Министерство образования и науки Российской Федерации
Московский физико-технический институт (государственный университет)
Факультет управления и прикладной математики
Кафедра Информатики

Петров Михаил Юрьевич

Распараллеливание алгоритмов решения задач сейсмики на многопроцессорных машинах с общей памятью

010900 — Прикладные математика и физика

Выпускная квалификационная работа бакалавра

Научный руководитель:
к. ф.-м. н. Хохлов Николай Игоревич

Москва
2014

Содержание

1 Введение **3**

2 Обзор **3**
2.1 Типы параллельных машин . . . 3
2.2 Типы архитектура памяти . . . 4
2.2.1 Разделяемая память . . . 4
2.2.2 Распределённая память . . . 5
2.2.3 Распределённо-разделяемая память . . . 5
2.3 Архитектура кэша . . . 6
2.4 Технологии . . . 7

3 Алгоритмы оптимизации **8**
3.1 Loop Tiling . . . 8
3.2 SPMD . . . 8
3.3 Цикл for . . . 9
3.4 Функция barrier . . . 9

4 Некоторые задачи **9**
4.1 Пересчёт узлов сетки . . . 9
4.2 Graph Coloring (раскраска графа) . . . 10
4.3 bsp деревья . . . 10

5 Задача **11**
5.1 Постановка задачи . . . 11
5.2 Решение задачи . . . 12
5.2.1 Характеристики тестовой эвм . . . 12
5.2.2 Последовательная версия . . . 12
5.2.3 Параллельная версия . . . 13
5.2.4 Параллельная кэш-версия . . . 15
5.2.5 Параллельная кэш-версия плюс аллоцирование данных . . . 15
5.2.6 Affinity и noaffinity . . . 16
5.2.7 Финальные тесты . . . 17

6 Заключение **18**

1 Введение

В настоящие время стали очень распространены параллельные компьютеры или ЭВМ. Это связано с тем, что экономически намного выгоднее делать много ядер с низкой частотой, чем одно ядро с большой частотой. В связи с этим фактом возникло новое направление – параллельные вычисления. Они применяются в таких областях как data mining, графика, медицинская диагностика, физическое и финансовое моделирование. Все эти задачи объединяет одна общая деталь – огромный объём обрабатываемых данных. Эта деталь очень часто позволяет распараллелить обработку этих данных.

Ускорение S_p – это отношение времени выполнения программы на одном ядре T_1 к времени выполнения этой же программы T_p на p ядрах:

$$S_p = \frac{T_1}{T_p} \tag{1}$$

Теоретическое ускорение не может превышать количества ядер. Если ускорение получается выше линейного, то это значит, что последовательная версия программы написана неэффективно. Для оценки эффективности вводят параметр эффективность распараллеливания. Эффективность распараллеливания:

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \tag{2}$$

Также для оценки теоретически возможного ускорения используют закон Амдала. Закон Амдала иллюстрирует ограничение роста производительности вычислительной системы с увеличением количества вычислителей. Пусть необходимо решить некоторую вычислительную задачу. Предположим, что алгоритм её решения таков, что доля α от общего объёма вычислений может быть получена только последовательными расчётами, а доля $1-\alpha$ может быть распараллелена. Часть алгоритма может быть распараллелена, если вычисления необходимые в данной части могут быть равномерно распределенны между ядрами и могут считаться параллельно. Тогда ускорение, которое может быть получено на вычислительной системе из p процессоров, по сравнению с однопроцессорным решением не будет превышать величины:

$$S_p = \frac{1}{\alpha + \frac{1-\alpha}{p}} \tag{3}$$

На практике всё не совсем так. По ряду технических причин не всегда данный закон выполняется. Обзору причин и решений данной проблемы посвящена работа.

2 Обзор

В данном разделе будут описаны основные технологии, связанные с параллельными вычислениями.

2.1 Типы параллельных машин

Для классификации параллельных машин используется таксономия Флинна. Распределение по классам происходит по наличию параллелизма в потоках команд и данных. Каждый из двух признаков может принимать ровно два значение: single (непараллельный), multiple (параллельный). Поэтому всё разнообразие архитектур ЭВМ в таксономии Флинна сводится к четырём классам:

1. Single instruction и single data. Этот тип компьютеров непараллельный. Примеры таких компьютеров: IBM360, PDP1.

2. Single instruction and multiple data (SIMD). Это параллельный тип компьютеров. Подходит только для специализированных проблем, для которых характерна высокая степень регулярности, например обработка изображений. Примеры таких компьютеров: IDBM 9000, CrayX-MP.

3. Multiply instruction и single data (MISD). Это параллельный тип компьютеров. Компьютеры данного типа не распространены.

4. Multiply instruction multiply data (MIMD). Самый распространённый в наше время тип параллельных компьютеров. Большинство современных компьютеров относятся к компьютерам такого типа. Очень часто эта технология включает технологию SIMD как свою подкомпоненту.

2.2 Типы архитектура памяти

2.2.1 Разделяемая память

Первый тип архитектуры называется *Разделяемая память*. Основным признаком данной архитектуры является то, что все процессоры могут работать независимо, но имеют доступ к одной и той же памяти. У этой архитектуры выделяют два подтипа: UMA и NUMA. Разделение на два данных типа основано на времени доступа процессоров к памяти. Рассмотрим каждый из них поподробнее:

- Uniform Memory Access (UMA): Очень часто представлен SMP (Simmetric multiprocessor) машинами. Все процессоры идентичны и имеют одинаковое время доступа ко всем участкам разделяемой памяти. Стандартный вид данной архитектуры показан на рис. 1.

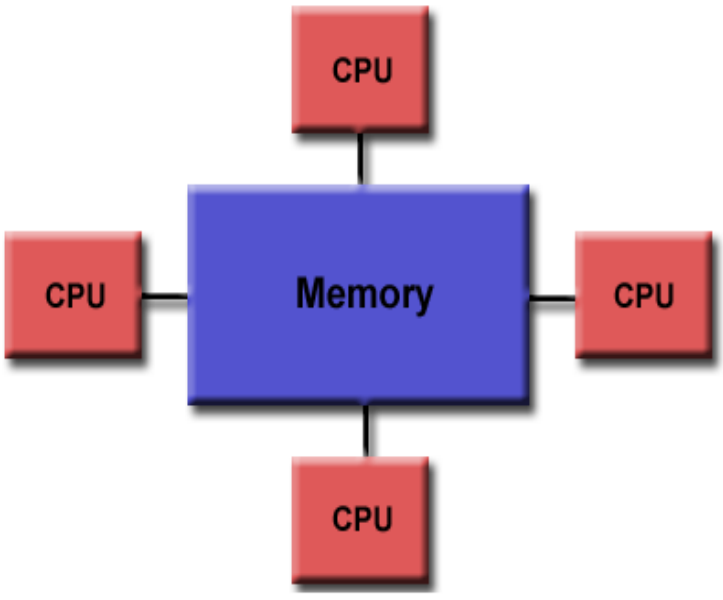

Рис. 1: Архитектура UMA

- Non-Uniform Memory Access: Разделяемая память разбита на области, каждая область принадлежит соответствующему ядру или группе ядер. Для данного

ядра или группы ядер эта область памяти называется «своей». Области памяти, которые не пренадлежат данному ядру или группе, называются «чужими» для данной группы. Все процессоры имеют доступ к разделяемой памяти, но время доступа к «чужой» памяти ниже чем к «своей». Стандартный вид данной архитектуры показан на рис. 2.

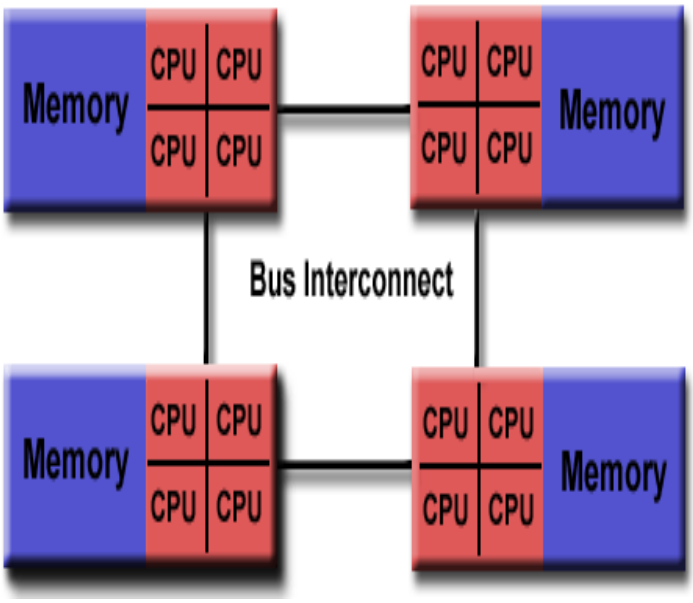

Рис. 2: Архитектура NUMA

Архитектура UMA имеет большую производительность в силу одинаково быстрого доступа к памяти, но она очень сильно ограниченна в количестве процессоров (в среднем не больше 4). Архитектура NUMA может поддерживать несколько десятков процессоров, но у неё возникает проблема *аллоцирования данных* (data allocated).

Проблема *аллоцирования данных* возникает вследствие неравнозначности доступа к областям памяти. Время доступа не к своей области памяти может занимать на порядок больше времени. Поэтому, если процессоры работают не со своей областью памяти, это может привести к сильному падению производительности. Для решения данной проблемы разработчики пытаются локализовать данные, с которыми работает данный процессор, в «своей» области памяти.

2.2.2 Распределённая память

Второй тип архитектуры называется "Распределённая память". Чаще всего это группа процессоров объединённых с помощью интернета. Каждый процессор имеет свою память. Все процессоры работают независимо, а именно изменения в памяти одного процессора не отображается в памяти у другого. Способ обмена информацией между процессорами полностью лежит на плечах программиста. Главным преимуществом данной архитектуры перед разделяемой памятью является лёгкость масштабирования. Для добавления ещё одного процессора требуется подключить ещё один компьютер со свободным процессором через интернет. Поэтому количество процессоров в данной архитектуре можно считать неограниченным.

2.2.3 Распределённо-разделяемая память

Третий тип называется "Распределённо-разделяемая память". Фактически это вторая архитектура, только вместо каждого процессора вставлена группа процессоров

с архитектурой Разделяемая память. Преимущества и недостатки этой архитектуры является объединением преимуществ и недостатков двух предыдущих архитектур.

2.3 Архитектура кэша

На рис. 3 показана схема устройства стандартной трёхуровневой кэш-памяти процессора. L1i – кэш-память для инструкции, L1 (или L1d) – кэш-память данных. На более низких уровнях кэша обычно нет явного разделения кэш-памяти на инструкции и данные.

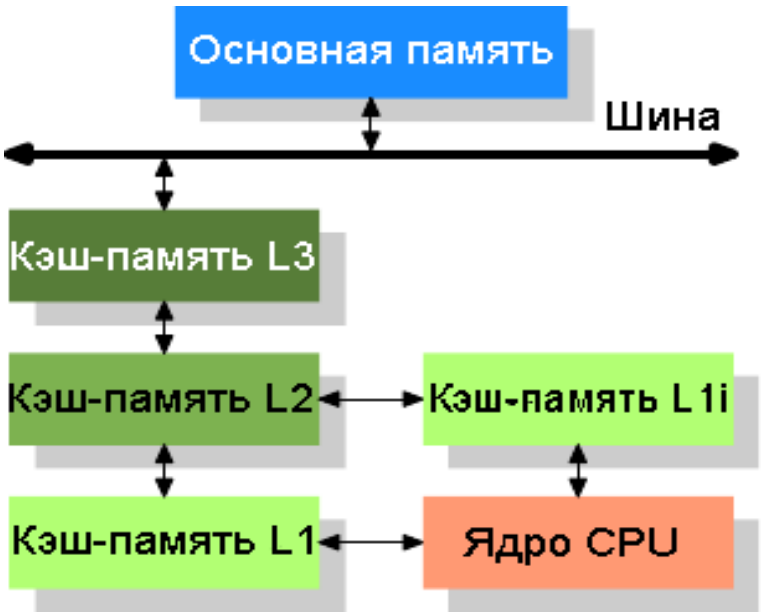

Рис. 3: устройство кэш-памяти

Стоит отметить, что обмен данными в системе производится порциями: кэш-строками или кэш-линиями. При загрузки переменной, загружается целая кэш-строка, содержащая данную переменную.

Чтобы процессор мог работать с переменной, она должна находиться в памяти L1 кэш-уровня. Размер кэша L1 очень ограничен, поэтому не всегда возможно загрузить все необходимые кэш-строки в данный кэш-уровень. Когда процессор не находит интересующую его переменную в L1, он начинает искать в L2, затем в L3, и в конце ищет уже в основной памяти. После нахождения переменной её необходимо загрузить в L1-кэш, а для этого очень часто необходимо произвести выгрузку данных из L1 (чтобы освободить место). Какая кэш-строка будет вытеснена определяет алгоритм вытеснения.

Событие ненахождения переменной в соответствующем кэше называется кэш-промахом. Для сравнения доступ к L1 памяти в среднем равняется 4 тактам процессора, к L2 10, а к L3 ещё больше. Если на каждом шаге будет происходить кэш-промахи первого уровня, то производительность может упасть более чем в два раза.

Проблема кэш-промахов возникает при работе с очень большим объёмом данных (когда данные не влазят в кэш L1). Поэтому проблемы кэш-промахов очень важна для задач вычислительной математики.

2.4 Технологии

Для начала разберём технологии для работы с "Разделяемой памятью". Для этого нам надо понять, что такое треды. Технически, треды это независимый поток инструкций. Треды можно сравнить с процессами в операционной системе, но треды всё-таки зависят от процесса создавшего их и имеют с ним общие данные (глобальные переменные и т.п.). Это (некоторые данные общие с parent process) позволяет создавать новый тред намного быстрее, чем новый процесс. К примеру создание нового треда на процессоре Intel 2.6GHz Xeon E5-2670 (16cores/node) занимает 0.9 мсек, а создание нового процесса 8.1 мсек.

Как мы можем заметить создание нового треда намного экономичнее по ресурсам и быстрее по времени, чем создание нового процесса. И для удобной работы с тредами был разработан стандарт, который называется Pthread. Плюсы уже были рассказаны выше, а главный минус этой технологии заключается в том, что синхронизация доступа к глобальным переменным лежит полностью на программисте.

Следующий технология или стандарт называется openmp. Она так же построена на тредах, но использует директивы компилятора, а не функции как это было с Pthread. Фактически это более удобная обёртка для Phread, которая включает в себя часто используемые параллельные конструкции. Главное преимущества данной технологии в том, что она предоставляет множество часто используемых параллельных конструкций и удобные методы синхронизации. Главных недостатка всего два. Первый это необходимость синхронизации доступа к памяти (конечно тут больше методов синхронизаций чем в Pthread, но всё равно надо следить за данными). Второй это скрытность работы с тредами (мы не знаем насколько оптимально написана та или иная функция).

Так же есть пара технологий от Intel. Первая из них Intel Silk Plus. Технология использует макросы вместо функций. Огромный плюс данной технологии это простота. В ней всего 3 ключевых слова. И фактически эти слова являются подсказками процессору, о дальнейших действиях. Минусом является синхронизация.

Ещё одна технология от Intel это Intel TBB. Технология использует шаблоны. Фактически это позволяет программисту абстрагироваться от проблем с синхронизацией и аналогичных проблем появляющихся при использование обычных тредов. Так же там реализовано множество datasafe контейнеров. Плюс обещается эффективное использование кэшей. Ярких минусов не имеет, кроме, быть может, портируемости.

Теперь поговорим о технологиях для работы с “Disturbed memory”. Главное отличие от “Shared memory”в том, что процессоры не имеют доступа к информации друг друга. И самый известный стандарт технологии, решающий данную проблему, есть MPI (message passing interface). Это стандарт обмена данными между процессорами. Технология использует функции.

Следует упомянуть несколько технологий другого типа.

Технология CUDA. Это программно-аппаратная архитектура параллельных вычислений использующая графические процессоры фирмы NVIDIA(по простому видиокарты). Написание программ выполняется на CUDA SDK языке, который является языком си с некоторыми ограничениями. В архитектуре CUDA используется модель памяти grid (сетка), кластерное моделирование потоков и SIMD-инструкции.

Технология векторизации. Для понимания этой технологии потребуется небольшое объяснение работы арифметики в процессоре. Чтобы сложить два числа, нужно записать одно число в регистр 1, а второе в регистр 2. И дальше процессор сложит два регистра. Надо заметить что размер регистра в современных процессорах больше размера int в несколько раз. И к примеру если мы хотим сложить три int с тремя другими int попарно, то мы можем записать первые три int в первый регистр и вто-

рые три int во второй регистр. Далее процессор сложит два регистра как раньше, то есть сложит три числа за одну операцию сложения регистров. И таким образом достигается распараллеливание. Процессор, в данном случае, можно рассматривать как архитектуру SIMD. Данная технология встроена на уровни компилятора icc, но также можно помогать компилятору находить места, где возможна векторизация.

3 Алгоритмы оптимизации

3.1 Loop Tiling

Один из алгоритмов оптимизации работы с кэшем (уменьшение кэш-промахов) называется Loop tiling. Данный метод оптимизации состоит в разбиении пространства итерирования исходного цикла (которое может проводиться по нескольким переменным) на небольшие блоки меньшего размера, что позволяет хранить используемые в этих небольших блоках данные в кэше полностью для их неоднократного использования в процессе выполнения блока.Рассмотрим данный алгоритм на примере перемножения матриц. Обычный код перемножения матриц выглядит следующим образом:

```
for (i = 0; i < N; i++)
    for (k = 0; k < N; k++)
        for (j = 0; j < N; j++) {
            z[i, j] = z[i, j] + x[i, k] * y[k, j]
        }
```

После Loop tiling:

```
for (k2 = 0; k2 < N; i += B)
    for (j2 = 0; j2 < N; j += B)
        for (i = 0; i < N; i++)
            for (k1 = k2; k1 < min(k2 + B, N); k1++)
                for (j1 = j2; k2 < min(j2 + B, N); j1++) {
                    z[i, j1] = z[i, j1] + x[i, k1] * y[k1, j1];
                }
```

При правильном выборе размера блоков, можно свести к минимуму количество кэш-промахов при работе с матрицей Y.

В работе [3] описанна модификация данного алгоритма. Главной причиной побудивших авторов написать эту модификацию является странность в поведении кэша первого уровня. Даже если размер блока равен или меньше размера кэша $L1$, то всё равно происходят промахи. Во избежании этого, они копируют вычисляемый в данный момент блок в новую область памяти. Так же для большего улучшения производительности они используют вторичный Tiling. Что позволяет улучшить работу для других уровней кэша.

Так же существует модификации данного алгоритма [4].

3.2 SPMD

Один из алгоритмов решения проблемы data allocated заключается в написании кода в SPMD стиле. Главная идея данного стиля написания кода состоит в распараллеливании данных по потокам. Благодаря этому на процессорах архитектуры Numa и ccNuma достигается огромный прирост производительности по сравнению с обычным стилем написания кода программ.

Главной проблемой данного стиля написания является общие данные(данные с которыми должно работать более одного процесса). В статье [5] представлены способы перевода кода в SPMD стиль и предложены способы решения проблемы общих переменных.

3.3 Цикл for

Ещё один алгоритм оптимизации связан с циклом for в openmp реализации. В стандартной реализации for можно вызвать с двумя параметрами $static\{chunksize\}$ и $dynamic\{chunksize\}$. Первый параметр говорит компилятору раздавать итерации в количестве $chunksize$ по очереди между процессорами. Это позволяет позволяет аллоцировать данные. Dynamic параметр раздаёт итерации свободным в данный момент процессам, что важно для циклов в которых итерации неравнозначны по времени выполнения. В работе [6] представлен дополнительный параметр adaptive. Он совмещает в себе плюсы стандартных параметров.

3.4 Функция barrier

Функция barrier вызывается в параллельных частях программы и означает следующие: выполнение треда продолжится как только все треды достигнут данного барьера. Данная функция очень важна, так как с помощью неё происходит подавляющие большинство синхронизаций. Она неявно вызывается в конце каждой паралельной секции. Однако если последующий код программы никак не зависит от выполнения данного параллельного региона можно включить дериктиву NOWAIT. Это говорит компилятору убрать barrier в конце соответствующего региона.

Так как функция barrier фактически заставляет простаивать треды, которые достигли барьера раньше, надо делать треды сбалансированными. Треды сбалансированны, если их времена выполнения примерно равны. Это позволяет избавиться от излишних простоев процессоров. Также можно оптимизировать саму функцию барьера. В работе [10] описаны несколько видов барьеров и проведён их анализ.

4 Некоторые задачи

4.1 Пересчёт узлов сетки

Openmp может применяться для решения уравнения Навье-Стокса. Как и большинство уравнений матфизики, решение уравнения состоит в применении разностной схемы к сетке. Одна из проблем возникающая при распараллеливании является data racing(сотояние гонки).

Data racing возникает в случае когда два процесса пытаются изменить одну общую переменную или же когда один процесс пишет, а другой читает данную переменную. В этом случае поведение памяти не определено, что приводит к непредсказуемым результатам данных операций.

Возникновение data racing связано с разностной схемой, используемой при пересчёте значений сетки. При подсчёте следующего значения узла сетки используется несколько предыдущих значений сетки, расположенных рядом. В следствии этого попытка пересчитать два соседних узла одновременно может привести к data racing.

В работе [7] предлагается разделять сетку на области, такие что любой элемент в пределах одной области стоит только с одной стороны разностной схем (только слева или только справа) используемых при пересчёте элементов данной группы.

Это позволяет избежать всех data racing, так как элемент либо только читается в данной группе, либо только пересчитывается.

Данное разбиение можно реализовать следующим способом:

- построить граф с вершинами равными узлами сетки;
- соединить вершины i и j только в том случае если существует формула, где при пересчёте одной из них используется другая;
- разбить граф на подграфы, вершины в каждом из которых не имеют рёбер между собой

Задача разбиения графа таким образом называется задача раскраски графа. Это задача очень распространена и имеет множество алгоритмов решения, что помогает в случае нерегулярных сеток, когда зависимости между вершинами могут быть достаточно сложными (простым вглядыванием не разбить).

После разбиения вершин на такие области, каждую область можно распараллеливать не опасаясь data racing. Но надо не забывать, что разные области надо считать последовательно.

4.2 Graph Coloring (раскраска графа)

Openmp может применяться для распараллеливания задачи раскраски графа. Задача: Необходимо раскрасить вершины графа в разные цвета, таким образом чтобы никакие две соседние(соединённые ребром) вершины не были одного цвета. В работе [8] предложен следующий алгоритм.

- разбиваем граф на p равных(по количеству вершин) частей;
- раскрашиваем в каждом подграфе вершины любым алгоритмом; (делается паралельно)
- для каждой вершины из каждого подграфа, просматриваются все соседи. Если цвет совпадает то минимальная из двух вершин помечается как недействительная и помещается в специальный массив A; (делается паралельно)
- В массиве A вершины раскрашиваются в правильные цвета.

Также в данной работе предложен способ уменьшения количества цветов.

4.3 bsp деревья

Также openmp применяется для распараллеливания bsp деревьев. Сбалансированные деревья можно распараллеливать деревья разбивая дерево на p(количество процессов) ветвей. Для несбалансированных деревьев данный способ неприемлем, так как некоторым процессам может достаться больше работы чем другим, что приведёт к простаиванию процессов закончивших раньше свою работу.

Для несбалансированных деревьев нужен несколько другой подход. В работе [9] предлагается несколько алгоритмов. Как показывают их тесты, в среднем, лучше всего справляется с работой встроенная в openmp 3.0 конструкция task.

Конструкция task создаёт отдельный поток и вызывает код, написанный внутри неё. Функция в которой была вызвана конструкция task продолжает свою работу без ожидания завершения своих tasks, или до деректривы taskwait. Также внутри себя,

task позволяет вызывать другие tasks. Также, на протяжении своей жизни task может мигрировать между процессами. Всё ранее перечисленное делает их очень удобными для обработки деревьев. Фактически концепт tasks позволяет легко использовать его для распараллеливания рекурсивных процедур, обычным заключение вызова данной процедуры в task конструкцию.

Также в данной работе были проведены тесты с ограничение максимального количества создаваемых тредов. Рассмотрим пример: пусть у нас есть только одно ядро, и 2 тредов. Политика процессора говорит, что никакой тред не должен ждать своей очереди слишком долго. Поэтому сначала процессор работает с 1 тредом, затем со 2, потом опять с 1 и т.д. Главная проблема в этом примере это подгрузка всех необходимых ресурсов во время переключений между тредами. Поэтому чтобы ограничить количества данных перегрузок, авторы решили ограничить число тредов.

Как показали тесты влияние ограничения максимального количества тредов минимально.

5 Задача

В данной работе производилось распараллеливание решения уравнения линейной динамической теории упругости.

5.1 Постановка задачи

Дана система уравнений:

$$\begin{cases} \rho\dot{\upsilon} = \bigtriangledown T \\ \dot{T} = \lambda(\bigtriangledown \upsilon)I + \mu(\bigtriangledown \otimes \upsilon + \upsilon \otimes \bigtriangledown), \end{cases} \tag{4}$$

где ρ – плотность среды, υ – вектор скорости смещения, T – тензор напряжений Коши, $\bigtriangledown$ – градиент по пространственным координатам, λ, μ – упругие постоянные Ляме, I – единичный тензор, $\otimes$ - оператор векторного произведения.

После преобразований, описанных в работе [11], данное уравнение приводится к стандартному виду гиперболических уравнений:

$$\frac{\partial u}{\partial t} = \Sigma_j A_j \frac{\partial u}{\partial \xi_j}, \tag{5}$$

где $u = (\upsilon_1, \upsilon_2, \upsilon_3, \delta_{11}, \delta_{12}, \delta_{13}, \delta_{22}, \delta_{23}, \delta_{33})^T$, а δ_{ij} элементы тензора напряжений T, ξ_j– ортогональная система координат, A_j матрицы 9 на 9. Для параллелепипедной сетки их вид можно посмотреть в работе [11].

Так как система является гиперболической, то каждая из матриц A_j обладает полным набором собственных векторов, поэтому систему можно переписать в следующем виде:

$$\frac{\partial u}{\partial t} = \Sigma_j \Omega_j^{-1} \Lambda \Omega_j \frac{\partial u}{\partial \xi_j}, \tag{6}$$

где матрица Ω_j – матрица из собственных векторов, а Λ – диагональная матрица, на диагонале которой стоят собственные значения матрицы A_j.

Поэтому если мы сделаем замену $\nu = \Omega u$, то каждая из систем распадётся на 9 независимых скалярных уравнений переноса вида:

$$\frac{\partial \nu}{\partial t} + \Lambda \frac{\partial \nu}{\partial x} = 0 \tag{7}$$

Каждое из которых можно решать одномерной разностной схемой.

После того как все компоненты ν пересчитаны, выполняем обратное преобразование координат $u^{n+1} = \Omega^{-1}\nu^{n+1}$.

Задача состояла в распараллеливании решения данной системы уравнений.

5.2 Решение задачи

5.2.1 Характеристики тестовой эвм

ОС: Linux, 64 bit.
Процессор: AMD Opteron 8431, 6 ядер.
Оперативной памяти: 256 Gb.
Количество процессоров: 8.
Частота процессора: 2.4 ГГц.
Суммарное количество ядер: 48.

5.2.2 Последовательная версия

Сначала был написан последовательная программа. Кратко её код выглядит следующим образом:

```
void main(..)
{
initGrid(..);

for(int i = 0; i < timeSteps; i++)
{
stepX(..);
stepY(..);
}
}
```

Две точки здесь и далее обозначают некоторые параметры, которые для понимания алгоритма не представляют особой важности. Шаги по X и Y выглядят в данной версии так:

```
void stepX(..)
{
for(int j = 0; j < YSize; j++)
for(int i = 0; i < XSize; i++)
{
Step(Gr[i-2][j], Gr[i-1][j], Gr[i][j], Gr[i+1][j], Gr[i+2][j], ..);
}
}

void stepY(..)
{
for(int i = 0; i < XSize; i++)
for(int j = 0; j < YSize; j++)
{
Step(Gr[i][j-2], Gr[i][j-1], Gr[i][j], Gr[i][j+1], Gr[i][j+2], ..);
```

```
}
}
```

5.2.3 Параллельная версия

Далее была написана простая параллельная версия:

```
void stepX(..)
{
#pragma omp parallel for
for(int j = 0; j < YSize; j++) {...}
}

void stepY(..)
{
#pragma omp parallel
for(int i = 0; i < XSize; i++) {...}
}
```

Прежде чем привести результаты тестов, давайте посчитаем идеальное теоретическое ускорение на n ядрах. Инициализация начальных данных происходит за время T_1, которое всегда одно и тоже (конечно оно зависит от размеров сетки, но очень слабо) и не параллелится. Время работы всех шагов по времени равно суммарному времени выполнения шага по X и Y, умноженному на количество шагов по времени:$T_2 = (ty + tx) \cdot timeSteps$. Так же заметим, что шаги по X и Y распараллеливаются полностью. Поэтому ускорение на p ядрах должно равняться:

$$S = \frac{T_1 + (ty + tx) * timeSteps}{T_1 + \frac{(ty+tx)*timeSteps}{p}} \xrightarrow{timeSteps \to \infty} p \tag{8}$$

Поэтому "идеальное"ускорение при достаточно большом количестве шагов по времени равняется количеству ядер.

Тесты проводились отдельно для шага X и шага Y. Результаты тестов показаны на рис. 4.

Как видно из графика, ускорение для X соответствует теоретическому, а Y сильно меньше. Это связано с организацией цикла в Y шаге. Существует две основные проблемы:

1. Кэш промахи. Чтобы было понятнее посмотрим на рис. 5. Пусть мы уже пересчитали все Y состовляющие до i, j включительно и пусть сетка у нас достаточно большого размера (Y размер сетки умножить на размер кэш линии больше кэша), тогда в данный момент загруженные кэш-линии выделенные красным. Следующим шагом мы пересчитываем элемент $i, j + 1$ и для этого нам нужна кэш-линия выделенная синим, поэтому происходит её загрузка из памяти, то есть кэш промах. Заметим, что если вместо элемента $i, j + 1$ пересчитывать $i + 1, j$ элемент, то кэш-промаха не случится. На этой идее основана параллельная кэш-версия. Так же именно поэтому у нас в пересчёте X не случается кэш-промахов.

2. Совместное использование одной кэш-линии. Посмотрим опять на рис. 5. Пусть первые 4 Y (вертикальные) линии считает первый процессор, а следующие 3 второй. Пусть первый посчитал i,j элемент сетки, а второй хочет посчитать

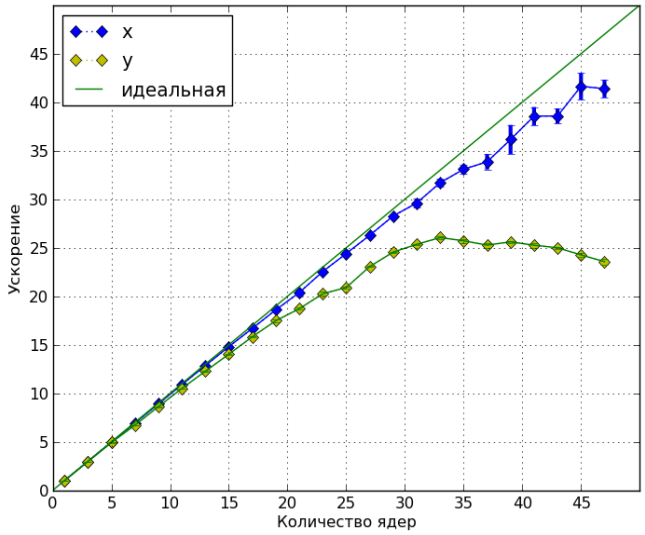

Рис. 4: Y и X ускорения

$i+3, j$ элемент. Тогда он считывает нужный элемент, пересчитывает и записывает его в свою хэш-линию. Но другой процесс также использует её, поэтому кэш линия первого процесса помечается как грязная. При следующей попытке первого процессора обратиться к ней произойдёт вынужденное обновление, а именно загрузка чистой версии данной кэш-линии из памяти. Это часто бывает хуже кэш-промаха, так как кэш-промах чаще всего 1 уровня (недостающая кэш-линия загружается из кэша 2 уровня), а в данном случае необходимо выполнить загрузку из общего кэша двух процессоров (чаще всего 3 уровень кэша). Также есть много других проблем связанных с использованием одной строки кэша двумя процессорами одновременно. Эта проблема становится очень ощутима при большом количестве процессов, так как количество пересекающихся кэш линий прямо пропорционально их количеству.

Заметим, что если использовать такую же последовательность пересчёта как в X координате, то эти проблемы пропадут.

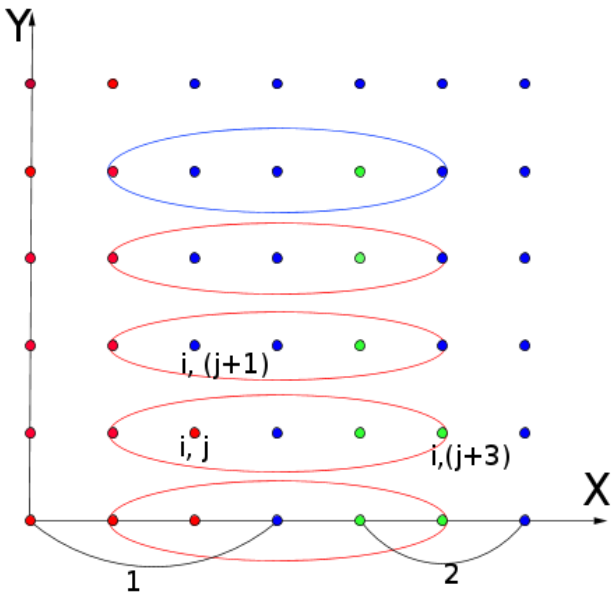

Рис. 5: Пересчёт сетки

5.2.4 Параллельная кэш-версия

В данной версии пересчёт координаты Y сделан по схеме координаты X. Что приводит к уменьшению кэш-промахов и совместного использования кэш-линий.

Для этого пришлось изменить структуру пересчёта.

Первое изменение состоит в том, что теперь каждому процессору отдаётся фиксированный последовательный набор линий, которые ему нужно пересчитать. Зачем это нужно будет объяснено позже.

Второе изменение состоит в необходимости хранить дополнительные строки информации. Взглянем на рис. Пусть процессору k необходимо пересчитать строки, выделенные красным. Для этого ему дополнительно понадобятся обведённые линии (две сверху и две снизу). Они будут необходимы для пересчёта первой и последней строки отданной. И дабы избежать совместное использование кэш-линий, значение обведённых линий копируется, и k процесс уже работает со своими собственными кэш-линиями. Для того чтобы хранить только 4 дополнительные линии и было введено первое изменение.

Результаты тестов приведены на рис. 6. Как видно из графика ускорение существенно возросло, но всё ещё не дотягивает до теоретического. Это связано с тем что теперь каждому процессору отдаётся фиксированное число линий, и всем процессам приходится ждать самого медленного. Подробнее об этом написано в части "Цикл for".

5.2.5 Параллельная кэш-версия плюс аллоцирование данных

Так как тесты проводятся на машине с архитектурой NUMA, то аллоцирование данных играет важную роль.

Для аллоцирования данных опять пришлось изменить структуру пересчёта.

Теперь пересчёт X происходит также как и Y. Это позволяет каждому процессу использовать одни и теже данные в X и Y шагах.

Инициализация сетки происходит в параллельном режиме. Сетка по прежнему shared переменная, но выделение памяти происходит в параллельном режиме. В этом случае openmp старается выделять память ближе к ядру на котором она выделяется.

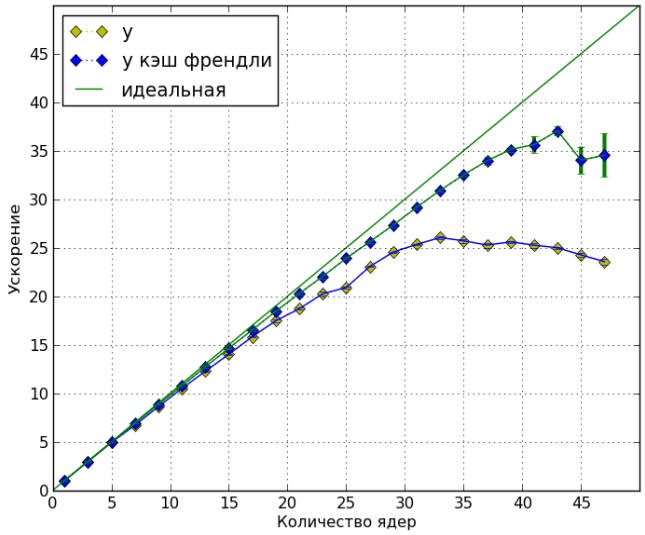

Рис. 6: Ускорение параллельной кэш версии

В нашем случае это означает, что openmp старается выделить память ближе к ядру на котором она в дальнейшем будет использоваться.

Результаты данных тестов представлены на рис. 7.

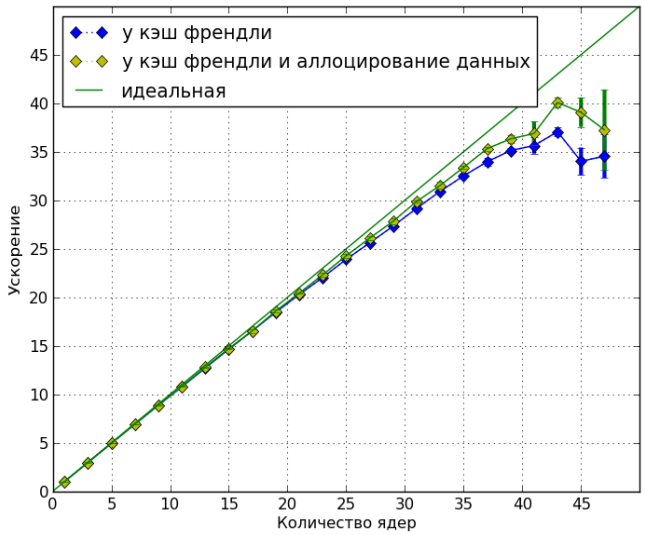

Рис. 7: Ускорение параллельной кэш версии плюс аллоцирование данных

5.2.6 Affinity и noaffinity

Технология openmp основана на тредах. У тредов есть возможность перемещаться между процессорами, что может свести на нет аллоцирование данных. Дабы убрать возможность перемещения, существует параметр affinity (affinity= *true* значит перемещать нельзя). Результаты тестов с affinity и без приведены на рис.8. Как видно из тестов при количестве процессов равных половине максимальных affinity оказывает наибольший эффект. Также включённый affinity даёт более предсказуемый результат (ошибки меньше).

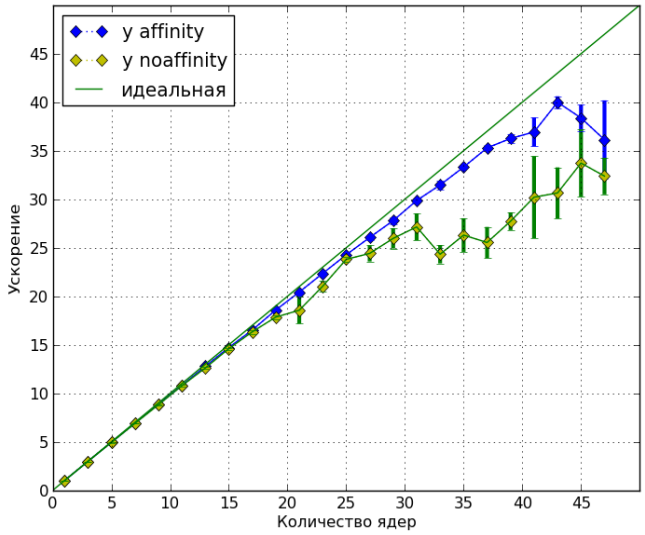

Рис. 8: Y affinity

5.2.7 Финальные тесты

Сравним простую версию и параллельную кэш-версию плюс аллоцирование. Результаты представлены на рис. 9. Как видно из графика нам удалось улучшить результаты первичного распараллеливания с 30 до 40 на 45 ядрах.

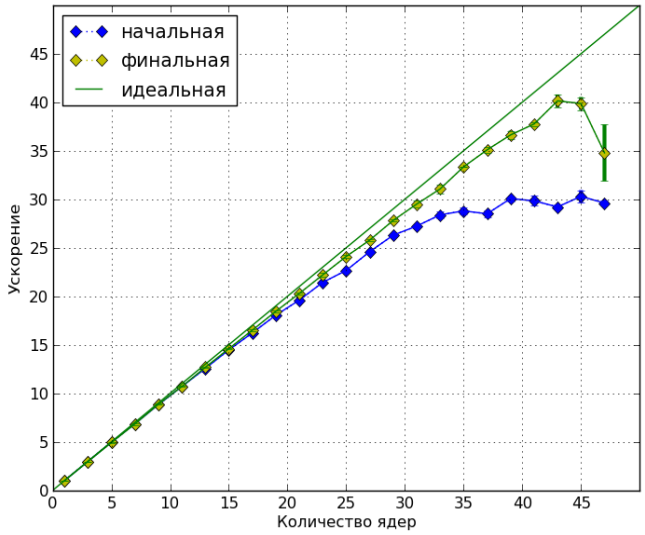

Рис. 9: Финальные тесты

6 Заключение

В данной работе были описаны следующие проблемы: кэш-промахи, аллоцирование данных, когерентность кэша. Были описаны некоторые алгоритмы для решения данных проблем. Также были описаны алгоритмы, которые могут быть применены для распараллеливания различных задач матфизики. Полученные знания были применены к решению задач сейсмики, а точнее для более эффективного распараллеливания их решения. В процессе решения поставленной задачи были продемонстрированы проблемы характерные для задач сейсмики. Подробно описаны способы решения данных проблем. Благодаря полученным знаниям, удалось улучшить результаты первичного распараллеливания с 30 до 40 на 45 ядрах.

Список литературы

[1] Introduction to Parallel Computing. Internet address:`https://computing.llnl.gov/tutorials/parallel_comp/`

[2] *Ulrich Drepper.* What Every Programmer Should Know About Memory.

[3] *Jens Breitbart.* An Approach for Semiautomatic Locality Optimizations Using OpenMP.

[4] *DaeGon Kim and Sanjay Rajopadhye.* Efficient Tiled Loop Generation: D-Tiling.

[5] *Zhenying Liu, Barbara Chapman, Tien-Hsiung Weng, Oscar Hernandez.* Improving the Performance of OpenMP by Array Privatization.

[6] *Marie Durand.* An Efficient OpenMP Loop Scheduler for Irregular Applications on Large-Scale NUMA Machines.

[7] *Yohei Sato, Takanori Hino Kunihide Ohashi.* Parallelization of an unstructured Navier–Stokes solver using a multi-color ordering method for OpenMP.

[8] *Fredrik Manne.* Parallel graph coloring using openmp.

[9] *Paul Kapinos and Dieter an Mey.* Parallel Simulation of Bevel Gear Cutting Processes with OpenMP Tasks.

[10] *Ramachandra Nanjegowda.* Scalability Evaluation of Barrier Algorithms for OpenMP.

[11] *Хохлов Николай Игоревич.* Численное моделирование сейсмических процессов на высокопроизводительных.