi.org/10.1016/j.compfluid.2020.104768

https://doi.org/10.1016/j.compfluid.2018.01.034

Вот на графике масштабирования (по 2 млн ячеек на узел) показано сравнение CPU-only (cpu), GPU-only (gpu) и гетерогенного CPU+GPU (het) режимов вычислений, а также синхронного (syn) и оверлап (ovl) режима обмена данными. За 100% тут взята максимально достигнутая гетерогенная производительность одного узла. Видно, какой выигрыш дает оверлап – сплошные линии против пунктирных. Также видно, что дает совместное использование CPU и GPU –получается выигрыш в скорости около 30 %.



Возьмем более современную конфигурацию. Вот, например, наш институтский кластер К60 ИПМ им. М. В. Келдыша РАН. Там гибридные узлы с двумя 16-ядерными “голдами” Intel Xeon 6142 и четырьмя NVIDIA V100 (Volta). Суммарный бэндвис двух CPU – 250 GB/s, GPU – 3.6 TB/s, потенциальный выигрыш получается всего порядка 7%. При том, что и этот выигрыш будет недостижим, поскольку на обслуживание обменов четырех мощных девайсов уйдет почти вся мощь проца. Ну а на какой-нибудь безумной GPU-шной машинке типа сберовского Кристофари на узлах аж по 16 Вольт против двух несчастных платинумов. Стоит ли говорить о бессмысленности гетерогенных вычислений на такой конфигурации?

Вывод: раньше в гетерогенных вычислениях был смысл, сейчас GPU сильно обогнали CPU, смысл пропал. Но, вероятно, скоро CPU нанесут ответный удар, разобравшись с пропускной способностью памяти. Либо поставят на борт быструю память, как у GPU (на Intel Xeon Phi уже пытались), либо прокачают каналы памяти, чтобы выйти на сопоставимый уровень пропускной способности. Более подробно об этом было во второй главе. Так вот, когда CPU подтянут производительность, снова появится смысл в гетерогенном режиме. А если CPU немного перестарается и достигнет паритета с GPU, то вычислительные GPU станут никому не нужны, и смысл снова пропадет, но уже в другую сторону. Такая вот непостоянная штука этот смысл. Время покажет.

## 6.5 Внеклассное чтение

Спецификации стандартов OpenCL живут тут:

- https://www.khronos.org/opencl/
- https://www.khronos.org/registry/OpenCL/

Стандарт 1.2 вполне подойдет:

- https://www.khronos.org/registry/OpenCL//sdk/1.2/docs/OpenCL-1.2-refcard.pdf – шпаргалка
- https://www.khronos.org/registry/OpenCL/specs/opencl-1.2.pdf – полная спецификация.

Статьи, материалы из которых использованы в этой главе:

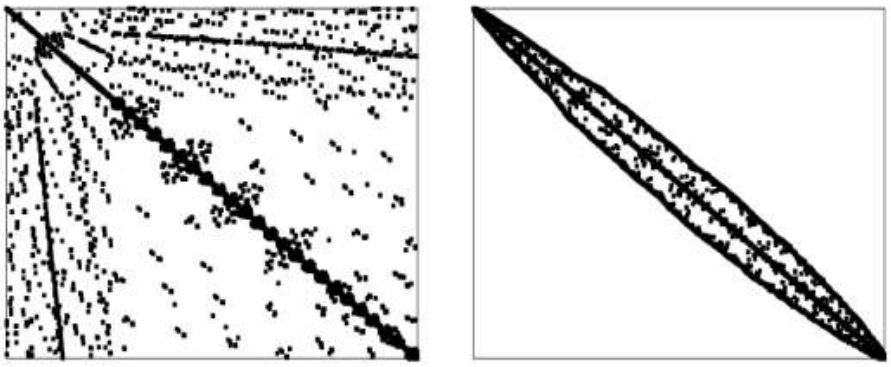

- A.Gorobets, S.Soukov, P.Bogdanov. Multilevel parallelization for simulating turbulent flows on most kinds of hybrid supercomputers. Computers & Fluids. Volume 173, Pages 171–177. 2018.https://doi.org/10.1016/j.compfluid.2018.03.011
- S. A. Soukov, A. V. Gorobets. Heterogeneous Computing in Resource-Intensive CFD Simulations. Doklady Mathematic. 2018. Volume. 98, No. 2, Pages 1–3. https://doi.org/10.1134/S1064562418060194
- X. Alvarez-Farre, A. Gorobets F. X. Trias. A hierarchical parallel implementation for heterogeneous computing. Application to algebra-based CFD simulations on hybrid supercomputers. Computers & Fluids. Volume 214, 2021, 104768. https://doi.org/10.1016/j.compfluid.2020.104768
- X.Alvarez, A.Gorobets, F.X.Trias, R.Borrell, and G.Oyarzun. HPC$^2$ - a fully portable algebra-dominant framework for heterogeneous computing. Application to CFD. Computers & Fluids. Volume 173. Pages 285–292. 2018. https://doi.org/10.1016/j.compfluid.2018.01.034

# 7 Алгоритмы, методы, технологии, приемы для реализации параллельных вычислений

## 7.1 Вспомогательные алгоритмы

### 7.1.1 Алгоритм Катхилла – Макки

Алгоритм Катхилла – Макки (Cuthill-McKee) нам нужен для переупорядочивания. Есть у нас матрица с симметричным портретом размера N×N. Переупорядочить, значит переставить у матрицы строки и столбцы. Поменять индексы i и j означает поменять местами строки i и j и столбы i и j. Этот алгоритм ужимает ширину ленты разреженной матрицы, переставляя ей строки и столбцы. Что это для нас значит? У нас есть граф связей ячеек (или еще каких-то объектов), с которым мы уже имеем дело все предыдущие главы. У графа есть матрица смежности. Если мы применим алгоритм Катхилла – Макки к матрице смежности, то получим новую нумерацию вершин графа. Чем уже лента, тем ближе расположены номера столбцов ненулевых коэффициентов в строках. Чем ближе расположены номера столбцов – тем ближе, компактнее, будет в памяти доступ к позициям входного вектора в матрично-векторном произведении. Тем меньше будет кэш промахов. А для графа это будет означать, что, если мы перебираем соседей какой-то вершины, эти соседние вершины будут расположены в памяти более компактно. Доступ к памяти будет более компактный, улучшится локальность данных. Для вычислений сеточным методом это означает, что, когда мы, например, считаем потоки через грани, инцидентные граням ячейки в основном будут расположены более компактно. Когда мы перебираем соседние ячейки, чтобы посчитать какой-нибудь градиент или еще что-нибудь, выборка данных из этих ячеек будет более эффективной за счет более локального доступа к памяти. Нарисуем портрет матрицы, поставив черные квадратики в ненулевые позиции. Применение алгоритма выглядит так:



Слева матрица в совершенно неприличном виде. Когда применили переупорядочивание, получился приличный вид – справа. Чем меньше расстояние между черными квадратиками в строке, тем ближе соответствующие данные расположены в памяти. Хорошо видно, насколько локальнее получается доступ к данным.

Эффект от применения алгоритма прямо-таки радикальный (особенно при плохой исходной нумерации). При этом алгоритм простой и дешевый. По сути, это с небольшими модификациями просто обход графа в ширину. Это когда берем одну вершину, добавляем в набор, обходим всех ее соседей, добавляем в набор, обходим всех соседей этих соседей, добавляем, обходим соседей тех соседей… пока не кончится весь граф.

Модификация состоит в том, что когда обходим соседей, вершины добавляются в набор не абы как, а упорядоченно по возрастанию степеней этих вершин (т.е. по количеству соседей у вершин). Ну и еще там заморочка выбрать “периферийную” вершину, с которой надо начинать нумерацию. Но обычно никто не заморачивается, просто берут в графе любую вершину с наименьшей степенью и от нее нумеруют. Для сеточного метода это вполне нормально работает. Периферийной вершиной оказывается какой-нибудь угол на отшибе.

Алгоритм широко известный, легко найти в интернете:
https://www.geeksforgeeks.org/reverse-cuthill-mckee-algorithm/
https://en.wikipedia.org/wiki/Cuthill%E2%80%93McKee_algorithm …
ну и в первоисточнике: Cuthill, E., McKee, J.: Reducing the Bandwidth of Sparse Symmetric Matrices. In: Proceedings of the 1969 24th National Conference. pp. 157–172. ACM ’69, ACM, New York, NY, USA (1969), https://doi.org/10.1145/800195.805928

Чтобы не лазать по интернету, приведем алгоритм тут. Пусть есть у нас граф из N вершин. Если у нас не граф, а симметричная матрица, то берем портрет, выкидываем главную диагональ, называем это матрицей смежности и строим по ней графа. Переупорядочим вершины так, чтобы они в графе были покомпактнее, тогда в матрице в строках будут покомпактнее столбцы. Итак, строим набор вершин V (V – vertices) в нужном порядке.

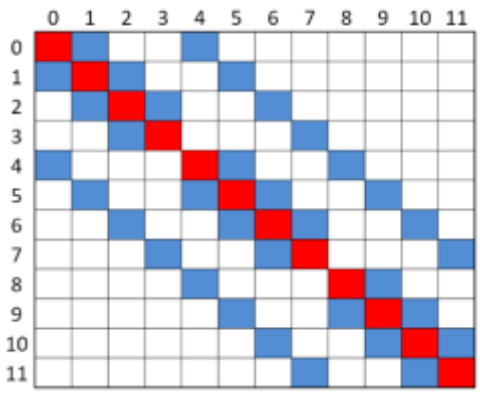
Берем первую попавшуюся вершину с минимальной степенью, добавляем ее номер в V. V[0] – хранит номер начальной вершины.
Повторяем для i = 0, …, N
Берем всех соседей вершины V[i], которых еще нет в V, упорядочиваем их по возрастанию степеней и пушбэкаем (*жарг.* дописываем, добавляем в конец) ровно в таком порядке в V.
Если размер V равен N – выходим, если нет, инкрементим i, повторяем манипуляцию.

А как по-быстрому узнавать, что вершины еще нет в V? Не будем же мы делать поиск по всему V!? Нет, можно сделать булочный массив (типа bool) по всем вершинам, занулить его, и маркировать при добавлении соответствующие позиции единичкой. Тогда сразу понятно, что уже добавлено, что еще нет.

Рассмотрим пример о том, как влияет нумерация на ширину ленты. Пусть есть сетка 3×4 из декартовой решетки, пронумеруем ячейки построчно сверху вниз:

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

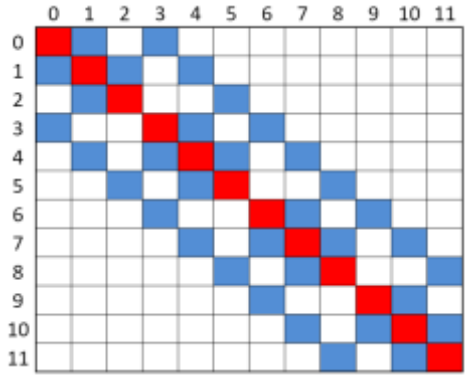
Матрица (смежности + главная диагональ) для такой нумерации:



Пронумеруем ячейки по-другому, не по строкам, а по столбцам:

| | | | |
|---|---|---|---|
| 0 | 3 | 6 | 9 |
| 1 | 4 | 7 | 10 |
| 2 | 5 | 8 | 11 |

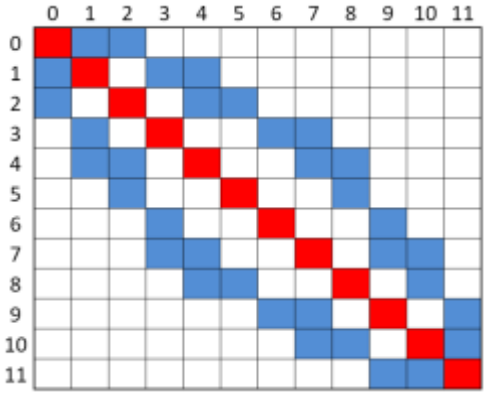
Матрица (смежности + главная диагональ) для такой нумерации:




Как видим, сетка та же, а ширина ленты стала меньше за счет более удобной нумерации. Ну с такой топологией решетки все просто, нумеруем по короткой стороне и все. Если строк больше, чем столбцов, то нумеруем построчно. И наоборот.

А вот что будет, если сделать Катхилла – Макки. Эта штука понятия не имеет о структуре сетки, о том, что это решетка. Она берет данные в общем виде и упорядочивает.

| | | | |
|---|---|---|---|
| 0 | 1 | 3 | 6 |
| 2 | 4 | 7 | 9 |
| 5 | 8 | 10 | 11 |

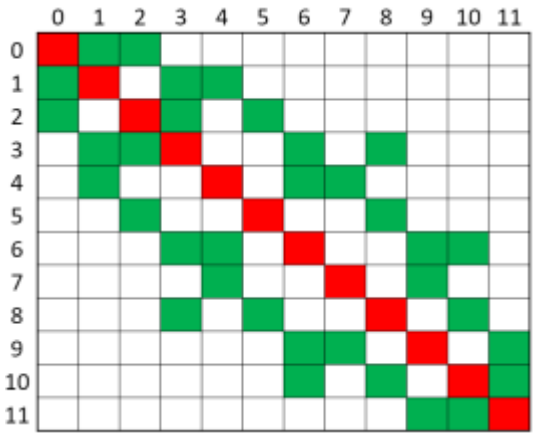



Получилось ну почти оптимально, немножко выперло посередине за минимальную ширину ленты. Вот такая штука, по-простому строит приличную нумерацию.

Обычно используют Reverse Cuthill – McKee вариант, в котором результат просто Cuthill – McKee разворачивают в обратном порядке. Получается немного получше с точки зрения доступа к памяти при переборе вершин (строк матрицы) по порядку.

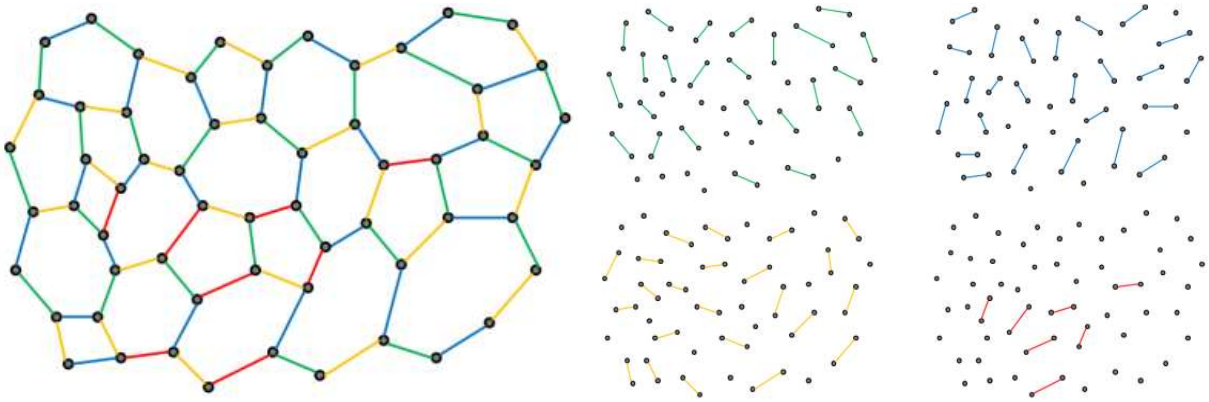
А если взять нумерацию просто обходом графа “в ширину”, то будет похуже:

| | | | |
|---|---|---|---|
| 0 | 1 | 4 | 7 |
| 2 | 3 | 6 | 9 |
| 5 | 8 | 10 | 11 |




### 7.1.2 Разрыв зависимостей раскраской графа

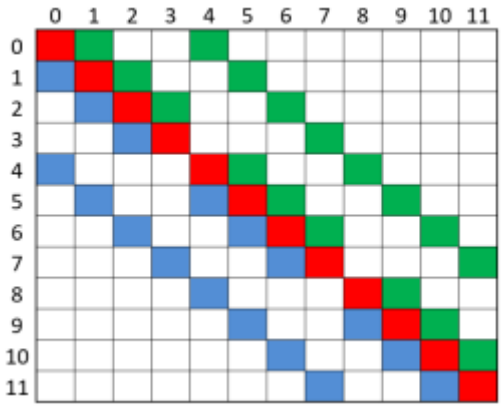
В разделе 3.3.5 мы уже красили графа для OpenMP распараллеливания. Красили ему ребра, чтобы можно было обрабатывать данные, ассоциированные с ребрами, и добавлять результаты в вершины, не опасаясь пересечений по данным.



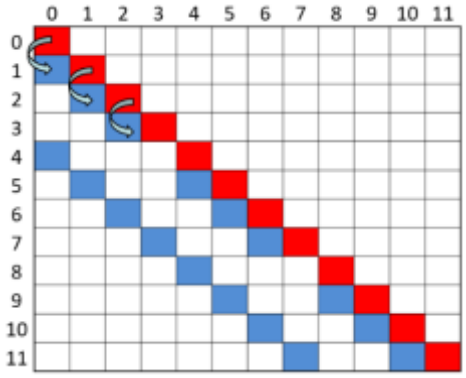
Иногда полезно покрасить графу вершины. Раскраска по вершинам – это когда у вершин нет соседей того же цвета. При этом цветов желательно чтобы было как можно меньше. А зачем это может понадобиться?

Самый обычный пример – всякие прямые или обратные подстановки при решении СЛАУ с разреженной матрицей треугольной формы (это когда все коэффициенты сверху или снизу от главной диагонали равны нулю). Вот возьмем ту же сетку из декартовой решетки и ее матрицу. Синее – это нижне-треугольная часть (L), зеленое – верхне-треугольная. Красненькое – это главная диагональ (D).

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |



Если нам надо решать систему с треугольной матрицей, пусть нижней, то мы берем строку, где всего один неноль, и сразу находим решение для этой позиции (поделив соответствующее значение в правой части на этот неноль). На каждой следующей строке у нас уже для всех ненулей, кроме одного, есть решение с предыдущих строк. Решение получается на халяву прямой подстановкой.




Но тут получается зависимость по данным. Мы не можем обработать строку, не обработав все предыдущие. И как тогда это параллелить? Приходится придумывать разные способы, анализировать структуру матрицы, делать всякие манипуляции с графом, делить его как-то на части и прочие бесчинства учинять, чтобы порвать с зависимостями.

Возьмем для примера метод Гаусса – Зейделя (Gauss – Seidel). Это распространенный итерационный метод решения СЛАУ. Часто используется как сглаживатель, aka smoother, в многосеточных методах, а также как предобуславливатель в проекционных методах.

$$\boldsymbol{Ax} = \boldsymbol{b}$$
$$\boldsymbol{A} = \boldsymbol{L} + \boldsymbol{D} + \boldsymbol{U}$$
$$(\boldsymbol{L} + \boldsymbol{D})\boldsymbol{x}^{k+1} = \boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{k}$$

Наверное, еще чаще используют SGS (Symmetric Gauss – Seidel), что примерно то же самое, только на одной итерации и в прямую, и в обратную сторону:

$$(\boldsymbol{L} + \boldsymbol{D})\boldsymbol{x}^{k+\frac{1}{2}} = \boldsymbol{b} - \boldsymbol{U}\boldsymbol{x}^{k}$$
$$(\boldsymbol{U} + \boldsymbol{D})\boldsymbol{x}^{k+1} = \boldsymbol{b} - \boldsymbol{L}\boldsymbol{x}^{k+1/2}$$

И вот эти все $(\boldsymbol{L} + \boldsymbol{D}), (\boldsymbol{U} + \boldsymbol{D})$ – это нижне- и верхне-треугольные матрицы, которые решаются прямой и обратной подстановкой, соответственно.

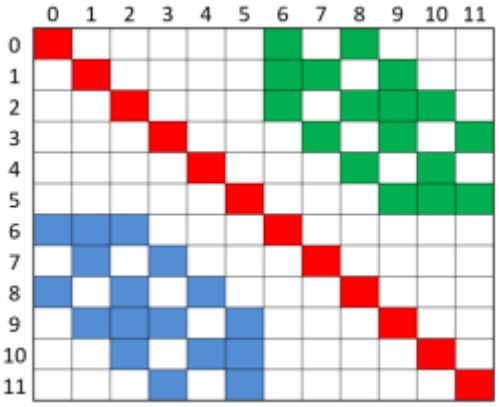
Порвем зависимости, переупорядочив неизвестные, то есть перенумеровав вершины графа, то есть ячейки сетки. Берем исходную нумерацию:

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 4 | 5 | 6 | 7 |
| 8 | 9 | 10 | 11 |

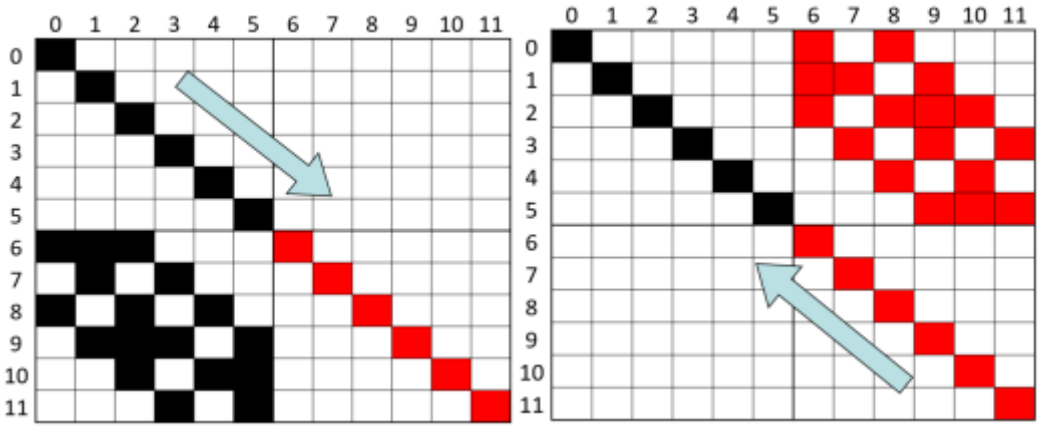
Красим вершины графа, а точнее, сразу ассоциированные с ними ячейки сетки в шахматном порядке в два цвета. Перенумеруем ячейки по цветам, сначала ячейки одного цвета, потом другого. Это будет красно-черное переупорядочивание (RB, Red-Black)

| | | | |
|---|---|---|---|
| 0 | 6 | 1 | 7 |
| 8 | 2 | 9 | 3 |
| 4 | 10 | 5 | 11 |

Матрица для такой нумерации получится интересного вида:




Теперь, если мы будем “решать” треугольную матрицу, то стало намного проще:




Решаем нижний треугольник – у черных ячеек блок диагональный, нет зависимостей. Идем сверху вниз, находим решение для всех черных позиций в параллельном режиме. Потом переходим дальше, у красных нет ненулей в “красном” блоке, нужны только черные, а их мы уже решили, просто подставляем. Опять-таки нет зависимостей, решаем в параллельном режиме красные. Обратная подстановка то же самое.

Получается, решаем систему в два этапа по цветам. Каждый этап не имеет внутренних зависимостей и легко делается параллельно, хоть в многопоточке OpenMP, хоть потоковой обработкой на GPU.

Правда, есть нюанс. С таким переупорядочиванием результат работы итерационного метода изменится. У SGS-то в чем фишка? “Возмущения”, “информация”, хорошо протаскиваются этой прямой-обратной подстановкой именно когда есть зависимости. Поэтому решение быстро устанавливается, итерационный метод быстро сходится (особенно если нумерация хорошо совпала с характерным направлением распространения возмущений). А раз мы зависимости порвали, формально это остался SGS, но, так сказать, потенция сильно провисла. От такой нумерации запросто может потребоваться вдвое больше итераций для получения решения той же точности. Но если на распараллеливании ускоримся раз в 100, то и какая разница, что итераций стало вдвое больше? Но когда будем мерять параллельное ускорение, надо быть честными и сравнивать с последовательной версией в оптимальной для итерационного метода нумерации.

А если у нас сетка неструктурированная? Тогда просто красим графа по вершинам в общем случае. Будет не красно-черный SGS, а цветной SGS. Потому что цветов уже будет не два, а непонятно сколько.

Графа от просто сетки можно красить простейшим жадным алгоритмом. Это, например, проходим графа “в ширину” (как Катхилла – Макки крутили), и раздаем вершинам первый доступный цвет. Если доступных цветов нет, добавляем еще. Получится, может, и не самым оптимальным образом, но для графа с сетки дискретизации по пространству обычно не сильно хуже. Цветом больше, цветом меньше – мало на что влияет. Это все очень простое, можно смотреть в любом интернете, в любой Википедии:

https://ru.wikipedia.org/wiki/Раскраска_графов

https://ru.wikipedia.org/wiki/Рёберная_раскраска

https://ru.wikipedia.org/wiki/Жадная_раскраска

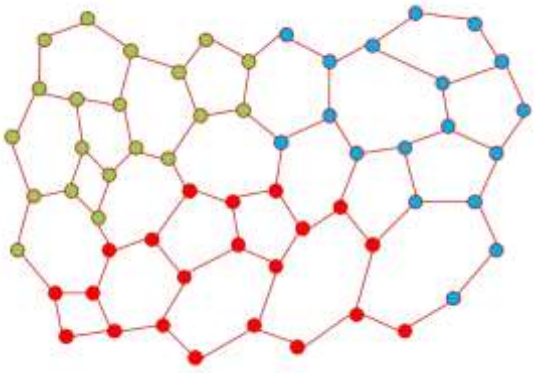
Этапов получится уже не 2, а сколько выйдет цветов. Внутри каждого цвета будет доступна параллельная обработка.

### 7.1.3 Прямой метод на основе дополнения Шура

То мы разбирались с итерационными методами, а если метод прямой? Например, метод прогонки же все знают? Кто не знает или не помнит, открывает интернет, вспоминает. Там такая же фигня, прямая-обратная подстановка с зависимостью по данным – чтоб находить следующее значение, надо знать предыдущие. Только так на халяву раскраской проблема уже не решается. Или же метод LU разложения, когда матрица $\boldsymbol{A}$ факторизуется на произведение нижне- и верхне-треугольных матриц: $\boldsymbol{A} = \boldsymbol{LU}$. Тогда система $\boldsymbol{Ax} = \boldsymbol{b}$ превращается в $\boldsymbol{A} = \boldsymbol{LUx} = \boldsymbol{b}$, которая решается влегкую прямой-обратной подстановкой: находится $\boldsymbol{Ly} = \boldsymbol{b}$, потом $\boldsymbol{Ux} = \boldsymbol{y}$. Но влегкую все выходит в последовательном случае. Матрицы L и U – это теперь не слагаемые, это множители. Они получаются сильно уже плотные, раскраской проблема тоже не решается. Есть широко используемый способ это дело распараллелить в рамках подхода декомпозиции сетки. Можно и под OpenMP, и под MPI применять.

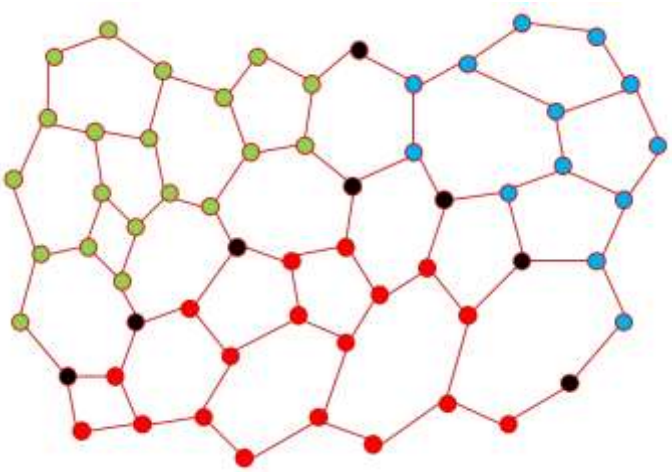
Пусть у нас есть сетка, по ней получается граф связей ячеек. Или есть просто граф, по которому строится матрица (матрица смежности + главная диагональ). Или просто матрица, по которой строится граф, беря по портрету матрицу смежности. В общем, так или иначе, матрице сопоставился граф. Откуда он взялся уже не важно.

Делим графа на подобласти алгоритмом декомпозиции (см. раздел 4.3.1).

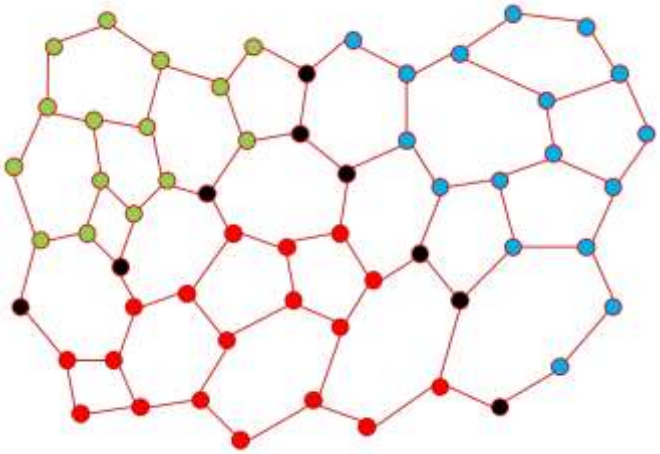


Вот поделили наш примерный граф на 3 подобласти. Выделим в нем интерфейсную зону. Но не как раньше. Это уже не как интерфейс-гало в 4 главе. Там было внахлест: интерфейс одной подобласти – это гало другой. А тут без нахлеста. Сделаем просто интерфейс, такое

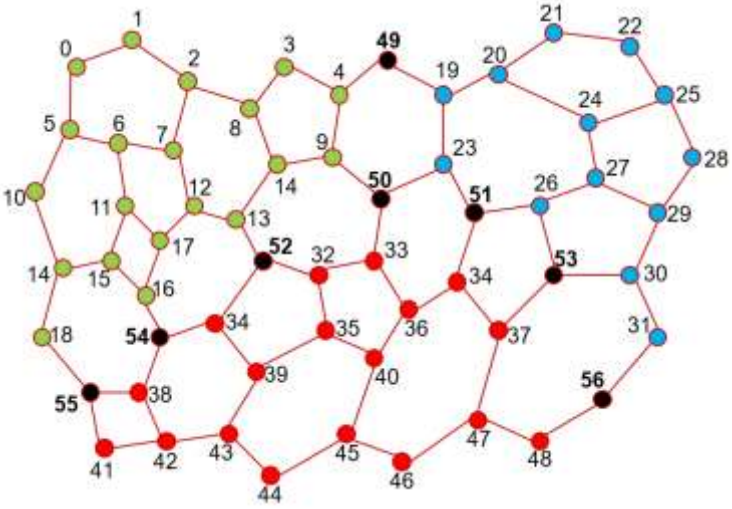
подмножество вершин в графе, что все остальные вершины, не входящие в это подмножество, не имеют связей с вершинами из других подобластей. Например так:



Черные вершины – интерфейс. Его можно выбрать по-разному. Вот так тоже можно:



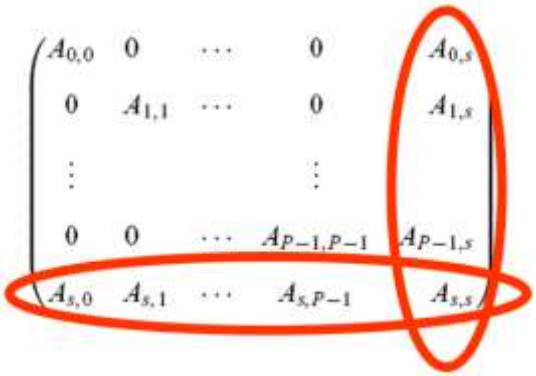
Обычно распределяют, чтоб подобластям досталось примерно поровну интерфейса. Те вершины, что не попали в интерфейс назовем внутренними. Пронумеруем вершины так, чтобы сначала шли внутренние вершины одной подобласти, потом другой, и так всех подобластей, а в конце идет интерфейс:



Пусть у нас есть СЛАУ и P процессов (или нитей). Пусть процессы умеют в последовательном режиме решать систему прямым методом. Например, используя LU (или прогонку, в случае трехдиагональной матрицы). Теперь им надо научиться это делать параллельно. Поделили графа от этой матрицы (по сути, поделили множество неизвестных, т.к. вершинам сопоставлены позиции в векторе) на P подобластей, используя алгоритм декомпозиции. Выделили интерфейс – S. Перенумеровали неизвестные – сначала внутренние по подобластям, потом интерфейс. Раздали внутренние позиции (строки матрицы, элементы вектора) процессам, интерфейс тоже как-то распределили примерно поровну. При такой нумерации строки и столбцы матрицы переставились согласно перенумерации, и СЛАУ получилась такой хитро-блочной структуры:

$$\begin{pmatrix} A_{0,0} & 0 & \cdots & 0 & A_{0,s} \\ 0 & A_{1,1} & \cdots & 0 & A_{1,s} \\ \vdots & & & \vdots & \\ 0 & 0 & \cdots & A_{P-1,P-1} & A_{P-1,s} \\ A_{s,0} & A_{s,1} & \cdots & A_{s,P-1} & A_{s,s} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{P-1} \\ x_s \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{P-1} \\ b_s \end{pmatrix}$$

Диагональные блоки соответствуют внутренностям подобластей, за исключением последнего, который интерфейсный. Внедиагональные блоки – это связи между подобластями процессов.



Алгоритм состоит, по сути, просто в решении системы методом исключения Гаусса, только блочным. Решаем систему $\boldsymbol{Ax} = \boldsymbol{b}$.

Используем блочное исключение Гаусса, чтобы привести систему к такому виду, который уже легко решается:

$$\begin{pmatrix} A_{0,0} & 0 & \cdots & 0 & A_{0,s} \\ 0 & A_{1,1} & \cdots & 0 & A_{1,s} \\ \vdots & & & \vdots & \\ 0 & 0 & \cdots & A_{P-1,P-1} & A_{P-1,s} \\ 0 & 0 & \cdots & 0 & \tilde{A}_{s,s} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_{P-1} \\ x_s \end{pmatrix} = \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_{P-1} \\ \tilde{b}_s \end{pmatrix}$$

На препроцессе (setup stage) долго и упорно находим этот интерфейсный блок:

$$\tilde{\boldsymbol{A}}_{s,s} = \boldsymbol{A}_{s,s} - \sum_{p=0}^{P-1} \boldsymbol{A}_{s,p} \boldsymbol{A}_{p,p}^{-1} \boldsymbol{A}_{p,s}.$$

Потом, на этапе решения (solution stage) находим правую часть для интерфейса

$$\widetilde{\boldsymbol{b}}_s = \boldsymbol{b}_s - \sum_{p=0}^{P-1} \boldsymbol{A}_{s,p}\boldsymbol{A}_{p,p}^{-1}\boldsymbol{b}_p,$$

из которого находим решение для интерфейса

$$\boldsymbol{x}_s = \widetilde{\boldsymbol{A}}_{s,s}^{-1}\widetilde{\boldsymbol{b}}_s ,$$

по которому находим решение для локальных подобластей

$$\boldsymbol{x}_p = \boldsymbol{A}_{p,p}^{-1}(\boldsymbol{b}_p - \boldsymbol{A}_{p,s}\boldsymbol{x}_s),\ p = 0, \dots, P-1.$$

Что тут получилось. Во-первых, дорогой препроцесс. Такой метод хорошо пользовать, когда систему надо решать много раз подряд. Тогда вес препроцесса в общем времени вычислений будет небольшим. В препроцесс также можно вынести обращение интерфейсного блока $\widetilde{\boldsymbol{A}}_{s,s}^{-1}$, чтобы каждый раз не решать плотную противную интерфейсную часть. Метод с дорогим препроцессом подходит, например, при моделировании сеточным методом какого-нибудь течения несжимаемой жидкости. Там на каждом шаге интегрирования по времени надо решать уравнение Пуассона с одной и той же матрицей.

На setup stage сделали $\widetilde{\boldsymbol{A}}_{s,s}^{-1}$, построили факторизацию для $\boldsymbol{A}_{p,p},\ p = 0, \dots, P-1$ (или что там надо для прямого метода, которым процессы умеют решать систему)

Разложим более подробно solve часть.

1. Находим локальное промежуточное решение из системы: $\boldsymbol{A}_{p,p}\boldsymbol{t}_p = \boldsymbol{b}_p,\ p = 0, \dots, P-1$
2. Находим локальный вклад в интерфейс: $\widetilde{\boldsymbol{b}}_s^p = \boldsymbol{A}_{s,p}\boldsymbol{t}_p,\ p = 0, \dots, P-1$
3. Суммируем, получаем интерфейс: $\widetilde{\boldsymbol{b}}_s = \boldsymbol{b}_s - \sum_{p=0}^{P-1} \widetilde{\boldsymbol{b}}_s^p$
4. Решаем интерфейс: $\boldsymbol{x}_s = \widetilde{\boldsymbol{A}}_{s,s}^{-1}\widetilde{\boldsymbol{b}}_s$
5. Решаем локальные подобласти: $\boldsymbol{A}_{p,p}\boldsymbol{x}_p = \boldsymbol{b}_p - \boldsymbol{A}_{p,s}\boldsymbol{x}_s,\ p = 0, \dots, P-1$

Получилось, что последовательный метод, которым процессы умеют решать систему применяется два раза – на этапах 1, 5. То есть, если последовательно система решалась методом LU, то теперь решение выйдет по цене не менее двух локальных LU.

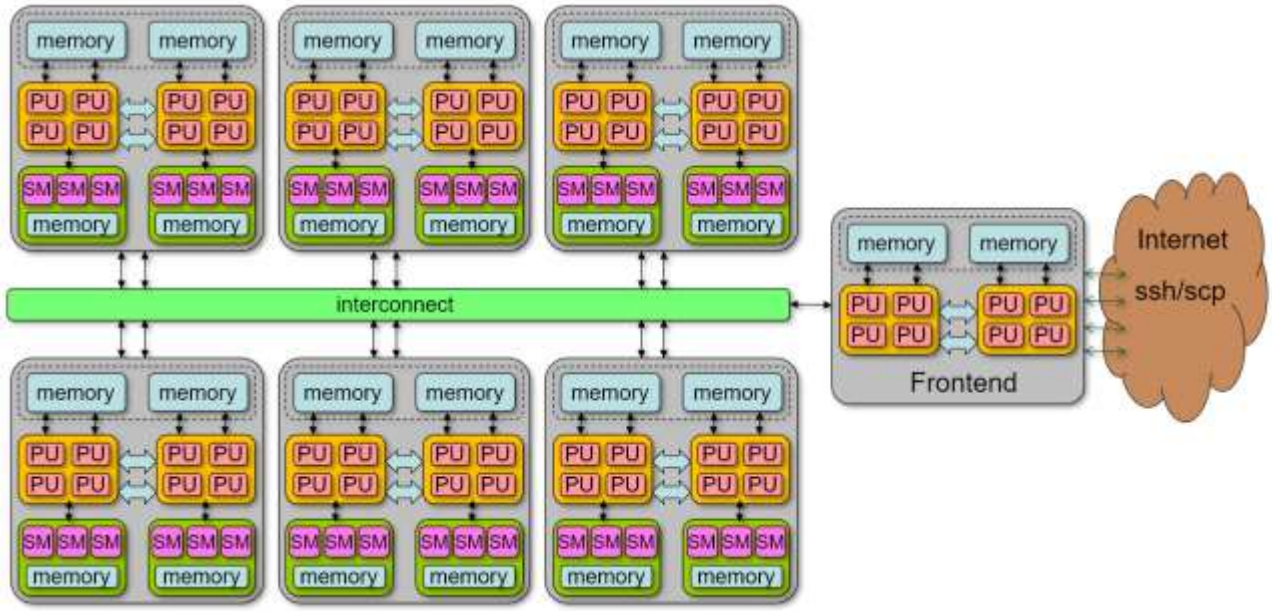
Кроме того, противное матрично-векторное произведение с плотной матрицей торчит на этапе 4. А для него нужен противный групповой обмен для нахождения глобальной суммы на этапе 3. Из-за того, что с ростом числа процессов растет размер интерфейсного блока параллельная эффективность метода быстро деградирует. Также она деградирует из-за роста группового обмена. Тем не менее, таким методом вполне получается решать системы быстро прямым методом и получать многократное ускорение.

## 7.2 Выполнения расчетов на кластерах

Тут уже будет совсем кратко, просто основные принципы. Чтобы, если вдруг придется работать на машинках ЦКП (Центр Коллективного Пользования), вам не пришло в голову, например, запускать на морде исполнимый файл расчетного кода (и потом огребать от удивленных админов).

Вычислительный кластер, суперкомпьютер – это (обычно) система коллективного пользования. То есть у нее много пользователей. Пользователи подключаются к системе по сети интернет, например, по протоколу SSH (Secure Shell) в режиме консоли (терминала), доступ к файловой системе осуществляется по протоколу SCP (Secure Copy Protocol). Но это доступ не прям ко всему кластеру, а только на управляющую машину (или машины, их может быть несколько), aka головной узел, голова, морда, frontend. С этой машины пользователи могут компилировать код и запрашивать ресурсы для своих задач. Для работы с кластером, естественно, понадобится умение работать в командной оболочке ОС Linux.

Для начала модифицируем диаграмму гибридного кластера из раздела 1.4, пририсуем управляющий узел (выбор цвета и формы для изображения сети интернет считать случайным):




За распределение ресурсов между задачами пользователей отвечает “система очередей”, aka планировщик, система управления прохождением заданий, менеджер ресурсов, workload manager. На разных кластерах стоят разные системы очередей (в основном slurm), используются разные скриптовые обертки, поэтому разбираться с конкретными командами бессмысленно, надо читать пользовательскую документацию от конкретной системы. Разберемся только с основными сущностями, так или иначе представленными в разных системах очередей.

Доступ к системе осуществляется по SSH либо по логину-паролю, либо по ключу. Когда зашли на систему, туда надо закачать все добро – код и данные для расчета. Первым делом нужно выяснить, на каком разделе файловой системы можно городить большие файлы. По умолчанию вы находитесь в домашнем каталоге, где обычно очень ограниченная квота, там нельзя плодить большие файлы и результаты расчетов.

Далее надо выбрать нужный компилятор, библиотеку MPI (например, используя module list, module avail, module load, module unload, см. документацию про конкретные команды).

Далее нужно выставить настройки компилятора. Поскольку флаги бывают разные, нужно правильно указать флаги под выбранный компилятор.

Если используем OpenCL, надо найти, где он находится и указать компилятору пути к заголовочным файлам и библиотеке.

Настроив компилятор в make-файле (или каком другом проектнике или скрипте), собираем код.

У системы очередей бывают следующие основные операции:

- отправить задачу в очередь (например, sbatch в slurm);
- отметить задачу (scancel);
- посмотреть состояние системы очередей (sinfo);
- посмотреть состояние своих задач (squeue).

Еще обычно бывают команды, чтобы занять узлы для интерактивного выполнения задачки (что полезно для отладки).

У запуска задачи, точнее, постановки в очередь, обычно бывают следующие основные параметры:

- сколько нужно запустить процессов;
- сколько задействовать узлов, то есть, по сколько процессов пускать на каждый узел;
- на какое время запустить задачу (чем больше, тем может быть дольше ждать в очереди);
- в какие файлы направить потоки stdout, stderr;
- на каком разделе (partition) запустить задачу, если кластер состоит из множественных разделов (частей с разными конфигурациями узлов, например).

Также можно указывать, как привязывать процессы (например --bind-to или --cpu-bind), но нам это не очень нужно, мы же сами выставляем affinity (см. раздел 3.2.3).

Какими именно командами это все делается, надо смотреть в мануале по системе.

Работа на системе коллективного пользования обычно имеет следующие особенности, которые необходимо учитывать при организации работы.

- **Ограниченное время выполнения**. Обычно время выполнения ограничено (например, сутками). При запуске указывается время счета, которое не должно превышать максимум, заданный на системе. При достижении запрошенного времени выполнения задача принудительно останавливается. Если задаче нужно считаться неделю, а максимальное время сутки, то придется ее перезапускать многократно.
- **Задача может быть остановлена в произвольный момент времени**. Внезапная профилактика, сбой в системе охлаждения, сбой в энергоснабжении, и много других причин.
- **Длительное время ожидания в очереди.** Это не просто взял, запустил, и оно считается. Это взял, поставил в очередь, и где-то, может быть, ночью, а, может быть, завтра оно стартанет.
- **Процессорное время стоит денег**. Много времени стоит много денег. Даже если лично вы не платите за время счета, кто-то все равно платит, даже опять-таки вы, как налогоплательщик. Процессорное время – ценный ресурс, его не надо зазря прожигать.
- **Ограниченная дисковая квота**. Большим расчетам надо хранить много данных. Эти много данных могут не лезть в квоту. Надо экономно расходовать дисковое пространство, использовать библиотеки сжатия данных.
- **Ненадежная файловая система для хранения большого объема данных.** Обычно раздел файловой системы, на котором ведутся большие расчеты, даже не бэкапится. Он часто даже так и называется – scratch. Что символизирует, что за оставленные в гардеробе ценные вещи никто ответственности не несет.

Из этих негативных факторов окружающей среды вытекают требования к программе. Помимо высокой степени параллелизма, хорошей масштабируемости, эффективной реализации вычислений, суперкомпьютерный код для больших ресурсоемких расчетов должен уметь все нижеперечисленное.

**Восстановление расчета.** Программа должна уметь поднимать расчет с файловой системы, поскольку время вычислений для одного запуска обычно меньше, чем необходимо для выполнения расчета. То есть надо сохранять на файловой системе точки восстановления, aka рестарты, чекпоинты, контрольные точки, рековырялки (recovery), в которых содержатся

все необходимые данные для продолжения расчета, как если бы он выполнялся непрерывно. Остановка и перезапуск не должны влиять на результат расчета. Из ограниченности дисковой квоты следует, что писать в рестарт нужно только необходимые вещи. Если какие-то сеточные функции, например, можно пересчитать по другим, уже имеющимся в рестарте, то их не надо писать на диск, их надо вычислить на ините (*жарг*. на этапе инициализации). В рестарт пишем не все подряд, а только необходимое, а все производное, вычислимое – пересчитываем при запуске. Например, если мы используем и физические переменные, и консервативные, то, естественно, запишем только один набор, а второй легко пересчитаем на рестарте. Чем чаще мы пишем чекпоинты, тем меньше будет потеря интервала вычислений при рестарте. Но тем больше будет потеря времени на запись данных на файловую систему. Поэтому интервал записи чекпоинтов выбираем из соображений, что за время вычислений одного запуска на кластере суммарное время записи на диск чекпоинтов не должно превосходить времени вычислений одного интервала между этими записями. То есть, чем на большее время мы можем запустить задачку на системе, тем реже надо писать рестарты.

**Надежность восстановления.** Программа должна уметь восстанавливать расчет во что бы то ни стало. Из условия, что программа может быть прибита в любой, вообще любой момент, в то числе, в момент записи на диск, следует, что записи восстановления должно быть как минимум две. Если грохнемся на записи одной, будет доступна предыдущая. Но иметь много записей нам не позволит ограниченная дисковая квота. Что же тогда делать? Просто делаем две переключающиеся записи, сначала пишем в одну, потом в другую, потом снова в первую, и так далее. Если задачка грохнется на записи, то просто откатимся на предыдущий чекпоинт. Для удобства восстановления хорошо бы научить программу автоматически определять, какая запись самая последняя, и проверять ее корректность и целостность. Тогда не надо будет каждый раз проверять, с чего подниматься, программа сама определит последнюю целую корректно записанную рековырялку и с нее восстановит расчет.

**Антиавост.** Программа должна избегать авоста (от аварийная остановка). Понятно, что в программе не должно быть ошибок, что надо хорошо отлаживаться, тестироваться и поддерживать качество кода. Но даже в совсем безошибочной программе могут возникать авосты. Для каждой конкретной задачи, особенно большого расчета, сложно заранее так хорошо подобрать конфигурацию численного метода, чтобы считало быстро и при этом не разваливалось. Например, моделируем какую-то CFD задачку. Из-за желания считать побыстрее поставили побольше шаг по времени. Оказалось, слишком побольше. Или ради экономии задали недостаточную точность солверу. Это негативно сказалось на устойчивости, расчет стал “разваливаться”. Где-то возникли нефизичные значения переменных, например, отрицательная плотность. Или вообще начались NaNo-технологии (NaN – Not a Number). Тогда программа должна аварийно убраться, совершить авост, aka crash. Мы тогда поправим шаг, повысим точность, перезапустим расчет в систему очередей. И она еще день-два будет болтаться в очереди. Как можно улучшить ситуацию? Да, можно сразу взять заведомо достаточно маленький шаг по времени, выставить заведомо высокую точность, но тогда мы можем прожечь многократно больше процессорного времени, что тоже нехорошо. Выход – научить программу самодиагностике, чтобы она понимала, что что-то идет не так (хотя бы, что возникли NaNы или нефизичность), откатываться на доступные чекпоинты и автоматически корректировать параметры. Например, программа уткнулась в отрицательное давление, сама откатилась на сколько-то шагов назад, взяв чекпоинт из оперативной памяти

или с файловой системы, сама подправила параметры, уменьшила там шаг по времени или еще что-нибудь подкрутила и продолжила расчет. И не пришлось тратить время на перезапуск с новыми параметрами.

**Интерактивное управление**. Пусть у нас идет долгий расчет. И вдруг нам захотелось подкрутить какие-то параметры, увеличить шаг, уменьшить скважность выдачи данных, поменять настройки солвера, или еще что-нибудь эдакое. Для этого надо остановить расчет, поправить инпутники (*жарг.* входные файлы параметров пользовательского ввода) и перезапустить задачку в систему очередей, где она еще несколько дней будет тухнуть в очереди. Это грустно. Поэтому надо всего-то научить программу перезачитывать инпутники раз в какое-то время. Не очень редко, чтобы можно было управлять, но и не слишком часто, чтобы не влияло на производительность. Тогда расчетом можно будет управлять, менять конфигурацию численного метода, не прерывая выполнение программы.

**Резервное копирование**, aka backup. Поскольку раздел файловой системы для больших расчетов обычно не гарантирует надежность хранения данных, наши результаты и чекпоинты от очень ресурсоемкого и дорогого расчета могут пропасть. Поэтому надо периодически выкачивать с кластера необходимые для восстановления расчета данные и важные результаты. Покупаем себе NAS (Network Attached Storage) побольше и регулярно выгребаем туда терабайты рестартов. Если канал связи не позволяет, то хотя бы регулярно перекладываем один чекпоинт на другой раздел файловой системы кластера. Иногда там бывают специальные разделы для бэкапов.

**Смена конфигурации вычислительных ресурсов.** Например, на системе коллективного пользования могут быть доступны разные вычислительные разделы. Можно задействовать разное число узлов кластера. Например, считали на одном числе узлов кластера, тут вдруг понабежало народу, позаняли систему, задача зависла в очереди. Уменьшаем число узлов, чтобы задача быстрее запустилась в изменившихся условиях загрузки. Или, наоборот, все резко свалили, и можно сделать all in. Чтобы иметь возможность запуститься в другой параллельной конфигурации, например, с другим числом MPI процессов, программа должна (легко) уметь подняться с рестарта, записанного ранее на другом числе процессоров. Пусть у нас сеточный метод, считаем какое-то CFD. Мы сделали декомпозицию сетки, раздали подобласти MPI процессам. Логично, что они каждый пишут себе рестарты по своей подобласти в распределенном формате. Потом с них поднимаются. Но если число процессов поменялось, происходит конфуз. Поэтому надо научить программу понимать, что изменилось число процессов, и тогда каждый процесс может, например, просто перечитать полный набор данных от всех процессов, и выбрать оттуда только то, что нужно для его новой подобласти.

**Выбор конфигурации ресурсов.** Раз уж речь зашла о том, чтобы менять число узлов на кластере, как вообще это число выбирать? Вот для этого и надо подробно исследовать свой расчетный код на предмет параллельной эффективности и масштабируемости. Об этом было в разделе 4.5. Ну и нужно соблюдать разумный баланс между выигрышем в скорости и потерях на неидеальной параллельной эффективности. Если время позволяет считать дольше, если дедлайн по сдаче расчета был не вчера, то можно брать поменьше узлов. Тогда эффективность будет выше, за счет чего суммарно на расчет прожжётся меньше процессорного времени (процессорное время – время вычислений, умноженное на число задействованных

процессоров). Но времени настенных часов (то есть просто времени, которое задача считалась), конечно, уйдет побольше. Если система не занята, ресурсы простаивают, то можно взять побольше, несмотря на некоторое (разумное) снижение параллельной эффективности. Если система забита, то считать на большом числе узлов с низкой параллельной эффективностью – это вандализм и прочее антисоциальное поведение.

## 7.3 Коллективная разработка параллельного кода

Осталось кратенько обсудить то, с чем, возможно, мало ли, всякие неприятности случаются в жизни, вам придется столкнуться в вашей профессиональной карьере. В этом разделе обсудим, как примерно разрабатывают и поддерживают расчетные коды. Ну и в какой-то степени и другие коды. Может быть, ручаться не могу, но и в других кодах тоже есть что-то общее в технологии работы.

### 7.3.1 Структура суперкомпьютерного CFD кода

Расчетный код, программный комплекс, который что-то предсказательно моделирует, обычно можно разделить на такие части, или этапы, или составляющие:

- **препроцесс** – то, что делается однократно для выполнения расчета;
- **расчет** – непосредственно выполнение расчета, aka flow solver, вычислительный движок, вычислительное ядро;
- **постпроцесс** – то, что делается после выполнения расчета, обработка результатов, анализ данных, визуализация, графики, картинки.

Вычислительная часть, которую обозвали расчет, в свою очередь, состоит из следующих этапов:

- **инициализация** – то, что делается однократно при запуске программы на исполнение (а запускаться программа в одном расчете может очень многократно в силу ограниченного кванта времени на кластере);
- **вычисления** – собственно выполнение расчета, то есть цикла интегрирования по времени.

Мы разбирались с этапом вычислений, как его сделать быстрее и эффективнее. Но для выполнения большого ресурсоемкого расчета не только непосредственно вычисления, но и все эти составляющие должны быть параллельными и работать быстро. Хотя для пре и поста не так критична скорость работы, поэтому можно не заморачиваться с низкоуровневой оптимизацией, можно городить удобные функции, объекты, ветвить дерево вызовов на нижнем уровне – ничего страшного. Лишь бы удобно и надежно. Но. Вычислительная сложность все равно должна быть минимальна. Неприемлемо сделать что-то по-простому с квадратичной стоимостью/ресурсоемкостью, если можно сделать с линейной. На больших задачах это не будет работать.

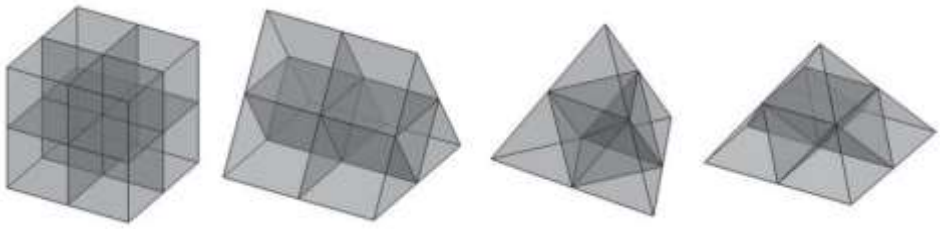
В общем препроцесс, постпроцесс, да и инит в общем-то могут себе позволить быть в несколько раз медленнее, чем это возможно. То есть, не стоит сильно жертвовать качеством кода, простотой устройства кода, удобством работы с кодом ради производительности. Но худшая асимптотика по вычислительной сложности – непозволительна. Тут уж приходится хорошо делать и продумывать алгоритм.

При этом качество распределенной параллельной реализации пре-постпроцесса тоже имеет существенное значение. Почему на последовательной реализации не выехать? Большая задачка не полезет в память одного узла. И результаты расчета с тысячи узлов будут

обрабатываться на одном узел дольше, чем был сам расчет. Это неприемлемо. Так что единая, так сказать, концепция распределенного распараллеливания (MPI) должна пронизывать весь цикл работы с сеточными данными (сверх)большого объема, и пре, и пост.

**Препроцесс**. Ресурсоемкие средства препроцессора, требующие распределенной (то есть с распределенной памятью) параллельной реализации, обычно связаны с обработкой сеточных данных, которых может быть (очень) много. На картинке ниже пример структуры.


Ресурсоемкость, требующая распараллеливания, начинается, когда нужно иметь дело с подробной расчетной сеткой. Обычно сеточный генератор строит сетку с ограниченным числом узлов (порядка нескольких десятков миллионов). Чтоб считать на миллиардных сетках, исходной базовой сетке делается равномерное измельчение. То есть все сеточные элементы разделяются на более мелкие. На середины ребер добавляются узлы, в центры четырехугольных граней, в центр гексаэдра тоже. Получается как-то так:



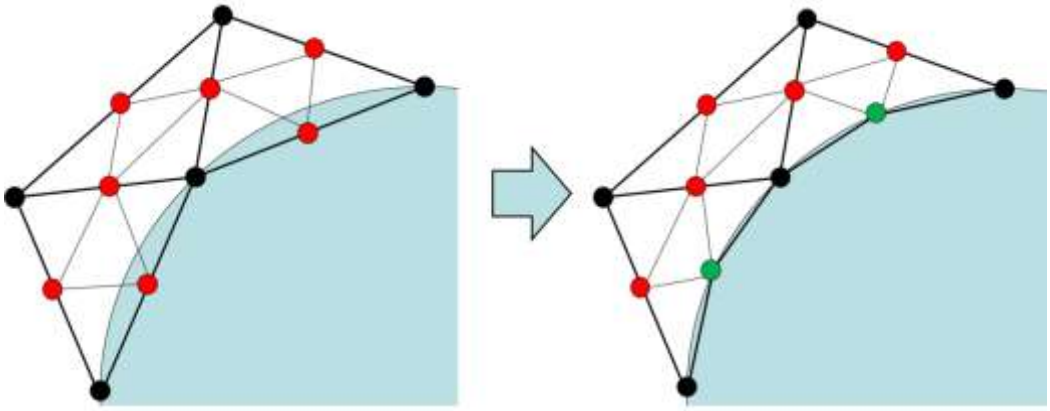
Картинка из статьи: S. A. Soukov. Adaptive Mesh Refinement Simulations of Gas Dynamic Flows on Hybrid Meshes. Doklady Mathematics, 2020, Vol. 102, No. 2, pp. 409–411.

За одно такое измельчение в сетке получается примерно в 8 раз больше ячеек, то есть примерно вдвое выше разрешающая способность по каждому пространственному направлению. При этом с каждым этапом измельчения ухудшается качество сетки (всякое там соотношение сторон у сеточных элементов и прочие неприятности). Обычно делают 2 – 3 уровня измельчения, что позволяет увеличить число узлов в 64 раза без существенного падения качества сетки.

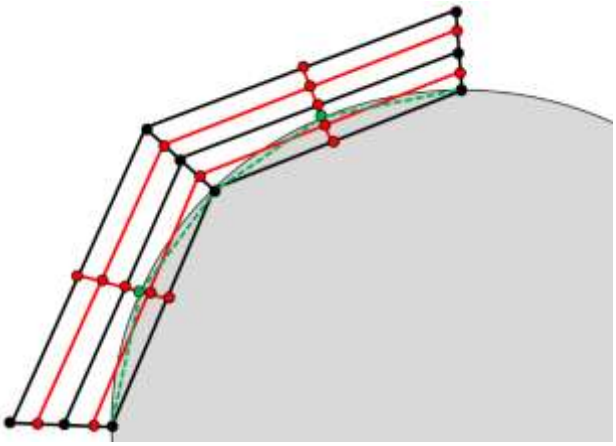
С базовой сеткой еще как-то можно было иметь дело в последовательном режиме, с рабочей измельченной сеткой уже нет. И сам измельчитель, естественно, должен быть

параллельным. Что несложно. Сетка делится на подобласти, подобласти измельчаются, потом сшиваются.

Измельченную сетку еще нужно поправить. Потому что от измельчения повысилось разрешение, но качество представления геометрии моделируемого объекта – нет. Добавленные на поверхности твердого тела узлы надо еще подвигать.



Но это был показан простой случай. Обычно сетка у твердой поверхности анизотропная для разрешения погранслоя. А там потом начинаются приколы, конфузы, узлы наползают на соседние элементы, надо это все двигать, разгребать, править, долго и упорно.



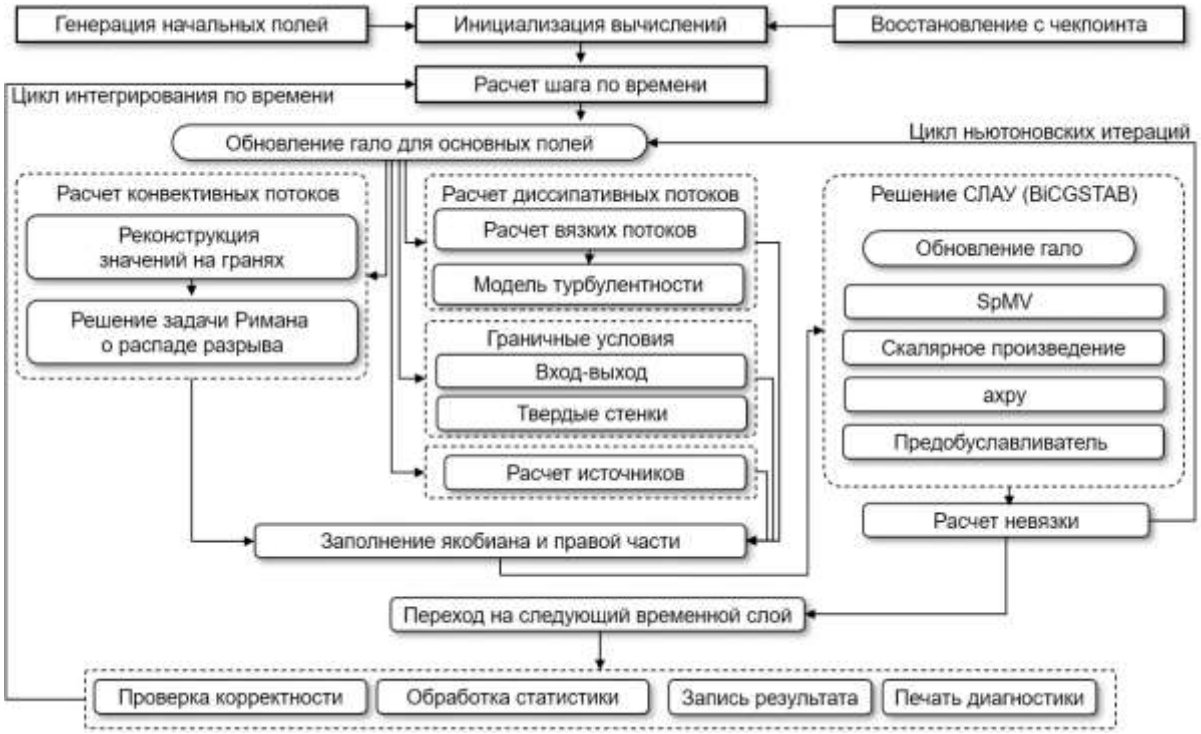
Получается нелокальная операция, надо двигать соседние узлы, править весь погранслой. Это можно назвать сглаживание измельченной сетки. Непременно параллельная операция.

Еще для корректной работы модели турбулентности надо для каждой сеточной ячейки знать ближайшее расстояние до твердой поверхности. Это надо для каждого узла, или центра масс элемента. А их – сотни миллионов, миллиарды, короче говоря, много. А к чему я это вообще? А, вот. Для примера. Если делать по-простому, то можно сделать набор граничных граней твердых поверхностей в расчетной области, то есть граней поверхностной сетки твердых тел. Потом для каждой ячейки сетки перебрать все эти грани и найти расстояние до ближайшей. Пусть ячеек N, а граней поверхности у объемной трехмерной сетки будет где-то $O(N^{2/3})$ (ну то объем – куб, а то площадь – квадрат, получаем 2/3). Умножим одно на другое, будет вычислительная сложность $O(N^{5/3})$. Но мы не будем так топорно делать. Построим какое-нибудь BSP (Binary Space Partitioning) дерево, или еще какое k-d дерево, ускорим поиск ближайшей грани с линейной сложности по числу граней до логарифмической, будет

O(N log(N)). Такая версия будет в дофига сколько раз быстрее на больших сетках. Поэтому выбор подходящего алгоритма – это важно. Вычислительная сложность, ресурсоемкость по памяти – важно. А вот детали реализации, лишние ветвления, вызовы функций – уже не критично. То есть, не так критично, как для вычислительной части. Можно отдать приоритет удобству реализации. Но, естественно, алгоритм должен быть параллельный. Что в данном случае делается элементарно – просто распределяется множество ячеек между процессами, поверхностная сетка реплицируется всем процессам и все. Ну и цикл по ячейкам параллелится по OpenMP.

А еще сетку надо поделить на подобласти – это параллельная декомпозиция. Но это делается внешней библиотекой типа ParMetis. А еще из сетки надо повырезать подсеток (поверхности твердых тел, сечения, огрубленные сетки) для визуализации и выдачи результатов. Все это обрабатывает большой объем данных.

**Вычислительное ядро**. А про эту часть мы говорили во введении, и других главах. Там ведется цикл интегрирования по времени, на каждой итерации которого считаются потоки для физических (или консервативных) переменных через все грани расчетных ячеек сетки, находятся значения переменных на новом временном слое, и так многие тысячи шагов.




**Постпроцесс**. А что вообще получаем по результатам расчета?

- **Сеточные функции** – то есть переменные, заданные в ячейках сетки, записанные на диск. Для удобства каждый MPI процесс пишет свой файл по своей подобласти и ни с кем не конфликтует и не пересекается.
- **Эволюция во времени** – выдача с некоторой скважностью (раз в какой-то промежуток времени) каких-то величин – значений в контрольных точках.
- **Интегральные характеристики** – просто осредненные скалярные значения типа коэффициента подъемной силы, сопротивления, значения моментов.

Тут вся ресурсоемкость в основном в обработке сеточных функций. Потому что ячеек сетки – очень много. Туда пишутся, например, моментальные поля, то есть значения в какой-то определенный момент времени, или осредненные поля – то есть проинтегрированные в каждой ячейке на каком-то временном интервале. Эти интервалы надо объединять – обрабатывать кучу файлов, просуммировать, поделить на время. Это ресурсоемко и делается параллельно. Также параллельно вычисляются всякие производные величины (всякие вектора завихренности, Q-критерии и прочее). То, что можно потом посчитать, мы не будем в расчете писать на файловую систему, конечно же. Ну вот, например, какая-то такая схема работы:


### 7.3.2 Коллективная разработка.

Из предыдущего пункта следует, что расчетные коды – это большая сложная штуковина. Когда много людей параллельно ~~гадят~~ модифицируют код, тем более, параллельный код, с этим надо что-то делать. При разработке расчетных кодов обычно имеется (ладно, кого я обманываю, должен иметься) некоторый регламент, как в этих кодах что-то делать, как вносить туда изменения, как их верифицировать и валидировать. Обычно есть такие штуки.

- Правила форматирования кода – aka style guide, шаблон, который говорит, как надо ставить пробелы, переводы строки, как и куда ставить скобочки, как именовать переменные, как надо смещать код в структурных блоках, как что выравнивать, какой длины строка, какая кодировка файла, чем смещать – табами или пробелами, и т.д.
- Система управления версиями – чтобы разбираться, кто когда куда что понасдавал, и кто именно что именно поломал в коде.
- Система управления проектами – чтобы знать, кто ничего не делает и задерживает работу.
- Система отслеживания ошибок – aka bug tracker, чтобы знать, где когда что было сломано, кто виноват и кто должен это починить.
- Система тестирования – aka QA (quality assurance), чтобы знать, что новые изменения ничего не сломали в коде и не изменили результаты;
- Набор валидационных тестов – чтобы знать, если результаты таки изменились, и это так и было задумано, что эти результаты необходимого качества.
- Внутренние средства в коде для диагностики, проверки корректности, отладки.
- Документация разработчика (хотя бы на уровне комментариев в коде).
- Регламент работы с кодом – QA, zero warning tolerance, статический анализ и т.д.

Если вы пришли куда-то работать, разрабатывать какой-то расчетный код, а там ничего такого нет, там не знают, что такое управление версиями и QA, то это повод задуматься, куда вы вообще попали, ну и, если это место вам дорого, навести там порядок.

Тут, конечно, не будет никаких подробных описаний, как эти системы работают. Этих систем, помогающих наладить жизненный цикл ПО, множество. Это все отдельная долгая история. Тут это приводится просто для того, чтобы, если вы вдруг с удивлением обнаружите, что не знаете, что такое Git или SVN, вы полезли с энтузиазмом читать про это в интернетах. А то приходят выпускники работать, и не знают, что такое системы управления версиями и как они работают. Зачастую даже не знают, что такое стайл гайд. Не осознают, что нельзя просто так взять и поставить в коде символ табуляции, если там приняты пробелы. Не знают, что нельзя перенести открывающую скобочку цикла на отдельную строку, если в этом коде так не делают. Не понимают, что нельзя оставить в коде неиспользуемую переменную или внести еще какие ворнинги (*жарг*. от warning – диагностических сообщений компилятора). Работодателю приходится удивляться, отменять ревизии, тратить время на разъяснение основных принципов работы с кодом в коллективе. Зачастую у начинающего разработчика это вызывает непонимание, недоумение – “а я так привык код писать”, “ну это же просто ворнинг, он же ни на что не влияет”, “ну и что, что кодировка 1251”. Это, в свою очередь, может приводить к кратковременному появлению на лице отпечатков клавиш клавиатуры (поскольку прикладывание разработчика лицом к клавиатуре с чтением мантры “да чему вас, *<разг., груб., неценз.>*, учили в ваших университетах”, ускоряет процесс постижения регламента)[1].

Рассмотрим это все на каком-то примере, хоть и (далеко) не самом удачном. Да, конечно, на этом разделе читатели, работающие в приличных IT конторах, могут только посмеяться над нами, бедными учеными, весьма отдаленно знакомыми с принципами разработки ПО. Ну что делать, нельзя же совсем ничего по столь важному вопросу не сказать? У нас в коллективе работа с кодом в настоящее время организована следующим образом.

**Шаблон оформления**. Стайл гайд представлен в виде примеров файлов исходного кода для h и cpp, из которых наглядно следует, что таб (символ табуляции) не используем, ширина табуляции 4 пробела; максимальная длина строки 100 (чтобы на обычном разрешении 1920 комфортно влезало 2 окна с кодом для сведения изменений); кодировка UTF-8 без сигнатуры; скобки от всяких for, if не отрываем (чтобы показывались подсказки на закрывающей скобке); указано, как выравнивать код; приводятся принципы именования переменных; сказано, что все свойства и методы объектов должны быть прокомментированы, указанно, где именно должны располагаться комментарии и т. д.

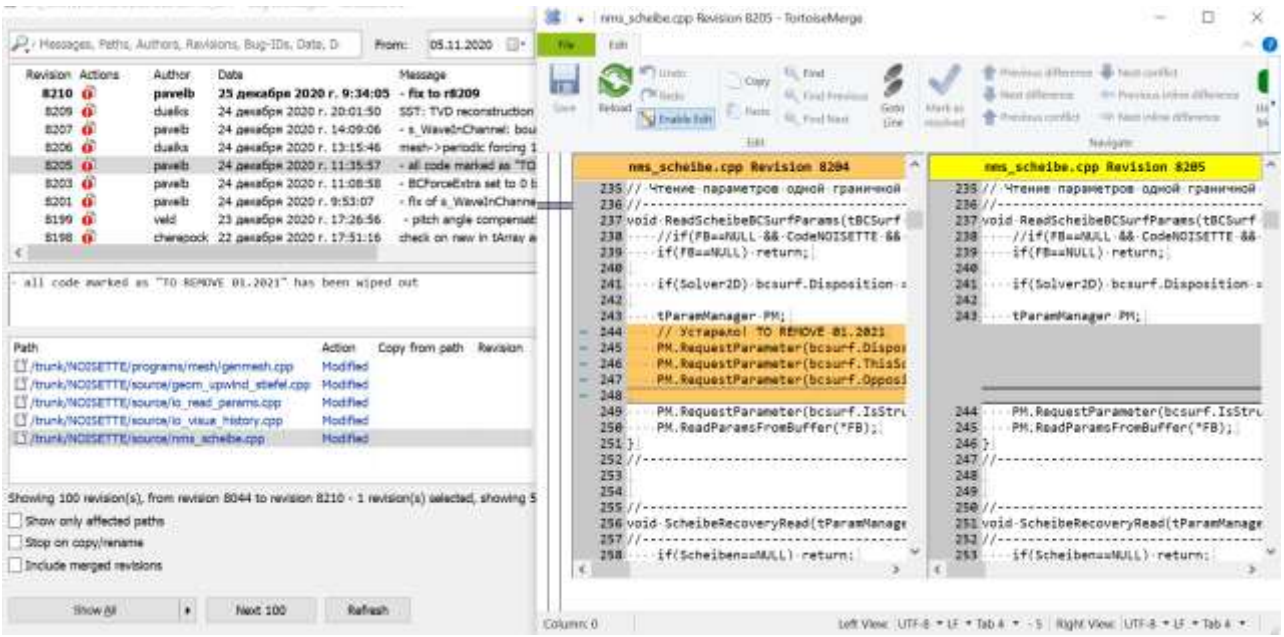
Выглядит это примерно так (небольшой фрагмент):

```
// Хотя бы краткое описание функции. Скобку не отрываем!
// Если надо - более подробное описание функции, входных и выходных параметров.
// FillMatrix - такая-то хрень,
// MeanFlow - сякая-то хрень.
void Viscosity(int FillMatrix, int MeanFlow){
    double var1; // каждое описание комментить!
    double var2; // в начале строчки тип обязательно
    int obviousVar1, obviousVar2; // если коммент не нужен – однотипы группировать в строку

    // условия
    if(condition){ // скобку структурного блока не переносим
```

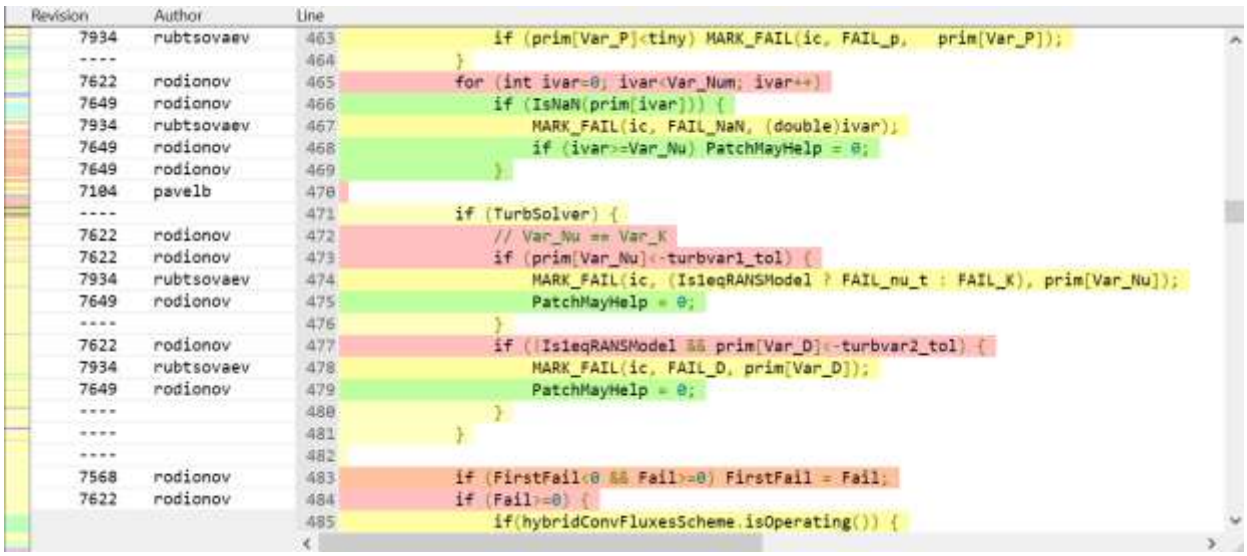
[1] Шутка, конечно же! Но в каждой шутке…

```
        int x = 0; // переменные, локальные для структурного блока, описываем внутри!
        function1(x); // у вызовов коммент, что происходит, если из названия неочевидно
    }
    else{ // если коммент к структурному блоку краткий - писать в той же строке
        int x = 0; // если переменная по сути локальна для блока,
        // но встречается в нескольких блоках - не ленимся, описываем каждый раз.
        Function2(x); // короткий коммент к вызову - в ту же строку

        // если надо длинный коммент – пишем перед вызовом.
        // комменты выравниваем!
        Function3();
    }

    // если действие короткое, то {} можно не использовать и делать one-лайнер.
    if(condition) function4();

    // если в одну строчку длинно, то по любому открыть структурный блок
    if(condition){
       function5();
    }

    // если переменная цикла имеет локальный смысл - описывать в заголовке цикла!
    // ставим пробелы после ;
    for(int jt=JT[0]; jt<JT[1]; jt++){ // если коммент к циклу короткий - в ту же строку.
        function1();
    }

    //если цикл без структурного блока - то только в одну строку!
    for(int jt=JT[0]; jt<JT[1]; jt++) function2();
```

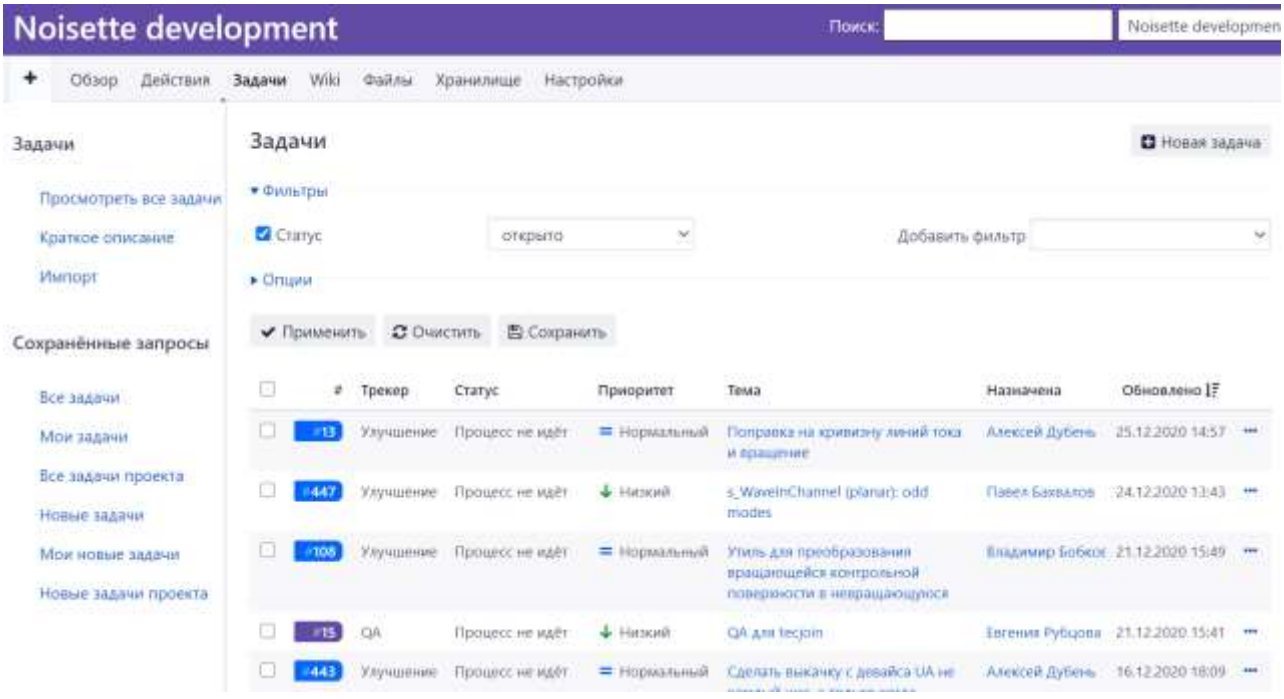
**Система управления версиями**. У нас используется SVN как-то издавна. Сейчас более распространен Git. Для тех, кто вдруг еще не знает, что это за системы, это такие штуки, которые показывают, кто когда какие файлы поменял в коде. А что именно в них поменяно в какой именно строке – показывают приложения для сравнения сорсов (*жарг*. от source code – исходный код программы), aka merge tool:




И, что особенно полезно, эта штука показывает, кто виноват. Точнее, кто какую строчку трогал в данном файле в данном диапазоне номеров ревизий.




**Система управления проектами и баг-треккер.** У нас используется Redmine. Эта штука ведет проекты, в каждом список задач, когда кому что надо сделать, кто кого задерживает, всякие диаграммы Ганта, ну и, конечно, кто во всем виноват. Redmine у нас работает в связке с SVN, можно сразу смотреть какие ревизии были сданы в код по задаче, aka тикету (ticket). Можно, сдавая ревизии в SVN, автоматом подцеплять их к задачам в Redmine, менять статусы задач ключевыми словами в комментариях к ревизии. Работает эта штука с web-интерфейса и выглядит так (особенно характерен статус выполнения задач):




**Система тестирования.** Задача тестирования – убедиться, что новая ревизия никому ничего не сломала, что все работает корректно. Поскольку эта вещь сильно важная, про организацию QA процедуры далее будет целый пункт 7.3.4. Тут только вкратце отметим, что QA тестирование делается это путем сравнения результатов тестовых расчетов (для вычислительной части) или просто разных выходных файлов (для пре- и постпроцессора) с эталонными результатами. Эти тесты покрывают (точнее, должны покрывать) всю основную реализованную функциональность. Чтобы, если что-то сломалось, по результатам

тестирования сразу можно было догадаться, что именно и где именно пошло не так. QA тесты делаются на очень маленьких сетках, чтобы они работали как можно быстрее. В этом тестировании важно показать не то, что результаты расчета правильные, а то, что они просто не изменились относительно предыдущей версии. Поэтому не обязательно брать подробные сетки, необходимые для получения хоть сколько-то точных результатов.

**Регламент работы с кодом.** Тут можно просто перечислить основные пункты регламента, который строго соблюдается всеми разработчиками:

- соблюдение стайл-гайда;
- полное отсутствие ворнингов на компиляторах Microsoft, Intel, GNU (-Wall);
- регулярная прочистка аналайзерами (типа clang), memory check под valgrind;
- QA тестирование на совпадение с эталонными данными;
- процедура обновления эталонных данных QA, если они таки поменялись;
- согласованность ревизий с шаблонами входных файлов пользовательского ввода;
- согласованность ревизий с документацией пользователя и разработчика;
- регулярное тестирование производительности параллельной эффективности;
- code review…

  Ох, ладно, кого я обманываю. Ничего у нас не соблюдается, бардак и хаос, как у всех.

### 7.3.3 Внутренняя инфраструктура для отладки и диагностики

А зачем она вообще нужна в коде? Есть gdb, valgrind, и прочие Intel Parallel Inspector. Когда ошибка воспроизводится на (очень) маленьком тесте, который можно запустить на декстопе, или на своем уютном сервачке, или хотя бы за разумное время на кластере на небольшом числе узлов в отладочной сборке – то с этим как-то можно справляться и без наворотов в коде. А что делать, когда ошибка возникает на “продуктовом” (*жарг.* от production) расчете? На третьем часу счета на нескольких тысячах процессоров на релизной сборке. И совершенно не воспроизводится на QA-шной мелочевке. На ~~хромой козе~~ валгринде к такой проблемке не подъедешь. Дебажная сборка под валгриндами будет безрезультатно ползать до скончания веков, прожигая всю квоту процессорного времени.

**Проверки.** Чтобы было легче локализовать проблемы, покроим всевозможные функции в коде всевозможными проверками – ассертами (assert). Особенно покроем проверкой на диапазон достукы к контейнерам-массивам, чтобы по квадратным скобкам проверялся диапазон индексов. Но если наплодить столько проверок в вычислительном ядре, то оно же будет заметно медленнее работать, запросто раза в полтора медленнее может стать. Что же делать? Разделим функции на функции верхнего уровня, низкочастотные, которые обрабатывают наборы сеточных объектов, и функции нижнего уровня, высокочастотные, которые обрабатывают один сеточный объект (объект – узел, грань, элемент и т.д.). Наличие проверок может сказываться на производительности только в высокочастотных вызовах. Ну и сделаем 2 релизных (release) конфигурации кода. В одной высокочастотные проверки выключены – для максимально быстрого выполнения продуктовых расчетов. В другой – включены – для случая, если в продакшене (серийном производство) что-то пошло не так. Назовем эту конфигурацию Safe mode. Оба конфига – сборки с оптимизацией, разница в производительности будет в проценты, а не на порядки (как в случае валгринда).

**Обработчик авоста.** Сделаем функцию-крашилку (Crash) для культурной аварийной остановки с печатью диагностики, печатью доступной информации о стеке вызовов, о месте возникновения проблемы, ну и с вызовом MPI_Abort, конечно же.

```
// Обработчик авоста
int Crash(const char *fmt,...); // печатаем в логии всю диагностику и валим с MPI_Abort
// выходим через exit в явном виде, чтобы компайлер знал о точке выхода
#define crash(...) exit(Crash(__LINE__, __FILE__, __VA_ARGS__))

// Макрос для высокочастотных проверок на нижнем уровне
#ifdef SAFE_MODE // В безопасной конфигурации включаем
#define SAFE_ASSERT(X, ...) if(!(X)) crash(__VA_ARGS__); // safe-mode only
#else
#define SAFE_ASSERT(X, ...) // на релизе выключаем высокочастотные проверки
#endif

// Базовые проверки – для верхнего уровня, где они не влияют на производительность
#define ASSERT(X, ...) if(!(X)) crash(__VA_ARGS__);
```

Утыкали весь код проверками, чтобы отлавливать возможные ошибки. Особенно покрылм проверкой на диапазон доступ ко всяким контейнерам-массивам:

```
inline T &operator[](int i){
    SAFE_ASSERT(i>=0 && i<N, "Array %s[%d] out of size %d", name.c_str(), i, NN);
    return V[i];
}
```

**Стек-трейсер.** А откуда брать информацию про стек вызовов? Можно какие-нибудь библиотеки для этого подцепить. Например, в Boost что-то было на эту тему. Но у нас сделано топорно, по-простому. Объектик-информатор по конструктору пихает данные о местоположении в общий стек с текстовыми метками, по деструктору – удаляет.

```
struct tStackAgent{
    tStackAgent(const char *Text, int Level=1); // Добавляем метку в глобальный список
   ~tStackAgent(); // Убираем метку из списка
};

// информатор для верхнего уровня
#define INFORM_KGB(X) tStackAgent KGB_agent(X/*name*/, 1/*level*/);

// информатор для нижнего уровня
#ifdef SAFE_MODE
#define INFORM_KGB_LOW(X) tStackAgent KGB_agent(X/*name*/, 2/*level*/);
#else
#define INFORM_KGB_LOW(X) // отключен
#endif
```

Утыкали весь код информаторами, чтобы код мог знать, где именно, в какой именно функции, в каком именно структурном блоке он находится, и как он туда пришел.

```
void Function1_low_level(…){
    INFORM_KGB_LOW(__FUNCTION__);
    // ... some calculations ...
    {
        INFORM_KGB_LOW("block1");
        // ... some calculations ...
```

```
    }
    {
        INFORM_KGB_LOW("block2");
        // ... some calculations ...
    }
}

void Function1_high_level(…){
    INFORM_KGB(__FUNCTION__);
    for(int i=ibeg; i<iend; ++ie)
        Function1_low_level(i);
}
```

**Обработчик сигналов.** Теперь Crash знает, где произошел авост, и каким путем мы дошли до жизни такой. Но это если ошибку мы отловили проверками. А если программа словила сигу (*жарг*. signal – сигнал операционной системы)? Самые популярные: SIGSEGV – Segmentation fault, когда мы полезли совсем не ту память; SIGFPE – Floating point exception, когда мы поделили на ноль или занялись NaN-о-технологиями. Надо настроить ловушки для сигналов, трапы (signal trap), чтобы своим обработчиком можно было выдать диагностику и вызвать свой Crash с внутренним стек-трейсером. Если совсем упрощенно, то как-то так:

```
#include <csignal>

typedef void (*tSigFunc)(int); // типок для обработчика сигналов

struct tSigData{ // храним данные для обработчика
    tSigFunc handler; // указатель на исходный обработчик, если надо вернуть как было
    const char *msg;  // наше текстовое сообщение о сигнале
    tSigData(){ handler = NULL; msg = NULL; }
};

static tSigData SigData[NSIG];  // данные для сигналов

static void SigHandler(int signo){ // наш обработчик, вызывающий наш авост
    string msg = SigData[signo].msg;
    msg = "Signal has been raised: " + msg;
    crash("%s", msg.c_str());
}

// инициализация ловушки для сигнала – вызываем при запуске программы
void SetSignalTrap(int sig, const char *sigmsg){
    if(SigData[sig].handler) return; // уже выставили
    SigData[sig].handler = signal(sig, SigHandler);
    if(SigData[sig].handler == SIG_ERR){
        fprintf(stderr, "SetSignalTrap: WARNING: can't set %s", sigmsg);
        return;
    }
    SigData[sig].msg = (sigmsg!=NULL ? sigmsg : "UNKNOWN");
}

// ставим лавушки интересующим нас сигналам
void SetSignalTraps(void){
    SetSignalTrap(SIGABRT, "SIGABRT - Abort signal\n");
    SetSignalTrap(SIGFPE,  "SIGFPE - Floating point error\n");
    SetSignalTrap(SIGILL,  "SIGILL - Illegal instruction\n");
    SetSignalTrap(SIGSEGV, "SIGSEGV - Segmentation fault\n");
#if !(defined(WIN32) || defined(WIN64))
    SetSignalTrap(SIGBUS,  "SIGBUS - Hardware fault");
```

```
    SetSignalTrap(SIGIOT,  "SIGIOT - I/O transfer Trap");
    SetSignalTrap(SIGTRAP, "SIGTRAP - Hardware fault \n");
    // …
#endif
    pprintf("Signal traps have been set\n");
}
```

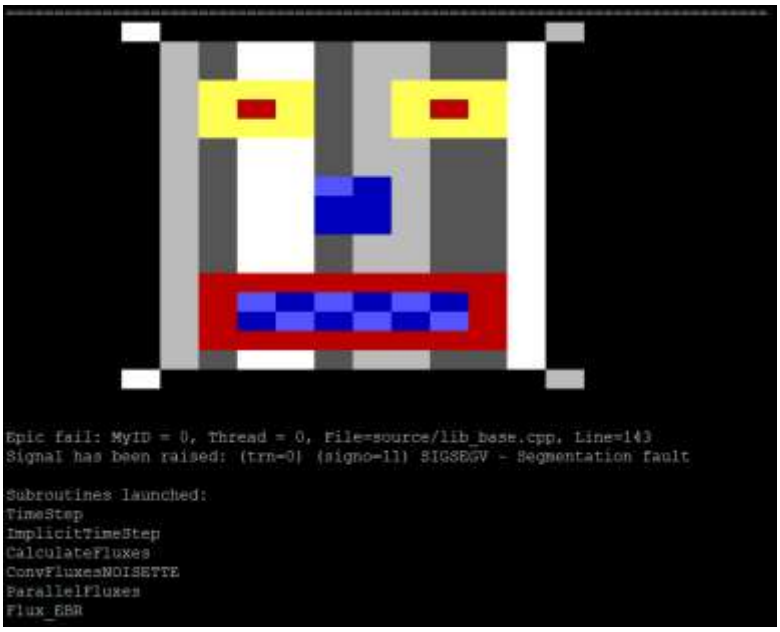
Да, чтобы ловить наш любимый и второй по популярности сигнал (после сегфолта), надо включить исключения для операций с плавающей точкой:

```
#if defined(WIN32) || defined(WIN64)
#include <float.h>
#include <windows.h>
#else
#include <fenv.h>
#endif

// включаем FPE при запуске программы в сейфмоде
void EnableFPExeptions(){
    #if defined(WIN32) || defined(WIN64)
        #if(_MSC_VER>=1700) // проверено, косяк 15-х вижлов. глючит!
            #define EM_VALUE ((_EM_ZERODIVIDE | _EM_OVERFLOW))
        #else
            #define EM_VALUE (_EM_INVALID | _EM_ZERODIVIDE | _EM_OVERFLOW)
        #endif
        _control87(~((unsigned)EM_VALUE), EM_VALUE); // включаем исключения
    #else
        feenableexcept(FE_DIVBYZERO | FE_INVALID | FE_OVERFLOW); // включаем исключения
    #endif
}
```

Проверим, как это все работает. Вставим какую-нибудь гадость куда-нибудь в код:

```
void Flux_EBR(const int iseg, double* Flux, int FillMatrix){
    INFORM_KGB_LOW(__FUNCTION__);
    *((int*)0) = 123; // гадость!
```

Запускаем. Код падает с культурной диагностикой:



**Монитор выделения памяти.** Чтобы ловить всякие утечки памяти в продакшене, а также, чтобы получать отчет о том, кто, где, когда и сколько потребляет памяти, обернем все выделение памяти своими функциями, которые будем вызывать вместо new:

```
// Memory allocation wrappers
template <typename T> T* GimmeMem(size_t N, const char* label=NULL); // Allocation
template <typename T> void FreeMem(T* &ptr); // Deallocation

// Allocation example
int* IA = GimmeMem<int>(N+1, "IA array in CSR matrix A"););
```

GimmeMem вносит текстовую метку, объем выделенной памяти и указатель (N) в глобальный (thread safe!) список. FreeMem – удаляет. Ну и все. Если будут утечки на релизе в продакшене, мы сразу это отловим своими силами без свяких валгриндов, мониторя количество аллокейтов. А если утечек не будет, то просто узнаем, кто, сколько съел памяти:

```
-------------------------------- MEMORY REPORT --------------------------------
     Current  %      MaxAlloc      MaxOnce   Ntot  Nact  Channel name
    17072296  1%     43027816     25955520     13     9  UNKNOWN CRAP
           0  0%      1431448       954296      2     0  CommData
    11346572  1%     11346572     10869424      2     2  Tadj
     7443172  0%      7443172      6966024      2     2  Nadj
    13932052  1%     13932052      6966024      3     3  Sadj
           0  0%     19548580     16304136      2     0  SegElements
    25955520  2%     25955520     25955520      1     1  SegNormals
    19548580  2%     22793020     16304136      3     2  Segments
   129777600 14%    129777600    129777600      1     1  SegIC
    49239316  5%     49239316     41796144      3     3  GreenGaussGradients
    38933280  4%     38933280     38933280      1     1  DerivativeSegCoeff
   348301200 40%    348301200    348301200      1     1  Matrix
    19085760  2%     19085760     14314320      2     2  DUXYZ_VISC
    38171520  4%     38171520     38171520      1     1  solver
    ...
-------------------------------------------------------------------------------
   870392518        872840303    348301200    341    97  == TOTAL ==
-------------------------------------------------------------------------------
```

**Таймирование.** Чтобы получать на реальном расчете распределение затрат времени по разным функциям и операциям, используется встроенное в код ручное инструментальное профилирование. Делаем объектик, который будет вести список открытых каналов измерения, идентифицируемых по текстовым меткам. В нем при открытии канала измерений запоминаем время начала измерений, при закрытии канала измерений находим время выполнения и плюсуем его к данному каналу. В коде расставляем замеры:

```
Timer.begin("BMatSpMV");
BMatSpMV(blk, ia, ja, a, x, y);
Timer.end("BMatSpMV");
```

В процессе расчета можем с нужной скважностью получать отчеты о распределении времени, что позволяет контролировать корректность вычислений и настраивать численный метод. Выглядит оно как-то так:

```
---------------------- CR_TIMER REPORT -------------------------------------
         Channel name                    N     tsum(min)  tsum(max)    %
 Total timestep                         10      12.472     12.472   100.000
  BeginTimeStep                         10       0.251      0.251     2.011
   CalcBeforeDT                         10       0.219      0.219     1.756
    Viscosity0                          10       0.210      0.210     1.683
   CalcDTL                              10       0.015      0.015     0.118
  MainTimeStep                          10      11.981     11.981    96.064
   Newton fluxes                        20       4.488      4.488    35.985
    ConvFluxes                          20       3.983      3.983    31.935
     ParFluxConVisEBR                   20       3.983      3.983    31.933
    BC                                  20       0.210      0.210     1.685
    CalcPhysVars                        10       0.010      0.010     0.077
    Viscosity0                          10       0.210      0.210     1.686
     ViscGradients                      10       0.183      0.183     1.471
      NodalGradient_gen                 10       0.160      0.160     1.280
     Visc MolViscosity                  10       0.027      0.027     0.215
   Newton solver                        10       6.904      6.904    55.360
    Solver BiCG-STAB                    10       6.904      6.904    55.360
     BVecDot                           172       0.062      0.062     0.495
     AllocPrecondM                       1       0.002      0.002     0.018
     InitPrecondM                       10       0.075      0.075     0.599
      BMatInvertDiagM                   10       0.075      0.075     0.599
     SolvePrecondM                     108       4.613      4.613    36.990
     BVecUpdate                        118       0.000      0.000     0.001
     BMatSpMV                          118       1.907      1.907    15.289
     BVecXYPBY                         108       0.092      0.092     0.739
  FinishTimeStep                        10       0.240      0.240     1.924
   Residuals and IO                     10       0.117      0.117     0.938
    History                             10       0.000      0.000     0.000
    Visualize                           10       0.000      0.000     0.000
    Recovery                            10       0.000      0.000     0.000
-----------------------------------------------------------------------------
```

Видим в общих чертах дерево вызовов, по вложенности автоматом добавляются смещения пробельчиками, выдается количество вызовов, минимальное время по всем процессам, максимальное (чтобы ловить дисбаланс), средне, можно печатать отдельно раскладку по каждому процессу (чтобы ловить тормозные узлы кластера, например). Это, конечно, штука для низкочастотных замеров. Таймирование расставляется только на верхнем уровне, чтобы не снижать производительность. Если нужно локализовать нагрузку на нижнем уровне, то используем какой-нибудь внешний статистический профилировщик.

### 7.3.4 Контроль качества

Проверка корректности кода состоит из двух частей:

- QA тесты, задача которых – убедиться, что новая ревизия кода ничего не сломала, что все работает корректно, то есть, что результаты совпадают с результатами эталонной ревизии;
- Набор валидационных тестов, которые нужны, чтобы проверять, что реализованные в коде численные методы и вся технология расчета в целом дают результаты необходимой точности.

QA тестирование делается это путем сравнения результатов тестовых расчетов (для вычислительной части) или просто разных выходных файлов (для пре- и постпроцессора) с эталонными результатами. У нас таких тестов несколько сотен. Эти тесты покрывают (точнее, должны покрывать) всю основную реализованную функциональность. Чтобы, если что-то сломалось, по результатам тестирования сразу можно было догадаться, что именно и где именно пошло не так.

QA тесты делаются на очень маленьких сетках, чтобы они работали как можно быстрее. В этом тестировании важно показать не то, что результаты расчета правильные, а то, что они просто не изменились относительно предыдущей версии. Поэтому не обязательно брать подробные сетки, необходимые для получения хоть сколько-то точных результатов. И уж тем более не обязательно выполнять весь расчет целиком, достаточно подняться с заранее заготовленного чекпоинта (рестарта) и проделать несколько шагов по времени. Более того, на полном расчете и не посравниваешься, потому что у параллельной версии быстро накапливаются расхождения, возникающие из-за округления чисел (ширина мантиссы-то конечная) при изменении порядка арифметических операций (от перемены мест слагаемых сумма-то меняется!). Начальное расхождение порядка $10^{-15}$ обычно быстро разрастается по мере интегрирования по времени, и через несколько тысяч шагов при моделировании сильно нестационарного процесса легко может стать порядка 1. При этом код будет работать корректно. Просто от распараллеливания меняется порядок обработки сеточных объектов, из-за чего просто меняется, например, порядок суммирования потоков в ячейки. Этого достаточно, чтобы разрастался расходняк (*жарг.* расхождение с эталонными результатами). Поэтому сравнение выполняют на каком-то фиксированном малом числе шагов (у нас обычно порядка 10).

Скорость работы QA важна, поскольку эту процедуру нужно делать часто. У нас QA уже раздулось непомерно, минут 30–40 выполняется на многоядерном сервере.

Поскольку код параллельный, тесты выполняются и в последовательном, и в параллельных режимах, и по MPI, и по OpenMP, и по OpenCL, и в комбинации. Важно, чтобы QA покрытие было надежно, потому что потом отлавливать ошибки в параллельном коде – это сильно неприятное занятие.

Наборы тестов снабжены скриптами для автоматического выполнения и выдачи отчета. Результаты тестирования по каждому набору тестов выдаются в логи, куда печатаются в основном нормы расхождений вида (фрагмент):

```
================================================
.\QA_3D\caseAngularPeriodics
ERR_UA = 2.081668e-17 (cell 78 var 4)
errL1UA = 1.414089e-15
================================================
.\QA_3D\caseBLUNCH_EBR5_EXPL
ERR_UA = 5.684342e-14 (cell 488 var 4)
errL1UA = 5.595784e-12
================================================
.\QA_3D\caseAngularPeriodics
```

Скрипт для QA тестирования имеет web-морду. Можно с интернетов смотреть результаты тестирования ревизий, сданных под SVN, и запускать тестирование для нужной ревизии, если оно еще не сделано. Если на какой-то ревизии что-то сломалось, ответственным

и виновным сервер QA рассылает письма счастья по электронной почте. Выглядит QA страница примерно так:

Current QA run revision

request QA run

8212

Testcases statistics:
540 passed, 4 failed (544 total)

STARTED AT 2020-12-25 23:00:01
NEW_REVISION PASSED 8212(2020-12-25 20:02:20 +0300 by veld)
CODE_CHECKOUT PASSED /qa/run.results/8212/code_checkout.log
QA_CHECKOUT PASSED /qa/run.results/8212/qa_checkout.log
COMPILATION PASSED /qa/run.results/8212/compilation.log
AUTOQA_LOG GENERATED /qa/run.results/8212/autoqa.log

| Название теста | Статус | Результаты | Проверка условий | Режим |
|---|---|---|---|---|
| QA_utils/extrusion | FAILED | QA_log.txt \| extrudemesh.std \| linkmeshes.std \| meshconvert.std \| submesh.std \| testcase.stdout \| transform1.std \| transform2.std \| transform3.std \| transform4.std | conditions.txt | SP |
| QA_STIEFEL_3D/caseHybridMeshVisc | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | SP |
| QA_STIEFEL_3D/caseHybridMeshVisc | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | 4xMPI |
| QA_STIEFEL_3D/caseScheibe05 | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | SP |
| QA_STIEFEL_3D/caseViscImplNoslipPeriodic | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | SP |
| QA_STIEFEL_3D/caseViscImplNoslipPeriodic | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | 4xMPI |
| QA_STIEFEL_3D/caseCHWENO5 | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | 4xMPI |
| QA_STIEFEL_3D/caseCHWENO5 | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | SP |
| QA_STIEFEL_3D/caseLEE_LO_RK01Incut | PASSED | QA_log.txt \| testcase.stdout | conditions.txt | 4xMPI |

Showing 1 to 544 of 544 entries

Если сдается ревизия, которая поменяла эталонные результаты QA, или сдается какая-то новая функциональность, реализация какого-то нового метода или модели, для которой еще нет эталонных результатов, то нужно проделывать валидацию. Если QA – это быстрые тестики, просто сделать несколько шагов интегрирования по времени на микроскопических сетках, то валидационные тесты – это уже тяжелые полные расчеты для получения результата необходимой точности. QA, соответственно, делается часто, а валидация – редко, по мере необходимости. Получив корректные результаты на валидации, нужно обновить QA тесты, перегенерировав референсные данные. Для новых вещей надо делать новые QA тесты. Сами эти QA тесты полностью живут под системой управления версиями, как и код.

Соблюдая регламент по контролю качества, можно более-менее успешно поддерживать корректную версию расчетного кода. Хотя ошибки все равно иногда просачиваются в продуктовые расчеты, и приходится использовать средства предыдущего пункта.

## 7.4 Общие замечания и рекомендации.

Во введении мы разбирались, зачем нужны CFD коды. Потом разбирались, как их, а также аналогичные расчетные коды из других областей, реализовать поэффективнее. Было много всякой оптимизации, эффективности, распараллеливания, доступа к памяти, … Из-за всего этого может сложиться неверное представление о сути происходящего. Да оно наверняка и сложилось. Попробуем это хоть немного исправить.

Вот что самое главное в HPC? В этом всем суперкомпьютерном моделировании. Достичь максимальной фактической производительности? Нет. Получить максимальную параллельную эффективность и масштабируемость? Тоже нет. Чтобы понять, что главное, надо вспомнить, для чего все это делается – для решения какой-то определенной задачи. Мы же делаем расчеты не ради расчетов, а ради того, чтобы выяснить что-то нужное для решения какой-то практической задачи. Так вот, при решении задачи главное – вообще не оказаться на суперкомпьютере! Пусть мы занимаемся предсказательным математическим

моделированием. То есть, хотим узнать, спрогнозировать свойства какого-то объекта в определенных условиях еще до того, как он будет существовать в реальности. Узнать, какой будет коэффициент подъемной силы и сопротивления у летательного аппарата, или как будет деформироваться кузов нового автомобиля при ударе о препятствие, или много еще чего, о чем был раздел Введение. Проследим следующую логическую цепочку.

- **Лучше всего, если можно предсказать вообще без вычислений.** Если нам надо предсказать, что будет с висящим на дереве яблоком, если оно оторвется от ветки, то нам даже не надо ничего моделировать. Оно, ясное дело, упадет.
- **Если не получается, то лучше, если можно посчитать на калькуляторе.** Пусть нам надо примерно узнать, как быстро упадет яблоко. Берем калькулятор, суем в него высоту и ускорение свободного падения, и задача решена.
- **Если не получается, то лучше, если можно посчитать десктопе.** Если надо найти поточнее, что будет происходить с яблоком в полете, какая на него будет действовать сила сопротивления, то может понадобиться газодинамический расчетик, который в грубой постановке можно прогнать и на обычном компьютере.
- **Если не получается, то лучше, если можно посчитать на мощном сервере.** Если надо найти аэродинамические характеристики с учетом тонких эффектов, влияния турбулентности, то может понадобиться стационарный расчет на подробной сетке или даже вихреразрешающий расчет, который потребует больше мощностей.
- **Если не получается, то лучше, если можно посчитать на маленьком кластере.** Если это летит не яблоко, а что-то более сложной формы и на более высокой скорости, то может понадобиться еще более подробная сетка и несколько вычислительных серверов.
- **Наконец, если совсем ничего не получается,** если по-другому никак, то только **от отчаяния и безысходности** мы полезем считать что-то на суперкомпьютере! Если нам надо посчитать нестационарную аэродинамику сложного объекта в сложных условиях, найти пульсационные нагрузки, акустические характеристики, то без суперкомпьютера уже сложно обойтись.

Основная проблема, самое главное, самое основное – минимизировать ресурсоемкость решения задачи! Если мы можем придумать математическую модель, которая позволит решать задачу на обычном компьютере, как, например, RANS модели турбулентности во многих случаях, то это оно – самое главное. Данный курс, по сути, находится в конце этой некой цепи. Кто-то придумывает методы, модели (кафедры Вычислительных методов, Математической физики – это про вас, между прочим), а наша задача, параллельных программистов, всего лишь придумать поэффективнее алгоритм и поэффективнее задействовать вычислительную систему. Это надо понимать.

А то часто приходится слышать, как кто-то гордо заявляет, что посчитал на миллиардной сетке с завидной параллельной эффективностью и производительностью на тысячах ядер какой-то расчет. И даже гордо пишет об этом статьи в научные журналы. А когда узнаешь, что именно он посчитал, хочется ответить – чувак, возьми не декартову сетку, а “неструкт” погранслойный, возьми не убогую схему первого порядка, а нормальную схему с приличной реконструкцией, возьми подходящую модель турбулентности прикрути, и решай свою задачу у себя на ноутбуке! И не лезь с этой универсальной субстанцией (*жарг. груб.*) прожигать очень дефицитные суперкомпьютерные ресурсы! Вот. Отсюда такая получается иерархия: главное в основе всего – метод, модель; потом алгоритм – от того, какая у алгоритма вычислительная

сложность сильно зависит ресурсоемкость; наконец, на последнем месте в цепи – реализация. Но без нее тоже никак не обойтись. Хорошая реализация сокращает ресурсоемкость в разы.

Пусть у нас теперь задача не такая глобальная – что-то промоделировать, а придумать что-то локальное, как сделать какое-то конкретное действие в рамках этого “промоделировать”. Придумать алгоритм какого-нибудь поиска по графу, какой-то раскраски, какой-то конкретной обработки данных, способа реализации чего-нибудь.

По результатам данного курса в голове наверняка образовалась каша, бардак и хаос из всяких эффективностей, оптимизаций, всяких доступов к памяти, параллельных моделей. Это все может сильно мешать придумывать алгоритм! Когда в голове одновременно крутится, помимо самой задачи, которую надо решить, еще и месиво из этих всяких оптимизаций, результат креативной деятельности может быть печальным. Поэтому не пытаемся придумывать что-то и оптимизировать это что-то одновременно!

Можно действовать поступательно, в такой последовательности. Сначала придумали хоть какой-нибудь алгоритм, который решает задачку. Потом уже думаем, а можно ли это сделать с меньшей вычислительной сложностью? Нет ли там квадратичной сложности по вычислениям или ресурсоемкости по памяти, например? Нельзя ли это сделать за линейную сложность? Первое, с чем заморачиваемся – боремся за асимптотику ресурсоемкости. Это повышает производительность на порядки. Когда разобрались со сложностью, думаем, как это распараллелить. А уже потом думаем, как это получше реализовать, как организовать структуру данных, какие лучше подходят контейнеры, куда вектора, куда сеты, что на мэп посадить, и т.д. Когда и с этим разобрались, занимаемся остальной оптимизацией. Вот. Смысл – действовать поступательно, а не пытаться одновременно. Если на последующем этапе столкнулись с неразрешимой проблемой, откатимся на предыдущий этап, еще подумаем. Такое вот рациональное предложение.

Ну и еще к вопросу оптимизации, можно уточнить, что – нужна ли вообще оптимизация? Следует ли необходимость заморочек с производительностью из нефункциональных требований – см Рик Касман - *абилити – 6 требований – может вообще не надо оптимизировать.

Атрибуты качества – см нефункциональные требования

Все должно вообще писаться в соответствии с требованиями заказчика, смотрите на бизнес-задачу, что надо клиенту. Применение данного курса не по делу может только все испортить и другие абилити просядут. Это тоже надо иметь в виду.

Ну и когда придете работать куда-то в разработку, первым делом выясняйте регламент работы с кодом, не подставляйтесь на такой ерунде. Вытрясайте из куратора шаблоны стайл-гайда, общие принципы работы, регламент сдачи ревизий, чтобы ничего не нарушать, никуда в код не гадить, не позориться. В общем, успехов в дальнейшей работе! Хотя… Для начала – успехов на экзамене.

# 8 Практические задания

## 8.1 Задание 1: многопоточная реализация операций с сеточными данными на неструктурированной смешанной сетке, решение СЛАУ

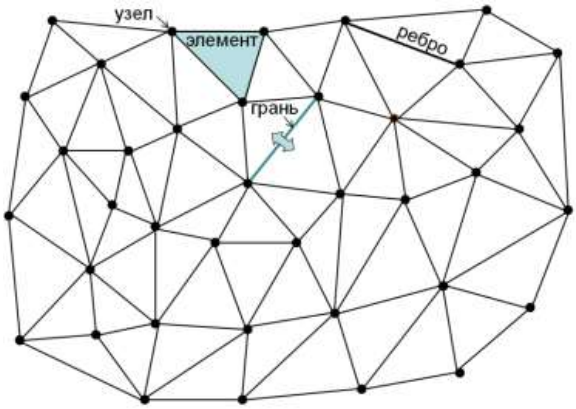
Цели задания:

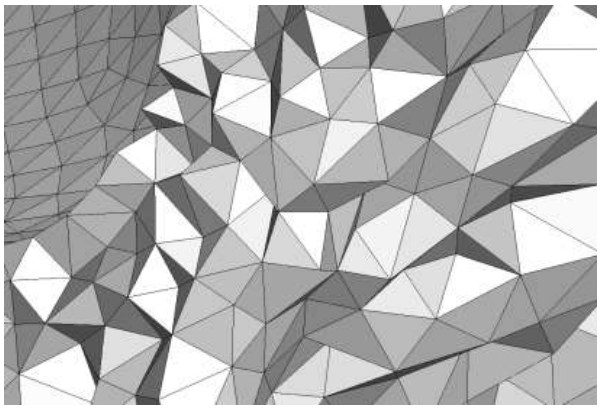
- освоение базовых структур данных для представления неструктурированной сетки, графа связей расчетных ячеек, портрета разреженной матрицы;
- постижение взаимосвязи между сеткой, графом и разреженной матрицей;
- освоение многопоточного распараллеливания простейших операций.

В качестве наиболее простой и репрезентативной операции выбрано матрично-векторное произведение с разреженной матрицей, портрет которой соответствует дискретизации на неструктурированной сетке. Задание будет разделено на этапы.

### 8.1.1 Введение

Для дискретизации по пространству используется неструктурированная сетка. Сетка – это разбиение некоторой пространственной расчетной области на сеточные элементы (в двухмерном случае – многоугольники, в трехмерном – многогранники). Сетка задана в виде набора узлов и набора элементов. Объединение сеточных элементов полностью заполняет нашу расчетную область, а сами элементы не наползают друг на друга, а стыкуются через общие грани и узлы. Ниже на картинке слева показан пример двухмерной треугольной сетки. Двухмерной – для простоты рисования картинок. В случае трехмерной сетки (справа) всё, по сути, то же самое, для наших целей никакой разницы.



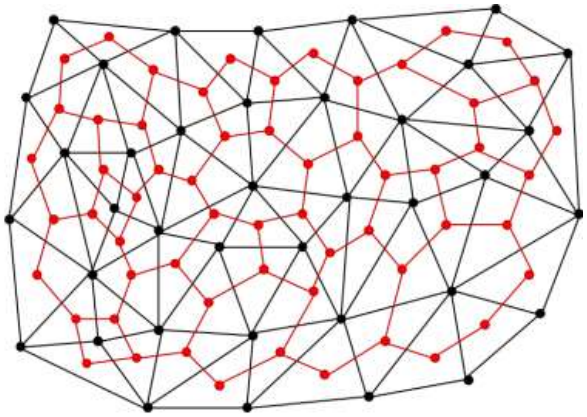



У нас есть несколько наборов сеточных объектов – узлов, элементов, ребер и граней этих элементов. На картинке (слева) элементы – это треугольнички, но могут быть и всякие другие многоугольники. А ребра и грани в двухмерном случае – это одно и то же. В трехмерном случае (как справа) грани станут многоугольниками – гранями многогранных элементов (их видно на картинке справа), а ребра останутся ребрами.

Теперь надо понять, где на сетке заданы сеточные функции, то есть, какие-то переменные, хранящиеся в сеточных объектах. Это будет дискретизация. В частности, можно задавать переменные 1) **в узлах**, а можно 2) **в элементах**. В первом случае контрольные объемы, то есть расчетные ячейки или просто ячейки, будут как-то строиться вокруг узлов, во втором – ячейками будут сами сеточные элементы.

Для дискретизации на сетке есть граф, описывающий связи между ячейками, то есть смежность ячеек (элементов или узлов). В таком графе связей (или смежности) ячеек вершины графа соответствуют ячейкам – элементам или узлам сетки. Ребрам графа соответствуют общие грани между ячейками. Если между ячейками i и j есть общая грань, то в графе вершины i и j соединены ребром. Когда ячейки – это элементы, то все понятно, есть общая грань, значит элементы смежные. Когда ячейки строятся вокруг узлов, получится некая двойственная (дуальная) сетка, грани которой будут проткнуты сеточными ребрами. Поэтому грани между ячейками можно взаимно однозначно сопоставить сеточным ребрам. Тогда ячейки вокруг узлов будут смежными, если узлы имеют соединение ребром – тоже все просто.

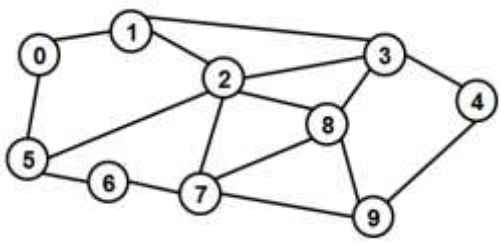
Наша сеточка на картинке, это, по сути, и есть граф. Но это узловой граф. Его вершины соответствуют узлам. Это как раз и будет граф для случая 1 с определением переменных в узлах. Для случая 2 с определением переменных в элементах нужен двойственный (dual) граф. Вот тут он красным нарисован.



Теперь у нас есть вершины графа, с которыми ассоциированы ячейки и заданные в них наборы переменных, и его ребра, которым соответствуют связи между ячейками. Если ячейки построены вокруг узлов, то это черный граф на картинке выше, если ячейками являются элементы, то это красный граф.

У графа есть матрица смежности. Это такая матрица N×N, где N – число вершин в графе, в которой ненулевые позиции соответствуют ребрам между вершинами графа. Коэффициент матрицы $a_{ij}$ в j-м столбце i-й строки равен 1, если между вершинами i и j графа есть ребро, иначе там ноль. Если такое ребро есть, то и $a_{ji}$ – неноль. Значит, число ненулей в матрице равно удвоенному числу ребер в графе.

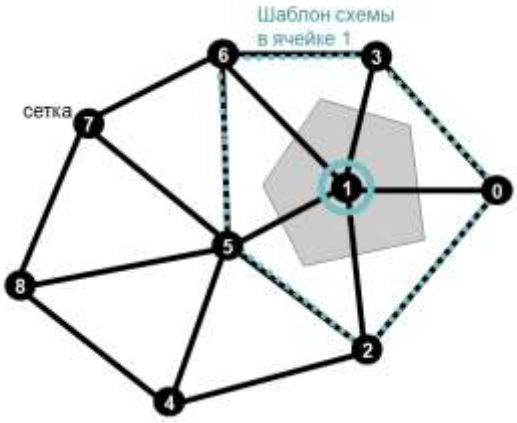
Возьмем какой-нибудь граф. Построим для него матрицу смежности и запишем ее портрет. Вот такой граф и его матрица для примера:




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |
| 3 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |
| 7 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 1 |
| 8 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| 9 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 |

Часто в вычислительных алгоритмах приходится иметь дело с разреженными матрицами, портрет которых имеет ненули на главной диагонали, а во внедиагональных позициях совпадает с портретом матрицы смежности. С такими матрицами мы и будем работать: **матрица смежности + главная диагональ**.

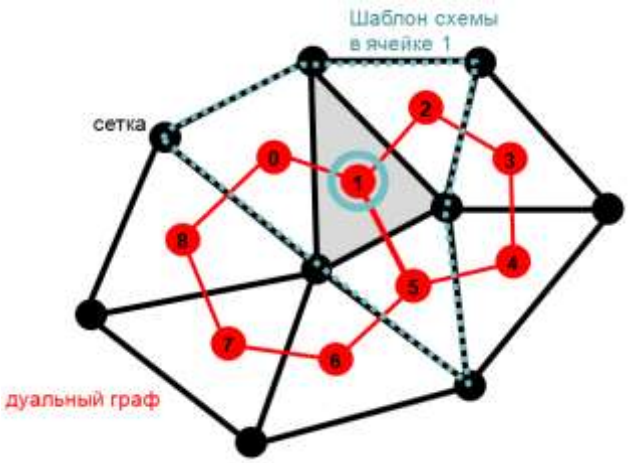
Вот пример треугольной сетки с определением сеточных функций в узлах сетки:




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | # | # | # | # | | | | | |
| 1 | # | # | # | # | | # | # | | |
| 2 | # | # | # | | # | # | | | |
| 3 | # | # | | # | | | # | | |
| 4 | | | # | | # | # | | | # |
| 5 | | # | # | | # | # | # | # | # |
| 6 | | # | | # | | # | # | # | |
| 7 | | | | | | # | # | # | # |
| 8 | | | | | # | # | | # | # |

В этом примере шаблон схемы в ячейке состоит из этой ячейки и соседних (смежных) с ней ячеек, т.е. ячеек, имеющих с данной ячейкой соединение ребром сетки. Номера в узлах обозначают номера неизвестных в векторе. Справа показан портрет матрицы. Номер узла в сетке соответствует номеру строки матрицы. Номера столбцов в строке узла соответствуют номерам соседних с ним узлов. Также присутствует диагональный элемент – каждый узел сам себе сосед.

На следующем рисунке показан пример для случая определения сеточных функций в элементах сетки. Пока нас интересует только портрет (то есть расположение ненулей). Значения коэффициентов матрицы определяются используемым численным методом. Способ их расчета в рамках данного задания нас не интересует, мы их нагенерим потом как попало.




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| 0 | # | # | | | | | | | # |
| 1 | # | # | # | | | # | | | |
| 2 | | # | # | # | | | | | |
| 3 | | | # | # | # | | | | |
| 4 | | | | # | # | # | | | |
| 5 | | # | | | # | # | # | | |
| 6 | | | | | | # | # | # | |
| 7 | | | | | | | # | # | # |
| 8 | # | | | | | | | # | # |

В этом примере шаблон схемы в ячейке состоит из этой ячейки и ее соседних ячеек, т.е. ячеек, имеющих с данной ячейкой общую грань. Справа показан портрет матрицы. Номера в узлах обозначают номера неизвестных в векторе. Номер элемента в сетке (красные кружки) соответствует номеру строки матрицы. Номера столбцов в строке узла соответствуют номерам

соседних с ним элементов. Также присутствует диагональный элемент – каждый элемент сам себе сосед.

**Формат хранения графа и разреженной матрицы**

Граф можно описать в виде портрета его матрицы смежности. Запишем портрет матрицы, то есть описание, где в матрице ненулевые позиции, в построчено-разреженном формате CSR (Compressed Sparse Row). Пусть в графе N вершин и E ребер. Для хранения этого добра нам нужны 2 массива:

**IA** – целочисленный, размера N+1 (также называют rowptr, xadj, ...). В нем для каждой строки храним позицию начала списка столбцов данной строки. Последний элемент в массиве, IA[N], хранит общее число ненулей в матрице (для матрицы смежности графа – равное удвоенному числу ребер графа – 2E).

**JA** – целочисленный, размера IA[N] (также называют colptr, adjncy, ...). В нем подряд для всех строк матрицы (упорядоченно по строкам) хранятся номера столбцов с ненулевыми значениями. И пусть для определенности номера столбцов каждой строки упорядочены по возрастанию.

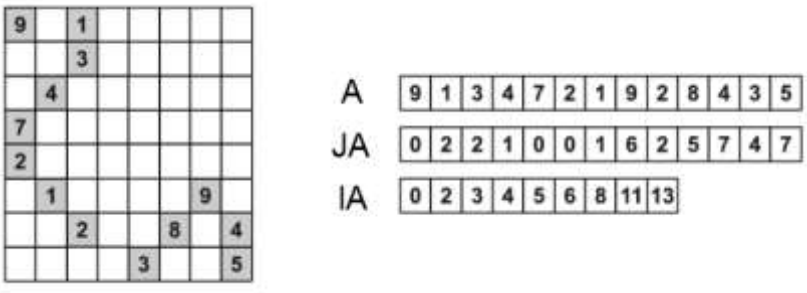
Как нам, например, перебрать все ненулевые столбцы i-й строки? Что, по сути, то же самое, что перебрать всех соседей i-й вершины. Вот так, например:

```
for(int col=IA[i]; col<IA[i+1]; ++col){
    j = JA[col]; // номер соседней вершины, т.е. соединенной с вершиной i ребром.
    ...
}
```

Это был портрет матрицы. Для того, чтобы это стало матрицей, нам нужны коэффициенты. Они хранятся в третьем массиве:

**A** – вещественный, размера IA[N]. В нем подряд по всем строкам хранятся коэффициенты матрицы.

Пример:




Доступ к ненулевым коэффициентам i-й строки:

```
for(int col=IA[i]; col<IA[i+1]; ++col){
    int j = JA[col]; // номер столбца
    double a_ij = A[col]; //значение коэффициента aij
    . . .
}
```

В случае произвольной неструктурированной сетки максимальное число смежных узлов никак не ограничено. Обычно на практике для тетраэдральной сетки в среднем у узлов где-то

по 15 соседей, а максимальное число может быть и больше 30 легко. Для случая 1 с определением переменных в узлах хорошо подходит формат CSR.

Для случая 2 число соседей ограничено максимальным числом граней элементов. Для тетраэдральной сетки – это всего 4. Для смешанной сетки с элементами до 6 граней число соседей тоже будет варьироваться не сильно, от 4 до 6, не считая граничных ячеек, которых мало, на них можно не обращать внимание. Тут может лучше подойти формат ELLPACK. Этот формат предполагает, что у всех строк равное число ненулевых коэффициентов. Раз оно равное, то мы просто берем CSR, выкидываем из него массив IA, который больше не нужен, а позицию строки в векторах JA и A находим, просто умножая номер строки i на константу, пусть m, равную числу ненулей в строке. Получим то же самое, но без IA:

```
for(int col=i*m; col<(i+1)*m; ++col){
    int j = JA[col]; // номер столбца
    double a_ij = A[col]; //значение коэффициента aij
    . . .
}
```

Но что делать со строками, в которых ненулей меньше, чем m? С граничными ячейками, например, у которых есть внешние грани, по которым нет соседей. Если нет ненулей, то доберем нулями. Вместо каких-то ненулевых коэффициентов окажутся нули, ну ничего страшного. Ну в A мы легко положим 0, а что подставить в JA? В JA подсунем, например, номер столбца последнего ненулевого коэффициента в строке, чтобы не надо было читать никаких лишних позиций из входного вектора.

Что такое сетка разобрались, как хранить матрицы, связанные с сеткой, тоже поняли. Можно перейти к постановке задачи.

### 8.1.2 Постановка задачи

Работа программы, которую будем делать, состоит из 4 этапов:

1. **Generate** – генерация графа/портрета по тестовой сетке;
2. **Fill** – заполнение матрицы по заданному портрету;
3. **Solve** – решение СЛАУ с полученной матрицей;
4. **Report** – проверка корректности программы и выдача измерений.

Сделать задание можно на языках C, C++. С новыми стандартами C++11 и выше (на момент написания этого текста) могут быть проблемы на кластерах ВМК, особенно на Блюджине.

**Этап 1: Generate – генерация портрета на основе тестовой сетки**

Для простоты будем иметь дело с двухмерной неструктурированной смешанной сеткой, состоящей из треугольников и четырехугольников.

Кто хочет поиграться с 3D – берем за основу двухмерную сетку и вытягиваем сколько угодно призматических слоев. Будет вполне себе трехмерная смешанная сетка их гексаэдров и треугольных призм.

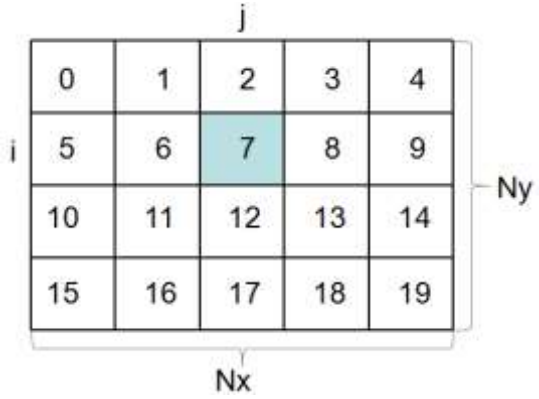
Есть два варианта определения сеточных функций

**A) в узлах;**

**B) в элементах.**

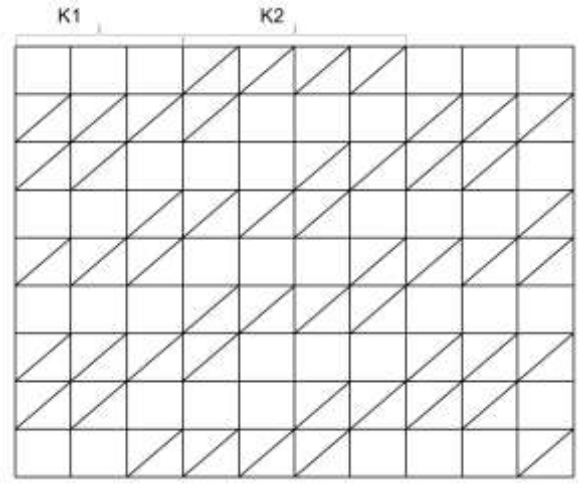
Для упрощения генерации тестовой сетки будем использовать самое просто, что только может быть – двухмерную решетку.

Решетка состоит из N = Nx×Ny клеточек. У каждой клеточки есть позиция (i, j), где i – номер строки в решетке, j – номер столбца. Нумерация клеточек в решетке построчно слева направо, сверху вниз. Номер клеточки в единой общей нумерации: I = i*Nx + j.




Итак, мы задали два параметра Nx, Ny, получили решетку. Теперь получим из решетки сетку. Чтобы сетка была смешанной, то есть чтобы в ней были разные типы элементов, часть квадратных клеток поделим на треугольники. Для этого введем еще два параметра K1, K2, чтобы регулировать соотношение числа треугольников и четырехугольников. Будем делить клетки следующим образом. K1 клеток не делим, K2 следующих клеток делим, K1 следующих клеток снова не делим, K2 следующих клеток снова делим, и так далее. Легко догадаться, что клетка поделена на треугольники, если I%(... впрочем, вы сами легко напишете формулу.

Пример для Nx=10, Ny=9, K1=3, K2=4:




Да, поделить четырехугольник можно двумя способами:

Способ 1 Способ 2

Получились следующие варианты:

А) ячейки, то есть вершины графа = **узлы**

Б) ячейки, то есть вершины графа = **элементы**

1) делим клетки решетки **способом 1**

2) делим клетки решетки **способом 2**.

Итого 4 варианта: **А1, А2, Б1, Б2**.

Ну это чтобы всем не делать одно и то же. Жизнь же должна быть сложной? И вам будет чуть сложнее передирать, и проверяющим сложнее проверять.

Вариант определяем, например, по списку группы по алфавиту. Берем порядковый номер в списке, находим остаток от деления на 4, по нему берем один из этих вариантов в указанном выше порядке.

На этапе 1 нужно сгенерировать "топологию" связей ячеек, то есть построить графа, и по нему сгенерировать портрет матрицы смежности. И дополнить этот портрет главной диагональю! Обязательно. **Не забываем главную диагональ!**

**Входные данные:**

**Nx, Ny** – число клеток в решетке по вертикали и горизонтали;

**K1, K2** – параметры для количества однопалубных и двухпалубных кораблей треугольных и четырехугольных элементов.

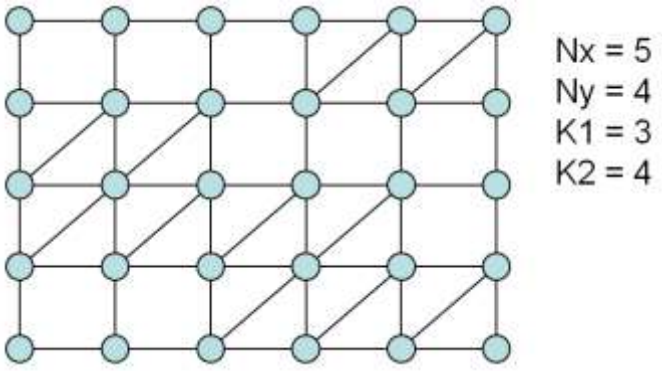
**Выходные данные:**

**N** – размер матрицы = число вершин в графе;

**IA, JA** – портрет разреженной матрицы смежности графа, дополненный главной диагональю (в формате CSR, при желании для случая Б можно делать ELLPACK).

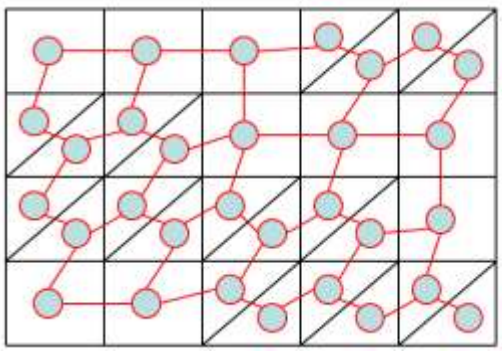
Чтобы было понятнее, разберемся еще подробнее с разными вариантами, что там будет графом, и как что нумеруется.

Вариант А – вершины графа = узлы сетки. Узлы сетки – это точки пересечения сеточных линий. Вот они кружочками отмечены:




Граф связей узлов совпадает с сеткой, он тут и нарисован на этой картинке. Вершин в графе N = (Nx+1)×(Ny+1). Нумерация узлов точно такая же – слева направо, сверху вниз. Удобство этого варианта – сразу понятно число вершин, оно не меняется от разбиения на треугольники. Неудобство – чтобы посчитать число соседей у вершины, надо смотреть, что делается в соседних клетках.

Вариант В – вершины графа = элементы сетки, то есть четырехугольники и треугольники. Граф связей – двойственный граф сетки. Элементы считаем смежными, если у них есть общая грань. Граф будет такой:




Нумерация элементов такая же, слева направо, сверху вниз. Что там из них левее/правее/выше/ниже – определяем по центру масс элемента. Удобство этого варианта – сразу понятно, сколько соседей у ячеек. Неудобство – число ячеек зависит от параметров разбиения на треугольники.

Во всех вариантах не забываем, что к портрету надо добавлять главную диагональ.

Генерацию надо сделать в виде функции с входными и выходными данными в качестве аргументов.

В main только считываются параметры пользовательского ввода (из командной строки или файла) и вызывается эта функция генерации. Интерактивный ввод не надо использовать. Ну то есть не надо никаких cin >> и прочих scanf. Обычно расчетные коды запускаются через систему очередей кластера не в интерактивном режиме, поэтому интерактивный ввод неуместен.

Программа должна брать хотя бы один аргумент командной строки. Если ввод параметров через файл, то это имя входного файла. Если программа запущена без аргументов, она должна напечатать хелп, как ей пользоваться.

Программа должна иметь проверку корректности пользовательского ввода. В случае ошибки она должна выйти с информативным сообщением, а не по какому-нибудь сегфолту.

Для проверки результатов программа должна уметь печатать выходные данные в текстовый файл или в stdout. Для этого должен быть параметр, включающий отладочную печать. Но по умолчанию печать должна быть выключена. Иначе на больших размерах будет печально.

Для проверки задания можно:

1) напечатать выходные данные для маленькой сетки, которую можно просто нарисовать на бумаге и проверить;
2) сравнить выходные данные с результатами других авторов с тем же вариантом задания;
3) проверить работоспособность с N порядка $10^6$, $10^7$, взять сеточку 2000x2000 хотя бы;
4) проверить, что расход памяти и время работы программы растут линейно с ростом числа ячеек;
5) сравнить время работы с другими авторами.

### Этап 2: Fill – построение СЛАУ по заданному портрету матрицы

**Входные данные:**

**N** – размер (квадратной матрицы NxN);

**IA, JA** – портрет матрицы.

**Выходные данные:**

**A** – массив ненулевых коэффициентов матрицы (размера IA[N]);

**b** – вектор правой части (размера N).

На этом этапе мы уже ничего не знаем о том, что сетка когда-то была решеткой. **Забыли про решетку!** Теперь мы работаем только с топологией, то есть с описанием связей между ячейками/объектами/вершинами графа в общем виде.

Для заполнения матрицы и правой части лучше использовать одинаковые формулы, чтобы можно было сравнивать результаты программы друг с другом для проверки (для одинаковых вариантов, естественно).

Например, для внедиагональных элементов строки можно взять какой-нибудь косинус от индексов строки и столбца, $a_{ij} = cos(i * j + i + j)$, $i \neq j$, $\boldsymbol{j} \in \boldsymbol{Col(i)}$, где Col(i) – множество номеров столбцов для строки i, которые входят в портрет матрицы (это то, что лежит в JA[IA[i],...,IA[i+1]–1]).

Диагональный элемент в каждой строке будем вычислять как сумму модулей внедиагональных элементов строки, умноженную на больший единицы коэффициент:

$$a_{ii} = 1.234 \sum_{j, j \neq i} |a_{ij}|.$$

Все остальные коэффициенты матрицы равны нулю (К.О.). Домножаем диагональ на коэффициентик, чтобы у матрицы было диагональное преобладание. Это нужно, чтобы решатель (солвер – от *англ.* solver), который мы дальше будем делать, устойчиво сходился к решению, а не болтался в проруби.

Заполняем матрицу так: идем по строкам, перебираем номера столбцов, если внедиагональный коэффициент – заполняем по формуле и прибавляем значение в сумму по строке, если диагональный – запоминаем позицию и пропускаем. Дошли до конца строки, домножили полученную сумму по строке на коэффициент и записали в диагональный элемент. Перешли к следующей строке, занулив сумму.

Вектор правой части заполняем по формуле $b_i = sin(i)$.

Когда функции для этапов 1 и 2 сделаны и корректно работают, нужно внедрить многопоточное распараллеливание средствами OpenMP. В связи с этим у программы появляется еще один **входной параметр:**

**T – число нитей** (threads).

Функции этапов 1 и 2 нужно распараллелить. Построение портрета, заполнение коэффициентов – всё должно выполняться параллельно T нитями.

Чтобы оценить эффект от распараллеливания, нужно добавить замеры времени. Для этого можно использовать функцию omp_get_wtime.

Замечания по выдаче. Программа может сообщать в лог о прохождении этапов, выдавать контрольные величины (например, число вершин графа, число ребер, число ненулей в портрете, ...), выдавать табличку замеров времени. По замерам будем проверять параллельное ускорение.

По умолчанию, без запроса пользователя, программа не должна никуда печатать никаких массивов, ни портрет, ни коэффициенты, ни в stdout, ни в файл. Выдача отладочной

информации – только в режиме отладки. Размеры сетки, которые надо будет тестировать, это 1 ~ 10 млн ячеек (К.О. сообщает: на таких размерах печатать портрет– плохая идея).

Результаты работы программы в параллельном и последовательном режиме должны быть полностью идентичны. Для проверки можно, например, печатать в файл массивчики и сравнивать файлы. Можно вычислять по массивам контрольные суммы и сравнивать просто числа, что удобнее, особенно на больших размерах. Можно и так, и эдак, но массивы – только в режиме отладки, а контрольные величины можно и всегда выдавать.

**Этап 3: Solve – решение СЛАУ итерационным методом**

**Входные данные:**

**N** – размер (квадратной матрицы N×N);

**IA, JA** – портрет матрицы;

**A** – массив ненулевых коэффициентов матрицы (размера IA[N]);

**b** – вектор правой части (размера N);

***eps*** – критерий остановки ($\varepsilon$), которым определяется точность решения;

***maxit*** – максимальное число итераций.

**Выходные данные:**

**x** – вектор решения (размера N);

**n** – количество выполненных итераций;

***res*** – L2 норма невязки (невязка – это вектор $\boldsymbol{r} = \boldsymbol{Ax} - \boldsymbol{b}$).

Поскольку матрица симметричная, будем использовать простейший метод сопряженных градиентов CG (Conjugate Gradient) с диагональным предобуславливателем.

Решаем СЛАУ $\boldsymbol{Ax} = \boldsymbol{b}$

Будем использовать такой алгоритм предобусловленного метода CG:

$\boldsymbol{x}_0 = 0$

$\boldsymbol{r}_0 = \boldsymbol{b}$

$k = 0$

**do**

$k = k + 1$

$\boldsymbol{z}_k = \boldsymbol{M}^{-1}\boldsymbol{r}_{k-1}$ // SpMV

$\rho_k = (\boldsymbol{r}_{k-1}, \boldsymbol{z}_k)$ // dot

**if** $k = 1$ **then**

$\boldsymbol{p}_k = \boldsymbol{z}_k$

**else**

$\beta_k = \rho_k / \rho_{k-1}$

$\boldsymbol{p}_k = \boldsymbol{z}_k + \beta_k \boldsymbol{p}_{k-1}$ // axpy

**end if**

$\boldsymbol{q}_k = \boldsymbol{A}\boldsymbol{p}_k$ // SpMV

$\alpha_k = \rho_k / (\boldsymbol{p}_k, \boldsymbol{q}_k)$ // dot

$\boldsymbol{x}_k = \boldsymbol{x}_{k-1} + \alpha_k \boldsymbol{p}_k$ // axpy

$\boldsymbol{r}_k = \boldsymbol{r}_{k-1} - \alpha_k \boldsymbol{q}_k$ // axpy

**while** $\rho_k > \varepsilon$ **and** $k <$ *maxit*

В формулах вектора и матрицы обозначены жирным шрифтом, из них матрицы обозначены прописными (заглавными) буквами.

В случае диагонального предобуславливателя матрица $\boldsymbol{M}$ – просто диагональная матрица, с диагональю из матрицы $\boldsymbol{A}$. Начальное приближение – нулевое.

Для отладки надо на каждой итерации вычислять L2 норму невязки и печатать в стандартный поток вывода с новой строки номер итерации и текущую норму невязки системы на данной итерации. Этот лог итерационного процесса можно сравнивать друг с другом для проверки (у кого одинаковые варианты, естественно). На выходе из солвера надо проверить финальную невязку и вернуть ее в переменную *res*.

Для решателя понадобится сделать несколько кернелов, то есть вычислительных функций для базовых операций в солвере. Основные из них:
**SpMV** – матрично-векторное произведение с разреженной матрицей (sparse matrix-vector);
**dot** – скалярное произведение;
**axpy** – поэлементное сложение двух векторов с умножением одного из них на скаляр, $\alpha\boldsymbol{x} + \mathbf{y}$.
Помимо этих трех базовых операций, там еще будут кернелы копирования векторов, заполнения константой.

Рекомендуется сделать проверочные тесты для каждой из базовых операций. Входные вектора можно заполнять по какой-то формуле от номера элемента. Проверочный тест для каждой операции должен выдать контрольные значения: для dot – просто результат, для axpy и SpMV контрольные значения по выходному вектору, например: сумма всех элементов, L2 норма – sqrt(dot(x, x)), и т.д. Последующие параллельные реализации можно будет сравнивать с исходными последовательными реализациями по этим контрольным значениям для подтверждения корректности.

Для решателя и составляющих его кернелов нужно сделать замеры времени, для чего можно использовать функцию omp_get_wtime().

Для каждой из трех базовых операций тоже нужно сделать замеры.

Нужно попробовать выполнить оптимизацию последовательных версий базовых операций, сохранив также исходные версии для сравнения. Можно применить развертку циклов, SIMD инструкции и т.д. Сделать таймирование по всем базовым операциям – кернелам, оценить улучшения.

Затем нужно сделать многопоточные параллельные реализации всех операций с векторами и матрицами с помощью OpenMP. Вообще все этапы 1 – 3 должны быть полностью распараллелены. В итоге надо получить многопоточную корректно работающую версию решателя, дающую адекватное ускорение при сохранении корректности результата.

**Этап 4: Report – проверка корректности программы и выдача измерений**

На этом этапе надо просто проверить, что невязка системы после работы решателя удовлетворяет заданной точности, выдать значение фактической невязки, и распечатать табличку таймирования, в которой указано, сколько времени в секундах затрачено на:

1. этап генерации;
2. этап заполнения;
3. этап решения СЛАУ;
4. каждую из трех кернел-функций, базовых алгебраических операций решателя СЛАУ.

Чтобы эти данные получить, по вашей программе надо расставить соответствующие замеры времени.

### 8.1.3 Отчет

По результатам нужно будет подготовить отчет, в котором привести данные по исследованию производительности решателя. Все вычислительные этапы должны адекватно ускоряться.

Для каждой из базовых операций и для всего алгоритма солвера (этап 3) исследовать зависимость достигаемой производительности от размера системы N, построить графики GFLOPS от N. Измерить OpenMP ускорение для различных N. Оценить максимально достижимую производительность (TBP) с учетом пропускной способности памяти и сравнить с фактической производительностью. Оценить достигаемую скорость передачи данных между процессором и памятью, получаемый процент от пропускной способности.

Сдавать код нужно вместе с инструкцией по компиляции и запуску, желательно сделать make-файл или какой-то скрипт сборки. Также надо, чтобы исполнимый файл печатал хэлп при запуске без аргументов. Отчет в формате PDF нужно загружать в систему отдельным файлом, а не в общем архиве с кодом.

**Требования к отчету:**

Титульный лист, содержащий
1.1 Название курса
1.2 Название задания
1.3 Фамилия, Имя, Отчество (при наличии)
1.4 Номер группы
1.5 Дата подачи

Содержание отчета:

2 Описание задания и программной реализации.
2.1 Краткое описание задания.
2.2 Краткое описание программной реализации – как организованы данные, какие функции реализованы (название, аргументы, назначение). Просьба указывать, как программа запускается – с какими параметрами, с описанием этих параметров.
2.3 Описание опробованных способов оптимизации последовательных вычислений (по желанию).

3 Исследование производительности
3.1 Характеристики вычислительной системы: описание одной или нескольких систем, на которых выполнено исследование (подойдет любой многоядерный процессор), тип процессора, количество ядер, пиковая производительность, пиковая пропускная способность памяти.
По желанию – промерять и на своем десктопе/ноуте, и на кластере.
Описание параметров компиляции под конкретную систему
3.2 Результаты измерений производительности
3.2.1 Последовательная производительность
Для всех этапов, включая генерацию и заполнение (то есть этапы инициализации), продемонстрировать линейный рост стоимости, линейный рост потребления памяти. Solve-часть (этап 3) надо исследовать подробнее. Для каждой из трех базовых операций и

для всего алгоритма солвера исследовать зависимость достигаемой производительности от размера системы N, построить графики GFLOPS от N.

Нескольких N достаточно: N = 10000, 100000, 1000000, 10000000.

Для повышения точности измерений, замеры времени лучше производить, выполняя операции многократно в цикле, чтобы осреднить время измерений. Суммарное время измерений лучше, чтобы получалось хотя бы порядка десятых долей секунды.

Оценить выигрыш от примененной оптимизации (по желанию).

3.2.2. Параллельное ускорение

Для этапов инициализации продемонстрировать наличие параллельного ускорения. Достигаемую производительность для этапов инициализации измерять не надо, считать ТВР не надо.

Для solve-части измерить OpenMP ускорение для различных N для каждой из 3-х базовых операций и для всего алгоритма солвера. Измерить достигаемую производительность. Можно сделать таблички или графики, показывающие GFLOPS от T, где T – число нитей. Можно представить данные по ускорению и производительности единой таблицей, в которой для разных N и T привести ускорение и достигаемую производительность.

4 Анализ полученных результатов

Оценить ТВР и получаемый процент от достижимой производительности для каждой из трех базовых операций, какой процент от пиковой производительности устройства составляет максимальная достигаемая в тесте производительность. Для этапов инициализации ничего оценивать и считать не требуется.

Приложение1: исходный текст программы в файлах C/C++

**Требования к программе:**

1) Программа должна использовать OpenMP для многопоточного распараллеливания и адекватно ускоряться.
2) Солвер должен корректно работать, т.е. показывать быструю сходимость.
3) Программа должна печатать хелп при запуске без аргументов.
4) При ошибках пользовательского ввода программа должна контролируемо завершаться с диагностическим сообщением.
5) Программа не должна валиться по сигналу или зависать (ни при каких параметрах пользовательского ввода).

## 8.2 Задание 2: параллельный решатель СЛАУ для кластерных систем с распределенной памятью

Цели задания:

- освоение организации распределенных вычислений и обмена данными в MPI;
- освоение работы с кластерной вычислительной системой.

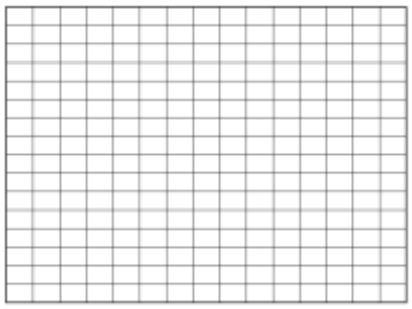
В этом задании необходимо расширить реализацию, сделанную в Задании 1, на вычисления в рамках параллельной модели с распределенной памятью. Это потребует существенной доработки. Для организации работы параллельных процессов и обмена данными используется MPI.

Распределенная реализация подразумевает, что расчетная область задачи разделяется на части – подобласти. Эти подобласти распределяются между параллельными процессами. Каждый процесс работает со своей частью расчетной области и обменивается с соседними процессами только информацией по интерфейсным ячейкам, граничащим с ячейками других подобластей.

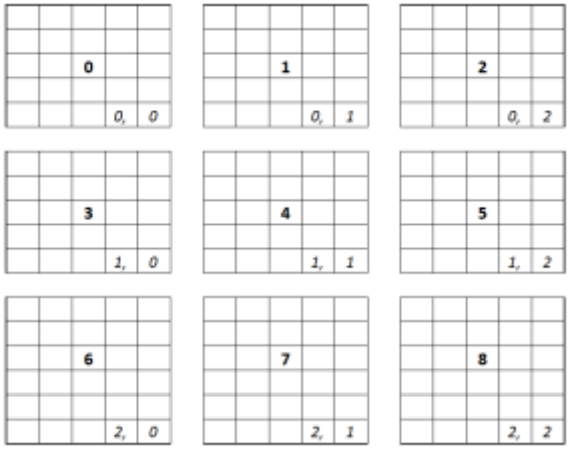
### 8.2.1 Введение

Рассмотрим на очень простом примере – на примере декартовой решетки. Возьмем и разделим нашу модельную расчетную область, представленную решеткой из N = Nx×Ny ячеек, на P = Px×Py подобластей путем декомпозиции по двум направлениям (осям координат).

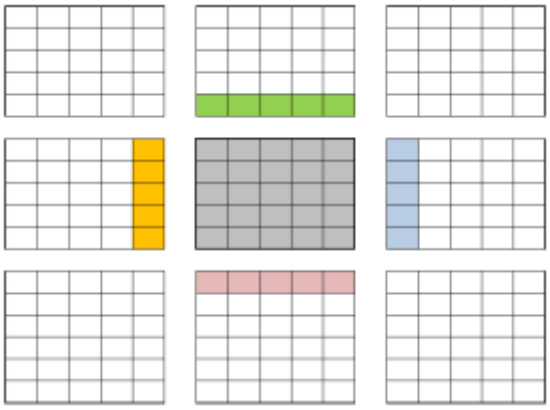
Каждый процесс работает только **со своей частью** расчетной области и с ее гало (т.е. приграничными ячейками из соседних подобластей). Рассмотрим простейший пример – делим на части двухмерную решетку, распределяем ее клеточки-ячейки. Вот есть какая-то решетка:



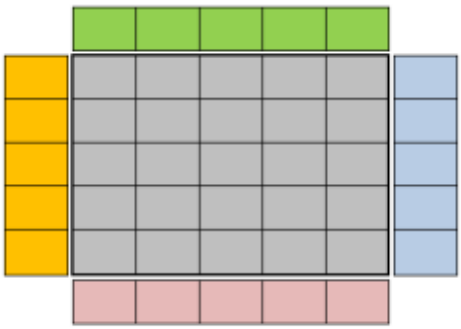
Поделим ее, например, на 9 частей 3×3, части нумеруем по тому же принципу, что и клетки, например, сверху вниз, слева направо (в каждой части по центру – номер процесса и курсивом в углу его позиция в решетке процессов px, py):



Рассмотрим один из процессов, например, который по середине – №4. Ячейки его подобласти отмечены серым. Другими цветами отмечены гало ячейки для этой локальной подобласти, принадлежащие чужим подобластям.



Так выглядит локальная часть, с которой имеет дело 4-й процесс:



О существовании других ячеек этот процесс не знает. В общем это был просто пример, чтобы понять принцип.

### 8.2.2 Постановка задачи

Делаем Задание 2 на основе Задания 1 (раздел 8.1). Работа программы, которую будем делать, теперь состоит из 5 этапов. Добавился еще этап построения обмена данными:

1. **Generate** – генерация графа/портрета по тестовой сетке;
2. **Fill** – заполнение матрицы по заданному портрету;
3. **Com** – построение схемы обменов;
4. **Solve** – решение СЛАУ с полученной матрицей;
5. **Report** – проверка корректности программы и выдача измерений.

**Этап 1: Generate – генерация портрета на основе тестовой сетки**

Имеется решетка состоит из **Ng** = Nx×Ny клеточек (g – global). На вход поступают два числа Px и Py, которые указывают, на сколько частей мы делим решетку по соответствующим направлениям. Общее число MPI процессов должно быть P=Px×Py.

Для начала по заданным Nx, Ny и Px, Py, зная ID процесса, определим подобласть данного процесса. Найдем диапазоны индексов подобласти по двум осям, которые мы делим: ib ≤ i < ie, jb ≤ j < je (b – begin, e – end). Диапазоны следуют из позиции данного процесса в решетке процессов, т.е. его порядковых номеров по двум осям. Поделить надо так, чтобы

дисбаланс по каждому направлению был не более 1 шт. (т.е. если Nx не делится нацело на Px, то остаток от деления надо раскидать по одной ячейке по процессам).

Для формирования подобласти процесса еще прибавим к своим клеткам один уровень гало – прилегающие клетки из соседних подобластей.

Модифицируем этап генерации, чтобы сетка строилась только для локальной подобласти (свои + 1 уровень гало). Там можно заменить диапазоны у двух циклов по двум осям на соответствующие границы, которые $ib \le i < ie$, $jb \le j < je$.

В зависимости от варианта мы распределяем узлы или элементы. В варианте по узлам – просто распределяем узлы исходной решетки. В варианте с элементами распределяем клетки исходной решетки, а элементам ставим владельца клетки (если клетку поделили на треугольники, то треугольники принадлежат тому же, кому и клетка).

Далее, чтобы не путаться в этих вариантах, вершинах и элементах, будем говорить в терминах графа. То, что распределяем – соответствует вершинам этого графа (которые сопоставлены узлам или элементам сетки, в зависимости от варианта).

Для всех вершин, попавших в подобласть (свои + гало) надо знать их номера в глобальном графе, то есть во всей сетке.

Отображение из номера вершины в локальной подобласти в номер вершины во всей глобальной области обозначим L2G (Local To Global). Обратное отображение – G2L.

**Упорядочим вершины** так, чтобы сначала были собственные, потом шли гало. Пусть число вершин в локальной области данного процесса будет **N**, из них собственных вершин – **No** (o – own), во всей глобальной области вершин **Ng** (g – global). Собственные вершины будут с номерами от 0 до No-1, гало вершины с номерами от No до N-1.

На выходе с генерации мы должны получить портрет, который хранит связи только для собственных вершин. IA у нас размера No+1, то есть мы будем распределять матрицу построчно и хранить No наших строк. В эти связи, естественно, входят гало вершины, то есть JA содержит все те же столбцы, что в исходной версии (только в локальной нумерации). Если на генерации JA делался в глобальных номерах, их надо перегнать в локальные, используя G2L.

Заполняем массив Part размера N, определяющий для каждой вершины, кому она принадлежит, т.е. вписываем каждой вершине номер владельца.

Как все это заполнять, все эти нумерации, описано в разделах 4.3.2, 4.3.3. Итак…

**Входные данные:**

**Nx, Ny –** число клеток во всей глобальной решетке по вертикали и горизонтали;

**K1, K2 –** параметры для количества треугольных и четырехугольных элементов;

**Px, Py –** параметры декомпозиция по двум осям;

Программа запускается через MPI с числом процессов P = Px×Py.

**Выходные данные:**

**N –** число вершин в локальной подобласти (собственные и гало);

**No –** число собственных вершин в локальной подобласти;

**IA, JA –** локальный портрет разреженной матрицы смежности графа, дополненный главной диагональю (в формате CSR), то есть только собственные строки для данной подобласти (но эти строки, естественно, включают столбцы для гало от соседей!);

**Part –** массив с номерами владельцев по каждой вершине в локальном графе;

**L2G –** массив с номерами вершин локального графа в глобальном графе, то есть отображение из локальной нумерации в глобальную.

**Этап 2: Fill – построение СЛАУ по заданному портрету матрицы**

**Входные данные:**

**N –** число вершин в локальной подобласти (собственные и гало);
**No –** число собственных вершин в локальной подобласти;
**IA, JA –** локальный портрет матрицы;
**L2G –** отображение из локальной нумерации в глобальную.

**Выходные данные:**

**A –** массив ненулевых коэффициентов матрицы (размера IA[No]), только свои строки;
**b –** вектор правой части (размера N).

Заполняем матрицу и вектора как в Задании 1, но индексы берем **глобальные**, иначе не будет совпадать с последовательной версией. Не cos(i*j…), например, а cos(L2G[i]*L2G[j]…).

**Этап 3: Com – построение схемы обмена данными**

**Входные данные:**

**N –** число вершин в локальной подобласти (собственные и гало);
**No –** число собственных вершин в локальной подобласти;
**IA, JA –** локальный портрет матрицы;
**Part –** массив с номерами владельцев по каждой вершине в локальном графе;
**L2G –** отображение из локальной нумерации в глобальную;

**Выходные данные:**

**Com** – схема обменов, то есть списки вершин на отправку всем соседям и на прием от всех соседей. Как её делать и какие контейнеры использовать – на ваше усмотрение.

Построение схемы обмена и организация обмена данными подробно описаны в разделе 4.4 (см. 4.4.1, 4.4.2). Просто делаем, как там написано.

**Этап 4: Solve – решение СЛАУ итерационным методом**

**Входные данные:**

**N –** число вершин в локальной подобласти (собственные и гало);
**No –** число собственных вершин в локальной подобласти;
**IA, JA –** локальный портрет матрицы;
**Com** – схема обменов;
**A –** массив ненулевых коэффициентов матрицы (размера IA[No]);
**b –** вектор правой части (размера N).
***eps*** – критерий остановки ($\varepsilon$), которым определяется точность решения;
***maxit*** – максимальное число итераций.

**Выходные данные:**

**x** – вектор решения (размера N);
**n** – количество выполненных итераций;
***res*** – L2 норма невязки (невязка – это вектор $\boldsymbol{r = Ax - b}$).

Делаем на основе Задания 1 (раздел 8.1). Добавляем обмены по схеме Com для обновления гало для SpMV (которое делается только для своих позиций)

Добавляем Allreduce обмен для сбора глобальной суммы в скалярном произведении dot (которое делается только для своих).

Axpy остается без изменений.

Проверяем, что солвер работает идентично последовательно версии, что все контрольные величины совпадают.

**Этап 4: Report – проверка корректности программы и выдача измерений**

Делаем полностью аналогично Заданию 1 (раздел 8.1).

### 8.2.3 Отчет

Отчет делаем по аналогии с Заданием 1 (раздел 8.1). Только исследование параллельной эффективности делаем на кластере для разного числа процессов, а не потоков.

Делаем тестовые запуски на разном числе процессов, для разных размеров сетки, проверяем, что все работает, что асимптотика по времени и потреблению памяти у всего кода линейная. Тестируем на системе с общей памятью, тестируем комбинированный режим распараллеливания – MPI+OpenMP.