Отчёт по практическому заданию

Михальцов Данила, студент 328 группы ВМК МГУ

Задача: Реализовать параллельную версию предложенного алгоритма с использованием технологий OpenMP и MPI

Предложенный алгоритм: Red-Black 3D — трёхмерная версия алгоритма Red-Black, предложенного в статье Takeshi Iwashita; Masaaki Shimasaki (2003). *Block Red-Black Ordering: A New Ordering Strategy for Parallelization of ICCG Method.*

Это **метод блочного упорядочивания**, который имеет всего две точки синхронизации на каждой итерации расчётов, за счёт чего он может быть эффективно распараллелен. Это достигается за счёт того, что каждая итерация разбивается на две части, каждая из которых использует результаты другой с прошлой итерации. При этом каждая часть итерации не имеет зависимостей по данным внутри себя, поэтому может быть легко распараллелена.

OpenMP

Для реализации параллельной версии программы с использованием OpenMP были проделаны следующие модификации программы:

1) Для ускорения программы порядок индексов в циклах был изменён с (k, j, i) на (i, j, k), чтобы внешний цикл шёл по первой размерности, средний по второй, а самый внутренней по третьей. Это известная оптимизация, которую нужно было бы применить и без распараллеливания.
2) Для цикла в функции `init` была добавлена директива
```
#pragma omp parallel for collapse(3) default(none) \
  private(i, j, k) shared(A)
```
3) Для циклов в функции relax были добавлены директивы
```
#pragma omp parallel for collapse(3) default(none) \
    private(i, j, k) shared(A, w) reduction(max:eps)
```
И
```
#pragma omp parallel for collapse(3) default(none) \
    private(i, j, k) shared(A, w)
```

Для того, чтобы сработала прагма `collapse(3)` и можно было запускать самую внутреннюю часть цикла на отдельной нити, пришлось убрать зависимость k от i и j — перебирать его от 1 до N-2 с шагом 1, поэтому также пришлось добавить внутрь цикла условие `if ((k + i + j) % 2 == 0)`, чтобы логика работы алгоритма не поменялась

4) В функции verify была добавлена прагма

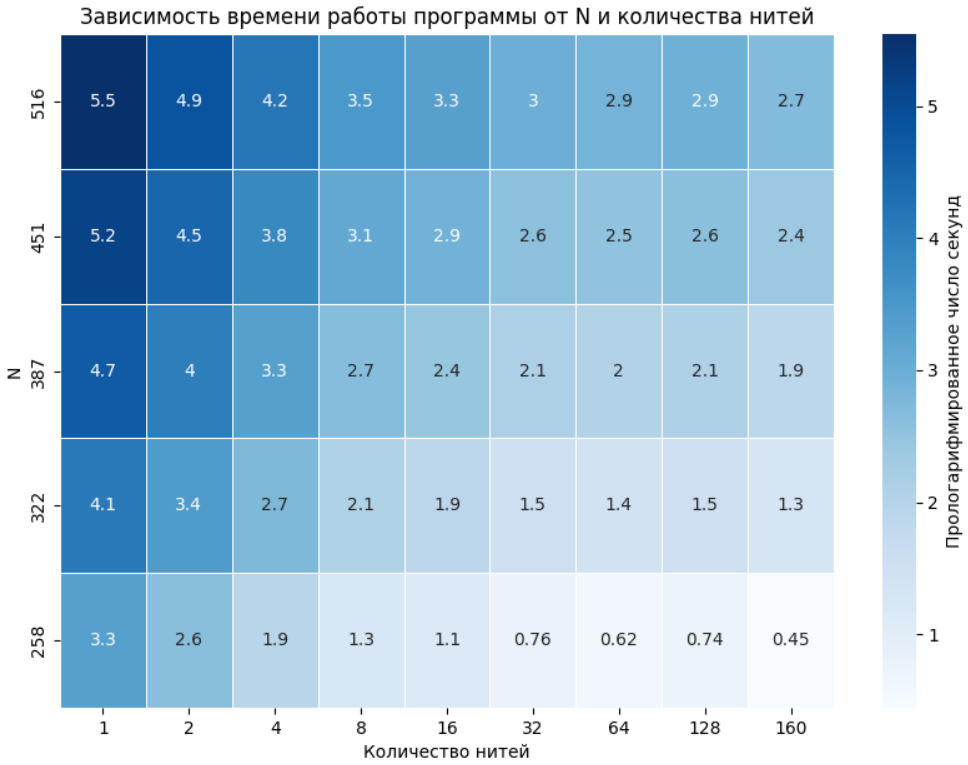
```
#pragma omp parallel for collapse(3) default(none) \
  private(i, j, k) shared(A) reduction(+:s)
```

Замер времени производился с помощью функций `gettimeofday` и `timersub`.

Результаты

Каждое значение времени, использованное для построения графика, является усреднённым из замеров для трёх запусков.
Помимо этого, для большей наглядности было решено прологарифмировать значения.

Получившийся график-heatmap:



Точные значения замеров сохранены в приложенном к отчёту python-скрипте для построения графика.

Стоит отметить, что с ростом N сложность задачи растёт как O(N^3).

Как видно из графика, для получившейся OpenMP-реализации алгоритма при данном максимальном количестве нитей (160):

- при увеличении числа нитей производительность растёт, но
- **при увеличении числа потоков с 64 до 128 производительность ухудшается**,
- однако при увеличении числа потоков со 128 до 160, производительность улучшается

Возможно, при большем максимальном количестве нитей мы бы более явно заметили, что для получения наилучшей производительности нельзя бесконечно увеличивать число потоков.

Программы

Для удобства запуска и для построения графиков были написаны bash и python скрипты, приложенные к отчёту. Основная программа также приложена к отчёту.

MPI

Для реализации параллельной версии программы с использованием технологии MPI в код были добавлены необходимые вызовы функций, определяющие взаимодействие разных процессов.

Изначально была идея разделить куб NxNxN на маленькие кубы KxKxK и отдать их разным процессам. Но после некоторых размышлений я пришёл к выводу, что логика взаимодействия процессов окажется довольно сложной, и что это может даже ухудшить результаты параллельной программы по сравнению с более простым разделением.

Поэтому каждый процесс обрабатывает одинаковое (кроме последнего процесса, он обрабатывает все оставшиеся) количество параллелепипедов 1x1xN (в коде для удобства они называются `sausages` в честь своей формы). Сначала я хотел поделить число “сосисок” на число `num_workers` и таким образом определить число “сосисок” на один процесс. Но опять же, эта схема оказалось излишне сложной по сравнению с выбранной в итоге: т.к. у нас есть квадрат N*N из сосисок, мы можем просто отдавать процессу определённое количество строк. При больших N это почти не будет отличаться от второй схемы (т.к. считаем, что N >> n_workers), но не потребует таких накладных расходов: всего лишь нужно отдать свои верхнюю и нижнюю строки. (В коде присутствуют закомментированные попытки реализовать вторую схему, но они в итоге не пригодились)

Итак, `sausages_rows_per_proc = (N-2) / num_workers;` (т.к. барьерные элементы нулевые). Каждый процесс, вообще говоря, обрабатывает несколько строк —

от `first_s_row` до `last_s_row` не включительно. После каждой итерации процессы обмениваются граничными слоями “сосисок” через `MPI_Send` и `MPI_Recv`.

В коде есть комментарии про то, кто что кому отправляет.

После каждой из двух частей итераций стоит барьер. В конце мы проводим редукцию `eps` среди всех `local_eps`. Один из процессов (с нулевым `rank`) печатает результат.

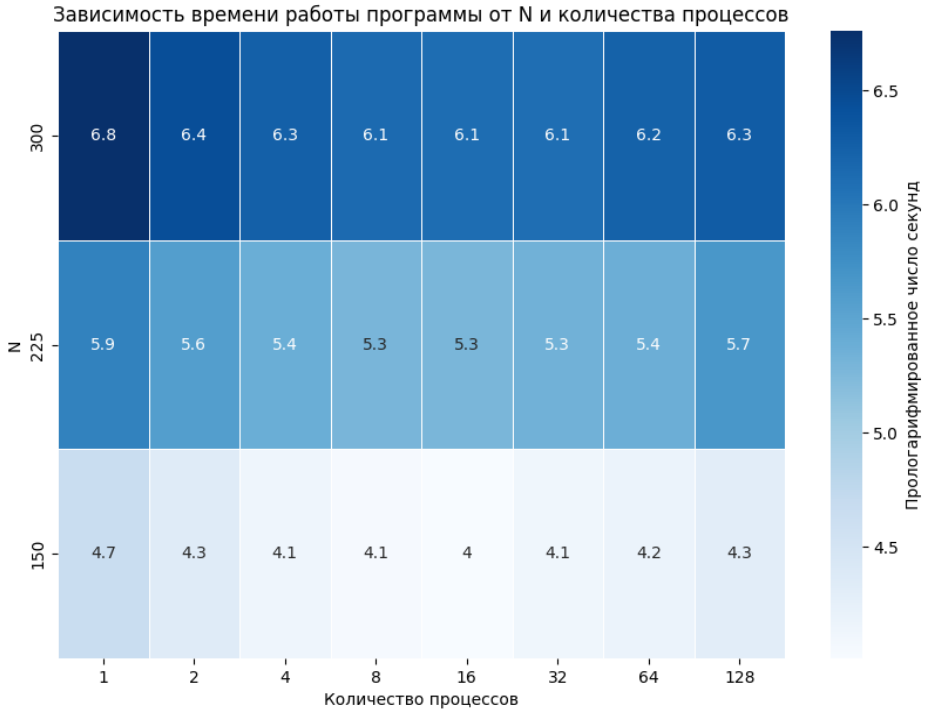
Результаты замеров усреднены по трём запускам. Значения, правда, отличались меньше, чем на 0.001 секунды.

Замеры времени работы программы в зависимости от N и числа процессов производились на Blue Gene с помощью `timersub`, как и в прошлой программе

Результаты

Получившиеся результаты можно увидеть на графике. Изначально планировалось большее количество разных N, но т.к. сложность задачи растёт как N^3, а Blue Gene с его производительностью и загруженностью был не очень рад бОльшим N, и с учётом того, что дробить промежутки между 150, 225, 300 почти бесполезно для выводов, я решил оставить этот набор N. К тому же, график получился довольно информативный:

Здесь чётко видно, что при **увеличении количества процессов до 32+ результат начинает ухудшаться.** Это обусловлено накладными расходами на распараллеливание программы.



Сравнение OpenMP и MPI

Если сравнивать получившиеся реализации программ на данных машинах с данными размерностями данных, то можно заметить, что **OpenMP** при увеличении числа тредов **сильно уменьшает время работы** программы (в 2 раза и больше), а **MPI-версия** при увеличении количества процессов **не даёт такого сильного прироста** в производительности. Возможно, для того, чтобы технология MPI дала больший прирост производительности, нужно сильно увеличить размер данных.