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

ЛАБОРАТОРНАЯ РАБОТА

«Параллельная сортировка»

по дисциплине «Параллельное программирование на суперкомпьютерных системах»

Выполнил: студент группы
3540201/20302 С.А. Ляхова

<подпись>

Проверил: А.А. Лукашин
доцент, к.т.н.

<подпись>

Санкт-Петербург
2023

Содержание

1. Постановка задачи 3
2. Теоретическая часть 4
3. Реализация 7
4. Результаты 11
5. Тестирование 13
6. Выводы 15

Приложение 1 16
Приложение 2 20
Приложение 3 22
Приложение 4 24
Приложение 5 28
Приложение 6 29

1. Постановка задачи

- Выбрать задачу и проработать реализацию алгоритма, допускающего распараллеливание на несколько потоков/процессов.
- 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).
- Реализовать алгоритмы с использованием выбранных технологий.
- Провести исследование эффекта от использования многоядерности / многопоточности / многопроцессности на СКЦ, варьируя узлы от 1 до 4 (для MPI) и варьируя количество процессов / потоков.
- Подготовить отчет в электронном виде.

Прикладная задача: параллельная сортировка.

Технологии:

1. C & Linux pthreads
2. C & MPI
3. Python & MPI
4. C & OpenMP

2. Теоретическая часть

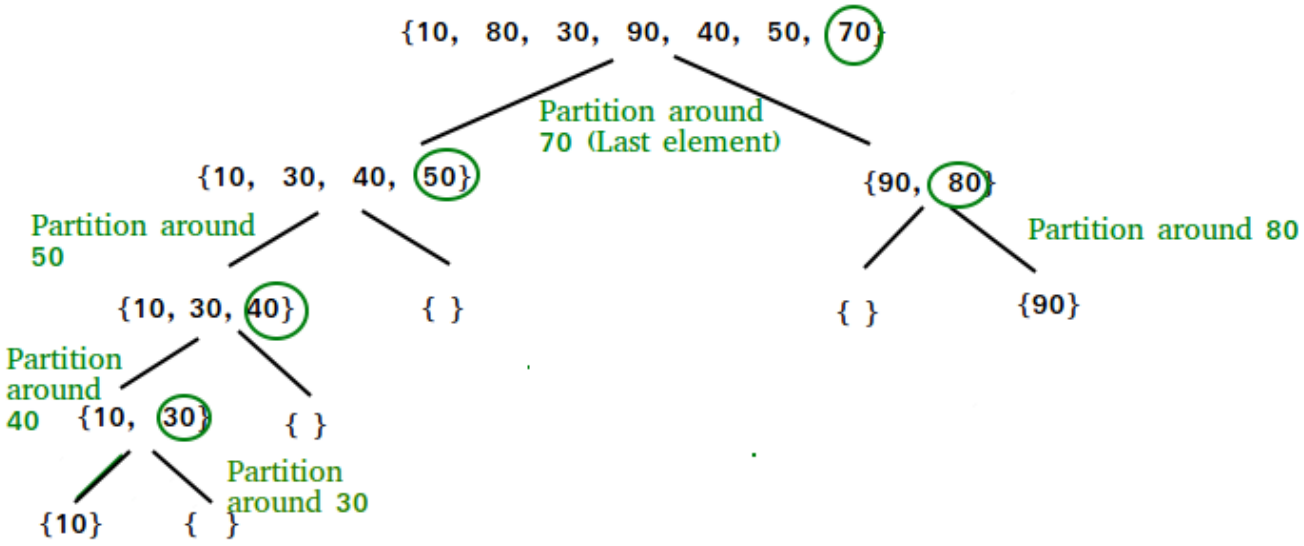
Алгоритм QuickSort известен как один из самых быстрых и эффективных алгоритмов сортировки. Быстрая сортировка использует подход "разделяй и властвуй". Его аспекты разделения делают QuickSort возможным для распараллеливания.

Последовательный алгоритм быстрой сортировки

Исходный несортированный список сначала делится на два подсписка таким образом, чтобы все элементы в первом подсписке были меньше, чем все элементы во втором подсписке. Это достигается путем выбора одного элемента, называемого pivot (опорный), с которым сравниваются все остальные элементы (разделяющим элементом может быть любой элемент в списке, но часто выбираются первый или последний элементы).

Затем к каждому подсписку применяется один и тот же метод разделения до тех пор, пока не получатся отдельные элементы. Из подсписков с надлежащим упорядочением получается окончательный отсортированный список.

Сложность алгоритма: O(nlogn), в худшем случае O(n^2). Затраты памяти O(n).



Параллельный алгоритм быстрой сортировки

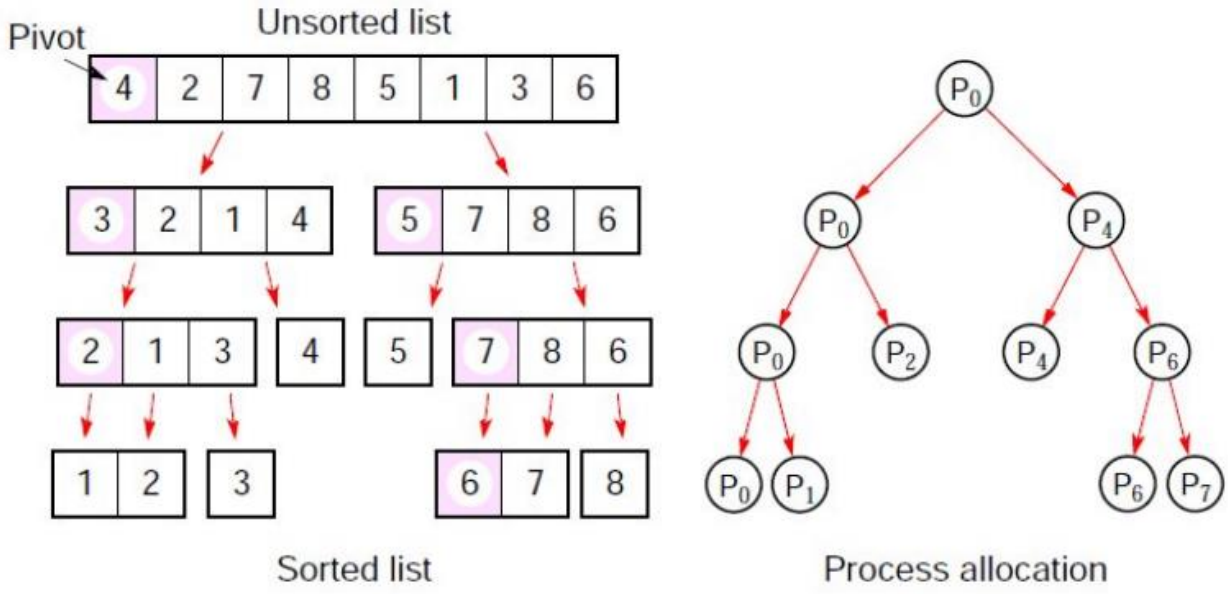
Параллельное программирование заключается в том, что программа разбивается на параллельные программы, которые выполняются одновременно в нескольких потоках процессора. Здесь требуется согласование или же синхронизация.

Рассмотрим случай распределенной памяти. Каждый процесс содержит сегмент несортированного списка. Несортированный список равномерно распределен между процессами.

Желаемый результат параллельного алгоритма быстрой сортировки:

- Сегмент списка, хранящийся в каждом процессе, сортируется
- Последний элемент в списке процесса i меньше, чем первый элемент в списке процесса i + 1

Идея распараллеливания состоит в том, чтобы воспользоваться древовидной структурой алгоритма для распределения работы между процессами.



Распределение процессов в древовидной структуре приводит к двум фундаментальным проблемам:

- В общем, дерево разделов не идеально сбалансировано (выбор хорошего разделяющего элемента имеет решающее значение для эффективности).
- Процесс распределения работы между процессами серьезно ограничивает эффективное использование доступных процессов (начальный раздел включает в себя только один процесс, затем второй раздел включает в себя два процесса, четыре процесса и так далее)

Опять же, если игнорировать время связи и считать, что выбор разделяющего элемента идеален, т.е. создает подсписки одинакового размера, то для этого требуется

$$\sum_{i=0}^{\log(n)} \frac{n}{2^i} \approx 2n$$

шагов для получения окончательного отсортированного списка в параллельной реализации, что соответствует временной сложности O(n). Выбор разделяющего элемента в наихудшем случае вырождается до временной сложности O(n^2).

Таким образом, лучшее, к чему можно стремиться с помощью алгоритма параллельной сортировки, использующего n процессорных блоков, - это временная сложность O(n log(n))/n = O(log(n)).

Обычно чисел (n) намного больше, чем блоков обработки (p), и в таких случаях каждому блоку обработки должен быть присвоен список из n/p номеров.

3. Реализация

Для создания исходных файлов была написана программа *gen_data.py*, генерирующая массив размерностью N и записывающая его в файл input.txt.

Было принято решение проводить тестирование с помощью средств для работы с массивами для языка программирования Python – *numpy*. Для этих целей была разработана программа *test.py*. Она загружает исходный массив и полученные в результате работы программ сортировки, после чего сверяет их с результатами работы метода *numpy.sort()*.

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

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

Для каждой технологии использовались свои методы, однако алгоритм действий отличался не сильно.

3.1. C & MPI

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

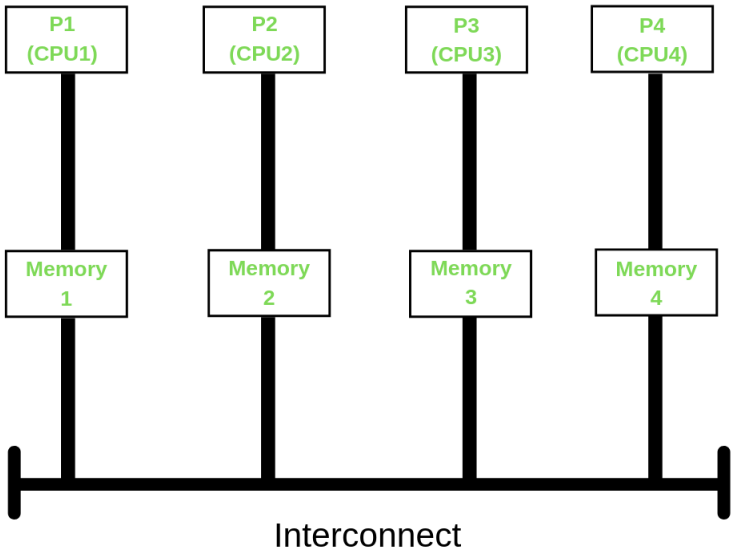
Каждому процессу присваивается раздел последовательности данных, подлежащих сортировке. Это делается путем выполнения операции разброса главным процессом. Каждый процесс работает со своей подпоследовательностью индивидуально, используя алгоритм быстрой сортировки, описанный в разделе 2, и отправляет свою отсортированную подпоследовательность для слияния.

Основное преимущество этого метода заключается в том, что время связи сокращается за счет отправки только подпоследовательности каждому процессу. Но это не оптимизированное решение.

Недостатки такой методики заключаются в следующем:

- Балансировка нагрузки не достигается, так как конкретный процесс может завершить свою сортировку и дожидаться операции слияния, пока другие процессы все еще сортируют свои подпоследовательности. Это приводит к простою процесса.
- Операции слияния, выполняемые для каждой отсортированной подпоследовательности, также требуют больших вычислительных затрат.

Здесь каждый процессор будет иметь свою собственную ячейку памяти для доступа и использования. Чтобы заставить их общаться, все независимые системы будут соединены вместе с помощью сети. MPI основан на распределенной архитектуре.



Из библиотеки MPI были использованы методы:

- MPI_Init – инициализация MPI.
- MPI_Comm_rank – получение текущего номера процесса.
- MPI_Comm_size – получение количества процессов.
- MPI_Send – отправка данных процессу.
- MPI_Recv – получение данных от другого процесса.
- MPI_Finalize – уничтожение окружения MPI.

После инициализации массива и получения необходимых данных программа делит вычисления между единственным узлом-хозяином (управляющим процессом) и множеством узлов-преемников. При этом узел-хозяин также выполняет свою часть сортировки. После завершения всех вычислений от всех потоков он записывает данные в результирующий массив, после чего она сохраняется в файле *mpi_c.txt.*

3.2. Python & MPI

Для языка Python работа программы была простроена аналогичным образом. Модуль библиотеки Python *mpi4py* предоставляет привязки Python для стандарта MPI.

Сортированный массив сохраняется в файле *mpi_python.txt.*

3.3. C & Linux pthreads

Для pthreads используется другой подход, где разделяемая память предоставляет альтернативу, позволяющую избежать операций слияния массивов.

Из библиотеки pthreads были взяты методы:

- pthread_create – функция, создающая поток.
- pthread_join – функция, принимающая данные от потока, который закончил свое выполнение.

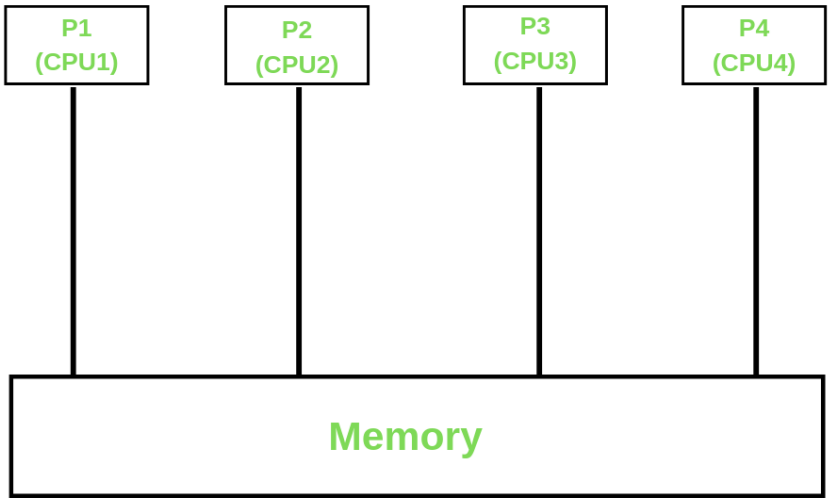
Таким образом, программа создает необходимое количество потоков, после чего каждый из них считает свою часть массива. Главная часть программы дожидается, пока все потоки закончат свое выполнения, и собирает их в выделенный массив данных. Далее полученный массив сохраняется в файл *pthread.txt*.

Основным преимуществом здесь является то, что достигается балансировка нагрузки, поскольку потоки работают над одной и той же последовательностью одновременно. Операция слияния не требуется, но недостатком является наличие критического раздела в коде при распределение

задачи между потоками, и, следовательно, требуются блокировки мьютексов (*pthread_mutex_lock*), которые замедляют процесс.

3.4. C & OpenMP

Open Multi-processing (OpenMP) — это метод распараллеливания разделов кода C/C++/Fortran. OpenMP относится к концепции разделяемой памяти. При этом разные процессоры будут иметь доступ к одной и той же ячейке памяти.



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.

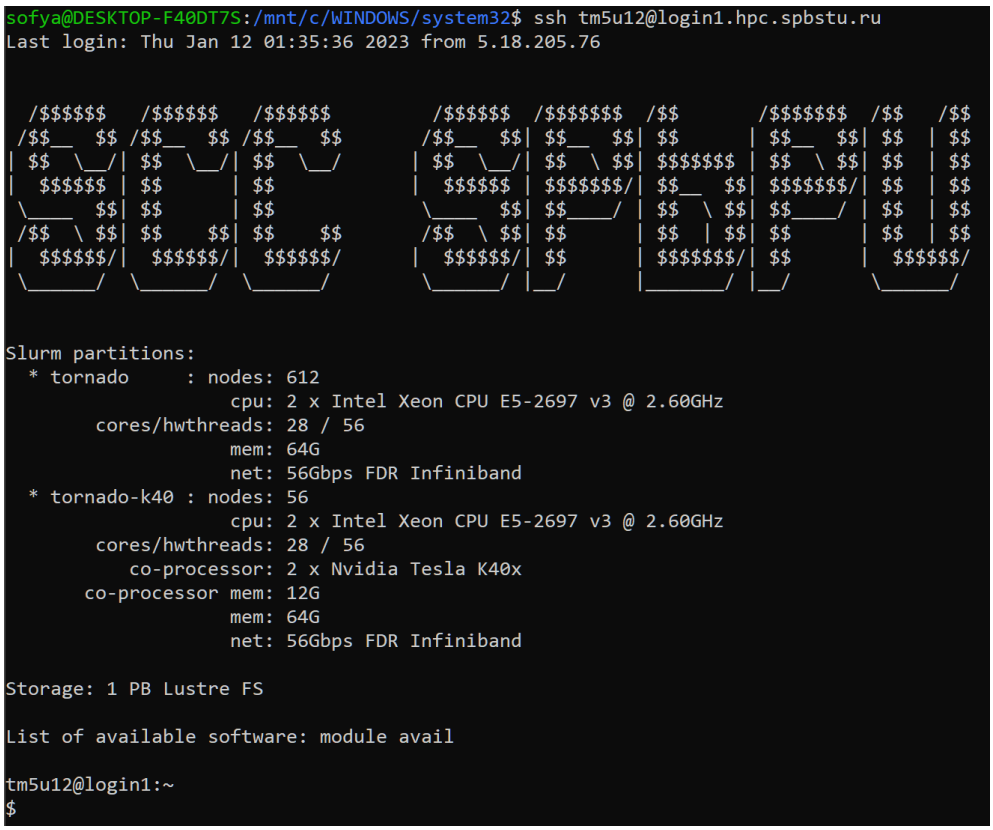
#include<omp.h> — это заголовочный файл для openmp, чтобы использовать его функции.

#pragma omp parallel section num_threads(thread_count) определяет параллельную область, содержащую код, который будет выполняться, используя несколько параллельных потоков. Этот код будет разделен между всеми потоками.

Как только все потоки закончили вычисления, результат записывается в файл *openmp.txt*.

4. Результаты

Сперва была выполнена авторизация с помощью ранее сгенерированного ключа SSH.



Для дальнейшей работы был выделен один свободный кластер

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

squeue

scancel <ID of job>

Сгенерированный массив для сортировки (при помощи написанного генератора):

python3 gen_data.py 1000 input.txt

```
$ python3 gen_data.py 50 input.txt
tm5u12@login1:~/parallel_sort/FINAL
$ cat input.txt
50
65988 91617 53514 71046 24568 39941 94205 81190 95445 58107 45480 86672 64436 65252 53838 222
16 31170 42901 99957 46733 93378 35163 36795 10236 60713 3259 82295 70250 30058 70315 6283 30
007 766 33906 46113 31275 12210 72760 65922 19038 41137 58069 81405 8606 17576 75826 67293 26
536 95856 80058 tm5u12@login1:~/parallel_sort/FINAL
```

Массив также сохраняется в формате *npy* для работы с ним в Python.

Пример работы pthreads:

```
tm5u12@login1:~/parallel_sort/FINAL
$ gcc Quick_sort_pthread.cpp -o Quick_sort_pthread -lpthread -D DEBUG
tm5u12@login1:~/parallel_sort/FINAL
$ ./Quick_sort_pthread 3
pivot is 33
pivot value is 33906
pivot is 0
pivot value is 24568
worker 0 (pthread id 47942404704000) has started
pivot is 8
pivot value is 17576
worker 1 (pthread id 47942406805248) has started
worker 2 (pthread id 47942408906496) has started
Main: completed join with worker 2 (pthread id 47942408906496) having a status of 0
Main: completed join with worker 1 (pthread id 47942406805248) having a status of 0
Main: completed join with worker 0 (pthread id 47942404704000) having a status of 0
Quicksort 50 ints on 3 procs: 0.000692 secs
Sorted array is:
766 3259 6283 8606 10236 12210 17576 19038 22216 24568 26536 30007 30058 31170 31275 33906 35
163 36795 39941 41137 42901 45480 46113 46733 53514 53838 58069 58107 60713 64436 65252 65922
 65988 67293 70250 70315 71046 72760 75826 80058 81190 81405 82295 86672 91617 93378 94205 95
445 95856 99957
```

Пример работы MPI (C):

module load mpi/openmpi/4.0.1/gcc/8

module load mpi/impi/5.1.3.210

```
$ mpirun -np 3 ./Quick_sort_MPI input.txt mpi_c.txt
Quicksort 50 ints on 3 procs: 0.000081 secs
```

```
Total number of Elements given as input : 50
Sorted array is:
766 3259 6283 8606 10236 12210 17576 19038 22216 24568 26536 30007 30058 31170 31275 33906 35
163 36795 39941 41137 42901 45480 46113 46733 53514 53838 58069 58107 60713 64436 65252 65922
 65988 67293 70250 70315 71046 72760 75826 80058 81190 81405 82295 86672 91617 93378 94205 95
445 95856 99957
```

Пример работы MPI (Python):

module load mpi/openmpi/4.0.1/gcc/8

module load mpi/impi/5.1.3.210

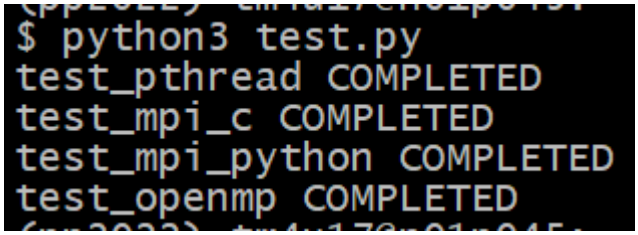
module load python/3.9

```
$ mpirun -np 2 python3 multi_mpi.py
--------------------------------------------------------------------------
By default, for Open MPI 4.0 and later, infiniband ports on a device
are not used by default.  The intent is to use UCX for these devices.
You can override this policy by setting the btl_openib_allow_ib MCA parameter
to true.

  Local host:              n01p045
  Local adapter:           mlx5_0
  Local port:              1

--------------------------------------------------------------------------
--------------------------------------------------------------------------
WARNING: There was an error initializing an OpenFabrics device.

  Local host:   n01p045
  Local device: mlx5_0
--------------------------------------------------------------------------
[1642953212.391493] [n01p045:90122:0]         mxm.c:196  MXM  WARN  The 'ulimit -s' on the system is set to 'u
mplications. Please set the stack size to the default value (10240)
[1642953212.391524] [n01p045:90123:0]         mxm.c:196  MXM  WARN  The 'ulimit -s' on the system is set to 'u
mplications. Please set the stack size to the default value (10240)
Time:0.019006013870239258s
[n01p045.cluster:90117] 1 more process has sent help message help-mpi-btl-openib.txt / ib port not selected
[n01p045.cluster:90117] Set MCA parameter "orte_base_help_aggregate" to 0 to see all help / error messages
[n01p045.cluster:90117] 1 more process has sent help message help-mpi-btl-openib.txt / error in device init
(pp2022) tm4u17@n01p045:~
```

Пример работы OpenMP:

```
$ gcc -fopenmp Quick_sort_OpenMP.cpp -o Quick_sort_OpenMP
(FINAL) tm5u12@n02p002:~/parallel_sort/FINAL
$ ./Quick_sort_OpenMP 3
Quicksort 50 ints on 3 procs: 0.000360 secs
(FINAL) tm5u12@n02p002:~/parallel_sort/FINAL
$ cat openmp.txt
766 3259 6283 8606 10236 12210 17576 19038 22216 24568 26536 30007 30058 31170 31275 33906 35
163 36795 39941 41137 42901 45480 46113 46733 53514 53838 58069 58107 60713 64436 65252 65922
 65988 67293 70250 70315 71046 72760 75826 80058 81190 81405 82295 86672 91617 93378 94205 95
445 95856 99957 (FINAL) tm5u12@n02p002:~/parallel_sort/FINAL
```

Все результаты работы программ выдали верный результат тестирования.



5. Тестирование

Различные технологии были протестированы на одинаковом количестве узлов, процессов и потоков, но на разном размере исходного массива.

В таблице представлены результаты тестирования для N=10, 100, 100000, 500000, 1000000, одного узла и двух процессов (потоков). Время работы программы представлено в секундах:

	10	**100**	**100000**	**500000**	**1000000**
MPI\C	0.000092	0.001104	0.112542	0.764321	1.126917
OpenMP\C	0.000206	0.000296	0.064368	0.253493	0.564943
pthread\C	0.001026	0.000633	0.016227	0.065738	0.082004
MPI\Python	0.000036	0.002150	0.342849	2.1593022	4.501786

По результатам, на небольшой размерности наилучшим образом справляется с задачей технология MPI, показывая наименьшее время работы программы. Однако при увеличении размера массива видно, что время работы MPI растет быстрее (особенно для Python), чем время работы программ, использующих многопоточность (pthreads, OpenMP).

Далее каждая технология была протестирована для различных исходных параметров (количество процессов, потоков, узлов) и фиксированной размерности массива 1000000.

Pthreads:

Количество потоков	**Время, сек**
2	0.135167
4	0.083472
8	0.082692
16	0.080979
32	0.073062
64	0.071201
100	0.068226

OpenMP:

Количество потоков	**Время, сек**
2	0.120472
4	0.075287
8	0.064848
16	0.084794
32	0.142963
64	0.140398
100	0.232306

MPI (C):

Количество процессов	**Время, сек**
2	0.131390
4	0.075161
5	0.069868
10	0.046366
20	0.035710
25	0.031510

MPI (Python):

Количество процессов	**Время, сек**
2	4.501786112785339
4	2.163034498691559
5	2.007234198299561
10	1.8612156629562377
20	1.148496150970459
25	0.858186149597168

Результаты тестирования MPI для различного количества узлов, время указано в секундах:

Количество узлов	**MPI (C)**	**MPI (Python)**
1	1.126917	4.550492
2	1.084888	3.545532
3	1.017309	3.434173
4	1.019141	3.789338

6. Выводы

В ходе лабораторной работы был разработан алгоритм сортировки Quicksort, использующий параллельные вычисления. Алгоритм был реализован для следующих технологий:

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

Каждая программная реализация была протестирована для различных значений размерности исходного массива, потоков, узлов, процессов. Для массивов небольшой размерности наилучшие результаты показала технология MPI, так как имела наименьшее время исполнения программы. Для массивов большой размерности >100000 наилучшие результаты показали технологии Pthreads, OpenMP.

Для случая, когда нужно было изменять количество потоков/процессов, наблюдалось, что при увеличении количества потоков/процессов производительность программы улучшилась. Но изменение количества узлов (для технологии MPI) слабо повлияло на производительность программ:

Для проверки корректной работы программ были разработаны программы генерирующие и проверяющая различные тесты. Для всех рассматриваемых случаев тесты были пройдены.

Приложение 1

Реализация Quicksort на языке C с технологией MPI

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
using namespace std;

// Функция перестановки двух чисел
void swap(int* arr, int i, int j)
{
	int t = arr[i];
	arr[i] = arr[j];
	arr[j] = t;
}

// Функция Quick Sort для массива arr[] с индексом начала start и концом end
void quicksort(int* arr, int start, int end)
{
	int pivot, index;

	if (end <= 1)
		return;

	// Выбираем pivot(как средний элемент) и меняем местами с первым
	pivot = arr[start + end / 2];
	swap(arr, start, start + end / 2);

	// Индекс разделения
	index = start;

	// Выполнить итерацию по диапазону [start, end]
	for (int i = start + 1; i < start + end; i++) {
		// Поменять местами, если элемент меньше, чем элемент pivot
		if (arr[i] < pivot) {
			index++;
			swap(arr, i, index);
		}
	}

	// Установить pivot на место
	swap(arr, start, index);

	// Рекурсивный вызов для подразделов массива
	quicksort(arr, start, index - start);
	quicksort(arr, index + 1, start + end - index - 1);
}

// Функция, которая объединяет два массива
int* merge(int* arr1, int n1, int* arr2, int n2)
{
	int* result = (int*)malloc((n1 + n2) * sizeof(int));
	int i = 0;
	int j = 0;
	int k;

	for (k = 0; k < n1 + n2; k++) {
		if (i >= n1) {
			result[k] = arr2[j];
			j++;
		}
		else if (j >= n2) {
			result[k] = arr1[i];
```

```
                        i++;
                }
                // Индексы в границах, поскольку i < n1 && j < n2
                else if (arr1[i] < arr2[j]) {
                        result[k] = arr1[i];
                        i++;
                }
                // arr2[j] <= arr1[i]
                else {
                        result[k] = arr2[j];
                        j++;
                }
        }
        return result;
}

int main(int argc, char* argv[])
{
        int number_of_elements;
        int* data = NULL;
        int chunk_size, own_chunk_size;
        int* chunk;
        FILE* file = NULL;
        double time_taken;

        MPI_Status status;

        if (argc != 3) {
                printf("2 files required first one input and second one output....\n");
                exit(-1);
        }

        int number_of_process, rank_of_process;
        int rc = MPI_Init(&argc, &argv); // Инициализация MPI

        if (rc != MPI_SUCCESS) {
                printf("Error in creating MPI program.....\n");
                MPI_Abort(MPI_COMM_WORLD, rc);
        }

        MPI_Comm_size(MPI_COMM_WORLD, &number_of_process); // определение числа процессов
        MPI_Comm_rank(MPI_COMM_WORLD, &rank_of_process); // определение номера процесса

        // главный процесс
        if (rank_of_process == 0) {
                // Открытие файла
                file = fopen(argv[1], "r");

                if (file == NULL) {
                        printf("Error in opening file.....\n");
                        exit(-1);
                }

                // Первое значение в файле - это количество элементов
                fscanf(file, "%d", &number_of_elements);
#ifdef DEBUG
                printf("Number of Elements in the file is %d \n", number_of_elements);
#endif

                // Вычисление размера блока
                chunk_size = (number_of_elements % number_of_process == 0) ?
                        (number_of_elements / number_of_process) :
                        (number_of_elements / (number_of_process - 1));

                data = (int*)malloc(number_of_process * chunk_size * sizeof(int));
```

```
		// Чтение самого массива из файла
		for (int i = 0; i < number_of_elements; i++){
			fscanf(file, "%d", &data[i]);
		}

		// Заполнение data нулями
		for (int i = number_of_elements; i < number_of_process * chunk_size; i++){
			data[i] = 0;
		}

#ifdef DEBUG
		printf("Elements in the array is : \n");
		for (int i = 0; i < number_of_elements; i++){
			printf("%d ", data[i]);
		}
		printf("\n");
#endif

		fclose(file);
		file = NULL;
	}

	// Блокирует весь процесс до достижения этой точки (синхронизация)
	MPI_Barrier(MPI_COMM_WORLD);

	// Начало таймера
	time_taken -= MPI_Wtime();

	// Передайте размер всем процессам из корневого процесса
	MPI_Bcast(&number_of_elements, 1, MPI_INT, 0, MPI_COMM_WORLD);

	// Вычисление размера блока
	chunk_size = (number_of_elements % number_of_process == 0) ?
		(number_of_elements / number_of_process) :
		(number_of_elements / (number_of_process - 1));

	// Вычисление общего размера блока
	chunk = (int*)malloc(chunk_size * sizeof(int));

	// Распределение данные о размере патрона по всему процессу
	MPI_Scatter(data, chunk_size, MPI_INT, chunk, chunk_size, MPI_INT, 0,
MPI_COMM_WORLD);
	free(data);
	data = NULL;

	// Вычисляет размер собственного блока, а затем сортирует с помощью quicksort
	own_chunk_size = (number_of_elements >= chunk_size * (rank_of_process + 1)) ?
		chunk_size : (number_of_elements - chunk_size * rank_of_process);

	//  Сортировка массива с quicksort для каждого фрагмента, вызванного процессом
	quicksort(chunk, 0, own_chunk_size);

	//Объединение фрагментов
	for (int step = 1; step < number_of_process; step = 2 * step)
	{
		if (rank_of_process % (2 * step) != 0) {
			MPI_Send(chunk, own_chunk_size, MPI_INT, rank_of_process - step, 0,
MPI_COMM_WORLD);
			break;
		}

		if (rank_of_process + step < number_of_process) {
			int received_chunk_size = (number_of_elements >= chunk_size *
(rank_of_process + 2 * step))
```

```
                        ? (chunk_size * step)
                        : (number_of_elements - chunk_size * (rank_of_process +
step));

                int* chunk_received;
                chunk_received = (int*)malloc(received_chunk_size * sizeof(int));
                MPI_Recv(chunk_received, received_chunk_size,
                        MPI_INT, rank_of_process + step, 0,
                        MPI_COMM_WORLD, &status);

                data = merge(chunk, own_chunk_size, chunk_received,
received_chunk_size);

                free(chunk);
                free(chunk_received);
                chunk = data;
                own_chunk_size = own_chunk_size + received_chunk_size;
            }
        }

        // Остановка таймера
        time_taken += MPI_Wtime();

        // Записать результат в файл
        if (rank_of_process == 0)
        {
            file = fopen(argv[2], "w+b");

            if (file == NULL) {
                printf("Error in opening file... \n");
                exit(-1);
            }

            for (int i = 0; i < own_chunk_size; i++) {
                fprintf(file, "%d ", chunk[i]);
            }

            fclose(file);

#ifdef DEBUG
            printf("Total number of Elements given as input : %d\n",
number_of_elements);
            printf("Sorted array is: \n");

            for (int i = 0; i < number_of_elements; i++) {
                printf("%d ", chunk[i]);
            }

            printf("\n");
#endif

            printf("Quicksort %d ints on %d procs: %f secs\n",
                number_of_elements, number_of_process,time_taken);
        }

        MPI_Finalize();
        return 0;
}
```

Приложение 2

Реализация Quicksort на языке Python с технологией MPI

```
from mpi4py import MPI
import numpy as np
import time
from operator import itemgetter
import psutil

comm = MPI.COMM_WORLD
size = comm.Get_size()
rank = comm.Get_rank()

def partition(array, low, high):
    pivot = array[high]
    i = low - 1

    for j in range(low, high):
        if array[j] <= pivot:
            i = i + 1
            (array[i], array[j]) = (array[j], array[i])

    (array[i + 1], array[high]) = (array[high], array[i + 1])

    return i + 1

def quickSort(array, low, high):
    if low < high:
        pi = partition(array, low, high)

        quickSort(array, low, pi - 1)
        quickSort(array, pi + 1, high)

if __name__ == '__main__':
    if rank == 0:
        numbers = np.load(f'input.npy')
        arraySize = len(numbers)
        chunks = np.array_split(numbers, size)
    else:
        chunks = None

    chunk = comm.scatter(chunks, root=0)

    # Start timer
    start = time.time()

    quickSort(chunk, 0, len(chunk) - 1)

    end = time.time()

    gathered = comm.gather(chunk, root=0)
    elapsed = end - start
    gatheredTime = comm.gather(elapsed, root=0)

    # Sorted array
    Arrays = comm.gather(chunk, root=0)
    if rank == 0:
        iteratorNumbers = np.zeros((len(Arrays),), dtype=int)
        sortedArray = []
        for myIndex in range(0, int(arraySize)):
            iterator = [
                (i, (99999999 if iteratorNumbers[i] >= len(Arrays[i]) else
Arrays[i][iteratorNumbers[i]])) for i
                in range(0, len(Arrays))]
            res = min(iterator, key=itemgetter(1))
```

```
        iteratorNumbers[res[0]] = iteratorNumbers[res[0]] + 1
        sortedArray.append(res[1])
        iterator = []

    np.savetxt('mpi_python.txt', sortedArray, fmt='%3.0d')
    totalTime = sum(gatheredTime)
    averageTime = totalTime / comm.size
    print('Quicksort', arraySize, 'ints on', size, 'procs:', averageTime, 'secs')
```

Приложение 3

Реализация Quicksort на языке C с технологией OpenMP

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <time.h>
#include<sys/time.h>
using namespace std;

// значения могут изменятся из вне другими потоками
volatile int arr_volume = 0;
int* volatile arr;

// Функция перестановки двух чисел
void swap(int* a, int* b)
{
	int t = *a;
	*a = *b;
	*b = t;
}

// Функция для выполнения разделения массива arr[]
int partition(int arr[], int start, int end)
{
	int pivot = arr[end];
	int i = (start - 1);

	// Перестановка массива
	for (int j = start; j <= end - 1; j++) {
		if (arr[j] < pivot) {
			i++;
			swap(&arr[i], &arr[j]);
		}
	}
	swap(&arr[i + 1], &arr[end]);

	// Возвращает соответствующий индекс
	return (i + 1);
}

// Функция для выполнения quicksort с использованием openmp
void quicksort(int arr[], int start, int end, int n)
{
	if (start < end) {

		// Получение индекса pivot путем разделения
		int index = partition(arr, start, end);

		// Параллельная секция
#pragma omp parallel sections num_threads(n)
		{
#pragma omp section
			{
				// Левая половина
				quicksort(arr, start, index - 1, n);
			}
#pragma omp section
			{
				// Правая половина
				quicksort(arr, index + 1, end, n);
			}
		}
	}
}
```

```
int main(int argc, char* argv[])
{
	float time_use = 0;
	struct timeval start_time;
	struct timeval end_time;

	int n = atoi(argv[1]);
	//omp_set_num_threads(n);

	FILE* open_file = fopen("input.txt", "r+");
	if (open_file)
	{
		fscanf(open_file, "%d", &arr_volume);

		arr = (int*)malloc(arr_volume * sizeof(int));
		for (int i = 0; i < arr_volume; i++)
		{
			fscanf(open_file, "%d", &arr[i]);
		}
		fclose(open_file);
	}

	gettimeofday(&start_time, NULL);

	// Вызов быстрой сортировки с параллельной реализацией
	quicksort(arr, 0, arr_volume - 1, n);

	gettimeofday(&end_time, NULL);
	time_use = (end_time.tv_sec - start_time.tv_sec) * 1000000 + (end_time.tv_usec -
start_time.tv_usec);


	FILE* close_file = fopen("openmp.txt", "w+");
	for (int i = 0; i < arr_volume; i++)
		fprintf(close_file, "%d ", arr[i]);

	printf("Quicksort %d ints on %d procs: %lf secs\n",
		arr_volume, n, time_use / 1000000);

	return 0;
}
```

Приложение 4

Реализация Quicksort на языке C с технологией pthread

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <time.h>
#include <sys/time.h>

#define MAX_WORKERS 100
#define MAX_PIVOTS 10

struct Pivot
{
	int index;
	int value;
};

struct WorkerData
{
	int id;
	int* start;
	int n;
	int size;
};

int size_array;
int* g_arrayData;
int num_workers;
pthread_t workers[MAX_WORKERS];

struct WorkerData g_workerData[MAX_WORKERS];
pthread_attr_t g_attr;

int g_activeWorkers = 0;
pthread_mutex_t g_lock;

double g_startTime;
double g_finalTime;

void initWorkerData()
{
	for (int i = 0; i < num_workers; i++) {
		g_workerData[i].id = 0;
		g_workerData[i].start = 0;
		g_workerData[i].n = 0;
		g_workerData[i].size = 0;
	}
}

int compare(const void* a, const void* b)
{
	return (*(int*)a - *(int*)b);
}

// Функция перестановки двух чисел
void swap(int* a, int* b)
{
	int tmp = *a;
	*a = *b;
	*b = tmp;
}

// функция сравнения, которая сортирует значения в порядке возрастания
```

```
int comparePivot(const void* a, const void* b)
{
	return (((struct Pivot*)a)->value - ((struct Pivot*)b)->value);
}

// выберите MAX_PIVOTS случайных pivot и выберите одну из них
int getPivot(int* start, int n)
{
	struct Pivot pivots[MAX_PIVOTS];
	int maxPivots = (MAX_PIVOTS > n) ? n : MAX_PIVOTS;

	for (int i = 0; i < maxPivots; i++){
		int index = rand() % n;
		pivots[i].index = index;
		pivots[i].value = start[index];
	}

	// сортируем pivot
	qsort(&pivots[0], maxPivots, sizeof(struct Pivot), comparePivot);

	int pivot = pivots[maxPivots / 2].index;
#ifdef DEBUG
	printf("pivot is %d\n", pivot);
	printf("pivot value is %d\n", start[pivot]);
#endif
	return pivot;
}

void parallel_quicksort(int* start, int n, int size);

// точка входа потока
void* startThread(void* data)
{
	struct WorkerData* p = (struct WorkerData*)data;
	int id = p->id;

#ifdef DEBUG
	printf("worker %d (pthread id %lu) has started\n", id, pthread_self());
#endif

	parallel_quicksort(p->start, p->n, p->size);
	pthread_exit(0);
}

// Параллельная quicksort принимает массив, начинающийся с указателя start и содержащий
количество элементов size
void parallel_quicksort(int* start, int n, int size)
{
	// блокируем мьютекс, чтобы проверить количество работников
	pthread_mutex_lock(&g_lock);
	if (g_activeWorkers < num_workers){
		// получаем id
		pthread_t worker = g_activeWorkers;
		g_activeWorkers++;
		pthread_mutex_unlock(&g_lock);

		// получаем pivot
		int pivotIndex = getPivot(start, n);
		int right = n - 1;

		// перемещаем pivot в конец
		swap(&start[pivotIndex], &start[right]);

		int storeIndex = 0;
		for (int i = 0; i < right; i++){
```

```
                        if (start[i] < start[right]){
                                swap(&start[i], &start[storeIndex]);
                                storeIndex++;
                        }
                }
                swap(&start[storeIndex], &start[right]);
                pivotIndex = storeIndex;

                // делим текущий массив на два подмассива
                g_workerData[worker].id = worker;
                g_workerData[worker].start = start + pivotIndex;
                g_workerData[worker].n = n - pivotIndex;
                g_workerData[worker].size = size;

                // создаем отдельный поток для работы с одним из вложенных массивов
                pthread_create(&workers[worker], &g_attr, startThread,
(void*)&g_workerData[worker]);

                // этот поток работает с другим вложенным массивом
                parallel_quicksort(start, pivotIndex, size);

                // дождаться завершения другого потока
                void* status;
                int rc = pthread_join(workers[worker], &status);
                if (rc){
                        printf("ERROR; return code from pthread_join() is %d\n", rc);
                        exit(-1);
                }
#ifdef DEBUG
                printf("Main: completed join with worker %ld (pthread id %lu) having a
status of %ld\n",
                        worker, workers[worker], (long)status);
#endif
        }
        else{
                pthread_mutex_unlock(&g_lock);
                // нет доступных потоков, выполняем обычную сортировку
                qsort(start, n, sizeof(int), compare);
        }
}

double readTimer()
{
        static bool initialized = false;
        static struct timeval start;
        struct timeval end;
        if (!initialized){
                gettimeofday(&start, NULL);
                initialized = true;
        }
        gettimeofday(&end, NULL);
        return (end.tv_sec - start.tv_sec) + 1.0e-6 * (end.tv_usec - start.tv_usec);
}

int main(int argc, const char* argv[])
{
        if (argc > 1)
                num_workers = atoi(argv[1]);
        else
                printf("Need number of workers....\n");

        // Инициализируем данные
        initWorkerData();

        // Чтение данных
```

```
	FILE* open_file = fopen("input.txt", "r+");
	if (open_file){
		fscanf(open_file, "%d", &size_array);

		g_arrayData = (int*)malloc(size_array * sizeof(int));
		for (int i = 0; i < size_array; i++){
			fscanf(open_file, "%d", &g_arrayData[i]);
		}
		fclose(open_file);
	}

	// Устанвливаем глобальные атрибуты потока
	pthread_attr_init(&g_attr);
	pthread_attr_setscope(&g_attr, PTHREAD_SCOPE_SYSTEM);
	pthread_attr_setdetachstate(&g_attr, PTHREAD_CREATE_JOINABLE);

	// Инициализируем мьютекс
	pthread_mutex_init(&g_lock, NULL);

	// Инициализируем таймер и установливаем время начала
	g_startTime = readTimer();

	// Начинаем параллельную quicksort
	parallel_quicksort(&g_arrayData[0], size_array, sizeof(int));

	// Остановка таймера
	g_finalTime = readTimer() - g_startTime;

	//printf("The execution time is %g sec\n", g_finalTime);
	printf("Quicksort %d ints on %d procs: %g secs\n",
		size_array, num_workers, g_finalTime);

	// Выводим массив в файл
	FILE* close_file = fopen("pthread.txt", "w+");
	for (int i = 0; i < size_array; i++) {
		fprintf(close_file, "%d ", g_arrayData[i]);
	}

#ifdef DEBUG
	printf("Sorted array is: \n");
	for (int i = 0; i < size_array; i++) {
		printf("%d ", g_arrayData[i]);
	}
	printf("\n");
#endif

	return 0;
}
```

Приложение 5

Генерация данных

```
# Use: ./gen_data.py <input_length_desired> <output_file_name>

import random
import sys
import numpy as np

if len(sys.argv) != 3:
    print ("Usage: ", sys.argv[0], " <length> <output_file>")
    exit()

#output file
f = open(sys.argv[2], 'w')

#length of the list needed
l = sys.argv[1]

f.write(l + "\n")
l = int(l)

array = []

for i in range(l):
    r = random.randrange(100000)
    f.write(str(r) + " ")
    array.append(r)

f.close()

np.save('input.npy', array)
```

Приложение 6

Тестирование результатов

```
import numpy as np


def test_pthread(expected_c):
    actual_c = np.loadtxt("pthread.txt")
    actual_c = [int(item) for item in actual_c]
    if np.array_equal(actual_c, expected_c[0]):
        print("test_pthread COMPLETED")
    else:
        print("test_pthread FAILED")


def test_mpi_c(expected_c):
    actual_c = np.loadtxt("mpi_c.txt")
    actual_c = [int(item) for item in actual_c]
    if np.array_equal(actual_c, expected_c[0]):
        print("test_mpi_c COMPLETED")
    else:
        print("test_mpi_c FAILED")


def test_mpi_python(expected_c):
    actual_c = np.loadtxt("mpi_python.txt")
    actual_c = [int(item) for item in actual_c]
    if np.array_equal(actual_c, expected_c[0]):
        print("test_mpi_python COMPLETED")
    else:
        print("test_mpi_python FAILED")

def test_openmp(expected_c):
    actual_c = np.loadtxt("openmp.txt")
    actual_c = [int(item) for item in actual_c]
    if np.array_equal(actual_c, expected_c[0]):
        print("test_openmp COMPLETED")
    else:
        print("test_openmp FAILED")


if __name__ == '__main__':
    #array = np.loadtxt("input.txt")

    array = []

    with open("input.txt") as f:
        size = f.readline()
        array.append([int(x) for x in f.readline().split()])

    expected_c = np.sort(array)


    test_pthread(expected_c)
    test_mpi_c(expected_c)
    test_mpi_python(expected_c)
    test_openmp(expected_c)

    f.close
```