Министерство науки и высшего образования Российской Федерации
Санкт-Петербургский Политехнический Университет Петра Великого
—
Институт компьютерных наук и кибербезопасности
Высшая школа технологий искусственного интеллекта

# Отчёт

по заданию «Параллельная сортировка»

по дисциплине «Параллельное программирование на суперкомпьютерных системах»

Выполнил: студент группы
5140201/30301 В.А. Ефременко

*<подпись>*

Проверил: А.А. Лукашин
д.т.н., профессор

*<подпись>*

Санкт-Петербург
2024

# 1 Постановка задачи

1. Выбрать задачу и проработать реализацию алгоритма, допускающего распараллеливание 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.

Вариант задания: Параллельная сортировка

Технологии:

- C & Linux pthreads
- C & MPI
- Python & MPI
- C & OpenMP

# 2 Описание алгоритма

В качестве алгоритма параллельной сортировки был выбран алгоритм быстрой сортировки QuickSort.

*Базовый алгоритм быстрой сортировки:*

1) Из массива выбирается опорный элемент (чаще – серединный)
2) Массив разделяется на 2 подмассива: один содержит элементы меньше опорного, другой – больше.
3) Рекурсивно применяем этот алгоритм к 2 подмассивам
4) Когда рекурсивные вызовы завершаются, объединяем подмассивы и опорный элемент, получая полностью отсортированный массив.

Затраты памяти – O(n). Сложность алгоритма в худшем случае O(n^2), в среднем – O(n log n).

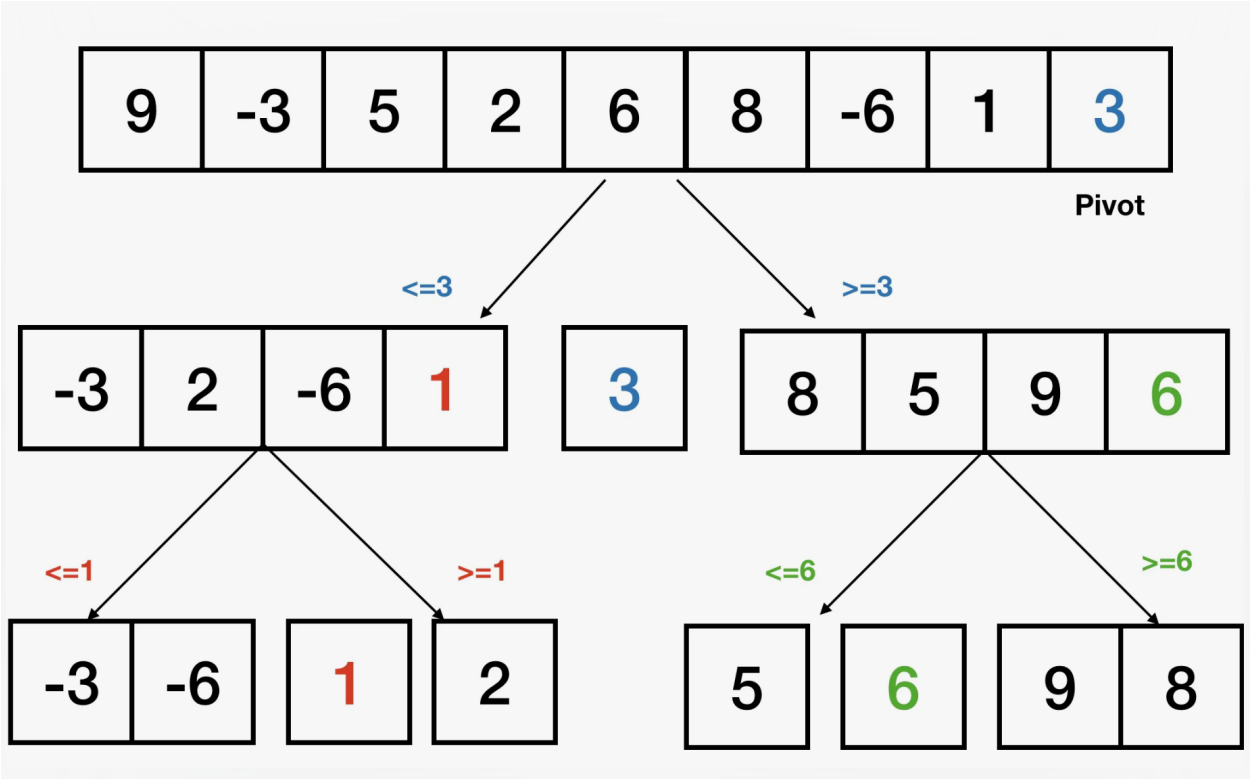



Рис 1 Пример работы алгоритма быстрой сортировки

*Алгоритм параллельной быстрой сортировки:*

Исходный массив разделяется на $n$ более мелких подмассивов, направляемых к $p$ удаленным процессам для параллельного выполнения. По завершении выполнения на удаленных процессах отсортированные подмассивы отправляются обратно на центральный узел обработки, который объединяет результаты, создавая полностью отсортированный массив.

Шаги алгоритма:

1. Разделить массив размером $n$ на несколько подмассивов, совместимых с числом доступных процессоров $p$.
2. Создать $p$ потоков в соответствии с количеством доступных процессоров.
3. Назначить подмассив каждому из $p$ потоков так, чтобы каждый имел $n/p$ последовательных элементов из исходного массива.
4. Случайным образом выбрать опорный элемент и передать его всем соответствующим процессам.
5. Каждый процесс разбивает свои элементы и делит их на две группы в соответствии с выбранным опорным элементом, как в базовом алгоритме. Это происходит параллельно на всех процессах одновременно.
6. Каждый процесс в верхней половине списка процессов отправляет свой массив с меньшими значениями процессу в нижней половине списка процессов и в ответ получает список с большими значениями.
7. Нижняя половина будет содержать значения, меньшие опорного, а верхняя половина - значения, большие опорного.
8. После $logP$ рекурсий у каждого процесса есть неотсортированный подмассив, не пересекающийся со значениями других процессов.

9. На этом этапе подмассив достаточно мал, каждый процесс сортирует свои значения последовательно, и основной процесс объединяет отсортированные результаты для каждого процесса.

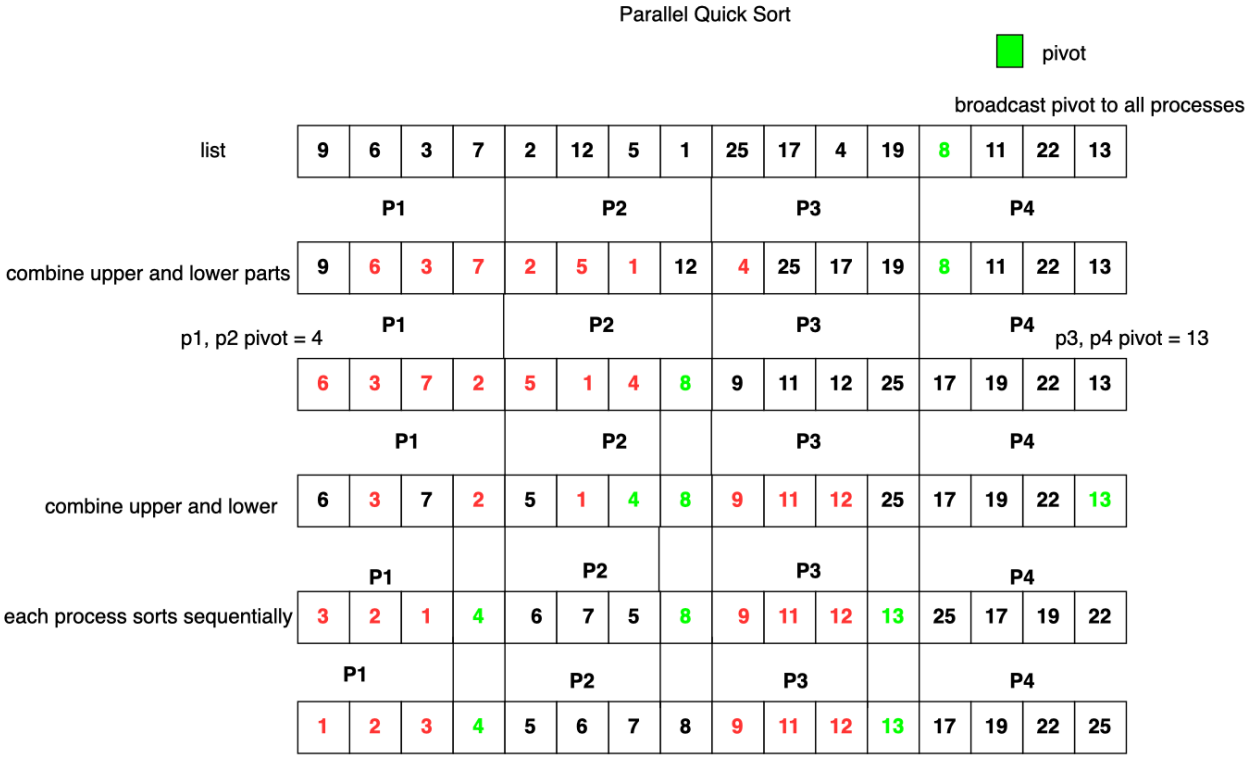



Рис 2 Пример работы алгоритма параллельной быстрой сортировки

Общая временная сложность алгоритма составляет $O(n/p * \log n)$.

Особенности алгоритма:

– Алгоритм плохо справляется с балансировкой нагрузки, поэтому важен выбор подходящего среднего значения в качестве опорного элемента для сохранения баланса.

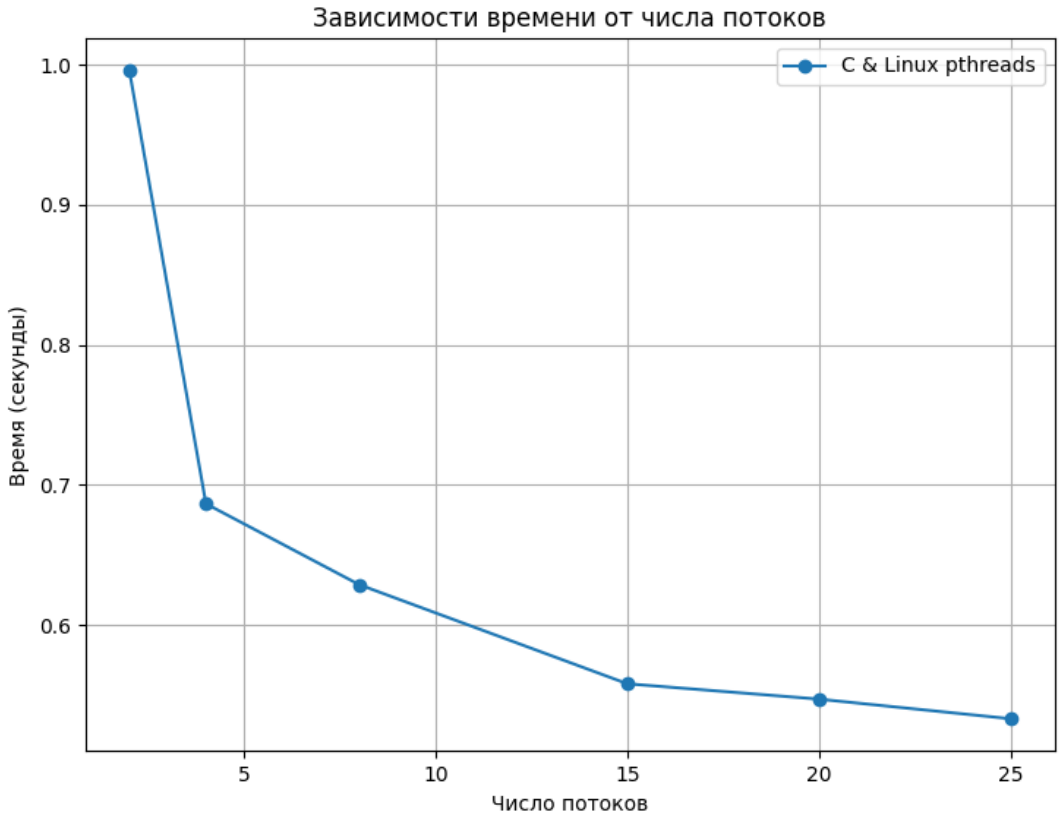
– Каждый блок объединяется в порядке процесса, определяя, где начинается и заканчивается каждый блок, и соединяя его конец с началом следующего процесса.

# 3 Тестирование алгоритма

Тестирование всех реализаций алгоритма осуществлялось на узлах кластера tornado.

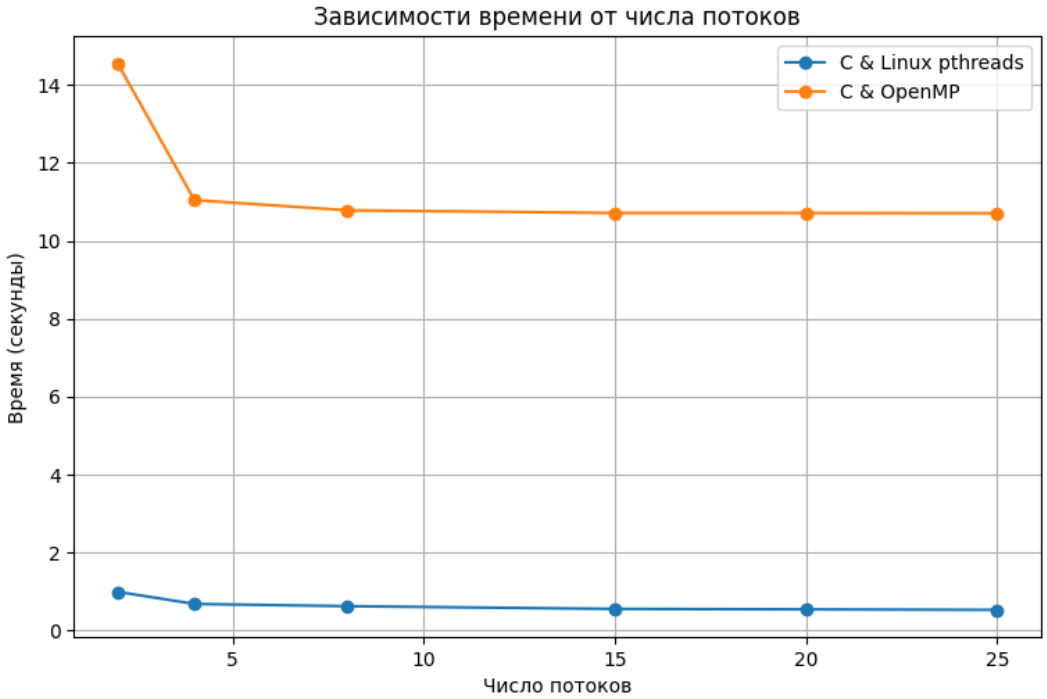
- C & Linux pthreads

Программа реализации pthreads создает необходимое число потоков. Каждый поток считает свою часть массива, и когда расчеты всех потоков заканчиваются, эти части объединяются. Ниже приведена зависимость времени от числа потоков, тестирование производилось на массиве размером n=10^7, число потоков – 2, 4, 8, 15, 20, 25:



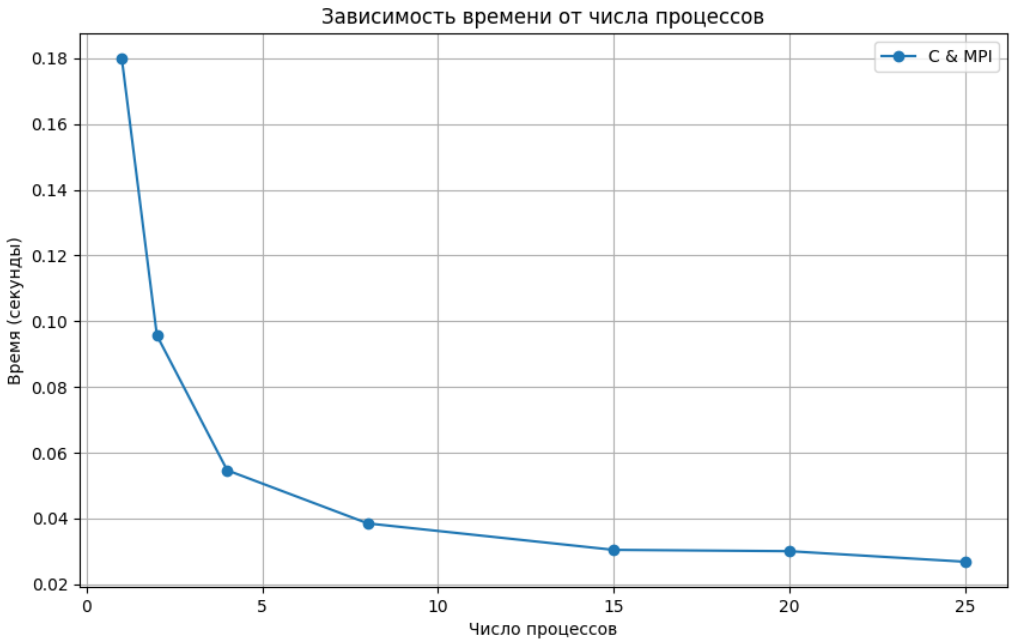

- C & OpenMP

Реализация – все потоки выполняют задачи из очереди задач, полученной от основного потока программы. Ниже приведена зависимость времени от числа потоков и сравнение с pthreads, тестирование также производилось на массиве размером n=10^7, число потоков – 2, 4, 8, 15, 20, 25:



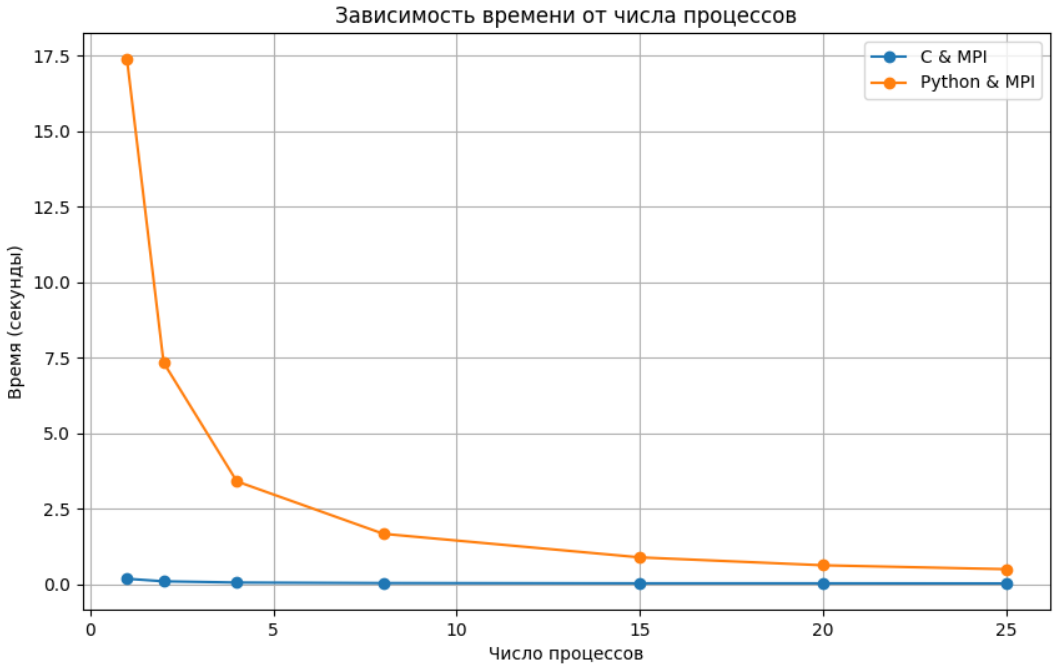

- C & MPI

Массив данных делится на части, которые нужно отсортировать. Каждая часть отправляется главным процессом на другие процессы, где они сортируют свою часть и, отсортировав, отправляют назад для слияния главным процессом. Оценка зависимости времени от числа процессов для C & MPI проводилась на массиве размера n=10^6, число процессов – 1, 2, 4, 8, 15, 20, 25:



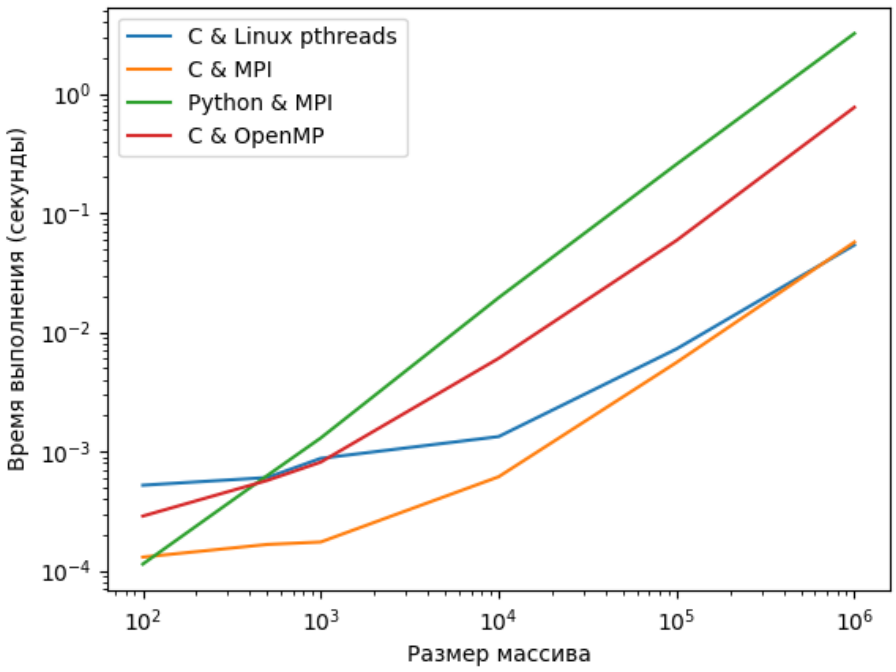
- Python & MPI

Аналогично C & MPI, ниже представлено сравнение их зависимостей от числа процессов с параметрами, указанными выше для C & MPI:




- Зависимость от размера массива

Также было проведено тестирование программ для разных размеров массива. На вход каждой программе поддавались одинаковые массивы разной размерности для каждого эксперимента – 100, 500, 10^3, 10^4, 10^5, 10^6. Запуск производился на 1 узле при одинаковом числе процессов и потоков, равном 4. На графике представлено сравнение всех подходов:




- MPI – зависимость от числа узлов (от 1 до 4, при фиксированном числе процессов 12)

C & MPI:

1 узел: 0.035893 secs

2 узла: 0.036545 secs

3 узла: 0.036072 secs

4 узла:  0.035754 secs

(Результат, полученный для 3 узлов, 30 процессов: 0.027694)

Python & MPI:

1 узел: 1.075608 secs

2 узла: 0.838063 secs

3 узла: 0.811741 secs

4 узла: 0.895692 secs

# Выводы

В результате данной работы был реализован алгоритм быстрой сортировки. Для реализации параллельных вычислений были использованы разные технологии, такие как C & Linux pthreads, C & MPI, Python & MPI, C & OpenMP. Для каждой из них были проведены тесты на зависимость времени их выполнения от различных параметров – числа процессов, потоков, узлов, размера массива, и также проведена проверка на правильность результата, которая была успешной для всех случаев.

При исследовании зависимости от числа процессов/потоков наилучшим образом себя проявила технология C & MPI, показав наименьшее время выполнения. Также, если говорить об общих результатах, с увеличением числа процессов/потоков все реализации показали улучшение производительности.

При изменении размера массива, как и ожидалось, наблюдалось увеличение времени работы программ, причем Python & MPI имеет самый быстрый рост среди всех. На массивах меньшей размерности самой быстрой также была технология C & MPI, но ее рост оказался быстрее чем у многопоточной C & Linux pthreads с увеличением размера массива.

При тестировании MPI технологий, наблюдалось, что изменение числа узлов не сильно влияет на производительность алгоритма. Но если брать большее число процессов, чем доступно при 1 узле, то производительность можно улучшить.

Реализация алгоритмов:
https://github.com/Eva53/Parallel-Programming/tree/main