Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Нижегородский государственный университет им. Н.И. Лобачевского»

**Институт информационных технологий, математики и механики**

**Отчет**

по лабораторной работе №3

**«Сортировка Шелла с простым слиянием»**

**Выполнил:**

студент группы 381608

Нифадьев В.С.

**Проверил:**

Кустикова В.Д.

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#__RefHeading___Toc4084_397246760)

[Метод решения 4](#__RefHeading___Toc4086_397246760)

[Схема распараллеливания 5](#__RefHeading___Toc4088_397246760)

[Описание программной реализации 6](#__RefHeading___Toc4090_397246760)

[Руководство пользователя 6](#__RefHeading___Toc5662_397246760)

[Руководство программиста 6](#__RefHeading___Toc5664_397246760)

[Подтверждение корректности 7](#__RefHeading___Toc4092_397246760)

[Результаты экспериментов 8](#__RefHeading___Toc4094_397246760)

[Заключение 9](#__RefHeading___Toc4096_397246760)

[Приложение 10](#__RefHeading___Toc5668_397246760)

[main.c 10](#__RefHeading___Toc5670_397246760)

[shell\_sort.h 12](#__RefHeading___Toc5672_397246760)

[shell\_sort.c 13](#__RefHeading___Toc5674_397246760)

# Постановка задачи

Имеется неупорядоченный массив, состоящий из N положительных случайных целых чисел от 0 до 100. Необходимо реализовать линейную и параллельную версию алгоритма сортировки Шелла с использованием простого слияния, сравнить время работы версий при различных значениях N и числа процессов, сделать вывод об эффективности использования параллельной версии алгоритма.

# Метод решения

Общая концепция сортировки Шелла – сравнение 2 значений, расположенных в массиве довольно далеко друг от друга. С каждым проходом расстояние между сравниваемыми элементами уменьшается и на последнем проходе равно 1 (то есть сравниваются соседние элементы).

Этапы алгоритма:

* На первом этапе алгоритма элементы N / 2 пар вида (, ) при 1 ≤ i ≤ N / 2 отсортированы
* На втором этапе элементы в N / 4 группах по четыре элемента (, , , ) при 1 ≤ i ≤ N / 4 сортируются
* …
* На последнем этапе элементы всего массива (, ,…, ) отсортированы

Число этапов сортировки Шелла –

Общая сложность алгоритма - O()

# Схема распараллеливания

Для распараллеливания алгоритма используется метод простого слияния.

Этапы параллельной версии:

* Рассылка каждому процессу подмассива, длина которого зависит от числа используемых процессов
  + Если невозможно равномерно разделить исходный массив на равные подмассивы, то последний процесс получает подмассив большего размера ( с учетом оставшихся элементов)
* Каждый процесс сортирует свой подмассив, используя последовательный алгоритм сортировки Шелла
* Слияние отсортированных подмассивов в единый результирующий массив
  + В цикле вводе так называемы шаг, увеличивающийся в 2 раза после каждой итерации и изначально равный 1
  + На первой итерации цикла сливаются подмассивы в соседних процессах (подмассив нулевого процесса с первым, подмассив второго с третьим и так далее) . Полученный массив хранится на четном процессе
  + На второй итерации сливаются подмассивы процессов, у которых разность рангов равна шагу (то есть двум)
  + …
  + На последней итерации сливаются 2 подмассива размера N / 2. В итоге в нулевом процессе получается результирующий отсортированный массив
  + Выход из цикла происходит, когда размер шага становится больше числа процессов

# Описание программной реализации

## Руководство пользователя

Для запуска программы необходимо ввести в консоли следующую команду:

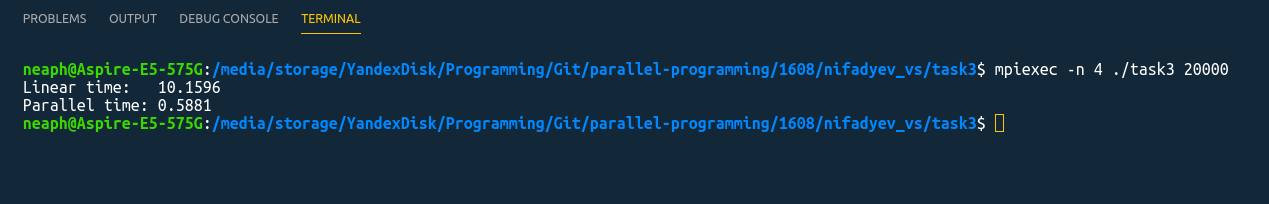
mpiexec -n <process\_number> <path\_to\_binary\_file> <array\_length>

Аргумент <array\_length> является необязательным, при его отсутствии размер массива будет равен 10 000

## Руководство программиста

Код программы можно посмотреть в разделе «[Приложение](#_Приложение)».

# Подтверждение корректности

Для подтверждения корректности в программе используется функция AreResultsEqual(…). В нулевом процессе происходит поэлементное сравнение массива, отсортированного в линейной версии, с массивом, полученным в параллельной версии.

*Рисунок 1. Демонстрация результата выполнения программы*

# Результаты экспериментов

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | N = 1 000 | N = 5 000 | N = 10 000 | N = 20 000 |
| Линейная версия, секунды | 0.0159 | 0.514 | 2.2839 | 10.2632 |
| Параллельная версия, секунды | 0.008 | 0.0316 | 0.1355 | 0.8226 |

*Таблица 1. Время выполнения линейной и параллельной версий*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | N = 1 000 | N = 5 000 | N = 10 000 | N = 20 000 |
| Ускорение параллельной версии | 1,9875 | 16,2658 | 16,8553 | 12,476537807 |

*Таблица 2. Ускорение параллельной версии*

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Р = 2 | Р = 3 | Р = 4 | Р = 8 | Р = 16 | Р = 32 |
| Время выполнения, секунды | 2.0085 | 1.3001 | 0.7861 | 0.4058 | 0.2478 | 0.3497 |

*Таблица 3. Время выполнения параллельной версии в зависимости от числа процессов*

Каждый опыт производился 3 раза, в таблицах указаны средние значения опытов

Эксперименты:

1. В параллельной версии используется 4 процесса
2. Формула вычисления ускорения: S(p) = Ts / Tp
3. Опыты проводились на компьютере с 4 физическими ядрами и размере массива равном 20 000

# Заключение

Во время выполнения лабораторной работы были реализованы линейный и параллельный алгоритмы сортировки Шелла для положительных чисел.

На основе данных, полученных в ходе выполнения экспериментов, можно сделать следующие выводы:

* Распаралелливание алгоритма сортировки Шелла имеет смысл при размере неупорядоченного массива большем, чем 1 000
* Оптимальное ускорение при достаточно большом размере массива достигнуто при количестве процессов, равному 16
* Параллельная версия сортировки Шелла при любом числе процессов, использованном в экспериментах, оказалась эффективнее линейной версии

# Приложение

## main.c

#include "shell\_sort.h"

// Task # 14: Shell sort with simple merge

int main(int argc, char \*\*argv)

{

int procRank, procNum;

int length;

int \*linearResultingArray, \*parallelResultingArray;

double startTime = 0.0, endTime = 0.0;

double linearTime = 0.0, parallelTime = 0.0;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &procNum);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &procRank);

if (procRank == 0)

{

length = (argc == 2) ? atoi(argv[1]) : 10000;

linearResultingArray = (int \*)malloc(sizeof(int) \* length);

GenerateArray(linearResultingArray, length);

parallelResultingArray = (int \*)malloc(sizeof(int) \* length);

}

ParallelShellSort(linearResultingArray, length, procNum, procRank,

parallelTime, parallelResultingArray);

if (procRank == 0)

{

startTime = MPI\_Wtime();

ShellSort(linearResultingArray, length);

endTime = MPI\_Wtime();

linearTime = endTime - startTime;

if (length < 30)

{

printf("Initial array: ");

PrintArray(linearResultingArray, length);

printf("Sorted array (parallel algorithm): ");

PrintArray(parallelResultingArray, length);

printf("Sorted array (linear algorithm): ");

PrintArray(linearResultingArray, length);

}

printf("Linear time: %.4f\n", linearTime);

printf("Parallel time: %.4f\n", parallelTime);

if (AreResultsEqual(linearResultingArray, parallelResultingArray, length))

{

printf("Results are not equal\n");

}

free(linearResultingArray);

free(parallelResultingArray);

}

MPI\_Finalize();

return 0;

}

## shell\_sort.h

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <mpich/mpi.h>

void GenerateArray(int \*array, const int length);

void PrintArray(int \*array, const int length);

void CopyArray(int \*sourceArray, int \*destinationArray, const int length);

int AreResultsEqual(int \*linearResultingArray, int \*parallelResultingArray,

const int length);

void ShellSort(int \*array, const int length);

int \*Merge(int \*firstArray, const int firstArrayLength, int \*secondArray,

const int secondArrayLength);

void ParallelShellSort(int \*array, int length, const int procNum,

const int procRank, double &time, int \*resultingArray);

## shell\_sort.c

#include "shell\_sort.h"

void GenerateArray(int \*array, const int length)

{

int i;

srand(time(NULL));

for (i = 0; i < length; i++)

{

array[i] = rand() % 100;

}

}

void PrintArray(int \*array, const int length)

{

int i;

for (i = 0; i < length; i++)

{

printf("%d ", array[i]);

}

printf("\n");

}

void CopyArray(int \*sourceArray, int \*destinationArray, const int length)

{

int i;

for (i = 0; i < length; i++)

{

destinationArray[i] = sourceArray[i];

}

}

void ShellSort(int \*array, const int length)

{

int gap; // Distance between compared array elements

int i, j, k, temp;

for (gap = length / 2; gap > 0; gap /= 2)

{

for (i = gap; i < length; i++)

{

j = i - gap;

while (j >= 0)

{

k = j + gap;

if (array[j] > array[k])

{

temp = array[j];

array[j] = array[k];

array[k] = temp;

j -= gap;

}

else

{

j--;

}

}

}

}

}

int \*Merge(int \*firstArray, const int firstArrayLength, int \*secondArray, const int secondArrayLength)

{

int i, k = 0, j = 0;

int \*resultingArray = (int \*)malloc(sizeof(int) \* (firstArrayLength + secondArrayLength));

for (i = 0; i < firstArrayLength + secondArrayLength; i++)

{

if (j > firstArrayLength - 1)

{

resultingArray[i] = secondArray[k];

k++;

}

else if (k > secondArrayLength - 1)

{

resultingArray[i] = firstArray[j];

j++;

}

else if (firstArray[j] <= secondArray[k])

{

resultingArray[i] = firstArray[j];

j++;

}

else

{

resultingArray[i] = secondArray[k];

k++;

}

}

return resultingArray;

}

int AreResultsEqual(int \*linearResultingArray, int \*parallelResultingArray, const int length)

{

int i;

for (i = 0; i < length; i++)

{

if (linearResultingArray[i] != parallelResultingArray[i])

{

return 1;

}

}

return 0;

}

void ParallelShellSort(int \*array, int length, const int procNum,

const int procRank, double &time, int \*resultingArray)

{

int i;

int step; // Helps to sync send and receive operations between processes

int receivedSubArrayLength;

int elementsPerProc;

int \*subArrayPerProc;

int \*receivedSubArray;

int \*subArrayShifts; // Arrays of start indexes for each subArrayPerProc

int \*subArrayLengths; // Array of lengths for each subArrayPerProc

double startTime, endTime;

MPI\_Status status;

subArrayShifts = (int \*)malloc(sizeof(int) \* procNum);

subArrayLengths = (int \*)malloc(sizeof(int) \* procNum);

if (procRank == 0)

{

elementsPerProc = length / procNum;

for (i = 0; i < procNum; i++)

{

subArrayShifts[i] = i \* elementsPerProc;

subArrayLengths[i] = elementsPerProc;

}

// Last process handles subArray with remainder (if there is one)

subArrayLengths[procNum - 1] = elementsPerProc + length % procNum;

}

startTime = MPI\_Wtime();

MPI\_Bcast(&length, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(&elementsPerProc, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(subArrayShifts, procNum, MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Bcast(subArrayLengths, procNum, MPI\_INT, 0, MPI\_COMM\_WORLD);

if (procRank == procNum - 1)

{

elementsPerProc += length % procNum;

}

subArrayPerProc = (int \*)malloc(sizeof(int) \* elementsPerProc);

MPI\_Scatterv(array, subArrayLengths, subArrayShifts,

MPI\_INT, subArrayPerProc, subArrayLengths[procRank],

MPI\_INT, 0, MPI\_COMM\_WORLD);

MPI\_Barrier(MPI\_COMM\_WORLD);

free(subArrayShifts);

free(subArrayLengths);

ShellSort(subArrayPerProc, elementsPerProc);

for (step = 1; step < procNum; step \*= 2)

{

// Even processes (included root) receive subArrays from not even processes

// And store merged subArray

if (procRank % (2 \* step) == 0)

{

if (procRank + step < procNum)

{

MPI\_Recv(&receivedSubArrayLength, 1, MPI\_INT, procRank + step, 0, MPI\_COMM\_WORLD, &status);

receivedSubArray = (int \*)malloc(receivedSubArrayLength \* sizeof(int));

MPI\_Recv(receivedSubArray, receivedSubArrayLength, MPI\_INT, procRank + step, 1, MPI\_COMM\_WORLD, &status);

subArrayPerProc = Merge(subArrayPerProc, elementsPerProc, receivedSubArray, receivedSubArrayLength);

free(receivedSubArray);

elementsPerProc += receivedSubArrayLength;

}

}

else

{

MPI\_Send(&elementsPerProc, 1, MPI\_INT, ((procRank - step) < 0) ? 0 : (procRank - step), 0, MPI\_COMM\_WORLD);

MPI\_Send(subArrayPerProc, elementsPerProc, MPI\_INT, ((procRank - step) < 0) ? 0 : (procRank - step), 1, MPI\_COMM\_WORLD);

}

}

if (procRank == 0)

{

CopyArray(subArrayPerProc, resultingArray, length);

endTime = MPI\_Wtime();

time = endTime - startTime;

}

free(subArrayPerProc);

}