МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Информационные технологии и прикладная математика»
Кафедра 806 «Вычислительная математика и программирование»

Лабораторная работа №3
по курсу «Параллельная обработка данных»

Технология MPI и технология OpenMP

Выполнил: М.А. Бронников
Группа: М8О-407Б
Преподаватели: К.Г. Крашенинников,
А.Ю. Морозов

Москва, 2020

Условие

Цель работы: Совместное использование технологии MPI и технологии OpenMP. Реализация метода Якоби. Решение задачи Дирихле для уравнения Лапласа в трехмерной области с граничными условиями первого рода.

Вариант №2: Распараллеливание в общем виде с разделением работы между нитями вручную (“в стиле CUDA”).

Программное и аппаратное обеспечение компьютера:

Device: GeForce GT 545
Размер глобальной памяти: 3150381056
Размер константной памяти : 65536
Размер разделяемой памяти: 49152
Регистров на блок: 32768
Максимум потоков на блок: 1024
Количество мультипроцессоров : 3
OS: Linux Mint 20 Cinnamon
Редактор: VSCode
Машины в кластере:
1. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 1050, 2 Gb
2. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 16 Gb, GeForce GT 545, 3 Gb
3. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 16 Gb, GeForce GTX 650, 2 Gb
4. Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz, 12 Gb, GeForce GT 530, 2 Gb
5. Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 8 Gb, GeForce GT 530, 2 Gb
Все машины соединены гигабатным ethernet и находятся в подсети 10.10.1.1/24. В качестве операционной системы установлена Ubuntu 16.04.6 LTS. Версии софта: mpirun 1.10.2, g++ 4.8.4, nvcc 7.0

Метод решения

Общая схема решения аналогична заданию из 7 лабораторной:
1. На первом этапе происходит обмен граничными слоями между процессами.
2. На втором этапе выполняется обновление значений во всех ячейках.
3. Третий этап заключается в вычислении погрешности: сначала локально в рамках каждого процесса а потом через обмены и во всей области.

Однако второй этап можно распараллелить на каждом из процессов с помощью технологии OpenMP. Теперь каждый поток в процессе будет перерассчитывать значения для отдельного участка памяти в блоке.

Описание программы

По сравнению с 7 лабораторной добавились директивы препроцессора #pragma parallel, которые позволяют задавать участки, которые будут выполнятся в многопоточном режиме для каждого из процессов. Для того, чтобы каждый поток отвечал за отдельный участок, мне пришлось переписать циклы с той целью, чтобы не было простаивающих потоков. Определение следующих значений i, j, k делает следующий макрос:

```
#define next_ijk(i, j, k, step) { \
--i; --j; --k; \
i += step; \
int addition = i / (blocks[dir_x] - 2); \
i -= addition * (blocks[dir_x] - 2); \
j += addition; \
addition = j / (blocks[dir_y] - 2); \
j -= addition * (blocks[dir_y] - 2); \
k += addition; \
++i; ++j; ++k; \
} \
```

Узким местом стало то, что теперь затруднился расчет максимального модуля разности значений в процессе, поскольку необходимо синхронизировать это значение от попыток разных потоков его изменить. Для этого я создал отдельный массив для с максимальными значениями для каждого из потоков:

```
int thrd_max = omp_get_max_threads();
double* norm_data = new double[workers_count];
```

После конца параллельного блока основной поток проходит по этому массиву и определяет наибольшее значеие.

Результаты:

Сравним время выполнения трёх разных программ: написанных для CPU и MPI из 7 лабораторной и текущей реализацией. Будем делать честное сравнение, поэтому замерим полное время выполнения программы, а не только основной цикл. Посмотрим на результат запуска на одном процессе:

Сетка \ Расчет	MPI	CPU	MPI+OMP
1 1 1 | 40 40 40	19861ms	9843.8ms	7574.2ms

Как видно, результат превосходит всех остальных, несмотря даже на то, что эта программа внутри себя сталкивается со всеми издержками, что и обычная MPI программа.

В тоже время запуск программы на нескольких процессах дал мне неприлично долгое время ожидаения, что может быть связано с тем, что учебный кластер довольно слабый, или что написанная мной программа неэффективна в силу моей неопытности.

Я подумал, что время польза mpi+omp будет нагляднее, если запускать процессы с большим объемом вычислений в каждом, поэтому запустил программу с большим количеством данных. Первые иттерации действительно превосходили по времени обычную программу MPI, однако позже этот выйгрышь спал на нет. Я думаю, что проблема связанна с тем, что кластер стал не справляться с возросшей нагрузкой.

Выводы

Данный метод Дирихле являетя одним из конечно-разностных методов, которые широко используются для решения дифференциальных уравнений на заданной сетке с высокой степенью точности. Без этих методов сложно представить современную физику, которая использует эти методы для расчета уравнений теплопроводности при разных условиях среды.

Эта лабораторная познакомила меня с новой для меня технологией OMP, позволяющей легко и быстро распараллеливать свой код с помощью легковесных потоков. Более того я научился использовать эту технологию в связке с MPI, что позволяет разбивать пррограмму на процессы, каждый из которых будет иметь несколько потоков исполнения.

Прирост полученный мной при однопроцессорном запуске меня приятно удивил, однако запуск программы на нескольких процессах меня огорчил своим безумно медленным выполнением, что объясняется тем, что кластеру не хватило ресурсов, чтобы справиться с сильно возросшей нагрузкой, а также повлияло время, которое тратиться на создание и инициализацию потоков на CPU, что более ресурсоёмко, нежели, чем на CPU. Скорее всего, если бы я изначально выбрал другую стратегию по организации своего кода, я бы смог добиться лучших результатов за счет уменьшения накладных расходов на распараллеливание участков моего кода.