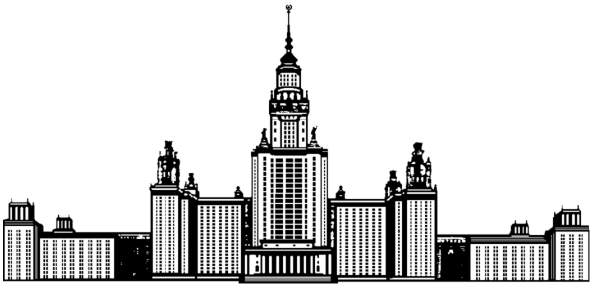


Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра вычислительных методов

Параллельные высокопроизводительные вычисления

Задание 2. Параллельная сортировка Бэтчера

Выполнила:
студентка 504 группы
Погосбекян Мария Михайловна

Дата подачи:
11.01.2023

Москва, 2023

1 Постановка задачи

Дана структура

```
Point {
    float coord[2];
    int index;
}
P[n1*n2]; // n1*n2 <= 2^30
```

Данная структура будет использоваться для работы с регулярной сеткой. Узлы сетки имеют координаты:

$$P[i \cdot n_2 + j].coord[0] = x(i, j),$$

$$P[i \cdot n_2 + j].coord[1] = y(i, j),$$

где i = 0, ..., $n_1 - 1, j = 0, \ldots, n_2 - 1$.
Индекс определяется соотношением $P[i \cdot n_2 + j].index = i \cdot n_2 + j$.

Особенности: для инициализации координат используются функции, принимающие на вход параметры ($int\ i$, $int\ j$) следовательно, координаты каждого узла сетки однозначно определяются парой (i, j).

На входе: на каждом процессе одинаковое количество элементов структуры Point. (Если на некоторых процессах элементов структуры Point меньше чем во всех остальных, тогда необходимо ввести дополнительные фиктивные элементы, например, с отрицательным значением индекса).

Цель: разработать и реализовать алгоритм, обеспечивающий параллельную сортировку методом Бэтчера массива или части массива структур Point, вдоль каждой из координат (x или y) в соответствии с заданным параметром. Следует реализовать и сортировку на каждом отдельном процессе и сеть сортировки Бэтчера.

На выходе: на каждом процессе одинаковое количество элементов структуры Point. Каждый элемент структуры Point одного процесса находится левее по координате x (или y) по сравнению с элементом структуры Point любого процесса с большим рангом, за исключением фиктивных элементов.

2 Описание метода решения

На вход программе поступает размер сетки – параметры n_1 и n_2. Затем происходит инициализация элементов типа Point с помощью функции InitPoints.

На каждый процессор раскидывается $m = \frac{n}{p}$ элементов типа Point, где p - количество процессоров, причем m - целое число. В зависимости от количества процессоров добавляются фиктивные элементы с отрицательным индексом -1 (если требуется), а количество элементов на каждом процессоре вычисляется. Далее на каждом процессоре сортируется вектор элементов типа Point, затем они будут сливаться в соответствии с расписанием.

Расписание строится на основе алгоритма Бэтчера. Сети Бэтчера – наиболее быстродействующие из масштабируемых сетей сортировки. Для построения сети обменной сортировки со слиянием используется следующий рекурсивный алгоритм. Чтобы отсортировать массив, состоящий из p элементов с номерами $[1, \ldots, p]$, нужно поделить его на две части: в первой будет $n = \lceil \frac{p}{2} \rceil$ элементов с номерами $[1, \ldots, n]$, во второй – $m = p - n$ элементов с номерами $[n+1, \ldots, p]$. Далее с помощью функции $Sort$ сортируется каждая из частей, а затем с помощью функции $Join$ происходит объединение отсортированных частей.

Функция $Sort$ – рекурсивное построение сети сортировки группы линий. Делит массив на две части. Одна состоит из n элементов, другая из m, дальше функция $Sort$ вызывается для каждой части массива, а затем вызывает функцию $Join$ для объединения упорядоченных частей.

Функция $Join$ – рекурсивное слияние двух групп линий. В сети нечетно-четного слияния отдельно объединяются массивов с нечетными номерами и отдельно – с четными. Далее с помощью заключительной группы компараторов обрабатываются пары соседних элементов с номерами вида $(2i, 2i+1)$, где i – натуральные числа от 1 до $\lfloor \frac{p}{2} \rfloor - 1$.

Запуск функции $Sort(0, 1, iCountProc)$ строит расписание сети слияний, то есть формирует вектор компаратаров – $vComparators$. Затем вызывается функция $CreateTackts$, которая формирует вектор тактов – $vTackts$ на основе вектора компараторов $vComparators$.

В соответствии с расписанием происходит слияние между двумя процессорами - функция $Merge$, за основу взят алгоритм функции $Join$ из лекций и книги М.В. Якобовского (лекция №5 слайд 74, «Введение в параллельные методы решения задач» с. 152).

Начальная сортировка на каждом процессоре выполняется с помощью $std :: qsort$, а параллельная сортировка между процессорами выполняется по алгоритму Бэтчера в соответствии с расписанием и использованием технологии MPI. В процессе расчета определяется время сортировки с использованием функции MPI_Wtime.

3 Описание используемой вычислительной системы

Тестирование программы выполнялось на суперкомпьютере «IBM Polus». Polus – параллельная вычислительная система, состоящая из 5 вычислительных узлов.

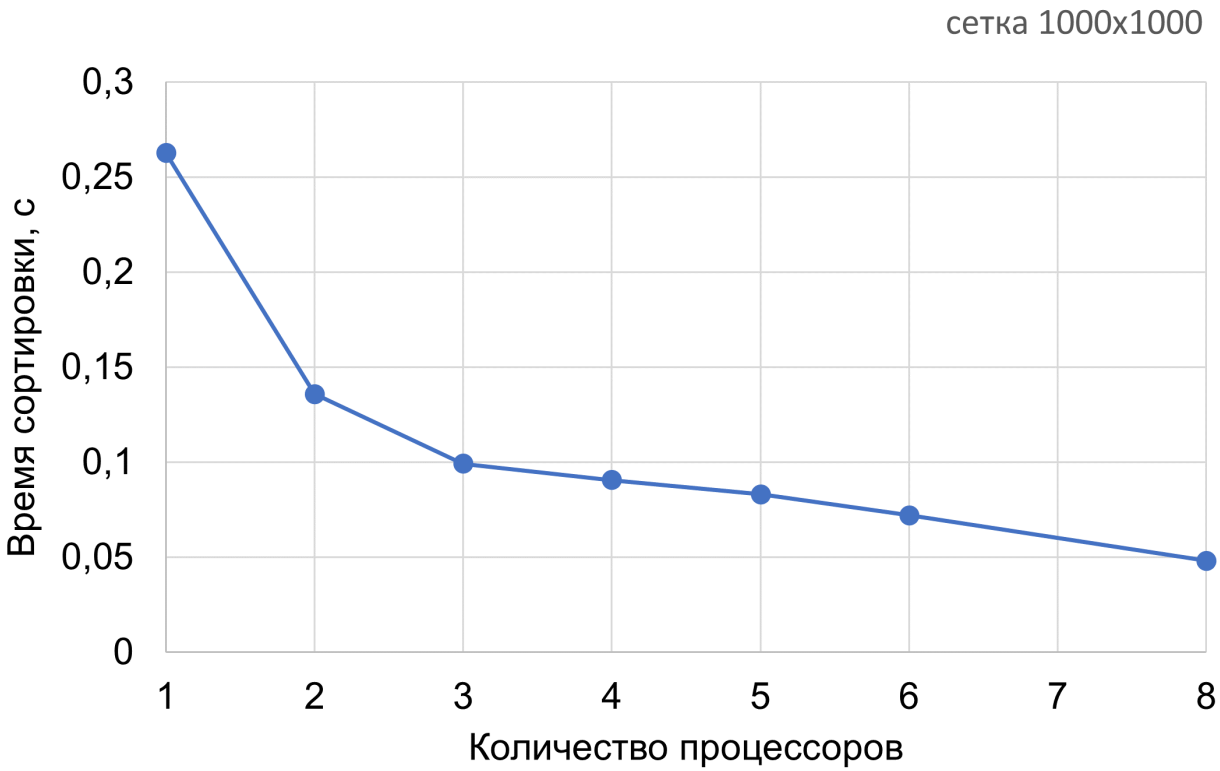
Пиковая производительность	55.84 TFlop/s
Производительность (Linpack)	40.39 TFlop/s
Вычислительных узлов	5

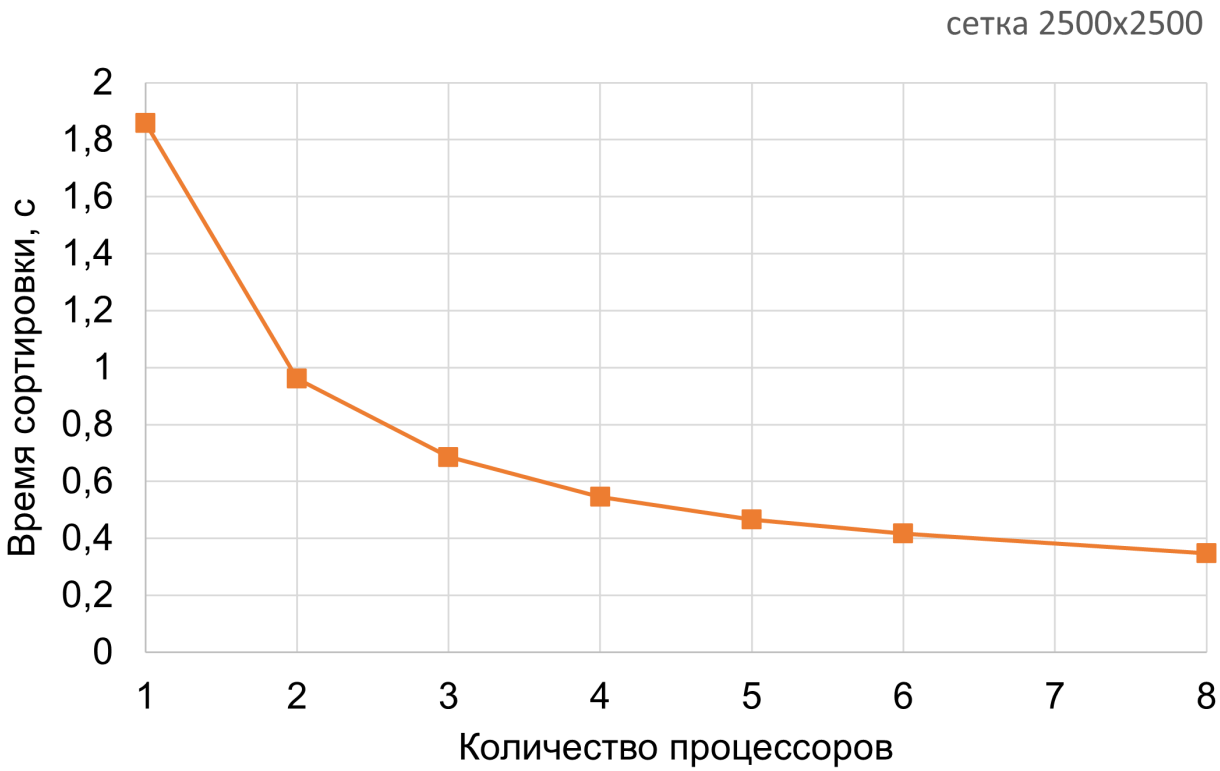
На каждом узле:

Процессоры IBM Power 8	2
NVIDIA Tesla P100	2
Число процессорных ядер	20
Число потоков на ядро	8
Оперативная память	256 Гбайт (1024 Гбайт узел 5)

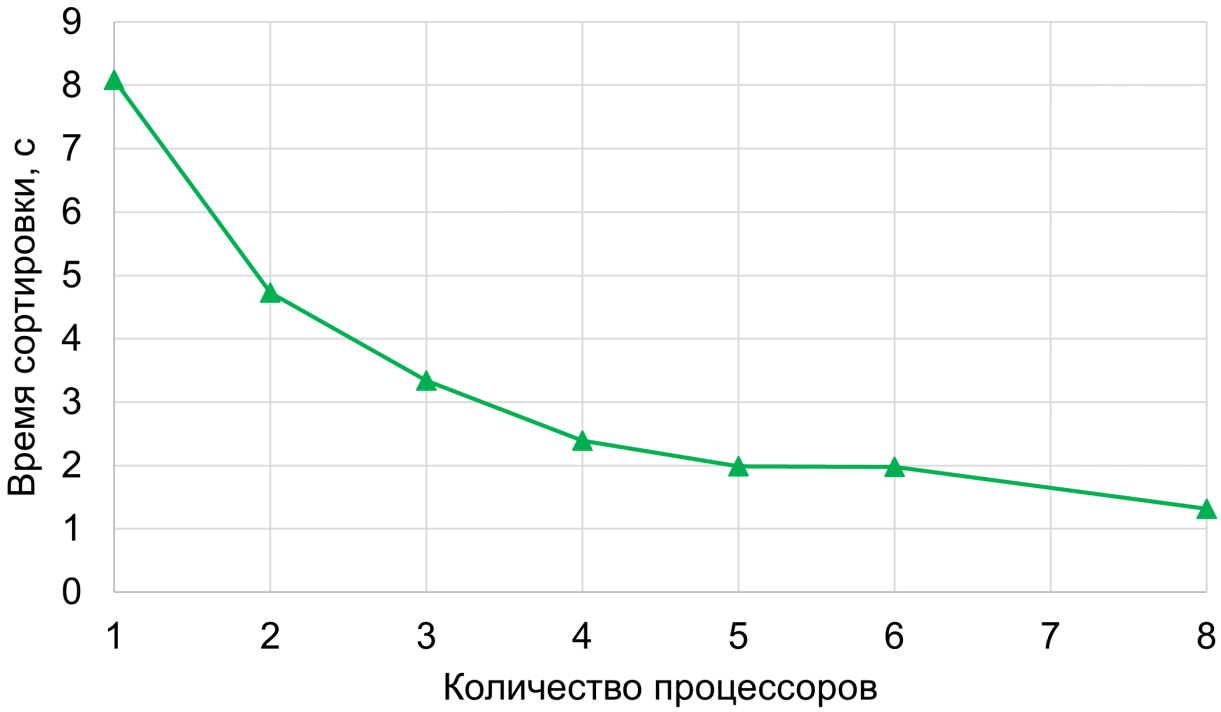
Коммуникационная сеть	Infiniband EDR/ 100 Gb
Система хранения данных	GPFS
Операционная система	Linux Red Hat 7.9

4 Результаты расчетов и их анализ

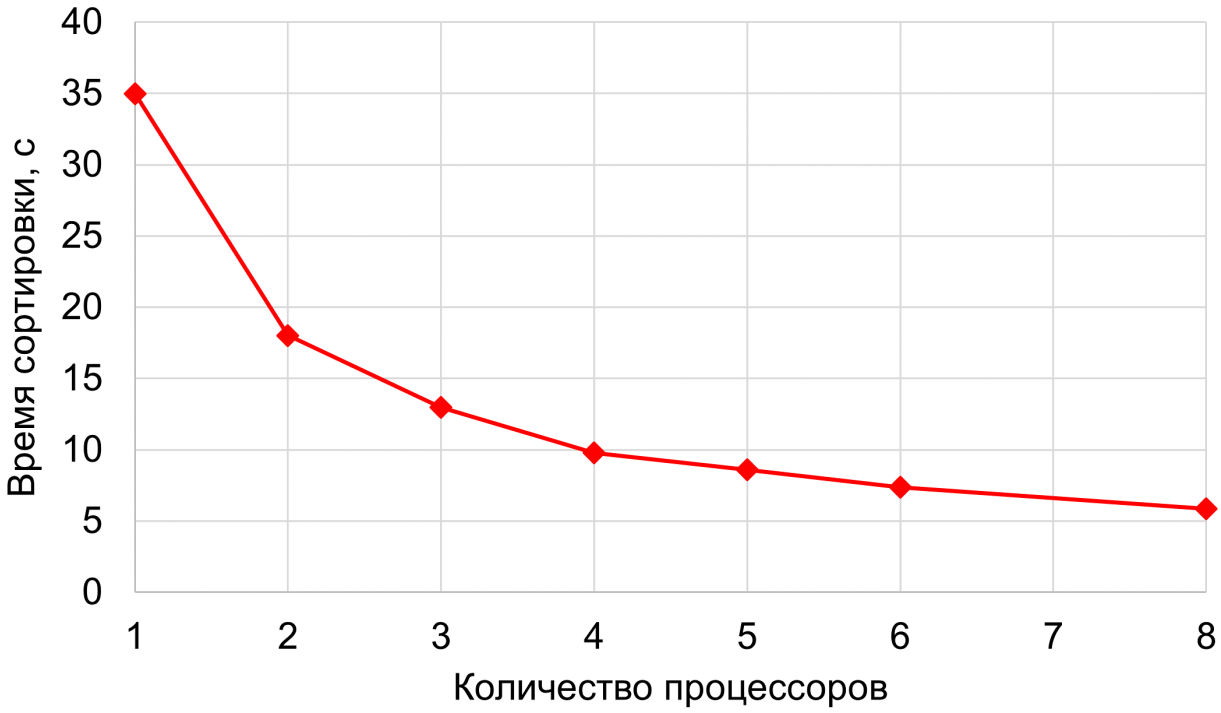


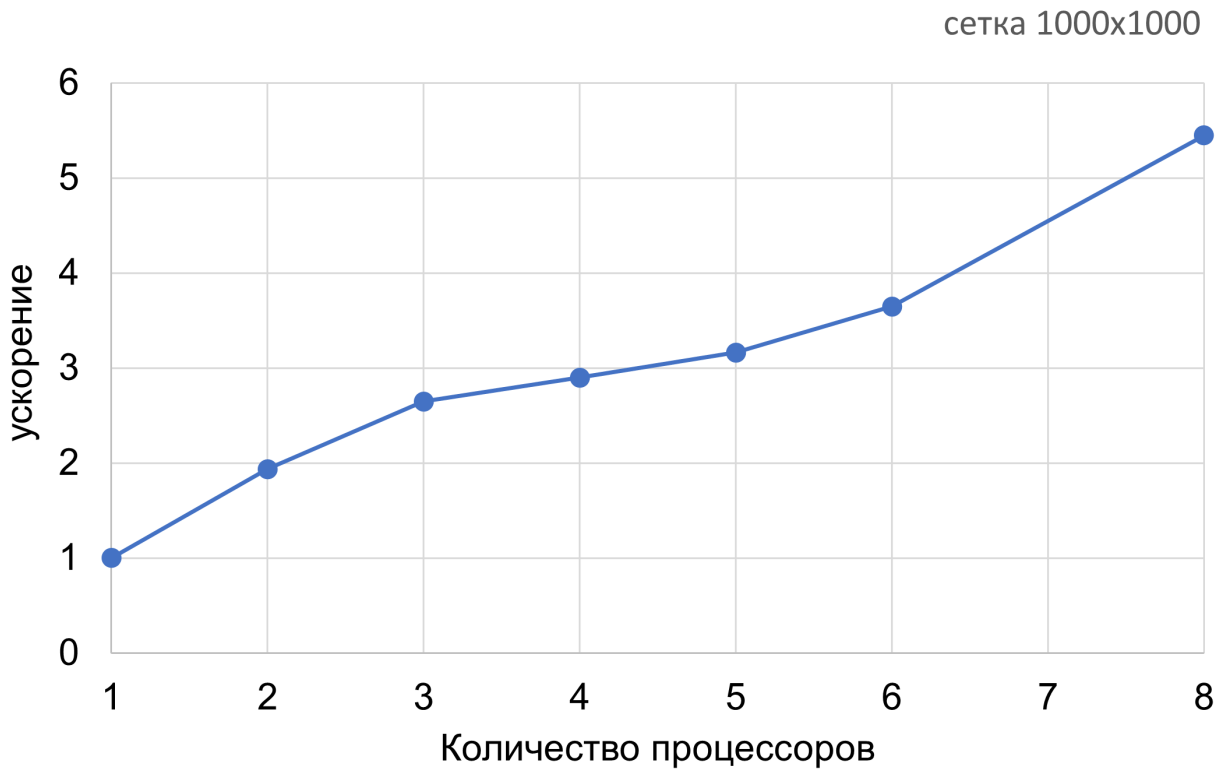


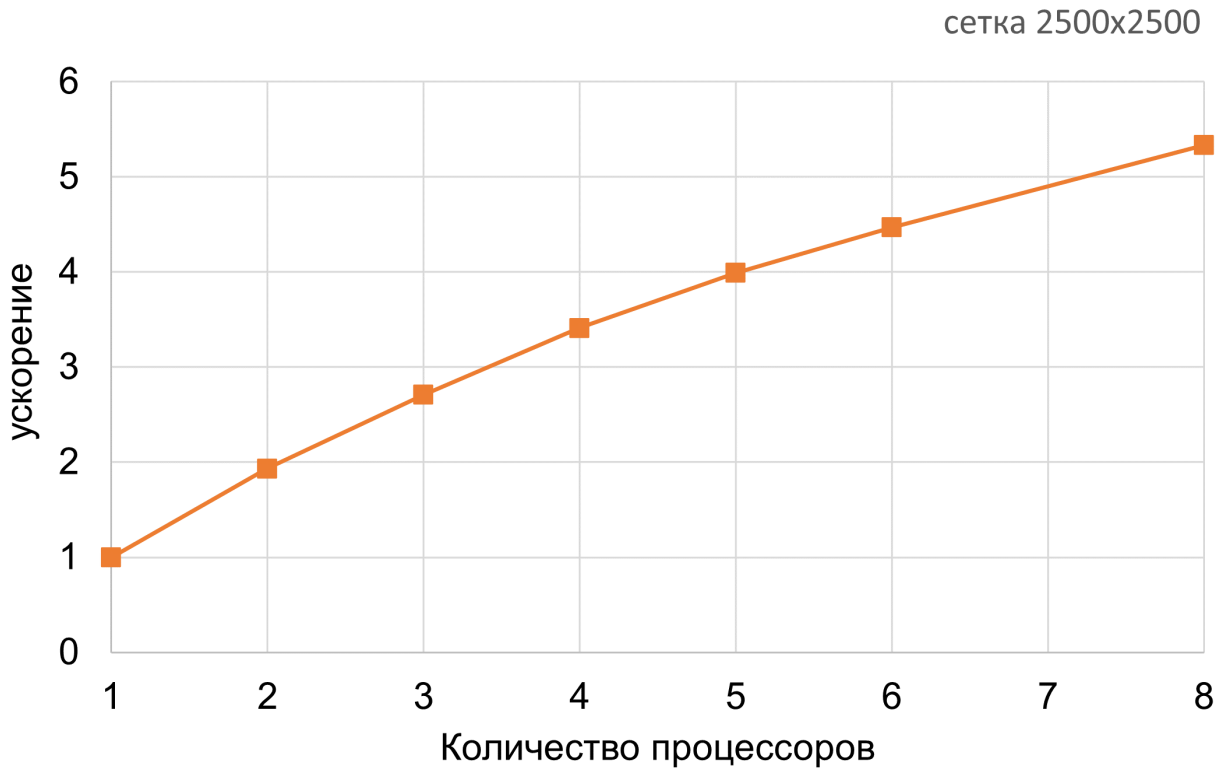
сетка 5000x5000



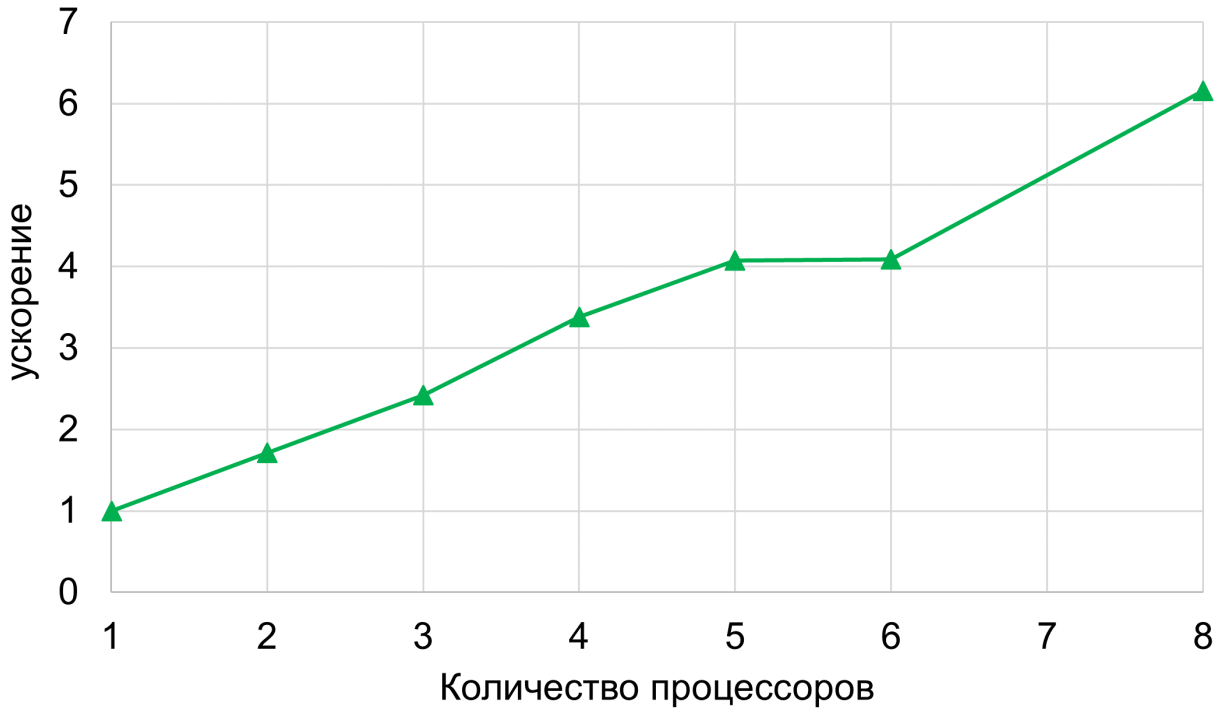
сетка 10000x10000



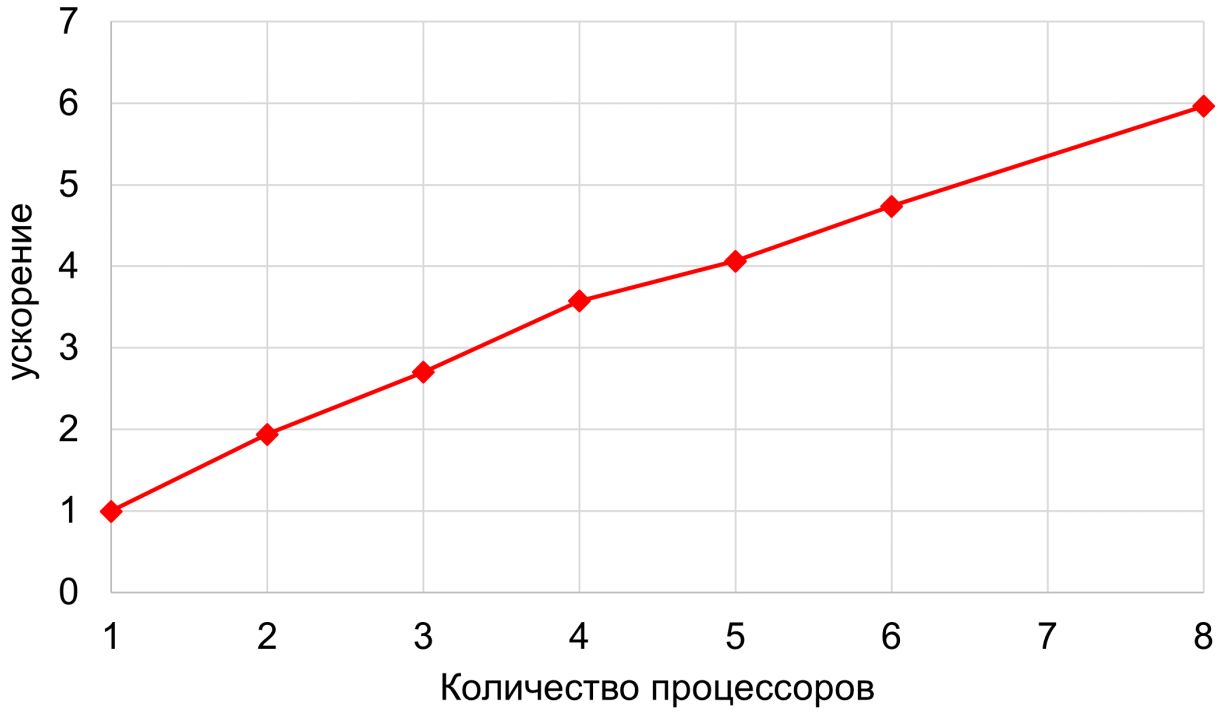

сетка 1000x1000
ускорение
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
Количество процессоров


сетка 2500x2500
ускорение
0
1
2
3
4
5
6
1
2
3
4
5
6
7
8
Количество процессоров

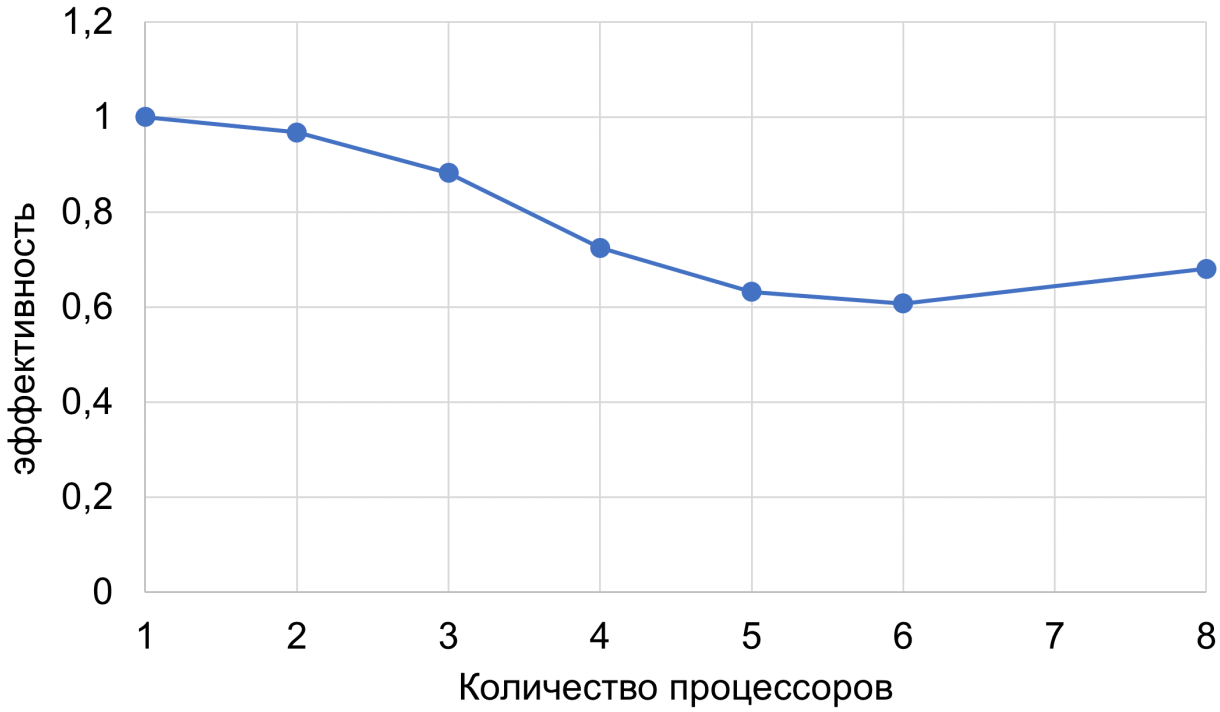
сетка 5000x5000



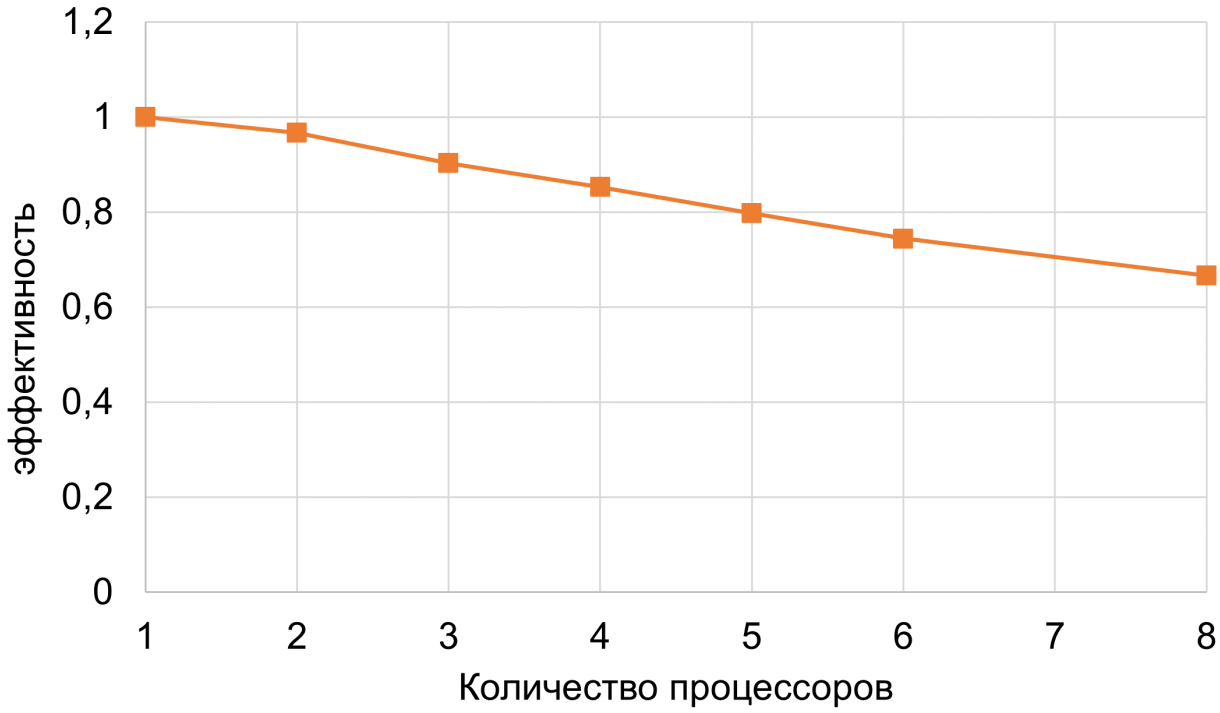
сетка 10000x10000

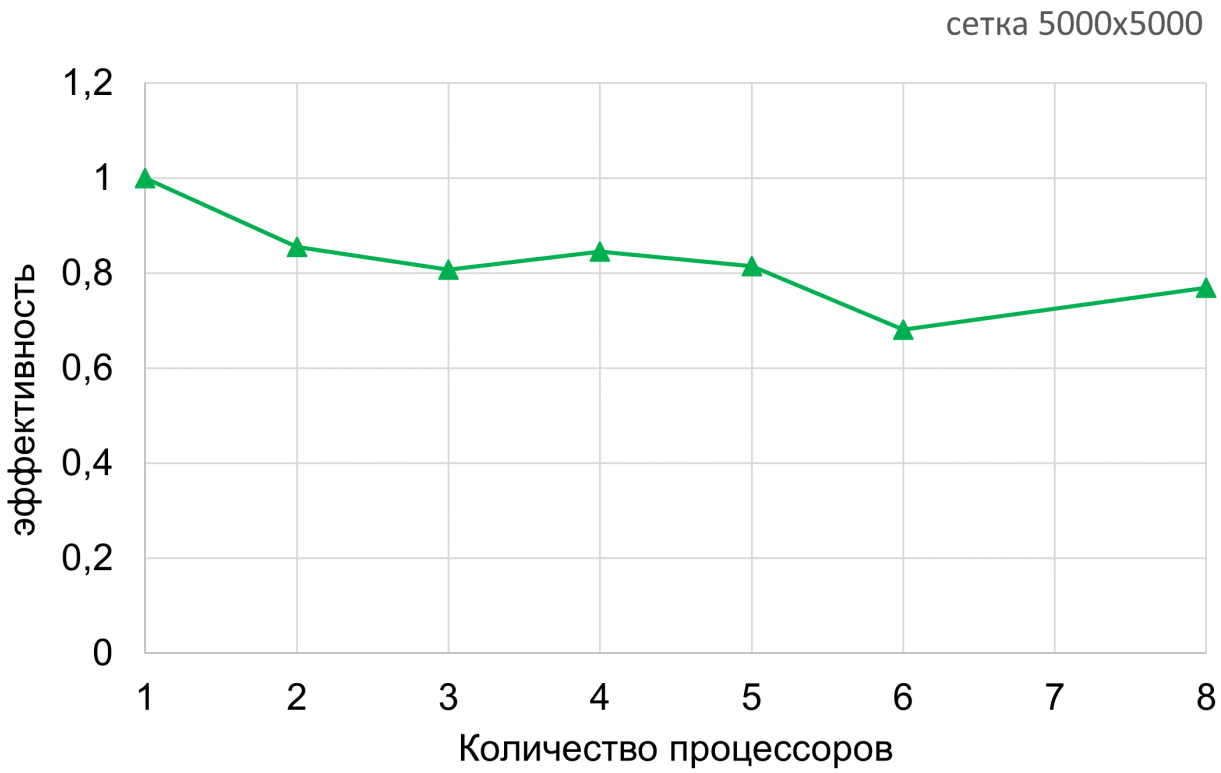


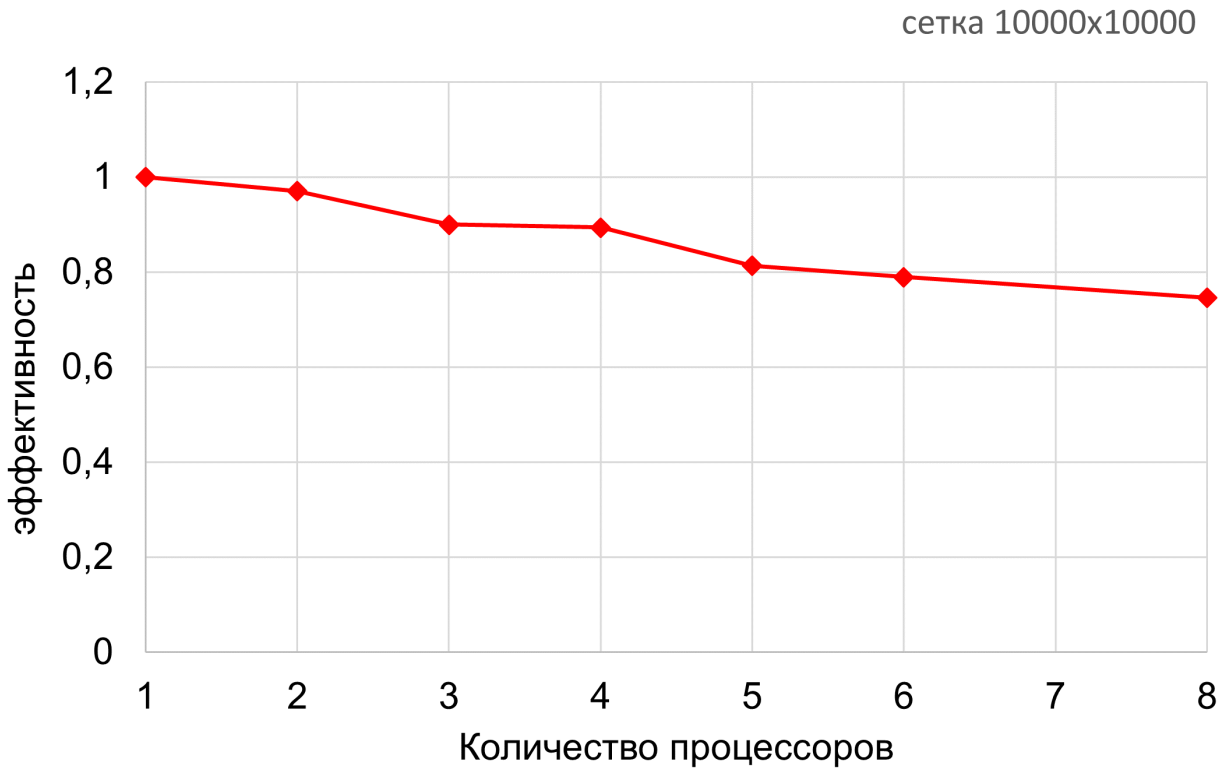
сетка 1000x1000



сетка 2500x2500







Максимально возможная эффективность всего алгоритма, полученная в предположении отсутствия затрат на передачу данных (например, при сортировке на системах с общей памятью), составляет:

$$E^{\max}(n,p) \approx \left(1 + \frac{\log_n p}{2} \log_2 \frac{p}{2}\right)^{-1}.$$

Тогда максимально возможное значение ускорения вычисляется следующим

образом:

$$S^{\max}(n,p) = p \cdot E^{\max}(n,p),$$

где n - число элементов массива, а p - число процессоров.

Ниже в таблицах приведено время расчета T, сек в зависимости от числа используемых процессоров p, а также сравнение расчетных значений ускорения S и эффективности E с их максимально возможными теоретическими оценками S^{max} и E^{max}. Параметр iCountTackts означает число тактов в параллельной сортировке Бэтчера.

На основе представленных таблиц можно сделать вывод о хорошей эффективности параллельной версии программы.

Таблица 1: Параметры расчета для сетки 1000x1000

p	T, сек	S	E	S^{max}	E^{max}	iCountTackts
1	0,2628	1,0000	1,0000	1,0000	1,0000	0
2	0,1358	1,9350	0,9675	2,0000	1,0000	1
3	0,0993	2,6473	0,8824	2,9318	0,9773	3
4	0,0907	2,8985	0,7246	3,8089	0,9522	3
5	0,0831	3,1612	0,6322	4,6425	0,9285	5
6	0,0720	3,6477	0,6079	5,4408	0,9068	6
8	0,0482	5,4472	0,6809	6,9534	0,8692	6

Таблица 2: Параметры расчета для сетки 2500x2500

p	T, сек	S	E	S^{max}	E^{max}	iCountTackts
1	1,8586	1,0000	1,0000	1,0000	1,0000	0
2	0,9605	1,9351	0,9676	2,0000	1,0000	1
3	0,6857	2,7106	0,9035	2,9396	0,9799	3
4	0,5449	3,4108	0,8527	3,8303	0,9576	3
5	0,4661	3,9875	0,7975	4,6817	0,9363	5
6	0,4162	4,4661	0,7444	5,5008	0,9168	6
8	0,3483	5,3356	0,6669	7,0616	0,8827	6

Таблица 3: Параметры расчета для сетки 5000x5000

p	T, сек	S	E	S^{max}	E^{max}	iCountTackts
1	8,0870	1	1,0000	1,0000	1,0000	0
2	4,7272	1,7107	0,8554	2,0000	1,0000	1
3	3,3387	2,4222	0,8074	2,9445	0,9815	3
4	2,3909	3,3825	0,8456	3,8436	0,9609	3
5	1,9861	4,0719	0,8144	4,7061	0,9412	5
6	1,9795	4,0854	0,6809	5,5383	0,9231	6
8	1,3132	6,1580	0,7698	7,1297	0,8912	6

Таблица 4: Параметры расчета для сетки 10000x10000

p	T, сек	S	E	S^{max}	E^{max}	iCountTackts
1	34,9846	1,0000	1,0000	1,0000	1,0000	0
2	18,0224	1,9412	0,9706	2,0000	1,0000	1
3	12,9588	2,6997	0,8999	2,9486	0,9829	3
4	9,7858	3,5751	0,8938	3,8549	0,9637	3
5	8,6021	4,0670	0,8134	4,7270	0,9454	5
6	7,3854	4,7370	0,7895	5,5706	0,9284	6
8	5,8631	5,9669	0,7459	7,1885	0,8986	6

Список литературы

[1] Якобовский М.В. Введение в параллельные методы решения задач: Учебное пособие / Предисл.: В. А. Садовничий. – М.: Издательство Московского университета, 2012. – 328 с., илл. – (Серия «Суперкомпьютерное образование»)

[2] Горобец А.В. Параллельные методы решения задач. Учебный курс для студентов магистратуры ВМК МГУ. – 232 с.

[3] Корнеев В.Д. Параллельное программирование в MPI. - Москва-Ижевск: Институт компьютерных исследований, 2003. - 304 с.

[4] А.С.Антонов Параллельное программирование с использованием технологии MPI.-М.: Изд-во МГУ, 2004.-71 с.

[5] В.В.Воеводин, Вл.В.Воеводин Параллельные вычисления - СПб.: БХВ-Петербург, 2002.-608 с.

A Приложение

Код программы

```
#include <mpi.h>
#include <stdio.h>
#include <cmath>
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>

using namespace std;

vector<pair<int, int>> vComparators; // вектор компаратаров
typedef vector<pair<int, int>> typeVecPair;
vector<typeVecPair> vTackts; // вектор из тактов

int iCountProc, my_rank; // количество процессоров и ранг текущего процесса
int idxSort = 0; // индекс сортировки, по умолчанию 0 - координата x

int iCountPointsOnProc; // количество точек на процесс

struct Point {
    float coord[2];
    int index;
};

// определяем новый тип MPI для структуры Point
void CreateMPITypePoint(MPI_Datatype& MPI_Type_Point) {
    int CountFields = 2;
    int LengthField[2] = { 2, 1 };
    MPI_Datatype TypeField[2] = { MPI_FLOAT, MPI_INT };
    MPI_Aint OffsetField[2];
    OffsetField[0] = offsetof(Point, coord);
    OffsetField[1] = offsetof(Point, index);
    MPI_Type_create_struct(CountFields, LengthField, OffsetField, TypeField, &
    MPI_Type_Point);
    MPI_Type_commit(&MPI_Type_Point);
}

float x(int i, int j) {
    return (float)i * i + j * j;
}

float y(int i, int j) {
    return (float)i * i - j * j;
}

// генерация точек, n1, n2 - размер сетки, iCountProc -  число процессоров
void InitPoints(vector<Point>& vPoints, int n1, int n2, int iCountProc) {
    Point curPoint;

    // количество точек, которые надо добавить на последний процессор
    int iAdd = ((n1 * n2) % iCountProc == 0) ? 0 : iCountProc - (n1 * n2) %
    iCountProc;

    // генерируем точки для основной сетки
    for (int i = 0; i < n1; i++) {
        for (int j = 0; j < n2; j++) {
            curPoint.index = n2 * i + j;
            curPoint.coord[0] = x(i, j);
            curPoint.coord[1] = y(i, j);
```

```
58                  vPoints.push_back(curPoint);
59              }
60          }
61          // добавляем фиктивные точки для последнего процессора
62          for (int i = 0; i < iAdd; i++) {
63              curPoint.index = -1;
64              curPoint.coord[0] = 0;
65              curPoint.coord[1] = 0;
66              vPoints.push_back(curPoint);
67          }
68      };
69
70      // печать точек
71      void PrintPoints(const vector<Point>& vPoints) {
72          //cout << "Points (size: " << vPoints.size() << "): " << "rank " << my_rank
            << endl;
73          for (int i = 0; i < vPoints.size(); i++) {
74              cout << vPoints[i].coord[0] << " " << vPoints[i].coord[1] << " " <<
            vPoints[i].index << " - rank " << my_rank << endl;
75          }
76      }
77
78      // печать компараторов
79      void PrintComparators(const typeVecPair& vComparators) {
80          cout << "Comparators (size: " << vComparators.size() << "):" << endl;
81          for (int i = 0; i < vComparators.size(); i++) {
82              cout << vComparators[i].first << " " << vComparators[i].second << endl;
83          }
84
85      }
86
87      // печать тактов
88      void PrintTackts(const vector<typeVecPair>& vTackts) {
89          cout << "Tackts (size: " << vTackts.size() << "):" << endl;
90          for (int i = 0; i < vTackts.size(); i++) {
91              for (int j = 0; j < vTackts[i].size(); j++) cout << "(" << vTackts[i][j
            ].first << " " << vTackts[i][j].second << ") ";
92              cout << endl;
93          }
94      }
95
96      // сравнение точек для стандартной функции сортировки
97      int ComparePoints(const void* First, const void* Second)
98      {
99          Point FPoint = *((const Point*)First);
100         Point SPoint = *((const Point*)Second);
101
102         if (FPoint.coord[idxSort] < SPoint.coord[idxSort]) return -1;
103         if (FPoint.coord[idxSort] > SPoint.coord[idxSort]) return 1;
104         return 0;
105     }
106
107     // возвращает true, если первая точка больше второй; idxSort - индекс сортировки (0,
        если x; 1, если y)
108     bool bGreater(Point First, Point Second, int idxSort) {
109         return First.coord[idxSort] > Second.coord[idxSort];
110     }
111
112     // возвращает true, если первая точка меньше либо равна второй; idxSort - индекс
        сортировки (0, если x; 1, если y)
113     bool bLess(Point First, Point Second, int idxSort) {
114         return First.coord[idxSort] <= Second.coord[idxSort];
115     }
```

```
116
117 // Join - S - слияние массивов Бэтчер
118 void Join(int iFirst1, int iFirst2, int iStep, int iCount1, int iCount2) {
119     int iCountOdd1, iCountEven2, i;
120
121     if (iCount1 * iCount2 < 1) return;
122     if (iCount1 == 1 && iCount2 == 1) {
123         vComparators.push_back(make_pair(iFirst1, iFirst2));
124         return;
125     }
126
127     iCountOdd1 = iCount1 - (iCount1 / 2);
128     iCountEven2 = iCount2 - (iCount2 / 2);
129
130     Join(iFirst1, iFirst2, 2 * iStep, iCountOdd1, iCountEven2);
131     Join(iFirst1 + iStep, iFirst2 + iStep, 2 * iStep, iCount1 - iCountOdd1,
    iCount2 - iCountEven2);
132
133     for (i = 1; i < iCount1 - 1; i += 2) {
134         vComparators.push_back(make_pair(iFirst1 + iStep * i, iFirst1 + iStep *
     (i + 1)));
135     }
136     if (iCount1 % 2 == 0) {
137         vComparators.push_back(make_pair(iFirst1 + iStep * (iCount1 - 1),
    iFirst2));
138         i = 1;
139     }
140     else i = 0;
141
142     for (; i < iCount2 - 1; i += 2) {
143         vComparators.push_back(make_pair(iFirst2 + iStep * i, iFirst2 + iStep *
     (i + 1)));
144     }
145
146 }
147
148 // Sort - B - сортировка массива Бэтчер
149 void Sort(int iFirst, int iStep, int iCount) {
150     if (iCount < 2) return;
151     if (iCount == 2) {
152         vComparators.push_back(make_pair(iFirst, iFirst + iStep));
153         return;
154     }
155
156     int iCount1;
157     if (iCount % 2 != 0) iCount1 = iCount / 2 + 1;
158     else iCount1 = iCount / 2;
159
160     Sort(iFirst, iStep, iCount1);
161     Sort(iFirst + iStep * iCount1, iStep, iCount - iCount1);
162     Join(iFirst, iFirst + iStep * iCount1, iStep, iCount1, iCount - iCount1);
163 }
164
165 // проверям можно ли добавить компаратор в текущий такт
166 bool bAddComp(pair<int, int> Comp, typeVecPair vTact) {
167     bool bOK = true;
168     int n = vTact.size();
169     int i = 0;
170     while (bOK && i < n) {
171         bOK = (Comp.first != vTact[i].first && Comp.first != vTact[i].second &&
     Comp.second != vTact[i].first && Comp.second != vTact[i].second);
172         i++;
173     }
```

```
174     return bOK;
175 }
176 
177 // раскидываем компараторы по тактам
178 void CreateTackts(typeVecPair& vComps, vector<typeVecPair>& vTackts) {
179     reverse(vComps.begin(), vComps.end()); // инвертируем вектор компараторов
180 
181     vTackts.resize(1);
182     vTackts[0].push_back(vComps[vComps.size()-1]);
183     vComps.pop_back();
184 
185     while (!vComps.empty()) { // пока вектор компараторов не пуст
186         int n = vTackts.size();
187         int idxCurComps = vComps.size()-1;
188         int i = n - 1;
189         // определяем номер такта в который можно вставить текущий компаратор
190         while (i >= 0 && bAddComp(vComps[idxCurComps], vTackts[i])) i--;
191         i++; // номер такта в который нужно добавить текущий компаратор
192 
193         if (i >= n) { // если тактов не хватает, то создаем новый такт
194             vTackts.resize(++n);
195         }
196         vTackts[i].push_back(vComps[idxCurComps]); // добавляем компаратор в
    нужный такт
197         vComps.pop_back(); // удаляем текущий компаратор из вектора компараторов
198     }
199 }
200 
201 // Merge - слияние массивов между процессорами, книга стр. 152
202 void MergeOld(vector<Point>& vFirst, vector<Point>& vSecond, int rank1, int
    rank2) {
203     int iSize = vFirst.size();
204     vector<Point> vResult(iSize);
205 
206     if (my_rank == rank1) { // формирование массива, содержащего меньшие элементы
    массивов iFirst и iSecond
207         for (int iF = 0, iS = 0, k = 0; k < iSize;) {
208             if (bLess(vFirst[iF], vSecond[iS], idxSort)) vResult[k++] = vFirst[
    iF++];
209             else vResult[k++] = vSecond[iS++];
210         }
211         vFirst = vResult;
212     }
213     else if (my_rank == rank2) { // формирование массива, содержащего большие
    элементы массивов iFirst и iSecond
214         for (int iF = iSize - 1, iS = iSize - 1, k = iSize - 1; k >= 0;) {
215             if (bGreater(vFirst[iF], vSecond[iS], idxSort)) vResult[k--] =
    vFirst[iF--];
216             else vResult[k--] = vSecond[iS--];
217         }
218         vSecond = vResult;
219     }
220 }
221 
222 void Merge(vector<Point>& vFirst, vector<Point>& vSecond, int rank1, int rank2)
    {
223     int iSize = vFirst.size();
224     vector<Point> vResult(iSize);
225 
226     if (my_rank == rank1) { // формирование массива, содержащего меньшие элементы
    массивов iFirst и iSecond
227         for (int iF = 0, iS = 0, k = 0; k < iSize;) {
228             if (bLess(vFirst[iF], vSecond[iS], idxSort)) vResult[k++] = vFirst[
```

```
    iF++];
229             else vResult[k++] = vSecond[iS++];
230         }
231         vFirst = vResult;
232     }
233     else if (my_rank == rank2) { // формирование массива, содержащего большие
    элементы массивов iFirst и iSecond
234         int iF = 0, iS = 0, k = 0;
235         // пропускаем младшие элементы, которые ушли в массив vFirst
236         for (; k < iSize;) {
237             if (bLess(vFirst[iF], vSecond[iS], idxSort)) {
238                 k++; iF++;
239             }
240             else {
241                 k++; iS++;
242             }
243         }
244         k = 0;
245         // формируем массив vSecond из оставшихся элементов
246         for (; k < iSize;) {
247             if (iF < iSize && iS < iSize) {
248                 if (bLess(vFirst[iF], vSecond[iS], idxSort)) vResult[k++] =
    vFirst[iF++];
249                 else vResult[k++] = vSecond[iS++];
250             }
251             else
252                 if (iF < iSize)  vResult[k++] = vFirst[iF++];
253                 else vResult[k++] = vSecond[iS++];
254         }
255         vSecond = vResult;
256     }
257 }
258
259 // параллельная сортировка Бэтчера
260 void ParallelSort(int n1, int n2) {
261     if (my_rank == 0) cout << n1 << ":" << n2 << endl;
262
263     MPI_Status status;
264
265     vector<Point> vPoints; // все точки
266     InitPoints(vPoints, n1, n2, iCountProc); // инициализируем точки
267     iCountPointsOnProc = vPoints.size() / iCountProc; // определяем количество
    точек на процесс
268
269     vector<Point> vPointsOnProc(iCountPointsOnProc); // основные точки на процессе
270     vector<Point> vBufOnProc(iCountPointsOnProc); // буфер для приема точек
271
272     // определяем новый тип MPI для структуры Point
273     MPI_Datatype MPI_Type_Point;
274     CreateMPITypePoint(MPI_Type_Point);
275
276     //if (my_rank == 0) cout << "iCountPointsOnProc:" << iCountPointsOnProc <<
    endl;
277
278     // рассылаем точки по процессам
279     MPI_Scatter(vPoints.data(), iCountPointsOnProc, MPI_Type_Point,
    vPointsOnProc.data(), iCountPointsOnProc, MPI_Type_Point, 0, MPI_COMM_WORLD
    );
280
281     double StartTime = MPI_Wtime(); // время старта сортировки
282
283     // сортируем массивы на каждом процессоре
284     qsort(vPointsOnProc.data(), vPointsOnProc.size(), sizeof(Point),
```

```
ComparePoints);
285  MPI_Barrier(MPI_COMM_WORLD);
286
287  //PrintPoints(vPointsOnProc);
288   //if (my_rank == 0) PrintPoints(vPoints); // печать точек
289
290  Sort(0, 1, iCountProc); // формируем расписание сети слияния
291  // if (my_rank == 0) PrintComparators(vComparators); // печать компараторов
292
293  CreateTackts(vComparators, vTackts); // формируем такты
294  reverse(vTackts.begin(), vTackts.end());
295
296  int iCountTackts = vTackts.size(); // сохраняем число тактов
297
298  //if (my_rank == 0) PrintTackts(vTackts); // печать тактов
299
300  MPI_Barrier(MPI_COMM_WORLD); // все процессы готовы к параллельной сортировке
301
302  while (!vTackts.empty()) { // пока не исчерпали все такты
303      int iCurTackt = vTackts.size() - 1;
304      int iSize = vTackts[iCurTackt].size(); // размер текущего такта
305
306      for (int i = iSize - 1; i >= 0; i--) { // бежим по всем компараторам
текущего такта
307          int rank1 = vTackts[iCurTackt][i].first;
308          int rank2 = vTackts[iCurTackt][i].second;
309          vTackts[iCurTackt].pop_back(); // удаляем текущий компаратор
310
311          // пересылкаприем/ массивов если my_rank == rank1
312          if (my_rank == rank1) {
313              //cout << "(" << rank1 << ", " << rank2 << ")" << endl;
314              MPI_Send(vPointsOnProc.data(), vPointsOnProc.size(),
MPI_Type_Point, rank2, 0, MPI_COMM_WORLD);
315              MPI_Recv(vBufOnProc.data(), vBufOnProc.size(), MPI_Type_Point,
rank2, 0, MPI_COMM_WORLD, &status);
316              Merge(vPointsOnProc, vBufOnProc, rank1, rank2); // слияние
массивов
317          }
318          // пересылкаприем/ массивов если my_rank == rank2
319          if (my_rank == rank2) {
320              MPI_Recv(vBufOnProc.data(), vBufOnProc.size(), MPI_Type_Point,
rank1, 0, MPI_COMM_WORLD, &status);
321              MPI_Send(vPointsOnProc.data(), vPointsOnProc.size(),
MPI_Type_Point, rank1, 0, MPI_COMM_WORLD);
322              Merge(vBufOnProc, vPointsOnProc, rank1, rank2); // слияние
массивов
323          }
324      }
325      MPI_Barrier(MPI_COMM_WORLD); // окончания работы такта
326      //cout << "iCurTackt: " << iCurTackt << endl;
327      vTackts.resize(iCurTackt);
328  }
329
330  //собираем отсортированный массив в vPoints на my_rank == 0
331  //MPI_Gather(vPointsOnProc.data(), iCountPointsOnProc, MPI_Type_Point,
vPoints.data(), iCountPointsOnProc, MPI_Type_Point, 0, MPI_COMM_WORLD);
332  MPI_Barrier(MPI_COMM_WORLD);
333
334  double EndTime = MPI_Wtime(); // время окончания сортировки
335  double SortTime = EndTime - StartTime; // время сортировки
336
337  if (my_rank == 0) {
338      cout << "Sort time: " << SortTime << endl;
```

```
339             cout << "iCountTackts: " << iCountTackts << endl;
340         }
341 
342         //PrintPoints(vPointsOnProc);
343         //if (my_rank == 0) PrintPoints(vPoints);
344 }
345 
346 // последовательная сортировка
347 void SerialSort(int n1, int n2) {
348         cout << n1 << ":" << n2 << endl;
349 
350         vector<Point> vPoints; // все точки
351         InitPoints(vPoints, n1, n2, iCountProc); // инициализируем точки
352 
353         double StartTime = MPI_Wtime(); // время старта сортировки
354         // сортируем массивы на каждом процессоре
355         qsort(vPoints.data(), vPoints.size(), sizeof(Point), ComparePoints);
356         double EndTime = MPI_Wtime(); // время окончания сортировки
357         double SortTime = EndTime - StartTime; // время сортировки
358         cout << "Sort time: " << SortTime << endl;
359 }
360 
361 int main(int argc, char* argv[]) {
362 
363    MPI_Init(&argc, &argv);  // инициализация MPI
364 
365         int n1 = 3, n2 = 2; // размер сетки по умолчанию
366 
367         if (argc > 2) {
368             n1 = atoi(argv[1]);
369             n2 = atoi(argv[2]);
370         }
371 
372         // определяем количество процессоров и ранг текущего процесса
373         MPI_Comm_size(MPI_COMM_WORLD, &iCountProc);
374         MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
375 
376         if (iCountProc > 1) ParallelSort(n1, n2);
377         else SerialSort(n1, n2);
378 
379         MPI_Finalize(); // закрытие MPI
380 }
```