Министерство образования и науки Российской Федерации

Государственное образовательное учреждение высшего профессионального образования «Нижегородский государственный университет   
им. Н.И. Лобачевского»

#### **Институт информационных технологий, математики и механики**

#### **Кафедра: Программная инженерия**

Специальность (направление): Программная инженерия

**Отчет**

по лабораторной работе

по дисциплине «Параллельное программирование»

тема:

**«Быстрая сортировка с четно-нечетным слиянием Бэтчера. Параллельная реализация»**

**Выполнил:**

студент группы 381508

Тарасов О.А.

\_\_\_\_\_\_\_Подпись

Нижний Новгород  
2018

**Введение**

Сортировка данных является одной из наиболее важных задач в программировании. Разработкой алгоритмов для параллельной сортировки стали сразу, как только появились методы параллельных вычислений. С тех пор было предложено много алгоритмов параллельной сортировки.

В данной лабораторной работе будет рассмотрена быстрая сортировка (QuickSort) с использованием чётно-нечётного слияния Бэтчера в трех версиях: последовательная, параллельная с использованием технологии OpenMP и параллельная с использованием технологии TBB.

Перед выполнением работы были поставлены следующие задачи:

1. Исследовать алгоритм быстрой сортировки
2. Исследовать метод объединения двух массивов при помощи четно-нечетного слияния Бэтчера
3. Реализовать последовательную версию быстрой сортировки, а также параллельные версии с использованием технологий OpenMP и TBB.
4. Проверить корректность выполнения всеми тремя программами поставленной задачи.
5. Сравнить скорость выполнения работы последовательной и параллельных версий алгоритма.
6. **Разработка тестовой версии программы**

В данной части будет приведено описание последовательной реализации алгоритма быстрой сортировки, а также программ для автоматической генерации тестов и проверки правильности проведенной сортировки.

1. **Последовательный алгоритм быстрой сортировки**

Быстрая сортировка (QuickSort) – это алгоритм упорядочивания данных, разработанный Ч. Хоаром. Является одним из самых быстрых универсальных алгоритмов сортировки: в среднем О(n log n) операций обмена при упорядочивании n элементов.

Общая идея алгоритма заключается в следующем:

1. Выбрать из массива элемент, называемый опорным. Это может быть любой из элементов массива (обычно выбирается центральный элемент).
2. Сравнить все остальные элементы с опорным и переставить их в массиве так, чтобы все элементы меньшие опорного располагались левее от него, а все элементы большие опорного – правее от него. После данного шага массив оказывается разделен на две части – в одной располагаются все элементы меньшие опорного, в другой – все элементы большие его. Опорный элемент теперь находится на своем месте.
3. Для обоих частей массива выполнить рекурсивно данную последовательность операций (если их размер больше единицы).

Программная реализация последовательной версии:

void QuickSort(int\* arr, int size)

{

int i = 0, j = size - 1;

int pivot = arr[size / 2];

do {

while (arr[i] < pivot) i++;

while (arr[j] > pivot) j--;

if (i <= j) {

int tmp = arr[i];

arr[i] = arr[j];

arr[j] = tmp;

i++;

j--;

}

} while (i <= j);

if (j > 0)

QuickSort(arr, j + 1);

if (i < size)

QuickSort(&arr[i], size - i);

}

1. **Генератор тестовых данных**

Ниже представлен код программы, генерирующей массив случайных целочисленных чисел. Всего доступно 20 вариантов (от выбранного варианта зависит число элементов в генерируемом массива).

Отдельно стоит сказать про вариант номер 10. Его особенностью является генерация массива одинаковых элементов для проверки сортировки на нем.

int n\_tests[] = { 1, 2, 3, 5, 7, 8, 10, 100, 100, 1000, 5000, 10000, 10000, 500000, 100000, 1000000, 10000000, 25000000, 50000000, 100000000 };

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

{

if (argc != 2 || atoi(argv[1]) < 1 || atoi(argv[1]) > 20)

{

cout << "Use 'generator [test number (1 - 20)]'" << endl;

return 1;

}

int N = n\_tests[atoi(argv[1])-1];

freopen(argv[1], "wb", stdout);

default\_random\_engine generator(static\_cast<unsigned int>(chrono::system\_clock::now().time\_since\_epoch().count()));

uniform\_int\_distribution <int> distribution(0, 1000);

fwrite(&N, sizeof(double), 1, stdout); // фиктивная запись (чтобы размеры входного и исходного файлов совпадали)

fwrite(&N, sizeof(N), 1, stdout);

int \*arr = new int[N];

if (atoi(argv[1]) == 10) { // одинаковые числа в массиве

int val = distribution(generator);

for (int i = 0; i < N; i++) arr[i] = val;

}

else // случайные числа в массиве

for (int i = 0; i < N; i++) arr[i] = distribution(generator);

fwrite(arr, sizeof(\*arr), N, stdout);

return 0;

}

1. **Проверка корректности сортировки**

Программа checker может работать в двух возможных режимах.

1. Проверка того, что все элементы массива расположены по возрастанию. Запустить: checker <файл массива>
2. Сравнение двух бинарных файлов (файла с отсортированным нами массивом и файла с заведомо корректно отсортированным массивом). Данный вариант надежнее, так как исключает ситуацию, когда программа сортировки может повредить исходный массив.

Запустить: checker <файл массива> <файл верно отсортированного массива>

1. **Описание алгоритма четно-нечетного слияния Бэтчера**

Данный алгоритм позволяет оптимизировать процедуру слияния отсортированных массивов. Далее описан его алгоритм с иллюстрацией на простом примере.

1. На вход подаются 2 отсортированных массива.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1 | 4 | 5 | 7 | 10 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 2 | 3 | 7 | 8 | 9 |

1. Из полученных массивов в возрастающем порядке выбираются отдельно четные и отдельно нечетные элементы. Таким образом из двух исходных отсортированных массивов образуются два новых (также отсортированных).

Массив четных элементов:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1 | 2 | 5 | 7 | 9 | 10 |

Массив нечетных элементов:

|  |  |  |  |
| --- | --- | --- | --- |
| 3 | 4 | 7 | 8 |

1. Массивы, полученные на шаге 2, сливаются: оба массива просматриваются слева направо и в результирующий массив поочередно записывается минимальный элемент.

Результат:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 7 | 7 | 8 | 9 | 10 |

Данный алгоритм, по сравнению с простым слиянием, значительно ускоряет процесс объединения отсортированных массивов. Достигается это за счет равномерного распределения нагрузки по всем процессорам.

**II. Параллельная реализация быстрой сортировки при помощи технологии OpenMP**

В данном этапе было необходимо провести распараллеливание алгоритма быстрой сортировки при помощи технологии OpenMP. Библиотека OpenMP предоставляет разработчику набор директив и библиотечных функций, позволяющих при помощи небольших усилий распараллелить последовательный код.

1. **Стратегия распараллеливания**

Основной подход к параллельной реализации быстрой сортировки массива заключается в том, что мы делим массив на равные (с точностью до округления) части, каждая из которых обрабатывается своим потоком.

Каждый поток, независимо от остальных потоков выполняет сортировку своей части массива последовательным методом быстрой сортировки.

Получив несколько отсортированных массивов, выполняется их объединение при помощи четно-нечетного слияния Бэтчера.

1. **Выполнение работы**
2. Прежде всего была создана вспомогательная функция

void SelectElements(elemType type, const int\* arr1, int size1, const int\* arr2, int size2, std::vector<int>& result),

принимающая на вход 2 массива, выбирающая из них четные или нечетные элементы (в зависимости от параметра type) и сохраняющая результат в вектор result.

enum elemType {

EVEN,

ODD

};

void SelectElements(elemType type, const int\* arr1, int size1, const int\* arr2, int size2, std::vector<int>& result) {

int i, j;

if (type == EVEN) i = 0, j = 0;

else i = 1, j = 1;

result.reserve(size1+size2);

while (i < size1 && j < size2) {

if (arr1[i] <= arr2[j])

{

result.push\_back(arr1[i]);

i += 2;

}

else

{

result.push\_back(arr2[j]);

j += 2;

}

}

if (i >= size1)

while (j < size2) {

result.push\_back(arr2[j]);

j += 2;

}

else

while (i < size1) {

result.push\_back(arr1[i]);

i += 2;

}

}

1. Функция void MergeAndSort(const std::vector<int> vec1, const std::vector<int> vec2, int\* write\_to) – выполняет слияние двух векторов (имеющих выбранные на шаге 1 четные и нечетные элементы соответственно), последовательно выбирая наименьший элемент и записывая в массив write\_to. Затем происходит сортировка парно стоящих элементов массива write\_to.

void MergeAndSort(const std::vector<int> vec1, const std::vector<int> vec2, int\* write\_to)

{

int i = 0, j = 0;

int size1 = vec1.size(), size2 = vec2.size();

while (i < size1 && j < size2) {

write\_to[i + j] = vec1[i];

write\_to[i + j + 1] = vec2[j];

++i; ++j;

}

while (i < size1) {

write\_to[size2 + i] = vec1[i];

i++;

}

while (j < size2) {

write\_to[size1 + j] = vec2[j];

j++;

}

i = 1; // Первый элемент проверять не нужно

while (i < size1 + size2 - 1) {

if (write\_to[i] > write\_to[i + 1]) {

j = write\_to[i];

write\_to[i] = write\_to[i + 1];

write\_to[i + 1] = j;

}

++i;

}

}

1. Основной блок кода с распараллеливание при помощи OpenMP:

int step; // Переменная для хранения шага (step = 2^(N-1))

std::vector<int>\* tempArray = new std::vector<int>[threads];

int \*shift = new int[threads], \*chunk = new int[threads]; // shift - массив сдвигов, chunk - массив, содержащий размеры частей массива

#pragma omp parallel shared(arr, step, shift, chunk, tempArray) num\_threads(threads)

{

int tid, thread\_index; // tid - переменная для хранения ID текущего потока, thread\_index - определяет необходимый сдвиг для получения парного потока (на шаге 1 = 1, на шаге 2 = 2 и т.д.)

tid = omp\_get\_thread\_num();

/\* Распределение частей исходного массива по потокам и сортировка данных частей \*/

shift[tid] = tid\*(size / threads);

chunk[tid] = (tid == threads - 1) ? size - tid \* (size / threads) : size / threads;

QuickSort(arr + shift[tid], chunk[tid]);

#pragma omp barrier // Ожидаем, пока все потоки отсортируют свою часть массива

step = 1;

while (step < threads) {

/\* На каждом шаге выбираем четные и нечетные элементы из парных потоков, записываем в tempArray \*/

thread\_index = (int)pow(2, step - 1);

if (tid % (thread\_index\*2) == 0) {

SelectElements(EVEN, arr + shift[tid], chunk[tid], arr + shift[tid + thread\_index], chunk[tid + thread\_index], tempArray[tid]);

}

else if(tid % thread\_index == 0) {

SelectElements(ODD, arr + shift[tid], chunk[tid], arr + shift[tid - thread\_index], chunk[tid - thread\_index], tempArray[tid]);

}

#pragma omp barrier // Ожидаем выполнения данной части всеми потоками

/\* Производим слияние и сортировку tempArray в парных потоках \*/

if (tid % (thread\_index\*2) == 0) {

MergeAndSort(tempArray[tid], tempArray[tid + thread\_index], arr + shift[tid]);

chunk[tid] += chunk[tid + thread\_index];

tempArray[tid].clear(); tempArray[tid].shrink\_to\_fit();

tempArray[tid + thread\_index].clear(); tempArray[tid + thread\_index].shrink\_to\_fit();

}

#pragma omp single

{

step \*= 2; // Переходим на следующий шаг первым освободившимся потоком

}

#pragma omp barrier // Ждем пока слияние и сортировку выполнят все потоки

}

}

delete[] tempArray;

delete[] chunk;

delete[] shift;

1. **Результаты работы**

Время выполнения программы было измерено на 1, 2 и 4 потоках. Результаты и полученное ускорение представлены в таблице:

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 2.75 | - |
| 2 | 1.02 | 2.7 |
| 4 | 0.81 | 3.4 |

Для тестирования брался массив размером 50 000 000 элементов.

Правильность выполнения работы была проверена при помощи программы checker.

Таким образом, можно сделать вывод, что применение технологии OpenMP позволило ускорить сортировку массива при сохранении правильности проведенной сортировки.

**III. Параллельная реализация быстрой сортировки при помощи технологии TBB**

На заключительном этапе было проведено распараллеливание алгоритма быстрой сортировки при помощи технологии TBB (Intel Threading Building Blocks). Для слияния, так же, как и в OpenMP версии, использовался алгоритм четно-нечетного слияния Бэтчера.

Библиотека TBB предоставляет разработчику возможность писать параллельные программы на уровне «логических задач», инкапсулируя тем самым практически всю логику работы с потоками.

При выполнении лабораторной работы были созданы классы EvenSelector и OddSelector, представляющие собой задачи по выделению из массива четных/нечетных элементов соответственно.

Ниже приведен только класс EvenSelector (реализация OddSelector) практически аналогична.

class EvenSelector : public task

{

private:

int \*arr;

int size1, size2;

public:

EvenSelector (int\* \_arr, int \_size1, int \_size2) : arr(\_arr), size1(\_size1), size2(\_size2) {

}

task\* execute()

{

int\* arr2 = arr + size1;

int num = (size1 + size2 + 1) / 2; // Число четных элементов

int\* tmp = new int[num]; // Массив для хранения четных элементов

int a = 0, b = 0, i = 0;

while (a < size1 && b < size2) {

if (arr[a] <= arr2[b])

{

tmp[i] = arr[a];

a += 2;

}

else

{

tmp[i] = arr2[b];

b += 2;

}

i++;

}

if (a >= size1)

for (int j = b; j < size2; j += 2, i++)

tmp[i] = arr2[j];

else

for (int j = a; j < size1; j += 2, i++)

tmp[i] = arr[j];

// Копирование из временного массива в основной

for (int j = 0; j < num; ++j)

arr[j \* 2] = tmp[j];

return NULL;

}

};

Для последнего этапа слияния необходим функтор Comparator, выполняющий попарное сравнение элементов полученного массива.

class Comparator

{

private:

int \*arr;

public:

Comparator(int \*\_arr) : arr(\_arr) {}

void operator()(const blocked\_range<int>& r) const

{

int begin = r.begin(), end = r.end();

for (int i = begin; i<end; i++)

if (arr[i-1] > arr[i])

{

int tmp = arr[i-1];

arr[i-1] = arr[i];

arr[i] = tmp;

}

}

};

Непосредственно task Sorter реализует основную логику распараллеливания быстрой сортировки, производя разделение массива на части, сортируемые последовательно и объединяя результаты при помощи четно-нечетного слияния Бэтчера.

class Sorter: public task

{

private:

int \*arr;

int size;

int portion;

public:

Sorter(int \*\_arr, int \_size, int \_portion) : arr(\_arr), size(\_size), portion(\_portion) {}

task\* execute()

{

if (size <= portion)

{

QuickSort(arr, size);

}

else

{

int s = size / 2 + (size / 2) % 2;

Sorter &sorter1 = \*new (allocate\_child()) Sorter(arr, s, portion);

Sorter &sorter2 = \*new (allocate\_child()) Sorter(arr + s, size - s, portion);

set\_ref\_count(3);

spawn(sorter1);

spawn\_and\_wait\_for\_all(sorter2);

EvenSelector &evenSelector = \*new (allocate\_child()) EvenSelector(arr, s, size - s);

OddSelector &oddSelector = \*new (allocate\_child()) OddSelector(arr, s, size - s);

set\_ref\_count(3);

spawn(evenSelector);

spawn\_and\_wait\_for\_all(oddSelector);

parallel\_for(blocked\_range<int>(1, size), Comparator(arr));

}

return NULL;

}

};

Время выполнения программы так же, как и в OpenMP реализации, было измерено на 1, 2 и 4 потоках. Результаты и полученное ускорение представлены в таблице:

|  |  |  |
| --- | --- | --- |
| Число потоков | Время работы | Ускорение |
| 1 | 2.75 | - |
| 2 | 1.62 | 1.7 |
| 4 | 1.31 | 2.1 |

Для тестирования брался массив размером 50 000 000 элементов.

Правильность выполнения работы была проверена при помощи программы checker.

Таким образом, можно сделать вывод, что применение технологии TBB также позволяет быстро и корректно решать поставленную задачу по сортировке исходного массива.

**Заключение**

В процессе выполнения работы мною были реализованы 3 программы, выполняющие быструю сортировку: последовательная версия, версия с реализацией при помощи технологии OpenMP, версия с реализацией при помощи технологии TBB.

Помимо этого, были реализованы две вспомогательные программы: для генерации тестовых данных и для проверки корректности выполняемой сортировки.

Было установлено, что все три программы верно решают поставленную задачу. Кроме того, глядя на результаты выполнения, можно сделать вывод, что применение технологий параллельного программирования - OpenMP и TBB - позволяет значительно ускорить процесс выполнения сортировки.

Можно констатировать, что поставленные перед началом выполнения работы задачи успешно выполнены.