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

«Быстрая сортировка со слиянием «Разделяй и властвуй»»

Выполнил:

студент группы 1606-1

Кудрин М.А.

Проверил:

Кустикова В.Д.

Нижний Новгород

2018

Содержание

Постановка задачи

Сортировкой (англ. sorting) называется процесс упорядочивания множества объектов по какому-либо признаку.

В данной работе мы будем сортировать массив из целых чисел в порядке возрастания.

Существуют различные алгоритмы сортировки, которые обладают теми или иными свойствами. Основные характеристики для алгоритмов сортировки:

1) Временная

   Эту классификацию обычно считают самой важной. Оценивают худшее время алгоритма, среднее и лучшее. Лучшее время — минимальное время работы алгоритма на каком-либо наборе, обычно этим набором является тривиальный [1…n]. Худшее время — наибольшее время. У большинства алгоритмов временные оценки бывают O(nlogn) и O(n2).

2) Память

   Параметр сортировки, показывающий, сколько дополнительной памяти требуется алгоритму. Сюда входят и дополнительный массив, и переменные, и затраты на стек вызовов. Обычно затраты бывают O(1), O(logn), O(n).

3) Количество обменов

   Количество обменов может быть важным параметром в случае, если объекты имеют большой размер. В таком случае при большом количестве обменов время алгоритма заметно увеличивается.

Основные алгоритмы сортировки: быстрая сортировка, сортировка пузырьком, сортировка вставками, сортировка Шелла, сортировка слиянием.

В нашей программе будет использоваться быстрая сортировка со слиянием «Разделяй и властвуй».

Была поставлена задача разработать программу, сортирующую массив элементов с помощью быстрой сортировки со слиянием «Разделяй и властвуй» с использованием методов параллельного программирования. Для этого разработать последовательный и параллельный алгоритм данной сортировки.

Программа должна иметь параллельную реализацию, выполненную при помощи MPI.

В программе необходимо:

1. Выполнить проверку совпадения результатов последовательной и параллельной реализаций.
2. Продемонстрировать корректность работы алгоритмов на задаче/задачах малой размерности.
3. Обеспечить генерацию данных для задач произвольной размерности. Параметры задачи должен задавать пользователь.
4. Вывести время работы последовательного и параллельного алгоритмов.

Метод решения

Алгоритм сортировки слиянием «Разделяй и властвуй» заключается в следующем:

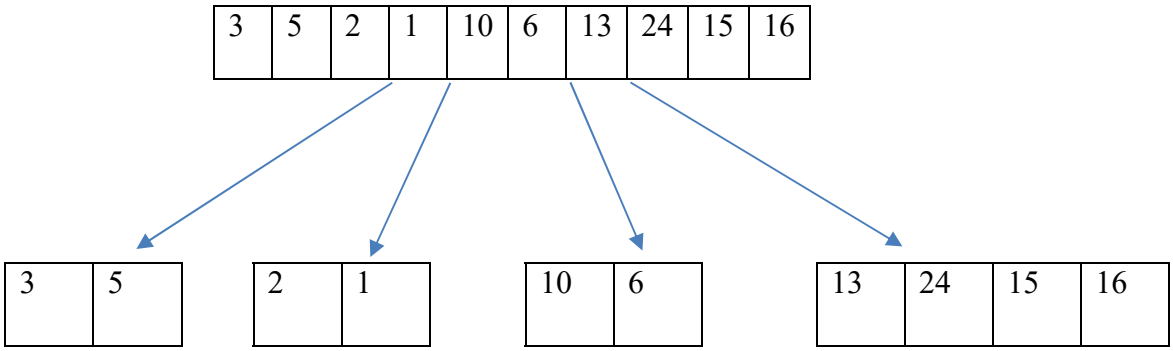
Задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи. Конкретно процедуру сортировки слиянием можно описать следующим образом:

1) Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.
2) Иначе массив разбивается на две части, которые сортируются рекурсивно.
3) После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

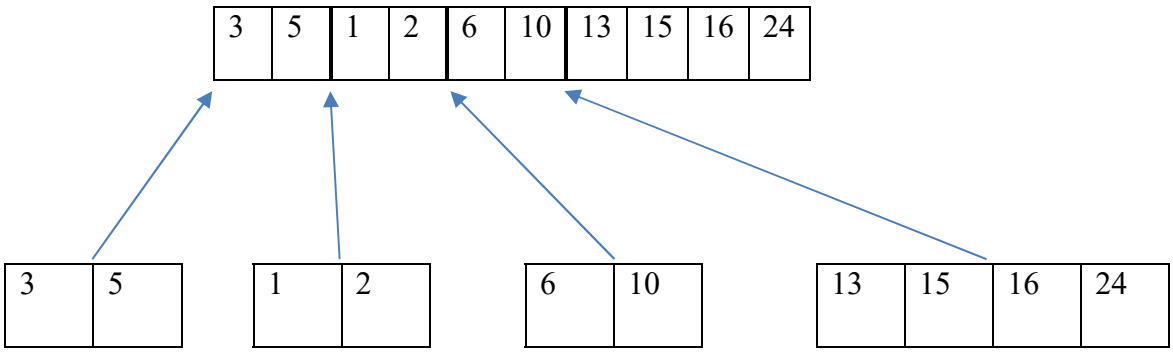
Теперь перейдем к алгоритму слияния. У нас есть два массива a и b (фактически это будут две части одного массива, но для удобства будем писать, что у нас просто два массива). Нам надо получить массив с размером |a|+|b|. Для этого можно применить процедуру слияния. Эта процедура заключается в том, что мы сравниваем элементы массивов (начиная с начала) и меньший из них записываем в финальный. И затем, в массиве у которого оказался меньший элемент, переходим к следующему элементу и сравниваем теперь его. В конце, если один из массивов закончился, мы просто дописываем в финальный другой массив. После мы наш финальный массив записываем вместо двух исходных и получаем отсортированный участок.

Схема распараллеливания

В начале мы делим массив на части. Количество частей равно количеству потоков. Если есть остаток от деления $\frac{n}{p}$, где n – количество элементов в массиве, а p – количество процессов, то этот остаток отдаем последнему потоку.

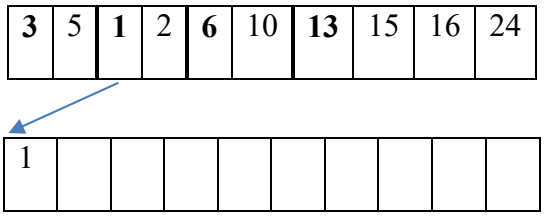


Каждый из процессов сортирует свой массив и отправляет в корневой.

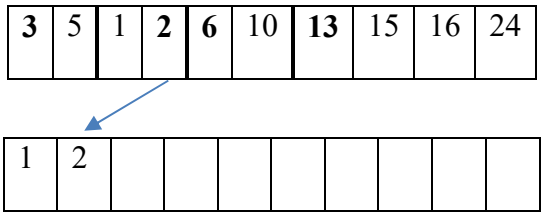


На последней стадии собираем слиянием новый отсортированный массив на корневом процессе. Имеем массив, в котором есть p последовательно расположенных отсортированных блоков. Создаем новый массив размером n, затем проходим по первым элементам каждого из блока, находим минимальный и записываем его в новый массив и сдвигаем в том блоке

1ая итерация



2ая итерация



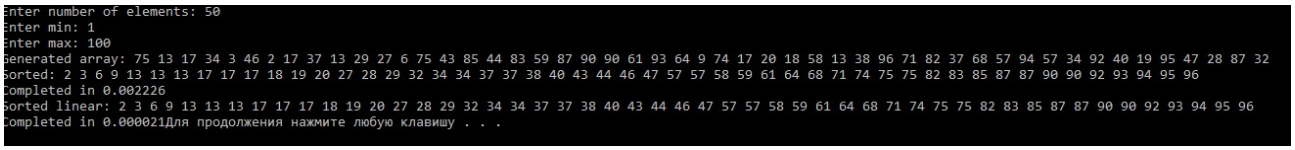
Описание программной реализации

Руководство пользователя

Для запуска программы в консоли необходимо выполнить следующую команду:

mpiexec –n p program.exe, где p – количество процессов

Затем в консоли надо будет ввести сначала размер массива, затем минимальный и максимальный элемент.



Руководство программиста

1) Функция `generateRandomArray` генерирует случайный массив размером elementsNumber, с элементами от min до max
2) Функция `generateAmountArray` генерирует массив, где каждому элементу соответствует количество отправляемых элементов основного массива. Индекс в сгенерированном массиве соответствует рангу процесса.
3) Функция `generateOffsetArray` генерирует массив смещений, где каждому элементу соответствует смещение внутри основного массива с которого будет осуществляться отправка каждому рангу. Индекс в сгенерированном массиве соответствует рангу процесса.
4) Функия `sort` осуществляет сортировку
5) Функция `combineArrays` нужна для финального слияния из массива, в котором есть p последовательно расположенных отсортированных блоков в один отсортированный массив.

Код программы можно просмотреть в разделе «Приложение».

Подтверждение корректности

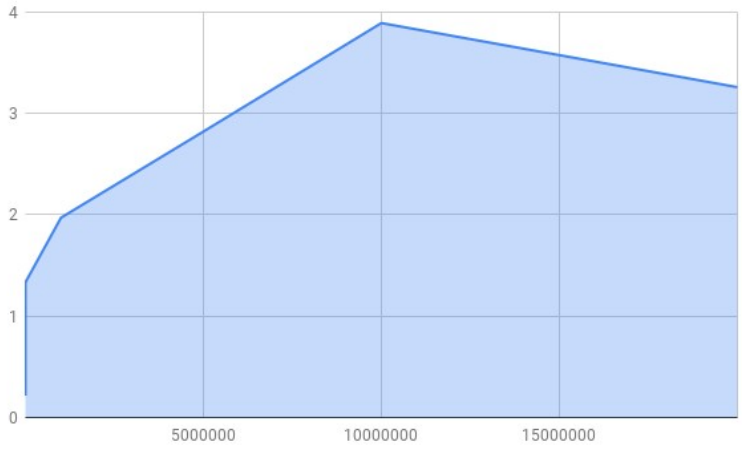
Для подтверждения корректности в программе выполняем сначала параллельный алгоритм, а затем последовательный. Проходим по этим массивам и поэлементно сравниваем, если есть несовпадение выводим на экран что произошла ошибка, если массивы пройдены полностью - успех.

Также результат сортировки выводится на экране.

Результаты экспериментов

По данным экспериментов видно, что при больших объемах данных распараллеливание имеет очень хороших показатель эффективности, однако при маленьких объемах параллельный алгоритм работает даже медленнее линейного.

Количество потоков	Количество элементов	Время параллельное	Время линейное	Ускорение
4	100	0,000258	0,000056	0,2170542636
4	1000	0,000454	0,000605	1,332599119
4	1000000	0,454631	0,895254	1,969188199
4	5000000	1,847745	5,207798	2,818461422
4	10000000	4,033836	15,682981	3,88785786
4	20000000	11,436448	37,236447	3,255945115



Заключение

Поставленная задача полностью выполнена. Была реализована программа, реализующая параллельный и последовательный алгоритм сортировки слиянием «разделяй и властвуй». Программа имеет хороший показатель ускорения, может быть, возможно улучшить параллельный алгоритм, если собирать блоки не в одном массиве, а по дереву.

Приложение

```
#include"stdafx.h"
#include<iostream>
#include<ctime>
#include"mpi.h"

#define _CRT_SECURE_NO_WARNINGS
using namespace std;

int* generateOffsetArray(int* amountArray, const int numproc) {
	int* arr = new int[numproc];
	int currOffset = 0;
	arr[0] = 0;
	for (int i = 1; i < numproc; i++) {
		currOffset = arr[i - 1];
		arr[i] = currOffset + amountArray[i - 1];
	}
	return arr;
}

int* generateAmountArray(const int elementsNumber, const int numproc) {
	int curr = 0;
	int* arr = new int[numproc];
	for (int j = 0; j < numproc; j++)
		arr[j] = elementsNumber / numproc;
	arr[numproc - 1] += elementsNumber % numproc;
	return arr;
}

int* generateRandomArray(const int elementsNumber, const int min, const int max) {
	int* arr = new int[elementsNumber];
	for (int j = 0; j < elementsNumber; j++)
		arr[j] = rand() % (max - min);
	return arr;
}

int* sort(int* array, int size) {
	int* res = new int[size];
	for (int i = 0; i < size; i++) {
		res[i] = 0;
	}

	if (size == 1) {
		return array;
	} else if (size == 2) {

		if (array[0] > array[1]) {
			res[0] = array[1];
			res[1] = array[0];
		}
		else {
			res[0] = array[0];
			res[1] = array[1];
		}

	}
	else {
		int* sz = generateAmountArray(size, 2);
		int size1 = sz[0];
		int size2 = sz[1];
```

```
		delete[] sz;

		int* arr1 = new int[size1];
		int* arr2 = new int[size2];

		for (int i = 0; i < size1; i++) {
			arr1[i] = array[i];
			arr2[i] = array[i + size1];
		}
		if (size1 != size2) {
			arr2[size2 - 1] = array[size - 1];
		}

		arr1 = sort(arr1, size1);
		arr2 = sort(arr2, size2);

		int c1 = 0;
		int c2 = 0;
		int i = 0;
		while (c1 + c2 < size) {
			if (c1 < size1&& c2 < size2) {
				if (arr1[c1] < arr2[c2])
				{
					res[i++] = arr1[c1++];
				} else if (arr1[c1] > arr2[c2])
				{
					res[i++] = arr2[c2++];
				}
				else if (arr1[c1] == arr2[c2]) {
					res[i++] = arr1[c1++];
					res[i++] = arr2[c2++];
				}
			}
			else if(c1 == size1){
				while (c2 < size2)
				{
					res[i++] = arr2[c2++];
				}
			}
			else if (c2 == size2) {
				while (c1 < size1)
				{
					res[i++] = arr1[c1++];
				}
			}

		}
	} return res;
}

int* combineArrays(int* array, int* amount, int* offset, int arraySize, int numproc) {
	int * res = new int[arraySize];
	for (int i = 0; i < arraySize; i++)
	{
		res[i] = 0;
	}
	int* counters = new int[numproc];
	for (int i = 0; i < numproc; i++)
	{
		counters[i] = 0;
	}
	int i = 0;
```

```
	while (i < arraySize) {
		int currmin = 100000;
		int currrank = 0;
		for (int j = 0; j < numproc; j++)
		{
			if (counters[j] < amount[j]) {
				if (array[offset[j] + counters[j]] < currmin) {
					currmin = array[offset[j] + counters[j]];
					currrank = j;
				}
			}
		}
		res[i++] = currmin;
		counters[currrank]++;
	}
	return res;

}


int main(int argc, char* argv[]) {
	int* array;
	int* amountArray;
	int* offsetArray;
	int* recvbuf;
	int numproc, rank;

	double starttime, endtime;
	int elementsNumber;
	MPI_Status status;
	MPI_Request request;
	MPI_Init(&argc, &argv);
	MPI_Comm_size(MPI_COMM_WORLD, &numproc);
	MPI_Comm_rank(MPI_COMM_WORLD, &rank);

	if (rank == 0) {

		printf("Enter number of elements: ");
		cin >> elementsNumber;
		int a, b;
		int* copy;
		copy = new int[elementsNumber];
		printf("Enter min: ");
		cin >> a;
		printf("Enter max: ");
		cin >> b;
		printf("Generated array: ");
		srand(time(nullptr));
		array = generateRandomArray(elementsNumber, a, b);

		for (int i = 0; i < elementsNumber ; i++)
		{
			copy[i] = array[i];
			printf("%i ", array[i]);
		}
		cout << endl;

		amountArray = generateAmountArray(elementsNumber, numproc);
		offsetArray = generateOffsetArray(amountArray, numproc);

		starttime = MPI_Wtime();
		MPI_Bcast(amountArray, numproc, MPI_INT, 0, MPI_COMM_WORLD);
```

```
            MPI_Bcast(offsetArray, numproc, MPI_INT, 0, MPI_COMM_WORLD);
            recvbuf = new int[amountArray[0]];

            MPI_Scatterv(array, amountArray, offsetArray, MPI_INT, recvbuf,
amountArray[0], MPI_INT, 0, MPI_COMM_WORLD);

            recvbuf = sort(recvbuf, amountArray[0]);


            MPI_Gatherv(recvbuf, amountArray[0], MPI_INT,
array,amountArray,offsetArray,MPI_INT, 0, MPI_COMM_WORLD);
            array = combineArrays(array, amountArray, offsetArray, elementsNumber,
numproc);
            endtime = MPI_Wtime();

            printf("Sorted: ");
            for (int i = 0; i < elementsNumber; i++) {
                  printf("%i ", array[i]);
            }

            printf("\nCompleted in %f", endtime - starttime);
            starttime = MPI_Wtime();
            array = sort(copy, elementsNumber);
            endtime = MPI_Wtime();

            printf("\nSorted linear: ");
            for (int i = 0; i < elementsNumber; i++) {
                  printf("%i ", array[i]);
            }
            printf("\nCompleted in %f", endtime - starttime);
      }
      else {
            amountArray = new int[numproc];
            offsetArray = new int[numproc];

            MPI_Bcast(amountArray, numproc, MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(offsetArray, numproc, MPI_INT, 0, MPI_COMM_WORLD);
            recvbuf = new int[amountArray[rank]];
            array = new int[amountArray[rank]];

            MPI_Scatterv(array, amountArray, offsetArray, MPI_INT, recvbuf,
amountArray[rank], MPI_INT, 0, MPI_COMM_WORLD);
            recvbuf = sort(recvbuf, amountArray[rank]);

            MPI_Gatherv(recvbuf, amountArray[rank], MPI_INT, nullptr, nullptr, nullptr,
MPI_INT, 0, MPI_COMM_WORLD);
      }
      MPI_Finalize();
      return 0;
}
```