****МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

имени М. В. Ломоносова

Факультет вычислительной математики и кибернетики

**Практикум по курсу**

**"Суперкомпьютеры и параллельная обработка данных"**

**Разработка параллельной версии программы для алгоритма Sor2D с использованием OpenMP**

**ОТЧЕТ**

**о выполненном задании**

студента 324 учебной группы факультета ВМК МГУ

Яндиева Абдуллаха Ахметгиреевича

Москва, 2021 г.

# Описание алгоритма

Вся программа делится на 3 составляющих: инициализация матрицы – init(), цикл по итерациям, в котором и происходит основная работа – relax(), подсчет и вывод результата – verify().

Сначала создается квадратная матрица N\*N вида (для N = 5):

где

Далее в цикле по итерациям с установленной точностью пересчитываются элементы матрицы А по формуле:

В конце, на основе полученной матрицы, считается величина S:

Итого, в программе создается 3 параллельной области. В первой происходит инициализация матрицы. Экспериментальным путем было получено, что до N = 2000 распараллеливать эту часть задачи не имеет смысла, что отражено в тексте программы: у соответствующей директивы parallelуказано условие if (N > 2000). Сам цикл распараллеливается при помощи директивы for, так как директива collapse(2) не приводит к улучшению результатов.

Вторая параллельная область создается в цикле по итерациям, то есть на самом деле создается не одна область, а столько, сколько итераций произойдет в программе. Внутри этой области происходит выполнение цикла с зависимостью по данным. Поэтому в качестве решения задачи распараллеливания программы организовано конвейерное выполнение цикла с массивом isync из нулей и единиц, позволяющему синхронизировать работу между нитями. Алгоритм отличается от того, что мы рассматривали на лекциях тем, что в нашей программе на каждой итерации считается точность как максимум из разницы преобразования данного элемента матрицы и предыдущего значения . В связи с этим в программе перед и после каждого переприсваивание , используется директива flush. Для того чтобы реже обращаться к общей переменной , ввелась локальная для каждой нити переменная .

Третья параллельная область создается для подсчета результата, то есть величины S, определенной выше. Здесь также получено, что при N < 2000 не имеет особого смысла распараллеливать эту часть программы, так как результаты получаются почти одинаковые, или же хуже с уменьшением объема данных. Сам цикл распараллеливается при помощи директивы for, так как директива collapse(2) не приводит к улучшению результатов.

Помимо распараллеливания программы, также были предприняты попытки оптимизировать исходный последовательных код. Во-первых, был изменен порядок обращения к матрице, то есть вместо:

for(int j=0; j<=N-1; j++)

for(int i=0; i<=N-1; i++)

A[i][j] = …

я написал следующее:

for(int i=0; i<=N-1; i++)

for(int j=0; j<=N-1; j++)

A[i][j] = …

После этого на 1 нити при N = 1000 имеем 1,071569 с вместо 1,228022 с, что является существенным ускорением программы даже на небольшом объеме данных.

Также была предпринята попытка убрать из программы функции, и переделать ее полностью в последовательный код. Это дало некоторое улучшение работы: 1,070456 с вместо 1,071569 с (при тех же конфигурациях). Но так как прирост оказался незначительным, то было принято решение отказаться от этого приема в пользу лаконичности и читаемости кода.

Замечание: Для подсчета времени выполнения используется функция omp\_get\_wtime(). Общее время работы определяется временем выполнения самой медленной нити.

# График зависимости времени выполнения от числа нитей для различного объема входных данных

# 

Замечание 1: График построен по таблице полученных результатов, которая приведена ниже (левая колонка – количество нитей, верхняя строчка – объем данных, см. замечание 2, значения в таблице – время работы программы в секундах):

Изображение выглядит как стол

Автоматически созданное описаниеЗамечание 2: По оси «объем данных» указан размер N, но, понятное дело, имеется в виду размер квадратной матрицы N\*N (это сделано с целью не загромождать график), то есть размер 1000, на самом деле – 1000\*1000, 2000 – 2000\*2000 и т. д.

Замечание 3: Результаты получены на вычислительном комплексе IBM Polus.

# Анализ результатов и выводы

Из графика зависимости времени выполнения от числа нитей для разного объема данных видно, что увеличение количества нитей не всегда приводит к уменьшению времени выполнения программы. Так, наибольшая эффективность для матриц с размерами 1000\*1000, 2000\*2000 и 3000\*3000 достигается на 8 нитях, для матриц с размерами 4000\*4000 и 5000\*5000 - на 16 нитях, а для матриц с размерами 10000\*10000 и 20000\*20000 – на 32 нитях. Также для матрицы размера 30000\*30000 проверено, что наиболее эффективным будет выполнение на 64 нитях (на графике это не представлено, так как на 1 и 2 нитях выполняется слишком долго). Таким образом, можно сделать вывод, что количество требуемых для наиболее быстрого выполнения программы нитей растет примерно с той же скоростью, как и размер задачи (по крайней мере, до N = 30000, дальше уже не проверялось). Тот факт, что скорость выполнения программы не всегда растет с увеличением количества нитей, можно объяснить ростом накладных расходов, связанных с созданием большего числа нитей (даже если надеется на то, что компилятор оптимизирует этот процесс, и нити, на самом деле, создаются заранее, а не каждый раз в цикле). Также с ростом количества нитей мы имеем дело с ростом конкуренции за доступ к данным. Так как ведется работа с общей матрицей А и с общей переменной , то при барьерной синхронизации, при организации конвейерного выполнения цикла с зависимостью по данным мы имеем все большее количество ожидающих работы нитей. Поэтому нужно четко оценивать cложность задачи, и не делать поспешных выводов о том, сколько нитей нужно использовать для той или иной задачи при том или ином наборе данных.

Также можно сделать вывод, что до достижения того количества нитей, на котором наша программа максимально ускоряется, время выполнения программы в зависимости от количества нитей растет почти линейно. Этот факт можно рассматривать как подтверждение того, что программа написана близко к правильной, ибо такие результаты мы и ожидаем. А то, что время выполнения программы все-таки убывает не линейно, а только близко к этому, объясняется теми же накладными расходами и конфликтом по данным.

Замечание: Текст программы прикреплен отдельным файлом с названием 324\_yandiev\_sor\_2d.c.