SF2568: Homework 3

Michael Hanke (*hanke@nada.kth.se*)

Omar Elshenawy (*omares@kth.se*)
Thomai Stathopoulou (*thomai@kth.se*)

March 16, 2016

Question 1:

(a) In order to sort N elements, we will need to have a processor grid of $N \times (N-1)$ as shown in Figure 1.

In the parallel algorithm described in the lectures, we use N processors for the sorting of N elements, where each processor is "responsible" for one element i and compares this element to the other $N-1$, while counting the number of elements that are lower than the ith element.

In this alternative, we have an entire row of processors that are responsible for an element i and assign the comparisons to all the $N-1$ processors of the ith row. So, basically each processor performs only one comparison, instead of $N-1$, while incrementing the element's rank if necessary.

Therefore $N-1$ processors are used to find the rank of **one** number. Incrementing the counter is done sequentially and requires maximum of N steps. We can adopt a tree-like structure, so that the necessary steps are reduced (Figure 2).

$$\begin{array}{ccccc} p_{0,0} & p_{0,1} & p_{0,2} & \cdots & p_{0,N-1} \\ p_{1,0} & p_{1,1} & p_{1,2} & \cdots & p_{1,N-1} \\ p_{2,0} & p_{2,1} & p_{2,2} & \cdots & p_{2,N-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ p_{N,0} & p_{N,1} & p_{N,2} & \cdots & p_{N,N-1} \end{array}$$

Figure 1: Mesh topology of $N \times (N-1)$ processors

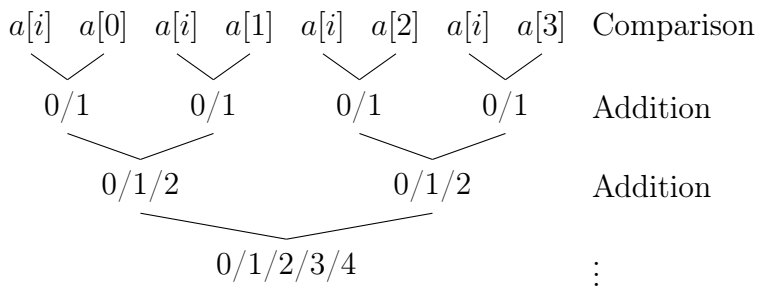

Figure 2: Incrementation of the counter in a tree-like manner

Using the above structure for the comparison of elements and incrementation of their ranks, we achieve an algorithm of time complexity of $\mathcal{O}(\log N)$.

(b) As mentioned before the number of processors that are needed for this implementation is $(N-1) \times N \simeq N^2$.

The sequential Rank Sort has time $T_S^* = \mathcal{O}(N^2)$. The speedup will be:

$$S_P = \frac{T_S^*}{T_P} = \frac{\mathcal{O}(N^2)}{\mathcal{O}(\log N)} = \mathcal{O}\left(\frac{N^2}{\log N}\right). \tag{1}$$

While the speedup is very high, when it comes to processor efficiency, we have:

$$\eta_P = \frac{S_P}{P} \simeq \frac{\mathcal{O}\left(\frac{N^2}{\log N}\right)}{N^2} \simeq \mathcal{O}\left(\frac{1}{\log N}\right), \tag{2}$$

which is a low number and decreases when N (the number of processors) increases.

(c) As mentioned before, the processor efficiency is quite low. Additionally the large demand on actual processors makes the algorithm not very practical to use. If there is a need for N^2 processors for the sorting of N elements, then for large Ns (which would indicate an actual need for parallelizing), we would have a huge demand in hardware, which would then be inefficiently used.

We would therefore **not** use this algorithm in practice, unless it was for experimental/research purposes.

Question 2:

The code is incorrect. Initially there would be caused a deadlock, since all processors try to send data before receiving. For that to be corrected, we only need to switch the order of sending/receiving in one "group" of processes (e.g. even processes first send and then receive, while odd processes first receive and then send).

Additionally, there is no check for processor "overflow", when each process sends or receives from the previous or next process. This would cause the program to crash.

Finally, the comparison and assignment of the elements is incorrect, in terms of what is supposed to be send or received. This would lead in an incorrect ordering of the elements at the end of the program.

A corrected code is shown in the following snippet:

```
1  evenprocess = (rem(i,2) == 0);
2  evenphase = 1;
3  for step = 0:N-1
4      if (evenprocess)
5          if (evenphase)
6          // even phase even processor
7              if (i != last_processor)
8                  send(a, i+1);
9                  receive(x, i+1);
10                 if x < a
11                     a = x;
12                 end
13             end
14         else
15         // odd phase even processor
16             if (i != first_processor)
17                 send(a, i-1);
18                 receive(x, i-1);
19                 if x > a
20                     x = a;
```

```
21                  end
22              end
23          end
24      else
25          if (evenphase)
26          // even phase odd processor
27              if (i != first_processor)
28                  receive(a, i-1);
29                  send(x, i-1);
30                  if x < a
31                      x = a;
32                  end
33              end
34          else
35          // odd phase odd processor
36              if (i != last_processor)
37                  receive(a, i-1);
38                  send(x, i-1);
39                  if x > a
40                      x = a;
41                  end
42              end
43          end
44      end
45      evenphase = ~evenphase;
46  end
```

odd_even.cpp

Question 3:

(a) For Question 3 we implemented the transposition sort algorithm. The code can be seen below. The number of elements is given to the program as an input from the command line. The program generates the elements and proceeds in sorting them. As can be seen both by the code and the example for small N, the program works even for the case where the number of elements cannot be equally distributed to the processes.

```
1  #include <mpi.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <math.h>
5  #include <string.h>
6
7  #define filename "sorted_list.txt"
8
9  int cmpfunc (const void * a, const void * b)
10 {
11   double da = *(double*) a, db = *(double*) b;
```

```
12   if (da > db)
13     return 1;
14   else if (da < db)
15     return -1;
16   else
17     return 0;
18 }
19
20 double* merge(double* A, double* B, int sizeA, int sizeB)
21 {
22   double* list;
23   list = (double *) malloc((sizeA + sizeB) * sizeof
       (double));
24
25   int a = 0, b = 0, idx = 0;
26
27   while (a < sizeA && b < sizeB)
28   {
29     if (A[a] < B[b])
30     {
31       list[idx++] = A[a++];
32     }
33     else
34     {
35       list[idx++] = B[b++];
36     }
37   }
38
39   while (a < sizeA)
40   {
41     list[idx++] = A[a++];
42   }
43
44   while (b < sizeB)
45   {
46     list[idx++] = B[b++];
47   }
48
49   return list;
50 }
51
52
53 int main(int argc, char **argv)
54 {
55   // Local variables
56   int P, p, N, I, i, step, evenphase, *Ip, msg = 1;
57   double *A, *B, start_time, end_time;
58   FILE *fp;
```

```
59
60    MPI_Init(&argc, &argv);
61    MPI_Comm_size(MPI_COMM_WORLD, &P);
62    MPI_Comm_rank(MPI_COMM_WORLD, &p);
63
64    /* Find problem size N from command line */
65    if (argc < 2)
66    {
67      fprintf(stdout, "No size N given");
68      exit(1);
69    }
70    N = atoi(argv[1]);
71
72    start_time = MPI_Wtime();
73
74    /* local size. Modify if P does not divide N */
75    Ip = (int *) malloc(P * sizeof (int)); // Number of
         elements of every processor
76    for (i = 0; i < P; i++)
77    {
78      Ip[i] = (N + P - i - 1) / P;
79    }
80    I = Ip[p]; // Number of elements for current processor
81
82    /* random number generator initialization */
83    srandom(p + 1);
84
85    /* data generation */
86    if (p % 2 == 0)
87    {
88      // Even process - Generate B
89      B = (double *) malloc(I * sizeof(double));
90      for (i = 0; i < I; i++)
91      {
92        B[i] = ((double) random())/((double) RAND_MAX + 1);
93      }
94      // Sort B
95      qsort(B, I, sizeof(double), cmpfunc);
96    }
97    else
98    {
99      // Odd process - Generate A
100     A = (double *) malloc(I * sizeof(double));
101     for (i = 0; i < I; i++)
102     {
103       A[i] = ((double) random())/((double) RAND_MAX + 1);
104     }
105     // Sort A
```

```
106     qsort(A, I, sizeof(double), cmpfunc);
107   }
108
109
110   // Start odd-even sort
111   evenphase = 1;
112   for (step = 0; step < P; step++)
113   {
114     if (p % 2 == 0)
115     {
116       // Even process
117       if (evenphase)
118       {
119         // Even phase
120         if (p < P - 1)
121         {
122           // Allocate A to receive
123           A = (double *) malloc(Ip[p + 1] * sizeof(double));
124           MPI_Recv(A, Ip[p + 1], MPI_DOUBLE, p + 1, 0,
                MPI_COMM_WORLD,
125              MPI_STATUS_IGNORE);
126           MPI_Send(B, I, MPI_DOUBLE, p + 1, 0,
                MPI_COMM_WORLD);
127
128           // Merge A and B
129           double *list;
130           list = (double *) malloc((I + Ip[p + 1]) *
                sizeof(double));
131           list = merge(A, B, Ip[p + 1], I);
132
133           // Keep first half of list
134           memcpy(B, list, I * sizeof(double));
135         }
136       }
137       else
138       {
139         // Odd phase
140         if (p > 0)
141         {
142           // Allocate A to receive
143           A = (double *) malloc(Ip[p - 1] * sizeof(double));
144           MPI_Recv(A, Ip[p - 1], MPI_DOUBLE, p - 1, 0,
                MPI_COMM_WORLD,
145              MPI_STATUS_IGNORE);
146           MPI_Send(B, I, MPI_DOUBLE, p - 1, 0,
                MPI_COMM_WORLD);
147
148           // Merge A and B
```

```
149           double *list;
150           list = (double *) malloc((I + Ip[p - 1]) *
                 sizeof(double));
151           list = merge(A, B, Ip[p - 1], I);
152
153           // Keep second half of list
154           memcpy(B, list + Ip[p - 1], I * sizeof(double));
155         }
156       }
157
158     }
159     else
160     {
161       // Odd process
162       if (evenphase)
163       {
164         // Even phase
165         // Allocate to receive B
166         B = (double *) malloc(Ip[p - 1] * sizeof(double));
167         MPI_Send(A, I, MPI_DOUBLE, p - 1, 0, MPI_COMM_WORLD);
168         MPI_Recv(B, Ip[p - 1], MPI_DOUBLE, p - 1, 0,
                    MPI_COMM_WORLD,
169          MPI_STATUS_IGNORE);
170
171         // Merge A and B
172         double *list;
173         list = (double *) malloc((I + Ip[p - 1]) *
                 sizeof(double));
174         list = merge(A, B, I, Ip[p - 1]);
175
176         // Keep second half of list
177         memcpy(A, list + Ip[p - 1], I * sizeof(double));
178       }
179       else
180       {
181         // Odd phase
182         if (p < P - 1)
183         {
184           // Allocate to receive B
185           B = (double *) malloc(Ip[p + 1] * sizeof(double));
186           MPI_Send(A, I, MPI_DOUBLE, p + 1, 0,
                      MPI_COMM_WORLD);
187           MPI_Recv(B, Ip[p + 1], MPI_DOUBLE, p + 1, 0,
                      MPI_COMM_WORLD,
188            MPI_STATUS_IGNORE);
189
190           // Merge A and B
191           double *list;
```

```
192           list = (double *) malloc((I + Ip[p + 1]) *
                 sizeof(double));
193           list = merge(A, B, I, Ip[p + 1]);
194
195           // Keep first half of list
196           memcpy(A, list, I * sizeof(double));
197         }
198       }
199
200     }
201     evenphase = !evenphase;
202   }
203
204   /// Print lists
205   if (p == 0)
206   {
207     // Create file
208     fp = fopen(filename, "w");
209
210     for (i = 0; i < I; i++)
211     {
212       fprintf(fp, "%f\n", B[i]);
213     }
214
215     // Close file
216     fclose(fp);
217
218     // Send signal to next process
219     if (P > 1)
220     {
221       MPI_Send(&msg, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
222     }
223   }
224   else
225   {
226     // Wait for signal from previous process
227     MPI_Recv(&msg, 1, MPI_INT, p - 1, 0, MPI_COMM_WORLD,
          MPI_STATUS_IGNORE);
228
229     // Open file to append
230     fp = fopen(filename, "a");
231
232     for (i = 0; i < I; i++)
233     {
234       if (p % 2 == 0)
235       {
236         fprintf(fp, "%f\n", B[i]);
237       }
```

```
238         else
239         {
240           fprintf(fp, "%f\n", A[i]);
241         }
242       }
243
244       // Send signal to next process
245       if (p != P - 1)
246       {
247         MPI_Send(&msg, 1, MPI_INT, p + 1, 0, MPI_COMM_WORLD);
248       }
249
250       // Close file
251       fclose(fp);
252     }
253
254
255     end_time = MPI_Wtime();
256
257     if (p == 0) {
258       printf ("runtime: %e on %d processors\n",end_time -
             start_time, P);
259     }
260     /* That's it */
261     MPI_Finalize();
262     exit(0);
263 }
```

transposition_sort.c

```
1   $ mpirun -np 4 ./transposition_sort 50
2   $ runtime: 4.482269e-04 on 4 processors
```

0.004012	0.214281	0.348307	0.506873	0.768230
0.066384	0.224983	0.360161	0.553970	0.783099
0.088795	0.230805	0.364784	0.561380	0.798440
0.114666	0.260081	0.393092	0.563554	0.809676
0.121479	0.271337	0.394383	0.587482	0.840188
0.133982	0.277775	0.419803	0.628871	0.864679
0.144781	0.285041	0.421962	0.642966	0.895402
0.191211	0.295718	0.443938	0.654078	0.911647
0.194366	0.309726	0.460581	0.699805	0.916458
0.197551	0.335223	0.477397	0.700976	0.990603

(b) In order to measure the efficiency of the program, we sorted 10^7 elements using different number of processors. We ran the program five times for each No of processors and averaged the times. The progression of runtimes is shown in Figure 3.

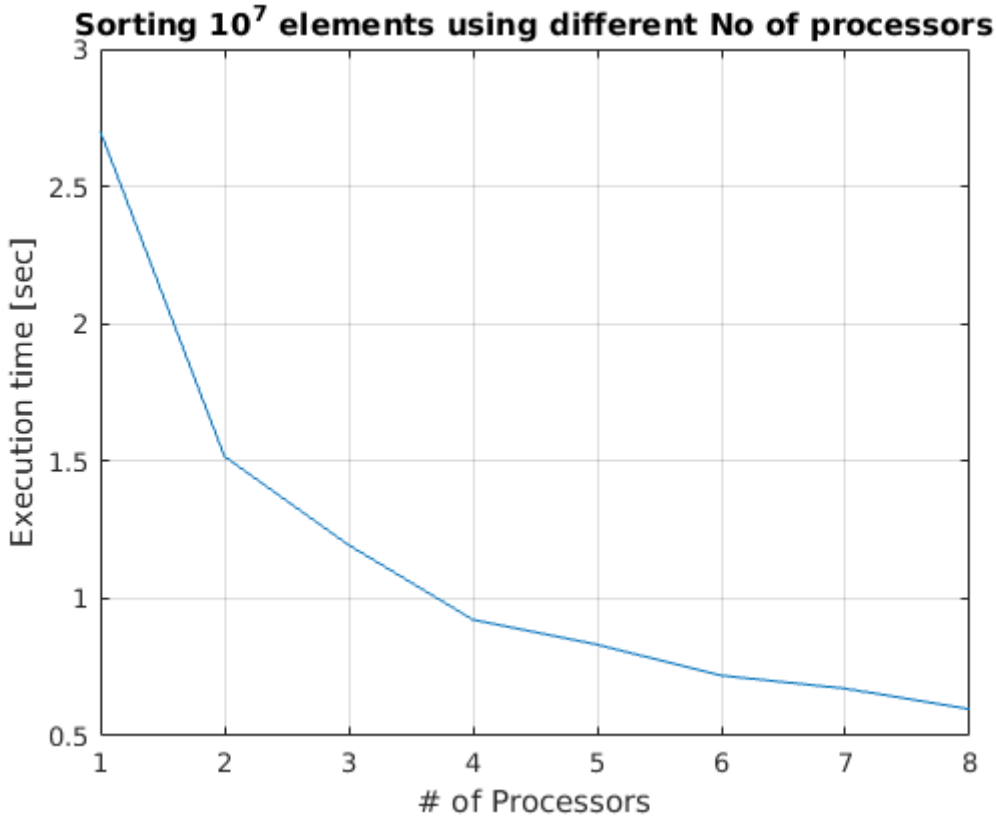

Figure 3: Running time as a function of No. of processors

We can see that in the beginning the time decreases significantly, however when we approach larger N, the curve seems to be stabilizing. This is what has been presented by Amdahl's law. Since the workload is fixed, the speedup will reach a constant value with the addition of more processors. For more processors to improve the efficiency of the program we would need to work with larger datasets.

Finally, in order to see how inter-node communication affects the efficiency of the program, we ran the program on 1 node, using 8 processors and on 2 nodes, using 8 processors divided into 4 per node. Once again we ran the program 5 times for each case. The average times are the following:

1 Node	$0.5974sec$
2 Nodes	$0.6089sec$

Even though the workload and number of processors are constant, we can see that using 2 nodes increases the running time by $0.01sec$. This is not a significant rise, it could even be argued that the times are almost the same. However for more processors and more nodes, this increase could significantly affect the efficiency of the program.