****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 MPI**

**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:

Сама программа написана при помощи алгоритма с теневыми гранями, описанным на лекции. Также функции для удобства были переделаны в последовательный код.

В самом начале каждый процесс узнает свой номер (MPI\_Comm\_rank()), количество процессов (MPI\_Comm\_size()) и встает на барьерную синхронизацию (MPI\_Barrier()).

**Инициализация.** Исходная матрица A делится на количество процессов. Каждый процесс имеет свою собственную матрицу, причем первый процесс имеет первые строки описанной выше матрицы A, второй – следующие строки и т. д. так, чтобы каждый процесс имел равное количество строк. Помимо этого, в каждой матрице каждого процесса предусмотрены 2 теневые грани: сверху и снизу. Очевидно, в первом и последнем процессе они не используются, зато матрицы первого и последнего процессов не приходится обрабатывать отдельно, что приближает код полученной программы к исходному последовательному коду.

**Основная работа.** Каждый процесс заходит в цикл по итерациям. Причем количество итераций разделено на количество процессов, чтобы величина максимального количества итераций не росло с количеством процессов, и вывод программы независимо от количества процессов был одинаков. Сразу отмечу, что в программе используются блокирующие send (MPI\_Send()) и receive (MPI\_Recv()), так как в цикле имеется зависимость по данным, то есть следующий процесс не может начать обработку своей матрицы на данной итерации, пока предыдущий процесс не завершит обработку последней строки (не считая теневую грань, конечно же) свой матрицы. Поэтому каждый следующий процесс должен блокироваться до получения необходимых данных. Сам цикл организован, как и в последовательной программе, только изменено условия выхода их цикла по строкам, так как количество строк в матрицах каждого из процесса меньше количества строк исходной матрицы в 1/(количество процессов) раз. Отправка данных происходит так: следующий процесс ждет отправки нужно строки от предыдущего, отрабатывает свой цикл, и отправляет последнюю строку (не теневую) предыдущему процессу, а сам ждет от следующего процесса нижнюю теневую грань (здесь можно и не использовать блокирующий receive, но он не сильно мешает). А затем сам отправляет верхнюю (не теневую) строку предыдущему процессу. И так, пока редуцированное по всем процессам значение eps не уменьшится до заранее определенного значения maxeps или до истечения количества итераций, максимальное значение которого ITMAX также определено.

**Подсчет и вывод результата.** После выхода из цикла по итерациям, каждый процесс независимо друг от друга входит в цикл по подсчету своего локального значения величины S, описанной выше. И далее при помощи MPI\_Allreduce() происходит подcчет уже итогового значения суммы: берется значение всех локальных сумм и суммируется в переменную S.

**Улучшение последовательного кода.** 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] = …

было написано следующее:

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

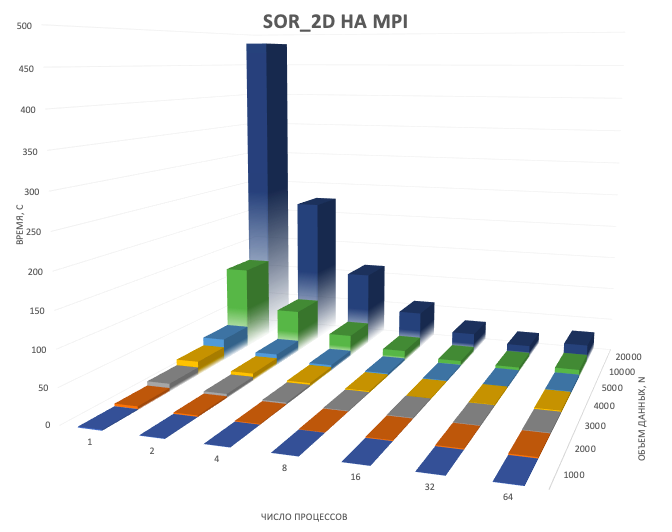
for(int j=0; j<=N-1; j++)

A[i][j] = …

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

Замечание: Для подсчета времени выполнения используется функция MPI\_Wtime(). Общее время работы определяется временем выполнения самого медленного процесса.

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



Изображение выглядит как стол

Автоматически созданное описание

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

Замечание 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. Также были сделаны подсчеты на машине BlueGene, но полученных результатов недостаточно, чтобы составить полноценный отчет: в какой-то момент начинали возвращаться пустые файлы с выводом (и так происходило за несколько подходов). Но даже этих тестов оказалось достаточно, чтобы сделать вывод, что Polus на заданном количестве процессов работает быстрее. Но в пользу BlueGene играет то, что там можно запустить существенно большее количество процессов.

# Сравнение эффективности OpenMP и MPI-версий параллельной программы sor2D

Что касается эффективности OpenMP и MPI-версий параллельной программы sor2D, то тут можно сделать простой вывод, что MPI-версии программы дольше разгоняются: на меньшем количестве нитей и процессов OpenMP-версия программы работает быстрее, причем довольно-таки существенно, чем соответственно MPI-версия. Но на дистанции, то есть при увеличении количества нитей и процессов, мы можем наблюдать, что MPI-версия программы начинает работать быстрее. Это объясняется самим алгоритмом решения. Алгоритм работы OpenMP-версии несколько сложнее устроен, и при росте количества нитей происходит все большее количество блокировок этих самых нитей. Хотя в MPI-версии программы у нас также имеются блокировки процессов, но они происходят реже и с более простой зависимостью (все таки в OpenMP мы просматриваем матрицу диагонально). И если учитывать, что непосредственно пересылки у нас происходят нечасто, то это только и настраивает нас на то, что стоит ожидать лучшего ускорения все-таки MPI-версии. Также в OpenMP-версии чаще происходит операция редукции, что тоже играет не в ее пользу. В общем, данную задачу определенно стоит распараллеливать при помощи технологии MPI.

# Анализ результатов и выводы

Здесь будут представлен анализ лишь MPI-версии, ибо анализ OpenMP-версии программы был предоставлен ранее и уже проверен.

Из графика зависимости времени выполнения от числа процессов для разного объема данных видно, что увеличение количества процессов 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 процессах, для матриц с размерами 4000\*4000 и 5000\*5000 - на 16 процессах, а для матриц с размерами 10000\*10000 и 20000\*20000 – на 32 процессах. Также для матрицы размера 30000\*30000 проверено, что наиболее эффективным будет выполнение на 64 процессах (на графике это не представлено, так как на 1 и 2 процессах выполняется слишком долго). Таким образом, можно сделать вывод, что количество требуемых для наиболее быстрого выполнения программы процессов 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 процессов, можно объяснить ростом накладных расходов, связанных с созданием большего числа процессов. Также с ростом количества процессов мы имеем дело с ростом количества пересылок данных, ведь в отличие от OpenMP, здесь мы имеем дело с распределенной, а не общей памятью. Поэтому нужно четко оценивать cложность задачи, и не делать поспешных выводов о том, сколько процессов нужно использовать для той или иной задачи при том или ином наборе данных.

Также можно сделать вывод, что до достижения того количества процессов, на котором наша программа максимально ускоряется, время выполнения программы в зависимости от количества процессов 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 пересылками данных.

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