1. Министерство науки и высшего образования Российской Федерации
2. Санкт-Петербургский Политехнический Университет Петра Великого
3. —

Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

**ЛАБОРАТОРНАЯ РАБОТА**

**«Решение СЛАУ с треугольными матрицами»**

по дисциплине «Параллельное программирование на суперкомпьютерных системах»

Выполнила

студентка гр. 3540201/20302 Обидина А.И.

Проверил

доцент Лукашин А.А.

Санкт-Петербург

2023

**Содержание**

[**1. Постановка задачи** 3](#_Toc128713599)

[**2. Теоретическая часть** 4](#_Toc128713600)

[**3. Реализация** 8](#_Toc128713601)

[**4. Результаты** 12](#_Toc128713602)

[**5.** **Тестирование** 15](#_Toc128713603)

[**6.** **Вывод** 18](#_Toc128713604)

# **1. Постановка задачи**

1. Выбрать задачу и проработать реализацию алгоритма, допускающего раcпараллеливание на несколько потоков / процессов.
2. Разработать тесты для проверки корректности алгоритма (входные данные, выходные данные, код для сравнения результатов). Для подготовки наборов тестов можно использовать математические пакеты, например, matlab (есть в классе СКЦ и на самом СКЦ).
3. Реализовать алгоритмы с использованием выбранных технологий.
4. Провести исследование эффекта от использования многоядерности / многопоточности / многопроцессности на СКЦ, варьируя узлы от 1 до 4 (для MPI) и варьируя количество процессов / потоков.
5. Подготовить отчет в электронном виде.

**Прикладная задача:** решение СЛАУ с треугольными матрицами.

**Технология**

* C & Linux pthreads
* С & MPI
* Python & MPI
* С & OpenMP

**3.1 C & Linux pthreads**

Из библиотеки pthreads были взяты методы:

* pthread\_create – функция, создающая поток.
* pthread\_join – функция, принимающая данные от потока, который закончил свое выполнение.

Таким образом, программа создает необходимое количество потоков, после чего каждый из них считает свою часть матричного умножения. Главная часть программы дожидается, пока все потоки закончат свое выполнения, и собирает их в выделенный массив данных. Далее посчитанная матрица сохраняется в файл *x\_pthread.txt*.

# **2. Теоретическая часть**

Система линейных алгебраических уравнений с квадратной матрицей записывается в виде:

Числа aij образуют таблицу, называемую матрицей, а сами эти числа называются элементами матрицы. Для обозначения матрицы используются заглавные буквы. Квадратные матрицы имеют следующую структуру (матрица размером 3x3):

xj – элементы вектора-столбца решений системы, bi – элементы вектора-столбца свободных членов.

Методы решения систем линейных алгебраических уравнений разделяют на два больших класса методов: прямые методы (их часто называют также точными) и итерационные (приближенные).

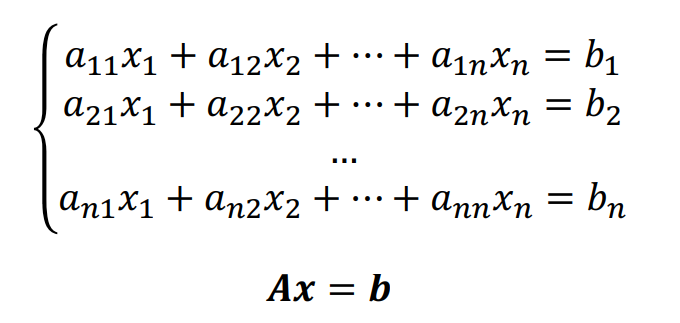
Прямые методы позволяют получить решение за конечное число арифметических операций (метод Крамера, метода Гаусса и его модификации и т. д.).

Однако число операций в этих методах существенно зависит от размерности системы уравнений, причем эта зависимость нелинейная. Например, метод Гаусса (решение систем уравнений с треугольной матрицей) требует порядка n3 операций для прямого хода и порядка n2 операций для обратного хода.

Итерационные методы позволяют получить решение лишь с заданной точностью, но у них число операций менее жестко связано с размерностью системы уравнений.

Основная идея метода Гаусса - приведение матрицы А посредством эквивалентных преобразований к треугольному виду, после чего значения искомых неизвестных могут быть получены непосредственно в явном виде.

**Прямой ход метода Гаусса** – это поочерёдное преобразования уравнений системы для последующего избавления от переменных неизвестных.



В системе уравнений с помощью умножения уравнения на константу, прибавления к уравнению другого уравнения обнуляют коэффициенты при a21, a31, a32, a41, a42, a43, …, an,n-1, получается верхняя треугольная матрица.

На рисунке 1 приведена общая схема состояния данных на i-й итерации прямого хода.

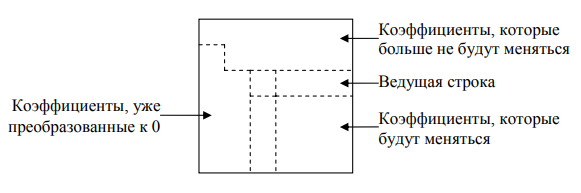


Рисунок 1 - Общая схема состояния данных на i-й итерации прямого хода

**Обратный ход метода Гаусса** – это вычисление неизвестных переменных от последнего уравнения к первому (из полученной треугольной матрицы при прямом ходе).

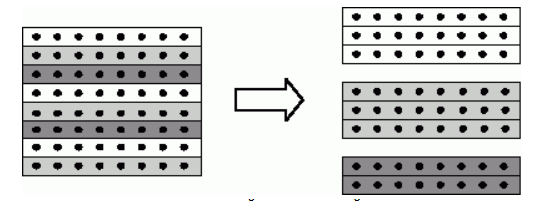
**Последовательное выполнение:**

При последовательном выполнении метода Гаусса для обнуления коэффициентов a21, …, an1 из строк с номерами 2, …, n вычитается первая строка, умноженная на константу. Для обнуления коэффициентов a32, …, an2 из измененных на первом шаге строк с номерами 3, …, n вычитается вторая строка, умноженная на соответствующую константу и т. д.

Далее из n-го уравнения получается значение xn, при его подстановке в (n-1)-е уравнение получается значение xn-1 и т. д. до получения значения x1.

**Параллельное выполнение:**

Идея параллельного выполнения метода Гаусса заключается в циклическом разделении строк между процессами для выполнения прямого хода, так как все вычисления сводятся к однотипным вычислительным операциям над строками матрицы коэффициентов системы линейных уравнений.



При обнулении коэффициентов из i-го столбца матрицы с помощью i-й строки потокам отдаются в соответствие строки с номерами (i+rank+1; n; numprocs), где rank – ранг процесса (потока), n – количество строк в матрице, numprocs – количество процессов (потоков). Аналогичные распределения строк между процессами происходят при дальнейшем обнулении коэффициентов из последующих столбцов (до столбца n-1 включительно). Такое циклическое распределение строк между процессами (потоками) обеспечивает лучшую балансировку вычислительной нагрузки между подзадачами.

Для решения задачи был разработан следующий алгоритм:

1. Инициализация матрицы коэффициентов (чтение из файла).
2. Распределение вычислений по потокам (процессам).
3. Вычисления.
4. Сбор полученных данных по потокам (процессам).
5. Запись результата (вектора решений) в файл.

# **3. Реализация**

Для создания исходных файлов была написана программа *gen.py*, генерирующая матрицу A размерностью N1x(N1+1) (со столбцом B) и записывающая её в файл m.txt (пример сгенерированной матрицы представлен на рисунке 2).

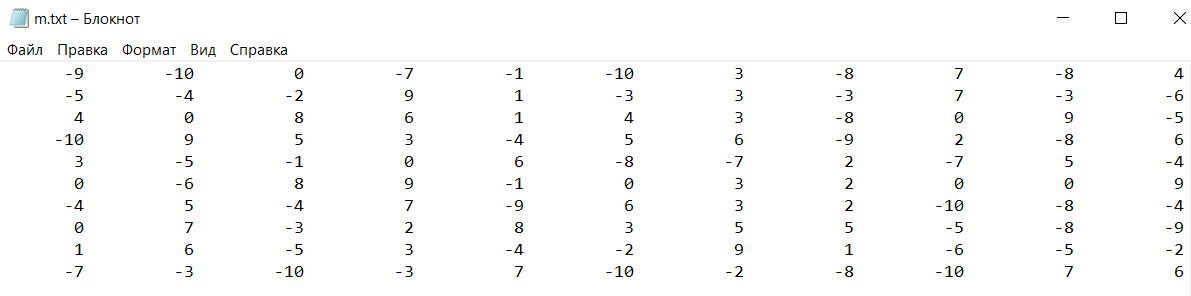


Рисунок 2 – Пример сгенерированной расширенной матрицы коэффициентов размером 10x11

Тестирование проводилось с помощью средств для работы с матрицами для языка программирования Python – numpy. Для этого была разработана программа *check.py*. Она загружает исходную матрицу и отделяет от нее вектор B, полученный вектор решений, после чего перемножает квадратную матрицу коэффициентов и вектор решений м помощью метода numpy.dot(). Если в результате был получен вектор B (выделенный из исходной расширенной матрицы коэффициентов), то вектор X был найден корректно. Иначе выводится ошибка с индексом не совпавших элементов полученных при вычислении векторов.

Согласно заданию, для реализации алгоритма необходимо было использовать следующие технологии:

* C & Linux pthreads
* С & MPI
* Python & MPI
* С & OpenMP

Для каждой технологии использовались свои методы, однако алгоритм действий отличался не сильно, кроме технологии C & MPI. Рассмотрим подробнее каждую из предложенных технологий.

**3.1 C & Linux pthreads**

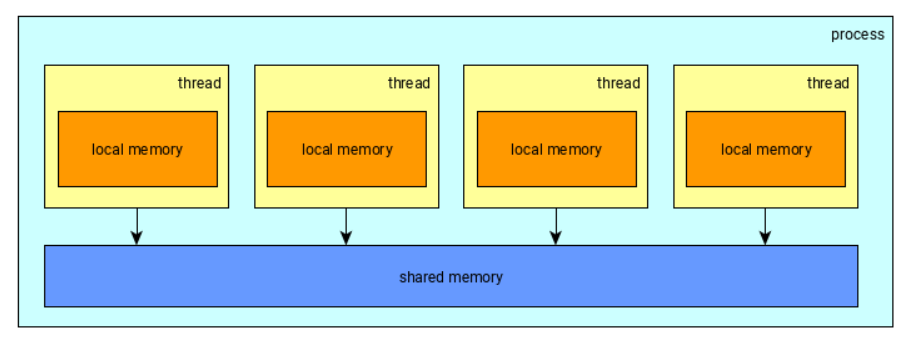
Разделяемая память позволяет избежать слияния массивов строк каждого потока.

Для реализации данной технологии из библиотеки pthreads были взяты методы:

* pthread\_create – функция, создающая поток;
* pthread\_join – функция, принимающая данные от потока, который закончил свое выполнение.

Программа создает нужное количество потоков (аргумент командной строки), после чего каждый из них считает номер строки матрицы коэффициентов, с которой он должен начать свои вычисления. Далее идет ожидание, пока все потоки закончат свое выполнение. Далее выполняется обратный ход метода Гаусса, найденный вектор решений сохраняется в файл *x\_pthread.txt*.

**3.2 C & OpenMPI**



Библиотека OpenMP подходит только для программирования систем с общей памятью, при этом используется параллелизм потоков. Потоки создаются в рамках единственного процесса и имеют свою собственную память. Кроме того, все потоки имеют доступ к памяти процесса.

Процесс состоит из одного потока (главного), создающего несколько задач для каждого уровня рекурсии, а затем все потоки могут выполнять задачи из этой сгенерированной очереди задач.

*#include<omp.h>* — заголовочный файл openmp для использования его функций.

*omp\_set\_num\_threads(threads\_count);* - фиксирование количества потоков.

*#pragma omp parallel for shared(a, b) private(i, j, scaling)* - определяет параллельную область, содержащую код, который будет выполняться, используя несколько параллельных потоков. Этот код будет разделен между всеми потоками, причем переменные shared – общие между потоками, а private – для каждого потока свои.

Распараллеливается прямой ход метода Гаусса, после выполнения вычислений всеми потоками выполняется обратный ход. Результат записывается в файл *x\_openmp.txt*.

**3.3 C & MPI**

MPI — это библиотека интерфейса передачи сообщений, позволяющая выполнять параллельные вычисления путем отправки кода на несколько процессоров.

Для реализации этой технологии из библиотеки С & MPI были использованы методы:

* MPI\_Init – инициализация MPI.
* MPI\_Comm\_rank – получение текущего номера процесса.
* MPI\_Comm\_size – получение количества процессов.
* MPI\_Bcast – широковещательная рассылка данных.
* MPI\_Finalize – уничтожение окружения MPI.

Каждый процесс хранит порядка n/numprocs строк матрицы коэффициентов, векторы B и X. Можно сказать, что вычислительная загрузка процессов выравнена.

Для обнуления коэффициентов при x1 процесс, содержащий первую строку матрицы (нулевой), рассылает её всем остальным процессам, они, в свою очередь, вычитают её из своей строки (при этом домножая её на соответствующее число).

Для обнуления коэффициентов при x2 процесс, содержащий вторую строку матрицы (первый), рассылает её всем остальным процессам, они, в свою очередь, вычитают её из своей строки (при этом домножая её на соответствующее число).

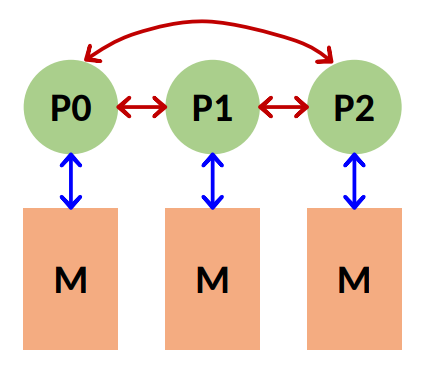
Аналогичные действия выполняются вплоть до строки с номером n-1.

После завершения прямого хода выполняется обратный ход метода Гаусса.

Каждый процесс инициализирует значения вектора решений соответствующими значениями вектора свободных членов СЛАУ, хранящимися в его памяти. После чего процессом, который отвечает за строку n, вычисляется значение xn (инициализированное значение делится на коэффициент при xn), это значение рассылается широковещанием, аналогично (из уравнений) вычисляются остальные значения вектора X. В итоге у каждого процесса имеется готовый вектор решений.

После выполненных вычислений вектор X записывается в файл x\_mpi\_c.txt.

Здесь каждый процессор имеет свою собственную ячейку памяти для доступа и использования, поэтому для их общения все независимые системы будут соединены вместе с помощью сети. MPI основан на распределенной архитектуре:



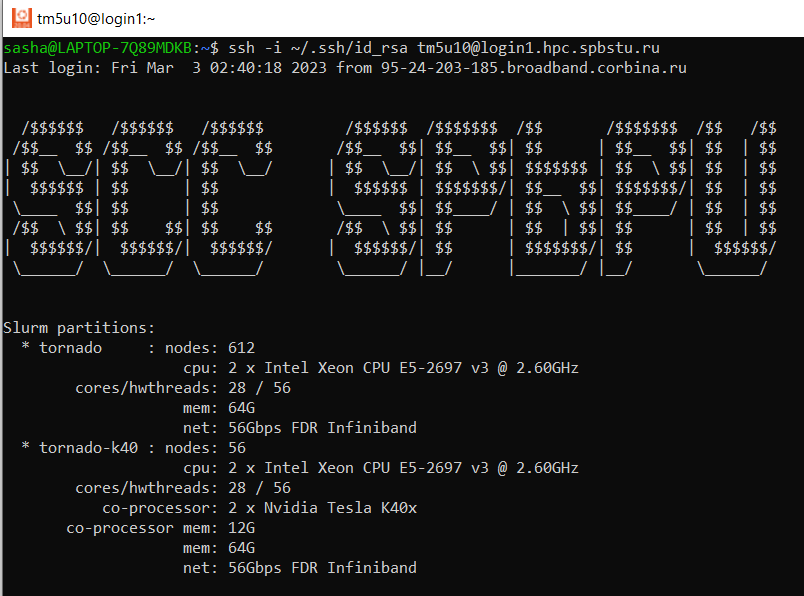
**3.4 Python & MPI**

Для языка python работа программы была простроена аналогичным образом (как описано в реализации алгоритма). Модуль библиотеки python mpi4py предоставляет привязки Python для стандарта MPI. Кроме того, используются функции send и recv (нулевой, главный процесс делит матрицу на массивы строк и передает их остальным процессом с помощью send, подтверждение получения - recv).

Вектор решений X сохраняется в файле mpi\_python.txt.

# **4. Результаты**

Сначала была выполнена авторизация с помощью заранее сгенерированного ключа SSH:

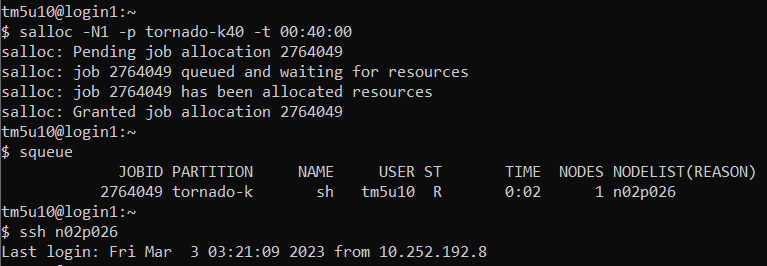


Для дальнейшей работы был выделен один свободный кластер:

salloc -N1 -p tornado-k40 -t 00:40:00

squeue

ssh <NODELIST (REASON)>



Далее были установлены все необходимые модули и сгенерирована расширенная матрица 100x101:

module purge

module add compiler/gcc/11.2.0

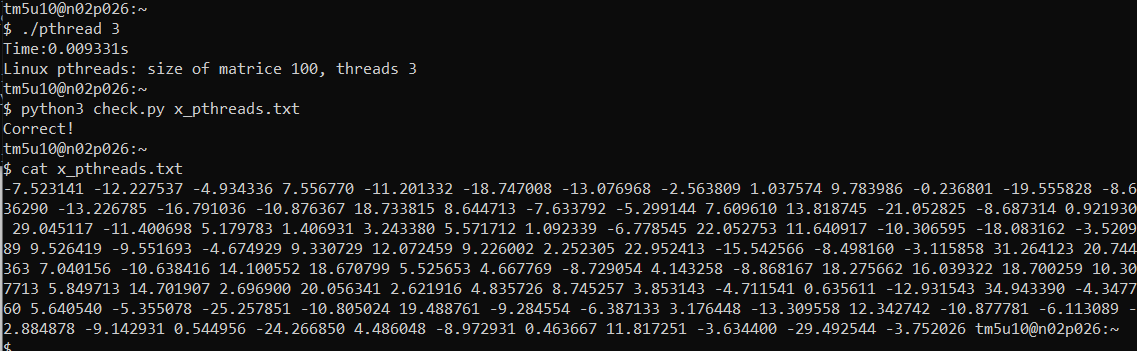
module add python/3.9

module add library/gsl/2.5.0

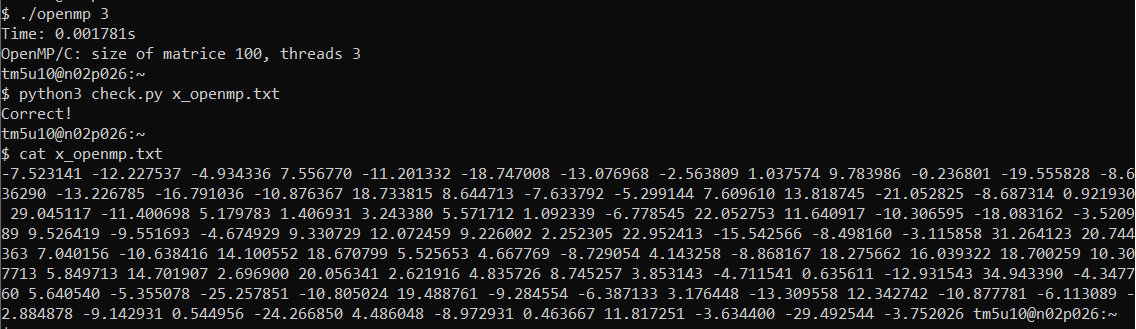
module load mpi/openmpi/4.0.1/gcc/8

python3 gen.py 100

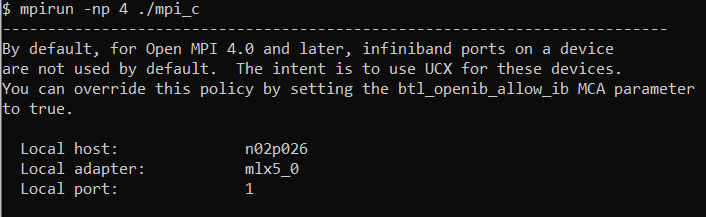
Пример работы pthreads с тремя потоками и тестирование:



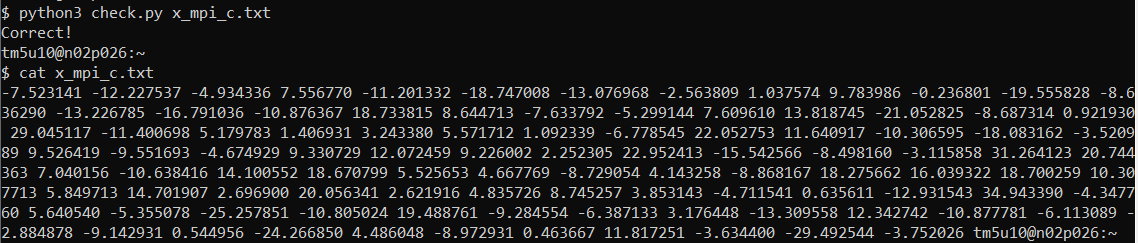
Пример работы для OpenMP с тремя потоками и тестирование:



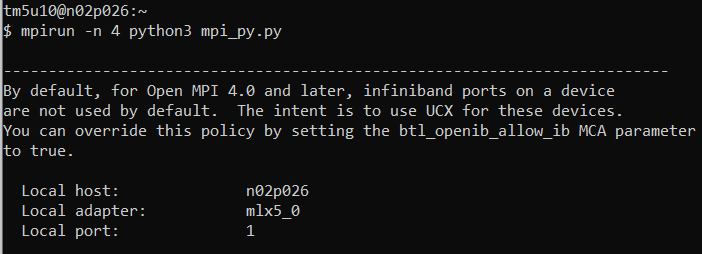
Пример работы MPI\C с 4 процессами и тестирование:







Пример работы для MPI\Python с 4 процессами и тестирование:







# **Тестирование**

Данные технологии были протестированы на одинаковом количестве узлов, процессов (потоков), но на разном размере исходной матрицы.

В таблице 1 представлены результаты времени работы программ (в секундах) для size = 10, 100, 1000, 2000, 5000, одного узла и двух процессов (потоков).

Таблица 1. Результаты тестирования на разных размерах исходной матрицы

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Технология\N** | **10** | **100** | **1000** | **2000** |
| *pthread\C* | 0.000566 | 0.007841 | 0.826806 | 5.575612 |
| *MPI\C* | 0.001038 | 0.000996 | 0.001117 | 4.197391 |
| *MPI\Python* | 0.009416 | 0.454702 | 312.567588 | 703.578321 |
| *OpenMP\C* | 0.000142 | 0.001608 | 0.516058 | 4.281362 |

Как можно наблюдать, на небольшой размерности N наилучшим образом справляется с задачей технологии pthread/C и MPI/C, так как показывают наименьшее время работы программы.

При увеличении размерности матрицы наихудшие результаты показывает технология MPI/Python, время очень увеличивается. Остальные технологии показывают примерно одинаковые результаты, работают намного быстрее MPI/Python.

Для всех исследуемых значений корректно были пройдены проверки результатов (check.py).

Затем каждая технология была протестирована для различных исходных параметров (количество процессов (потоков), узлов (в случае MPI)) и фиксированной размерности исходной матрицы size = 1000 (для случая MPI/Python – size = 500). Результаты приведены в таблицах 2 – 6.

Таблица 2. Результаты тестирования на разном количестве потоков, pthreads

|  |  |
| --- | --- |
| **Количество потоков** | **Время, секунды** |
| 2 | 0.838935 |
| 4 | 0.444882 |
| 8 | 0.354086 |
| 16 | 0.309896 |
| 32 | 0.297687 |
| 64 | 0.275898 |
| 100 | 0.264837 |

Таблица 3. Результаты тестирования на разном количестве процессов, MPI/C

|  |  |
| --- | --- |
| **Количество процессов** | **Время, секунды** |
| 2 | 0.514167 |
| 4 | 0.268094 |
| 5 | 0.241454 |
| 10 | 0.134659 |
| 20 | 0.095926 |
| 25 | 0.093050 |

Таблица 4. Результаты тестирования на разном количестве процессов, MPI/Python

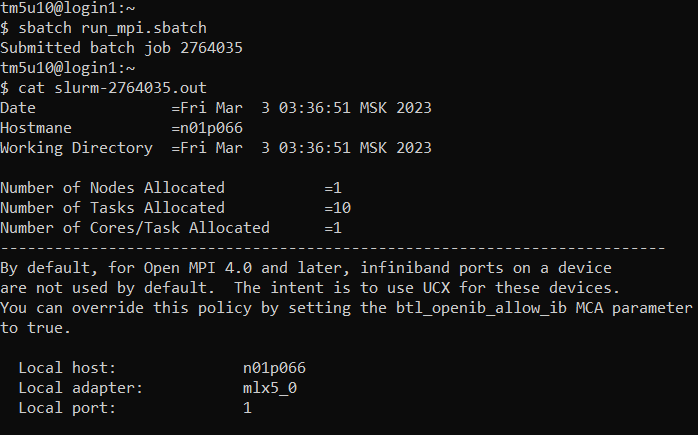
|  |  |
| --- | --- |
| **Количество процессов** | **Время, секунды** |
| 2 | 38.073034 |
| 4 | 20.891607 |
| 5 | 18.341000 |
| 10 | 11.756802 |
| 20 | 8.183309 |
| 25 | 7.642711 |

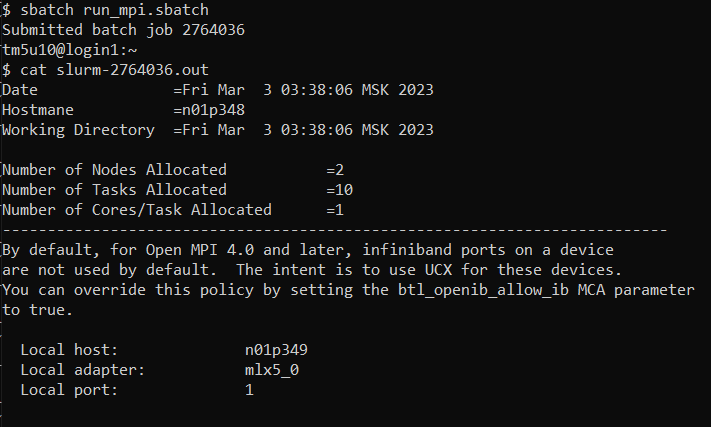
Таблица 5. Результаты тестирования на разном количестве потоков, OpenMP

|  |  |
| --- | --- |
| **Количество потоков** | **Время, секунды** |
| 2 | 0.512725 |
| 4 | 0.295297 |
| 8 | 0.162143 |
| 10 | 0.134547 |
| 16 | 0.108564 |
| 20 | 0.097993 |
| 25 | 0.092283 |

Таблица 6. Результаты тестирования MPI для различного количества узлов и 10 процессов

|  |  |  |
| --- | --- | --- |
| **Количество узлов** | **MPI/Python** | **MPI/C** |
| 1 | 11.810336112976074 | 0.272955 |
| 2 | 11.705430269241333 | 0.272390 |
| 3 | 11.669542789459229 | 0.237467 |
| 4 | 11.478615045547485 | 0.231128 |





# **Вывод**

При выполнении данной лабораторной работы был разработан алгоритм для решения систем уравнений с помощью верхней треугольной матрицы, использующий параллельные вычисления. Алгоритм был реализован для следующих технологий:

* C & Linux pthreads
* С & MPI
* Python & MPI
* С & OpenMP

Каждая программная реализация была протестирована для различных значений размерности исходных матриц, потоков, узлов, процессов.

Для матриц небольшой размерности (size ≤ 100) наилучшие (самые быстрые) результаты показали технологии pthread/C и MPI/C. Для матриц размерности size > 100 наилучшие результаты показали технологии Pthreads, OpenMP и MPI/C.

При изменении количества потоков/процессов наблюдалось, что при их увеличении производительность программы улучшилась.

Изменение количества узлов (для технологии MPI) слабо повлияло на производительность программ (улучшение наблюдалось для технологии MPI/C).

Корректная работа всех программ была проверена с помощью теста check.py.

Все использованные файлы программ и скриптов представлены в репозитории Github по ссылке: [GitHub - SashaObidina/ParallelProgramming: Parallel methods of solving equations systems using a triangular matrix.](https://github.com/SashaObidina/ParallelProgramming)