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 с простым слиянием»**

**Выполнил:**

студент группы 381608

Ленькин Вячеслав Александрович

Нижний Новгород

2018

**Содержание**

[Постановка задачи 3](#_Toc531367741)

[Метод решения 4](#_Toc531367742)

[Схема распараллеливания 5](#_Toc531367743)

[Описание программной реализации 6](#_Toc531367744)

[Подтверждение корректности 7](#_Toc531367745)

[Результаты экспериментов 7](#_Toc531367746)

[Заключение 9](#_Toc531367747)

[Приложение 10](#_Toc531367748)

Постановка задачи

Дан неупорядоченный массив, состоящий из n элементов. Требуется расположить элементы массива в порядке возрастания, реализовав упорядочивание с помощью быстрой сортировки с простым слиянием. При этом реализация должна содержать последовательную и параллельную версии алгоритма. Также необходимо сравнить время их работы и сделать вывод об эффективности использования параллельной версии.

Метод решения

Быстрая сортировка отличается от других операцией разбиением массива на две части относительно опорного элемента. Массив будет отсортирован независимо от того, какой элемент выбран в качестве опорного. Но наиболее удачным считается ситуация, когда опорный элемент делит исходный массив примерно на две равные части. Алгоритм быстрой сортировки включает в себя два основных этапа:

* разбиение массива относительно опорного элемента;
* рекурсивная сортировка каждой части массива.

*Общая схема разбиения массива:*

1. вводятся указатели *first* и *last* для обозначения минимального и максимального индексов массива, а также опорный элемент *mid*;
2. вычисляется значение опорного элемента (*first*+*last*)/2 и заносится в переменную *mid*;
3. указатель *first* смещается с шагом в 1 элемент к концу массива до тех пор, пока *Mas*[*first*]>*mid*.
4. указатель *last* смещается от конца массива с шагом в 1 элемент к его началу, пока *Mas*[*last*]<*mid*;
5. каждые два найденных элемента меняются местами;
6. пункты 3 и 4 выполняются до тех пор, пока first<last.

Общая сложность алгоритма определяется глубиной рекурсии. В лучшем случае (в наиболее сбалансированном варианте) при каждой операции разделения массив делится на две почти одинаковые части. Это даёт сложность алгоритма  O(n log2 n). Средняя сложность при случайном распределении входных данных составит O(n logn). В самом несбалансированном варианте каждое разделение дает два подмассива размерами 1 и n-1. Общее время работы составит O(n2).

Схема распараллеливания

Для распараллеливания сортировки используем простое слияние. Идея параллельной реализации с использованием простого слияния заключается в выполнении следующих шагов:

* Каждый процесс отсортирует свою часть исходного массива.
* После сортировки частей массива выполняется слияние этих частей

Слияние будет проводиться следующим образом: на первом шаге процессы делятся на пары, и правый массив передаёт свои данные левому. Процесс, получивший данные, сортирует их. На втором шаге также объединяем процессы в пары, но только те, которые получали данные на первом шаге. Точно также правый процесс передаёт свои данные левому, а тот их сортирует. Это выполняется до тех пор, пока не останется один процесс. Его массив и будет результатом параллельной сортировки. **{\displaystyle O(n\cdot \log \_{2}n)}**

*Пример:*

* 1. Дан исходный массив, который нужно разделить между процессами

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **12** | **7** | **31** | **1** | **9** | **55** | **10** | **5** | **8** | **23** | **6** |

* 1. Процессы получили части массива

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0-й процесс |  | 1-й процесс |  | 2-й процесс |  | 3-й процесс |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **12** | **7** | **31** |  | **1** | **9** | **55** |  | **10** | **5** | **8** |  | **23** | **6** |

* 1. Каждый процесс сортирует свой массив

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0-й процесс |  | 1-й процесс |  | 2-й процесс |  | 3-й процесс |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **7** | **12** | **31** |  | **1** | **9** | **55** |  | **5** | **8** | **10** |  | **6** | **23** |

* 1. Сливаем массивы по алгоритму, описанному выше

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 0-й процесс |  | 1-й процесс |  | 2-й процесс |  | 3-й процесс |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **7** | **12** | **31** |  | **1** | **9** | **55** |  | **5** | **8** | **10** |  | **6** | **23** |

|  |  |  |
| --- | --- | --- |
| 0-й процесс |  | 2-й процесс |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **1** | **7** | **9** | **12** | **31** | **55** |  | **5** | **6** | **8** | **10** | **23** |

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0-й процесс | | | | | | | | | | |
| **1** | **5** | **6** | **7** | **8** | **9** | **10** | **12** | **23** | **31** | **55** |

Описание программной реализации

**Руководство пользователя**

Для запуска программы необходимо в консоли выполнить следующую команду:

mpiexec –n <число процессов> <имя файла>

**Руководство программиста**

*Последовательность выполнения программы:*

1. Программа на вход получает размер исходного массива Size. Если этого не происходит, то используется значение по умолчанию.
2. В процессе с рангом 0 генерируется неупорядоченный массив mass.
3. Создаются массивы числа посылаемых элементов (sendCounts) и смещений (displs) для передачи частей исходного массива mass всем процессам, кроме нулевого, с помощью функции MPI\_Scatterv.
4. Нулевой процесс выполняет последовательную версию алгоритма, измеряет время своей работы и отправляет последнему процессу отсортированный массив mass, а также значения времени начала и конца выполнения алгоритма.
5. Процессы, участвующие в параллельной версии алгоритма (все, кроме нулевого) сортируют полученные части исходного массива.
6. Отсортированные массивы сливаются в один общий по алгоритму, описанному ранее.
7. Последний процесс измеряет время работы параллельного алгоритма, принимает результаты работы нулевого процесса и выполняет сравнение.

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

Подтверждение корректности

Подтверждение корректности работы параллельной версии алгоритма выполняет последний процесс: производится поэлементное сравнение полученных результатов последовательной и параллельной версий алгоритма. После чего программа выводит соответствующее сообщение.

Результаты экспериментов

Время работы последовательной и параллельной (с использованием 4-х процессов) версий.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | n = 5000 | n = 250000 | n = 500000 | n = 5000000 |
| Последовательная  версия | 0.182845ms | 9.34411ms | 18.8ms | 195.706ms |
| Параллельная версия | 0.329958ms | 4.69809ms | 8.26491ms | 80.6488ms |
| Ускорение | 0.554147 | 1.98891 | 2.27468 | 2.42664 |

При оптимальном выборе ведущих элементов, когда разделение каждого блока происходит на равные по размеру части, трудоемкость алгоритма составляет

*Tseq =*

Теперь выясним, сколько времени требуется для работы параллельной версии. Пусть число процессов равно p.

В таком случае возможное ускорение будет равно:

Заключение

Были реализованы последовательная и параллельная версии Быстрой сортировки. Проведены эксперименты с использованием массивов разных размеров, а также с разным числом процессов. Замерено время работы алгоритмов и на основании этого можно сделать ряд выводов:

1. Имеет смысл применять параллельную версию Быстрой сортировки при больших размерах массива и при большом количестве процессов.
2. Использование параллельной версии алгоритма дает хороший результат в плане ускорения, т.к. при увеличении числа процессов, ускорение будет расти. Но при малых размерах исходного массива ускорение будет меньше единицы. Это обусловлено тем, что время работы сортировки в параллельной версии будет существенно зависеть от механизмов взаимодействия между процессами. При большом количестве процессов механизмы взаимодействия между ними не окупаются.

Приложение

#include <mpi.h>

#include <stdio.h>

#include <iostream>

#include <ctime>

using namespace std;

#define endLine cout << endl

void quicksort(int \*mas, int first, int last)

{

int mid, count;

int f = first, l = last;

mid = mas[(f + l) / 2];

do

{

while (mas[f]<mid) f++;

while (mas[l]>mid) l--;

if (f <= l)

{

count = mas[f];

mas[f] = mas[l];

mas[l] = count;

f++;

l--;

}

} while (f<l);

if (first<l) quicksort(mas, first, l);

if (f<last) quicksort(mas, f, last);

}

int \*merge(int \*a, int n, int \*b, int m)

{

int \*result = new int[n + m];

int i = 0, j = 0;

int index = 0;

while (i<n && j<m)

{

if (a[i] < b[j])

{

result[index] = a[i];

i++;

}

else

{

result[index] = b[j];

j++;

}

index++;

}

while (i < n)

{

result[index] = a[i];

index++;

i++;

}

while (j < m)

{

result[index] = b[j];

index++;

j++;

}

return result;

}

void main(int argc, char \*argv[])

{

int Size = 5000;

int \*mass , \*sendCounts = nullptr, \*displs = nullptr, \*paralMass = nullptr, \*rez = nullptr;

int first, last, paralPart;

double timeStartOfSequential, timeEndOfSequential, timeStartOfParallel, timeEndOfParallel;

int ProcNum, ProcRank;

MPI\_Status status;

MPI\_Init(&argc, &argv);

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

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

if (argc > 1)

Size = atoi(argv[1]);

mass = new int[Size];

if (ProcRank == 0)

{

for (int i = 0; i < Size; i++)

mass[i] = rand() % 100;

if (Size < 11)

{

cout << "Array:" << endl;

for (int i = 0; i < Size; i++)

{

cout << mass[i] << " ";

}

}

}

// Массивы sendCounts и displs для передачи массива через Scatterv

if (ProcNum>1)

{

paralPart = Size / (ProcNum - 1);

int given = 0;

sendCounts = new int[ProcNum];

sendCounts[0] = 0;

for (int i = 1; i<ProcNum - 1; i++)

{

sendCounts[i] = paralPart;

given += paralPart;

}

sendCounts[ProcNum - 1] = Size - given;

displs = new int[ProcNum];

for (int i = 0; i<ProcNum; i++)

displs[i] = 0;

for (int i = 2; i<ProcNum; i++)

displs[i] += (i - 1)\*sendCounts[i - 1];

paralMass = new int[sendCounts[ProcRank]];

for (int i = 0; i<sendCounts[ProcRank]; i++)

paralMass[i] = 0;

rez = new int[Size];

for (int i = 0; i<Size; i++)

rez[i] = 0;

MPI\_Scatterv(mass, sendCounts, displs, MPI\_INT, paralMass, sendCounts[ProcRank], MPI\_INT, 0, MPI\_COMM\_WORLD);

}

//Последовательный алгоритм

if (ProcRank == 0)

{

timeStartOfSequential = MPI\_Wtime() \* 1000;

quicksort(mass, 0, Size - 1);

timeEndOfSequential = MPI\_Wtime() \* 1000;

/\*for(int i=0; i<Size; i++)

cout<<mass[i]<<" ";

endLine;\*/

if (ProcNum>1)

{

MPI\_Send(mass, Size, MPI\_INT, ProcNum - 1, 8, MPI\_COMM\_WORLD);

MPI\_Send(&timeStartOfSequential, 1, MPI\_DOUBLE, ProcNum - 1, 9, MPI\_COMM\_WORLD);

MPI\_Send(&timeEndOfSequential, 1, MPI\_DOUBLE, ProcNum - 1, 10, MPI\_COMM\_WORLD);

}

endLine;

cout << "Time of sequential algorithm: " << timeEndOfSequential - timeStartOfSequential << "ms" << endl;

}

else

{

if (ProcRank == ProcNum - 1)

timeStartOfParallel = MPI\_Wtime() \* 1000;

quicksort(paralMass, 0, sendCounts[ProcRank] - 1);

/\*

for(int i=0; i<sendCounts[ProcRank]; i++)

cout<<paralMass[i]<<" ";

endLine;

\*/

int s = ProcNum, m = 1;

int k = sendCounts[ProcRank];

while (s > 1)

{

s = s / 2 + s % 2;

if ((ProcRank - 1 - m) % (2 \* m) == 0)

{

MPI\_Send(&k, 1, MPI\_INT, ProcRank - m, 0, MPI\_COMM\_WORLD);

MPI\_Send(paralMass, k, MPI\_INT, ProcRank - m, 0, MPI\_COMM\_WORLD);

}

if (((ProcRank - 1) % (2 \* m) == 0) && (ProcNum - ProcRank > m))

{

int k1;

int \*paralMass2;

MPI\_Recv(&k1, 1, MPI\_INT, ProcRank + m, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

paralMass2 = new int[k1];

MPI\_Recv(paralMass2, k1, MPI\_INT, ProcRank + m, MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status);

paralMass = merge(paralMass, k, paralMass2, k1);

if (k + k1 == Size)

MPI\_Send(paralMass, Size, MPI\_INT, ProcNum - 1, 11, MPI\_COMM\_WORLD);

k = k + k1;

}

m = 2 \* m;

}

if (ProcRank == ProcNum - 1)

{

timeEndOfParallel = MPI\_Wtime() \* 1000;

MPI\_Recv(mass, Size, MPI\_INT, 0, 8, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&timeStartOfSequential, 1, MPI\_DOUBLE, 0, 9, MPI\_COMM\_WORLD, &status);

MPI\_Recv(&timeEndOfSequential, 1, MPI\_DOUBLE, 0, 10, MPI\_COMM\_WORLD, &status);

MPI\_Recv(rez, Size, MPI\_INT, MPI\_ANY\_SOURCE, 11, MPI\_COMM\_WORLD, &status);

//Сравниваем поэлементно результаты двух алгоритмов

int same = 0;

for (int i = 0; i<Size; i++)

if (rez[i] == mass[i])

same++;

if (Size < 11)

{

cout << "Sorted array: " << endl;

for (int i = 0; i < Size; i++)

{

cout << mass[i] << " ";

}

}

endLine;

cout << "Time of parallel algorithm: " << timeEndOfParallel - timeStartOfParallel << "ms" << endl;

if (same == Size)

cout << "The results of algorithms are the same" << endl;

else

cout << "The results of algorithms are different" << endl;

cout << "Acceleration: " << (timeEndOfSequential - timeStartOfSequential) / (timeEndOfParallel - timeStartOfParallel) << endl;

}

}

MPI\_Finalize();

if (ProcNum == 1)

system("pause");

}