

Министерство науки и Высшего образования Российской Федерации
Севастопольский государственный университет
Кафедра ИС

Отчет
по лабораторной работе №6
«Исследование алгоритмов сортировки данных методами пузырька и Шелла,
используемых при проектировании параллельных вычислительных программных
систем»
по дисциплине
«ТЕОРИЯ РАСПРЕДЕЛЕННЫХ СИСТЕМ И ПАРАЛЛЕЛЬНЫХ
ВЫЧИСЛЕНИЙ»

Выполнил студент группы ИС/б-17-2-о
Горбенко К. Н.
Проверил
Дрозин А.Ю.

Севастополь
2020

1 ЦЕЛЬ РАБОТЫ

Программно реализовать и исследовать эффективность алгоритмов параллельной сортировки с использованием функций библиотеки MPI в сравнении с последовательными версиями тех же алгоритмов.

2 ПОСТАНОВКА ЗАДАЧИ

Выполнить разработку и отладку программы параллельной сортировки данных с использованием вызовов требуемых функций библиотеки MPI в соответствии с вариантом, указанным преподавателем. Дополнительно реализовать последовательный вариант того же метода сортировки. Получить результаты работы программы в виде протоколов сообщений, комментирующих параллельное выполнение процессов и их взаимодействие в ходе выполнения. Оценить эффективность параллельного процесса сортировки в сравнении с последовательным на том же наборе исходных данных. Вариант №1 – Четная-нечетная.

3 ХОД РАБОТЫ

Текст программы:

```
1 #include <iostream>
2
3 using namespace std;
4
5 void OddEvenSort(int* vector, int size);
6 void ShowVector(int vector[], int size);
7
8 int main()
9 {
10     const int size = 15;
11     int A[size] = { 41, 67, 34, 0, 69, 24, 78, 58, 62, 64, 5, 45, 81, 27, 61 };
12
13     ShowVector(A, size);
14     OddEvenSort(A, size);
15     ShowVector(A, size);
16 }
17
18 void OddEvenSort(int* vector, int size)
19 {
20     for (int i = 0; i < size; i++)
21     {
```

```

22     if (i % 2 == 0)
23     {
24         for (int j = 0; j < size; j += 2)
25         {
26             if (j < size - 1)
27             {
28                 if (vector[j] > vector[j + 1])
29                 {
30                     int tmp = vector[j];
31                     vector[j] = vector[j + 1];
32                     vector[j + 1] = tmp;
33                 }
34             }
35         }
36     }
37     else
38     {
39         for (int j = 1; j < size; j += 2)
40         {
41             if (j < size - 1)
42             {
43                 if (vector[j] > vector[j + 1])
44                 {
45                     int tmp = vector[j];
46                     vector[j] = vector[j + 1];
47                     vector[j + 1] = tmp;
48                 }
49             }
50         }
51     }
52 }
53 }
54
55 void ShowVector(int vector[], int size)
56 {
57     for (int i = 0; i < size; i++)
58     {
59         cout << vector[i] << " ";
60     }
61     cout << endl;
62 }
63
64 Программа 2. Параллельный метод сортировки:
65 #include <iostream>
66 #include <mpi.h>
67
68 using namespace std;
69

```

```

70 const int data_tag = 2001;
71 const int N = 15;
72
73 void ShowVector(int vector[], int size);
74 int* GetHalfVector(int vector[], int size, bool mode);
75 int Partition(int vector[], int start, int end);
76 void Quicksort(int vector[], int start, int end);
77 int* MergeSort(int firstVector[], int secondVector[], int firstVectorSize, int
    secondVectorSize);
78
79 int main(int argc, char** argv)
80 {
81     // 41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
82     int rank, processes;
83     MPI_Init(&argc, &argv);
84     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
85     MPI_Comm_size(MPI_COMM_WORLD, &processes);
86     MPI_Status status;
87
88     int masterProcess = 0;
89     int vector[N];
90     int* sortedVector = new int[N];
91     int blockSize = N / processes;
92     int* blockVector = new int[blockSize];
93
94     if (rank == masterProcess)
95     {
96         for (int i = 0; i < N; i++)
97         {
98             vector[i] = 0 + rand() % 100;
99         }
100
101         cout << "Unsorted vector: ";
102         ShowVector(vector, N);
103         cout << endl;
104     }
105
106     MPI_Scatter(vector, blockSize, MPI_INT, blockVector, blockSize, MPI_INT,
        masterProcess, MPI_COMM_WORLD);
107
108 #pragma region DEBUG
109     cout << "P" << rank << "-unsorted: ";
110     ShowVector(blockVector, blockSize);
111     MPI_Barrier(MPI_COMM_WORLD);
112 #pragma endregion
113
114     Quicksort(blockVector, 0, blockSize - 1);
115

```

```

116 #pragma region DEBUG
117     cout << "P" << rank << "-sorted: ";
118     ShowVector(blockVector, blockSize);
119 #pragma endregion
120
121     MPI_Barrier(MPI_COMM_WORLD);
122
123     for (int i = 0; i < processes - 1; i++)
124     {
125         if (i % 2 == 0)
126         {
127             if (rank % 2 == 0)
128             {
129                 if (rank != processes - 1)
130                 {
131                     int* blockVectorFromNext = new int[blockSize];
132                     MPI_Send(blockVector, blockSize, MPI_INT, rank + 1,
133                             data_tag, MPI_COMM_WORLD);
134                     MPI_Recv(blockVectorFromNext, blockSize, MPI_INT, rank + 1,
135                             data_tag, MPI_COMM_WORLD, &status);
136
137                     int* merged = MergeSort(blockVector, blockVectorFromNext,
138                                             blockSize, blockSize);
139                     blockVector = GetHalfVector(merged, blockSize * 2, 0);
140
141                     #pragma region DEBUG
142                     cout << "P" << rank << "-merged-sorted: ";
143                     ShowVector(merged, blockSize * 2);
144                     #pragma endregion
145
146                     delete[] blockVectorFromNext;
147                     delete[] merged;
148                 }
149             }
150             else
151             {
152                 int* blockVectorFromPrev = new int[blockSize];
153                 MPI_Send(blockVector, blockSize, MPI_INT, rank - 1, data_tag,
154                         MPI_COMM_WORLD);
155                 MPI_Recv(blockVectorFromPrev, blockSize, MPI_INT, rank - 1,
156                         data_tag, MPI_COMM_WORLD, &status);
157
158                 int* merged = MergeSort(blockVector, blockVectorFromPrev,
159                                         blockSize, blockSize);
160                 blockVector = GetHalfVector(merged, blockSize * 2, 1);

```

```

158
159 #pragma region DEBUG
160         cout << "P" << rank << "-merged-sorted: ";
161         ShowVector(merged, blockSize * 2);
162
163         cout << "P" << rank << "-half: ";
164         ShowVector(blockVector, blockSize);
165 #pragma endregion
166
167         delete[] blockVectorFromPrev;
168         delete[] merged;
169     }
170 }
171 else
172 {
173     if (rank % 2 == 0)
174     {
175         if (rank != 0)
176         {
177             int* blockVectorFromPrev = new int[blockSize];
178             MPI_Send(blockVector, blockSize, MPI_INT, rank - 1,
179                     data_tag, MPI_COMM_WORLD);
179             MPI_Recv(blockVectorFromPrev, blockSize, MPI_INT, rank - 1,
180                     data_tag, MPI_COMM_WORLD, &status);
181
182             int* merged = MergeSort(blockVector, blockVectorFromPrev,
183                                     blockSize, blockSize);
184             blockVector = GetHalfVector(merged, blockSize * 2, 1);
185
186             #pragma region DEBUG
187             cout << "P" << rank << "-merged-sorted: ";
188             ShowVector(merged, blockSize * 2);
189
190             cout << "P" << rank << "-half: ";
191             ShowVector(blockVector, blockSize);
192             #pragma endregion
193
194             delete[] blockVectorFromPrev;
195             delete[] merged;
196         }
197     }
198     else
199     {
200         int* blockVectorFromNext = new int[blockSize];
201         MPI_Send(blockVector, blockSize, MPI_INT, rank + 1, data_tag,
202                 MPI_COMM_WORLD);
203         MPI_Recv(blockVectorFromNext, blockSize, MPI_INT, rank + 1,
204                 data_tag, MPI_COMM_WORLD, &status);

```

```

201
202         int* merged = MergeSort(blockVector, blockVectorFromNext,
203                                 blockSize, blockSize);
204         blockVector = GetHalfVector(merged, blockSize * 2, 0);
205 #pragma region DEBUG
206         cout << "P" << rank << "-merged-sorted: ";
207         ShowVector(merged, blockSize * 2);
208
209         cout << "P" << rank << "-half: ";
210         ShowVector(blockVector, blockSize);
211 #pragma endregion
212
213         delete[] blockVectorFromNext;
214         delete[] merged;
215     }
216 }
217
218     MPI_Barrier(MPI_COMM_WORLD);
219 }
220
221 MPI_Barrier(MPI_COMM_WORLD);
222 MPI_Gather(blockVector, blockSize, MPI_INT, sortedVector, blockSize,
223           MPI_INT, masterProcess, MPI_COMM_WORLD);
224 MPI_Barrier(MPI_COMM_WORLD);
225
226 if (rank == masterProcess)
227 {
228     cout << endl << "Sorted vector: ";
229     ShowVector(sortedVector, N);
230 }
231
232 delete[] blockVector;
233
234 MPI_Finalize();
235
236 return 0;
237 }
238
239 void ShowVector(int vector[], int size)
240 {
241     for (int i = 0; i < size; i++)
242     {
243         cout << vector[i] << " ";
244     }
245     cout << endl;
246 }

```

```

247
248 int* GetHalfVector(int vector[], int size, bool mode)
249 {
250     int* result = new int[size / 2];
251
252     if (mode)
253     {
254         copy(vector + size / 2, vector + size, result);
255     }
256     else
257     {
258         copy(vector, vector + size / 2, result);
259     }
260
261     return result;
262 }
263
264 int Partition(int vector[], int start, int end)
265 {
266     int pivot = vector[end];
267     int pIndex = start;
268
269     for (int i = start; i < end; ++i)
270     {
271         if (vector[i] < pivot)
272         {
273             swap(vector[i], vector[pIndex]);
274             pIndex++;
275         }
276     }
277
278     swap(vector[pIndex], vector[end]);
279
280     return pIndex;
281 }
282
283 void Quicksort(int vector[], int start, int end)
284 {
285     int* stack = (int*)malloc((end - start + 1) * sizeof(int));
286     int top = -1;
287     stack[++top] = start;
288     stack[++top] = end;
289
290     while (top >= 0)
291     {
292         end = stack[top--];
293         start = stack[top--];
294

```



```

295         int pivot_index = Partition(vector, start, end);
296
297         if (pivot_index - 1 > start)
298         {
299             stack[++top] = start;
300             stack[++top] = pivot_index - 1;
301         }
302         if (pivot_index + 1 < end)
303         {
304             stack[++top] = pivot_index + 1;
305             stack[++top] = end;
306         }
307     }
308 }
309
310 int* MergeSort(int firstVector[], int secondVector[], int firstVectorSize, int
    secondVectorSize)
311 {
312     int i = 0;
313     int j = 0;
314     int index = 0;
315     int* result = new int[firstVectorSize + secondVectorSize];
316
317     while (i < firstVectorSize && j < secondVectorSize)
318     {
319         if (firstVector[i] < secondVector[j])
320         {
321             result[index] = firstVector[i];
322             i++;
323         }
324         else
325         {
326             result[index] = secondVector[j];
327             j++;
328         }
329         index++;
330     }
331
332     while (i < firstVectorSize)
333     {
334         result[index] = firstVector[i];
335         index++;
336         i++;
337     }
338
339     while (j < secondVectorSize)
340     {
341         result[index] = secondVector[j];

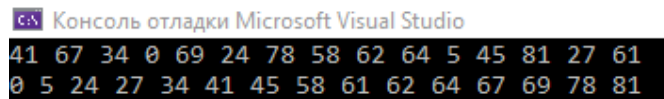
```

```

342         index++;
343         j++;
344     }
345
346     return result;
347 }

```

Первым этапом был разработан последовательный алгоритм чет-нечетной сортировки. Результат сортировки таким методом представлен на рисунке 1.



Консоль отладки Microsoft Visual Studio

41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
 0 5 24 27 34 41 45 58 61 62 64 67 69 78 81

Рисунок 1 – Результат выполнения последовательного алгоритма чет-нечетной сортировки

Далее было осуществлено моделирование параллельного алгоритма чет-нечетной сортировки. Каждый этап был проведен вручную, что позволило полностью понять, как работает данный алгоритм. Данные для сортировки были взяты из примера выше. Результат показан на рисунке 2.

0				1				2				3				4			
41	67	34		0	69	24		78	58	62		64	5	45		81	27	61	
34	41	67		0	24	69		58	62	78		5	45	64		27	61	81	
0	24	34		41	67	69		5	45	58		62	64	78		27	61	81	
0	24	34		5	41	45		58	67	69		27	61	62		64	78	81	
0	5	24		34	41	45		27	58	61		62	67	69		64	78	81	
0	5	24		27	34	41		45	58	61		62	64	67		69	78	81	

Рисунок 2 – Результат моделирования параллельного алгоритма чет-нечетной сортировки

Как видим, из рисунка 2 следует, что существует processes – 1 итераций из четной и нечетной фазы. На каждой четной фазе процессы (0 1), (2 3) и т.д. обмениваются блоками и составляют общий массив и сортируют его, после этого каждый оставляет себе половину по правилу: процесс с меньшим номером оставляет себе левую половину, процесс с большим – правую. Таким же образом действуют процессы на нечетной фазе, однако уже (1 2), (3 4) и т.д.

После всех итераций мы собираем данные с процессов и получаем отсортированный массив.

```

Unsorted vector: 41 67 34 0 69 24 78 58 62 64 5 45 81 27 61
P0-unsorted: 41 67 34
P4-unsorted: 81 27 61
P1-unsorted: 0 69 24
P2-unsorted: 78 58 62
P3-unsorted: 64 5 45
P2-sorted: 58 62 78
P4-sorted: 27 61 81
P3-sorted: 5 45 64
P1-sorted: 0 24 69
P0-sorted: 34 41 67
P0-merged-sorted: 0 24 34 41 67 69
P0-half: 0 24 34
P1-merged-sorted: 0 24 34 41 67 69
P1-half: 41 67 69
P3-merged-sorted: 5 45 58 62 64 78
P2-merged-sorted: 5 45 58 62 64 78
P3-half: 62 64 78
P2-half: 5 45 58
P2-merged-sorted: 5 41 45 58 67 69
P1-merged-sorted: 5 41 45 58 67 69
P2-half: 58 67 69
P4-merged-sorted: 27 61 62 64 78 81
P1-half: 5 41 45
P4-half: 64 78 81
P3-merged-sorted: 27 61 62 64 78 81
P3-half: 27 61 62
P2-merged-sorted: 27 58 61 62 67 69
P3-merged-sorted: 27 58 61 62 67 69
P2-half: 27 58 61
P3-half: 62 67 69
P1-merged-sorted: 0 5 24 34 41 45
P0-merged-sorted: 0 5 24 34 41 45
P1-half: 34 41 45
P0-half: 0 5 24
P1-merged-sorted: 27 34 41 45 58 61
P2-merged-sorted: 27 34 41 45 58 61
P4-merged-sorted: 62 64 67 69 78 81
P2-half: 45 58 61
P4-half: 69 78 81
P1-half: 27 34 41
P3-merged-sorted: 62 64 67 69 78 81
P3-half: 62 64 67
Sorted vector: 0 5 24 27 34 41 45 58 61 62 64 67 69 78 81

```

Рисунок 3 – Результат выполнения параллельного алгоритма чет-нечетной сортировки

ВЫВОДЫ

В ходе данной лабораторной работы были изучены основные понятия составления параллельных методов сортировок их последовательных аналогов. Программно реализован и исследован алгоритм чет-нечетной параллельной сортировки с использованием функций библиотеки MPI в сравнении с последовательными версиями тех же алгоритмов.