



UPPSALA UNIVERSITET

Parallel and Distributed Programming **Assignment 3**

Authors:

Aikaterini Manousidou

Moa Li Lekander

Uppsala

May 17, 2021

Contents

1	Introduction	1
1.1	Theory	1
1.1.1	The Quicksort Algorithm	1
1.1.2	The Parallel Quicksort Algorithm	1
1.1.3	Strong and Weak Scaling	1
2	Implementation	2
3	Numerical Experiments	4
4	Results	5
5	Discussion	10
6	Appendix	11
6.1	Parallelized Code	11
6.2	Performance Evaluation Code	18

1 Introduction

Fast and efficient algorithms for sorting lists are of great importance in computer science. Quicksort is a widely used sorting algorithm that also can be parallelized in order to increase its efficiency. In this assignment the quicksort algorithm is parallelized using C and MPI, and the performance of the program is analyzed by examining its strong and weak scaling.

1.1 Theory

1.1.1 The Quicksort Algorithm

The quicksort algorithm is based on the divide and conquer approach. The first step is to choose a pivot element in the list and thereafter partition the list so that all the elements smaller than the pivot are positioned to the left of the pivot, and the larger elements to the right of the pivot. Then, this process is repeated recursively for the part of the list to the left of the pivot and for the part to the right of the pivot.

1.1.2 The Parallel Quicksort Algorithm

In parallelized quicksort, the first step is to divide the list into equally sized parts and distribute one part to every process. The local list must then be sorted and thereafter a pivot element is chosen. On every local list, the data is partitioned into two parts, one containing the elements smaller than the pivot and one containing the elements larger than the pivot. The processes are then split into two groups and processes exchange data pairwise so that only one of the groups has the data less than the pivot and the other only has data larger than the pivot. This course of action is then repeated recursively for every group of processes until no more splits of the process groups can be made. Finally, the local lists of every process are then put together to form the final sorted list.

1.1.3 Strong and Weak Scaling

When analyzing the performance of a program, its strong and weak scaling can be taken into account. Strong scaling implies that the size of the problem is kept constant while the number of processes used increases. On the contrary, for weak scaling both the size of the problem and the number of processes is increased in a way so that the workload for the processes is kept constant. To calculate the strong scalability the expression presented in equation (1) can be used, where N is the size of the problem, p is the number of processes, S is the speed-up and T_{serial} and $T_{parallel}$ are the timings for the serial and parallelized codes, respectively.

$$S(N, p) = \frac{T_{serial}(N)}{T_{parallel}(N, p)} \quad (1)$$

Similarly, the weak scaling can be computed with equation (2).

$$S(N, p) = \frac{T_{serial}(N)}{T_{parallel}(N, p)} \times p \quad (2)$$

The parallel efficiency of a program is also something that can be analyzed, which can be thought of as the speed-up per process. The expression for computing the efficiency is presented in equation (3).

$$E(N, p) = \frac{S(N, p)}{p} = \frac{T_{serial}(N)}{p \times T_{parallel}(N, p)} \quad (3)$$

A strongly scalable program has a constant efficiency when the size of the problem stays the same, and a weakly scalable program has a constant efficiency when the problem size increases.

2 Implementation

The implemented program is presented in Appendix section 6.1. The program takes three input arguments, where the first one determines how the elements in the list are generated. This input argument is stored in the variable `seq` (for "sequence") and the list is given uniform random numbers if 0 is entered, an exponential distribution if 1 is entered, a normal distribution if 2 is entered and a list with decreasing values if 3 is entered. The second argument specifies the length of the list which is stored in the variable `len`. The last input argument which is stored in `strategy` determines according to which strategy the pivot element is chosen in the algorithm. If 0 is chosen then the median in one of the processes is used as pivot, if 1 is chosen the median in every process is computed and the median of those values is chosen as pivot. If 2 is entered the mean of all process's medians is chosen as the pivot instead.

The first part of the code initializes the MPI and then the input format and the assumption that the number of processes can be expressed as 2^k , where k is an integer, are checked. The list `data` is filled with values on the process of rank 0, according to the chosen sequence. Then, the number of values that are to be distributed to every process is computed. If the length of the list is not equally divisible with the number of processes, the process of rank 0 will receive the remaining values. Thereafter, the parts of the list are distributed to all the processes with the use of the function `MPI_Scatter`, and the scattered values are stored in a local list for every process. Every local list is then sorted using the serial quicksort function `quicksort`, and thereafter the recursive parallel quicksort function `p_quicksort` is called for the local list on every process.

The parallel quicksort function `p_quicksort` is implemented so that it takes a list and its length together with a communicator, the chosen strategy and the rank of the process in the `MPI_COMM_WORLD` communicator as input. At first, the size of the input communicator and the rank of the process calling the function (according to the input communicator) are stored in the variables `local_size` and `local_rank`. Then, the base case of the recursive function presents itself, which checks if the size of the input communicator is equal to 1, meaning that no more splits of the current communicator can be made. If so, the process will send its local list to rank 0 (according to the `MPI_COMM_WORLD` communicator) using `MPI_Isend` and then return. However if the base case condition is not fulfilled, the pivot element will be computed according to the chosen strategy. Then, the local list will be partitioned, i.e. the number of elements smaller and larger than the pivot are found. Using this information, the pairwise communication described in section 1.1.2 can be executed. This is done by assigning the processes into two groups, one that contains the processes that are to store the elements smaller than the pivot (the left group), and the other the elements larger than the pivot (the right group). From this, the processes in the left group will send their local values that are larger than the pivot to a process in the right group. Vice versa, the processes in the right group will send their values that are smaller than the pivot to its pair process in the left group. In order to be able to know how much memory to allocate for the incoming elements, the functions `MPI_Probe` and `MPI_Get_count` are used to obtain the number of values that are to be received on every process. Using this information, the values are then received in the allocated list `temp_data`. The values that were not sent (that are to be kept in the local list of the process) are transferred to the allocated list `old_data`, which then is merged together with the `temp_data` array using the function `merge`. Right before the `merge` function is called, the old `local_data` array is freed so that the resulting list that is produced by `merge` can be assigned to `local_data`. After this, the input communicator is split into two parts (defined by the groups described above) by the function `MPI_Comm_split`. The temporary arrays `temp_data` and `old_data` are then freed and thereafter a recursive call to the `p_quicksort` function is made, where the updated `local_data` list and the new communicator are passed as input.

Back in the main function, the next step is to receive the data that is sent by the processes from the `p_quicksort` function. Firstly, the number of elements that are to be received from every process is obtained and stored in the allocated array `recv_sizes`, by using the functions `MPI_Probe` and `MPI_Get_count`. Using this information, the elements are then received and put in the original list `data` using `MPI_Irecv`

and `MPI_Wait` calls. Lastly, the result is checked and all allocated memory is freed before a call to the function `MPI_Finalize` is made and the program terminates.

3 Numerical Experiments

A set of experiments was set up on the UPPMAX system Snowy for the purpose of analyzing the strong and weak scaling of the program, for combinations of the different sequences and strategies. The UPPMAX system enabled a total of 40 processes to be used in the experiments. When measuring the time for the parallel quicksort algorithm, time measurements were made on every process, and the longest measured time was considered as the final execution time.

A list of length 200000000 was used when the strong scaling experiments were carried out. As the assumption described in section 2 had to be fulfilled, time measurements could be made for when 1, 2, 4, 8, 16 and 32 processes were used. This was done for all possible combinations of the chosen sequence and strategy, giving a total of 12 strong scaling experiments where the number of processes was varied. The results are presented in section 4 where the outcome of the experiments for the different sequences are presented together in a plot for every strategy.

When the weak scaling experiments were carried out, the length of the local list (the workload) for every process was kept constant with the value 25000000. Hence, the program was run with list lengths of 25000000, 50000000, 100000000, 200000000, 400000000 and 800000000 for the number of processes 1, 2, 4, 8, 16 and 32 respectively. Thus, a constant list-length-per-process was kept through the different measurements. Also, the weak scaling experiments were carried out for all possible combinations of the sequence and strategy options, and the results for the different sequences are presented together in a plot for every strategy, which are displayed in section 4.

4 Results

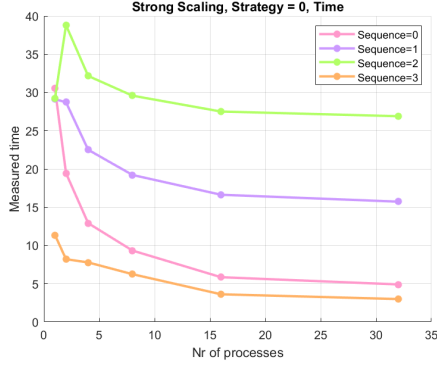
The speed-up of the strong scaling analysis was calculated with the help of equation (1). In Figures 1, 2 and 3 the results of the strong scaling experiments for strategies 0, 1 and 2, respectively, can be seen. The numerical values of the calculated speed-up for each strategy can be found in Table 1.

Table 1: Table over the number of processes, the problem sizes and the timings acquired during the strong scaling analysis and the calculated speed-up for each sequence and strategy. All values have been rounded to four significant digits.

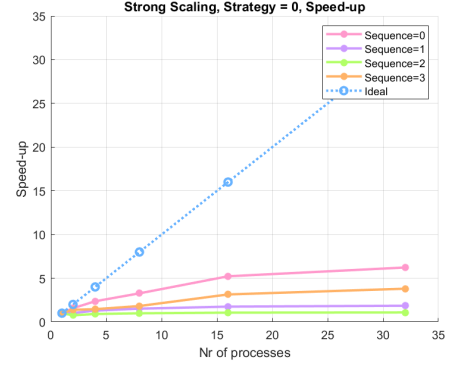
Strong Scaling Speed-up - Strategy = 0					
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	200000000	1.000	1.000	1.000	1.000
2	200000000	1.570	1.014	0.753	1.385
4	200000000	2.364	1.294	0.909	1.460
8	200000000	3.271	1.515	0.988	1.820
16	200000000	5.221	1.751	1.062	3.144
32	200000000	6.227	1.852	1.086	3.802

Strong Scaling Speed-up - Strategy = 1					
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	200000000	1.000	1.000	1.000	1.000
2	200000000	1.572	1.013	0.756	1.379
4	200000000	2.367	1.291	0.900	1.617
8	200000000	3.300	1.512	0.988	2.034
16	200000000	5.199	1.744	1.059	3.435
32	200000000	6.165	1.840	1.094	4.014

Strong Scaling Speed-up - Strategy = 2					
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	200000000	1.000	1.000	1.000	1.000
2	200000000	1.571	1.015	0.761	1.480
4	200000000	2.390	1.295	0.912	1.727
8	200000000	3.318	1.516	0.988	2.230
16	200000000	5.216	1.748	1.065	3.906
32	200000000	6.182	1.850	1.098	4.157

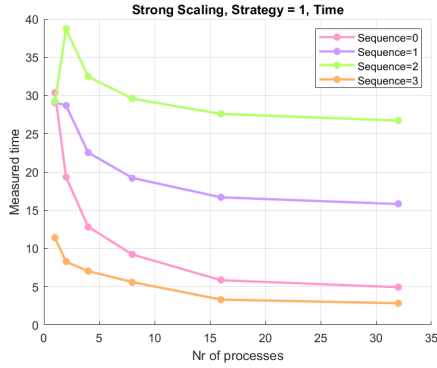


(a) Execution time

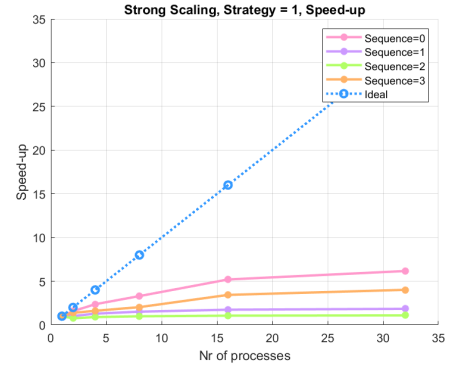


(b) Speed-up

Figure 1: Plots over the strong scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 0.

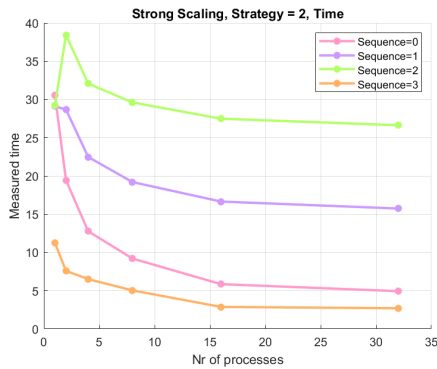


(a) Execution time

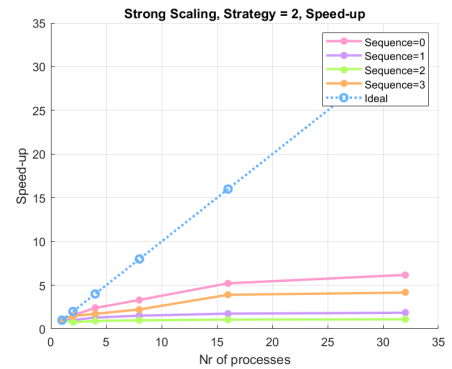


(b) Speed-up

Figure 2: Plots over the strong scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 1.



(a) Execution time



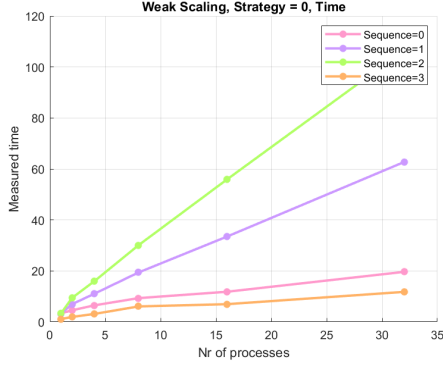
(b) Speed-up

Figure 3: Plots over the strong scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 2.

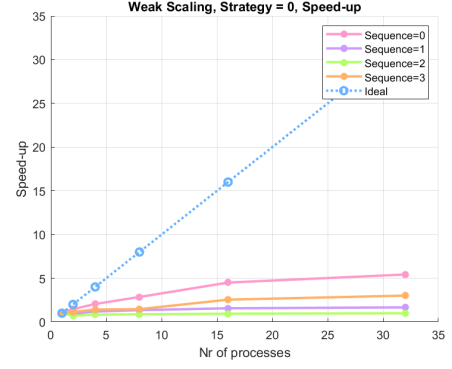
Moreover, the speed-up of the weak scaling analysis was calculated with the help of equation (2). In Figures 4, 5 and 6 the results of the weak scaling experiments for strategies 0, 1 and 2, respectively, can be observed. The numerical values of the calculated speed-up for each strategy are displayed in Table 2.

Table 2: Table over the number of processes, the problem sizes and the timings acquired during the weak scaling analysis and the calculated speed-up for each sequence and strategy. All values have been rounded to four significant digits.

Weak Scaling Speed-up - Strategy = 0					
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	25000000	1.000	1.000	1.000	1.000
2	50000000	1.461	0.938	0.700	1.135
4	100000000	2.051	1.185	0.830	1.420
8	200000000	2.855	1.347	0.878	1.461
16	400000000	4.501	1.560	0.945	2.547
32	800000000	5.426	1.665	0.991	3.011
Weak Scaling Speed-up - Strategy = 1					
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	25000000	1.000	1.000	1.000	1.000
2	50000000	1.475	0.939	0.701	1.138
4	100000000	2.083	1.195	0.832	1.514
8	200000000	2.8812	1.353	0.880	1.615
16	400000000	4.475	1.562	0.948	2.755
32	800000000	5.444	1.662	0.991	3.248
Weak Scaling Speed-up - Strategy = 2					
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	25000000	1.000	1.000	1.000	1.000
2	50000000	1.486	0.916	0.699	1.258
4	100000000	2.111	1.194	0.825	1.729
8	200000000	2.897	1.351	0.888	1.731
16	400000000	4.486	1.560	0.956	3.053
32	800000000	5.427	1.655	0.988	3.303

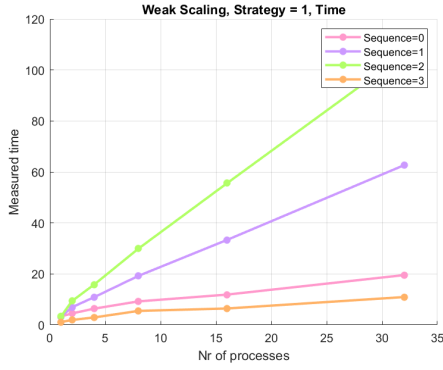


(a) Execution time

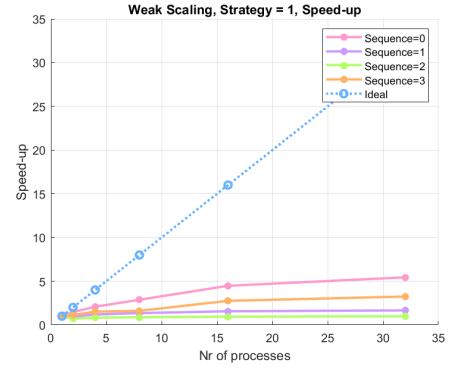


(b) Speed-up

Figure 4: Plots over the weak scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 0.

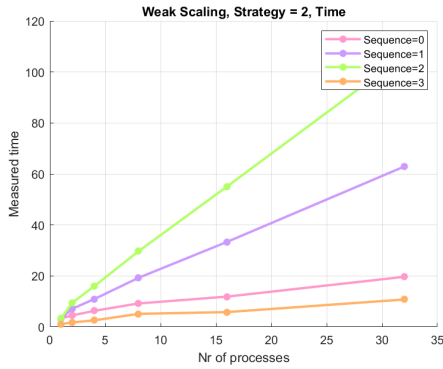


(a) Execution time

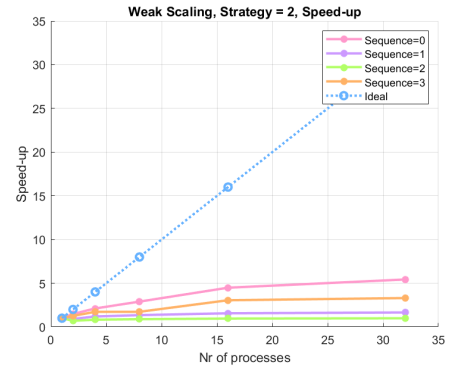


(b) Speed-up

Figure 5: Plots over the weak scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 1.



(a) Execution time



(b) Speed-up

Figure 6: Plots over the weak scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 2.

Lastly, the efficiency of the strong and weak scaling experiments was calculated with equation (3). Figure 7 shows the efficiency of the strong scaling analyses while Figure 8 depicts the efficiency of the weak scaling analyses. The Matlab code responsible for generating all the presented plots can be found in the Appendix, in section 6.2.

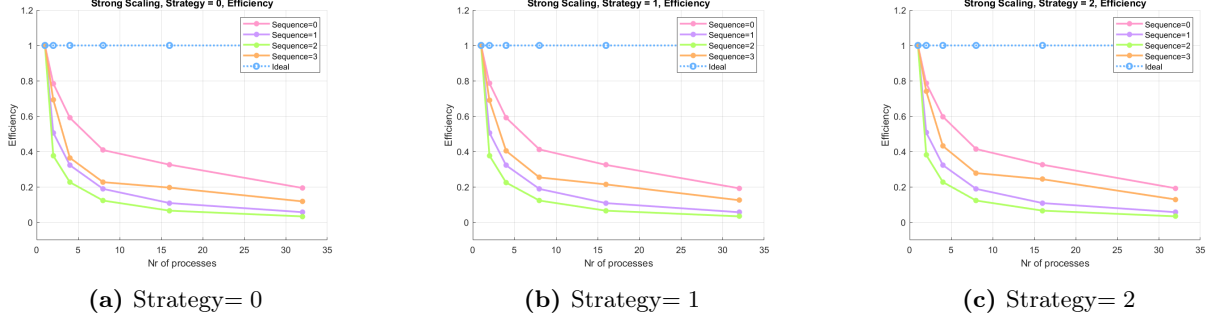


Figure 7: Plots over the efficiency of the strong scaling analyses for Strategies 0, 1 and 2.

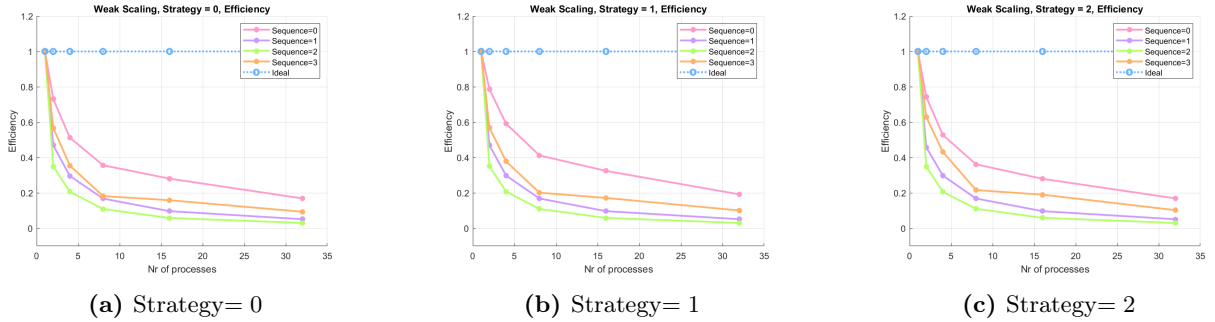


Figure 8: Plots over the efficiency of the weak scaling analyses for Strategies 0, 1 and 2.

5 Discussion

Figures 1a, 2a and 3a show that the execution time of the strong scaling experiments decreases as the number of processes increases, which can be expected. From Figures 1b, 2b and 3b, it is implied that there is a minimal increase in the speed-up with an increasing number of processes. The same plots also suggest that none of the available strategies achieve a speed-up as high as the ideal speed-up. The above conclusions indicate that this quicksort implementation is not strongly scalable.

By studying Figures 4a, 5a and 6a, it is obvious that the execution time increases while the number of processes and the problem size increases, even though the workload is kept constant. This results in the implementation obtaining only a minimal speed-up, which is depicted in Figures 4b, 5b and 6b. As a conclusion, the results of the weak scaling analyses suggest that this quicksort implementation is not weakly scalable.

Figures 7 and 8 illustrate the calculated strong and weak efficiency of the different sequence and strategy combinations. It is clear that this algorithm does not produce an efficiency close to the ideal one. The conclusions that this implementation is neither strongly nor weakly scalable are therefore verified.

A general observation was made when studying the different generated plots. There is a trend in the performance of the algorithm regarding the choice of sequence. Regardless the choice of strategy, the implementation seems to always perform best with sequence 0, which entails filling the initial list with uniformly random numbers. On the other hand the implementation seems to perform the worst when sequence 2 is chosen, which involves filling the initial list with values of a normal distribution. Also, a list with values generated according to an exponential distribution (sequence 1) seems to have a slightly better performance than for a normal distribution (sequence 2). This can be due to the fact that an exponentially generated list will have a larger amount of sorted elements in its original list than a list with a normal distribution, as the latter will have larger values in the middle of the list and smaller on the edges. Thus, less communications might have to be made in the algorithm for the exponential sequence which can be a cause of its better performance. Additionally, sequence 3 has the lowest execution times in all experiments. This can be explained by the fact that this list already is sorted, but in the wrong direction (values go from larger to smaller instead of smaller to larger). Hence, for sequence 3 the list only has to be changed to the right direction, which requires less work in the algorithm.

When observing the result of different strategies for choosing the pivot value, there seems to be no distinct difference in the performance of the program. Hence, it can be concluded that the method of choosing a pivot value in the parallel quicksort algorithm does not have a great importance when it comes to the efficiency of the algorithm.

6 Appendix

6.1 Parallelized Code

```
1  /*****
2   * Parallel Quick sort
3   * Usage: ./a.out sequence length strategy
4   *****/
5  #define PI 3.14159265358979323846
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <math.h>
9  #include <mpi.h>
10
11 int partition(double *data, int left, int right, int pivotIndex);
12 void quicksort(double *data, int left, int right);
13 double *merge(double *v1, int n1, double *v2, int n2);
14 void printList(double* data, int len);
15 void p_quicksort(double *local_data, int len, MPI_Comm comm, int strategy, int world_rank);
16 void writeOutput(char* filename, double* data, int n);
17
18 int main(int argc, char *argv[]) {
19     double start_time, execution_time, max_time, *data, *local_data, pivot, *temp_data;
20     MPI_Status status;
21     MPI_Request request;
22     int size, rank, len, seq, strategy, num_splits, chunk, last_chunk = 0, ...
        uneven_distribution = 0;
23
24     if(argc != 4){
25         printf("ERROR! Expected input: quicksort sequence length strategy\n");
26     }
27     seq=atoi(argv[1]);
28     len=atoi(argv[2]);
29     strategy=atoi(argv[3]);
30
31     MPI_Init(&argc, &argv); // Initialize MPI
32     MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the number of processors
33     MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get my number
34
35     // Check assumptions
36     if((seq < 0) || (seq > 3)){
37         if(rank == 0) printf("ERROR! Invalid sequence selection. Use 0, 1 or 2!\n");
38         exit(0);
39     }
40     if((strategy < 0) || (strategy > 2)){
41         if(rank == 0) printf("ERROR! Invalid strategy selection. Use 0, 1, 2 or 3!\n");
42         exit(0);
43     }
44     if(len <= 0){
45         if(rank == 0) printf("ERROR! Invalid length selection. Please use a positive ...
            integer!\n");
46         exit(0);
47     }
48     num_splits = log(size) / log(2);
49     if(pow(2, num_splits) != size){
50         if(rank == 0) printf("ERROR! Invalid number of processes.\n");
51         exit(0);
52     }
53
54     // Fill list
55     if(rank==0){
56         data = (double *)malloc(len*sizeof(double));
57         if(seq == 0){ // Uniform random numbers
```

```

58         for (int i = 0; i < len; i++)
59             data[i] = drand48();
60     }else if(seq == 1){ // Exponential distribution
61         double lambda = 10;
62         for (int i = 0; i < len; i++)
63             data[i] = - lambda*log(1-drand48());
64     }else if(seq == 2){ // Normal distribution
65         double x, y;
66         for (int i = 0; i < len; i++){
67             x = drand48(); y = drand48();
68             data[i] = sqrt(-2*log(x)) * cos(2*PI*y);
69         }
70     }else if(seq == 3){
71         for(int i = 0; i < len; i++)
72             data[i] = len - i;
73     }
74     // Print initial list
75     /*
76     printf("Initial list\n");
77     printList(data, len);
78     */
79 }
80
81 // Calculate local_data size and allocate memory
82 chunk = len/size;
83 if(len % size != 0){
84     uneven_distribution = 1;
85     if(rank==0){
86         last_chunk = len % size;
87     }
88 }
89 local_data = (double *)malloc((chunk+last_chunk)*sizeof(double));
90
91 // Start timer
92 start_time = MPI_Wtime();
93
94 // Scatter data to local_data arrays
95 MPI_Scatter(&data[0], chunk, MPI_DOUBLE, &local_data[0], chunk, MPI_DOUBLE, 0, ...
96           MPI_COMM_WORLD);
97 if(uneven_distribution == 1){ // add rest to rank 0
98     if(rank == 0){
99         int counter = 0;
100         for(int i = size*chunk; i < size*chunk+last_chunk; i++){
101             local_data[chunk+counter] = data[i];
102             counter++;
103         }
104     }
105 }
106
107 // Sort local lists
108 quicksort(local_data, 0, chunk+last_chunk-1);
109
110 // Call parallel quicksort
111 p_quicksort(local_data, chunk+last_chunk, MPI_COMM_WORLD, strategy, rank);
112
113 // Receive data
114 if(rank == 0){
115     int* recv_sizes = (int*)malloc(size*sizeof(int));
116     int position = 0;
117     for(int i = 0; i < size; i++){
118         MPI_Probe(i, 200+i, MPI_COMM_WORLD, &status); // fetch status thing for ...
119         // communication
120         MPI_Get_count(&status, MPI_DOUBLE, &recv_sizes[i]); // find number of elements ...
121         // to receive

```

```

120         MPI_Irecv(&data[position], recv_sizes[i], MPI_DOUBLE, i, 200+i, ...
121                 MPI_COMM_WORLD, &request);
122         MPI_Wait(&request, &status);
123         position += recv_sizes[i];
124     }
125     free(recv_sizes);
126 }
127 // Print final list
128 /*
129 if(rank == 0) {
130     printf("FINAL LIST \n");
131     printList(data, len);
132 }
133 */
134
135 // Compute time
136 execution_time = MPI_Wtime()-start_time; // stop timer
137 MPI_Reduce(&execution_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
138
139 // Display timing results
140 if(rank==0){
141     printf("%f\n", max_time);
142 }
143
144 // Check results
145 if(rank == 0){
146     int OK = 1;
147     for (int i = 0; i < len-1; i++) {
148         if(data[i] > data[i+1]) {
149             printf("Wrong result: data[%d] = %f, data[%d] = %f\n", i, data[i], i+1, ...
150                     data[i+1]);
151             OK = 0;
152         }
153     }
154     if (OK) printf("Data sorted correctly!\n");
155 }
156
157 // Clean up
158 if (rank == 0){
159     free(data);
160 }
161 MPI_Finalize();
162 return 0;
163 }
164
165 void p_quicksort(double *local_data, int len, MPI_Comm comm, int strategy, int world_rank){
166     int local_size, local_rank;
167     double pivot;
168     MPI_Request request, request_send, request_recv;
169     MPI_Status status;
170
171     MPI_Comm_size(comm, &local_size);
172     MPI_Comm_rank(comm, &local_rank);
173
174     // Basecase
175     if(local_size == 1){
176         MPI_Isend(&local_data[0], len, MPI_DOUBLE, 0, 200+world_rank, MPI_COMM_WORLD, ...
177                 &request);
178         return;
179     }
180
181     // Select pivot element according to chosen strategy
182     if(strategy == 0){ // median in one process
183         if(local_rank == 0){

```

```

182     pivot = local_data[len/2];
183 }
184 MPI_Bcast(&pivot, 1, MPI_DOUBLE, 0, comm);
185 } else if (strategy == 1) { // median of all medians
186     double *medians;
187     int *lengths;
188
189     // Find how many local lists that are NOT empty
190     if (local_rank == 0) {
191         lengths = (int*)malloc(local_size*sizeof(int));
192     }
193     MPI_Gather(&len, 1, MPI_INT, &lengths[0], 1, MPI_INT, 0, comm);
194
195     if (local_rank == 0) {
196         int num_medians = 0;
197         for (int i = 0; i < local_size; i++) {
198             if (lengths[i] > 0) {
199                 num_medians++;
200             }
201         }
202         medians = (double *)malloc(local_size*sizeof(double));
203     }
204
205     // If local list not empty: send median
206     if (len > 0) {
207         MPI_Isend(&local_data[len/2], 1, MPI_DOUBLE, 0, 500+local_rank, comm, &request);
208     }
209
210     // Receive medians
211     if (local_rank == 0) {
212         int counter = 0;
213         for (int i = 0; i < local_size; i++) {
214             if (lengths[i] > 0) {
215                 MPI_Irecv(&medians[counter], 1, MPI_DOUBLE, i, 500+i, comm, &request);
216                 MPI_Wait(&request, &status);
217                 counter++;
218             }
219         }
220
221         // Compute medians of medians
222         quicksort(medians, 0, counter-1);
223         pivot = medians[counter/2];
224         free(medians);
225         free(lengths);
226     }
227     MPI_Bcast(&pivot, 1, MPI_DOUBLE, 0, comm);
228
229 } else if (strategy == 2) { // mean of all medians
230     double median = local_data[len/2];
231     MPI_Reduce(&median, &pivot, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
232     pivot = pivot/local_size;
233     MPI_Bcast(&pivot, 1, MPI_DOUBLE, 0, comm);
234 }
235
236 // Find split in local list
237 int num_smaller_elements = 0;
238 for (int i = 0; i < len; i++) {
239     if (local_data[i] ≤ pivot) {
240         num_smaller_elements++;
241     } else {
242         break;
243     }
244 }
245 int num_larger_elements = len - num_smaller_elements;
246

```



```

247 // EXCHANGE DATA PAIRWISE
248 // Split processes in two groups
249 int rank_friend; // rank to exchange with
250 int group;
251 if(local_rank < local_size/2) {
252     group = 0; // left part of list
253     rank_friend = local_rank + local_size/2;
254 } else {
255     group = 1; // right part of list
256     rank_friend = local_rank - local_size/2;
257 }
258
259 // Send elements
260 if(group == 0){ // left group
261     MPI_Isend(&local_data[num_smaller_elements], num_larger_elements, MPI_DOUBLE, ...
262         rank_friend, 100+rank_friend, comm, &request_send);
263 } else { // right group
264     MPI_Isend(&local_data[0], num_smaller_elements, MPI_DOUBLE, rank_friend, ...
265         100+rank_friend, comm, &request_send);
266 }
267
268 MPI_Probe(rank_friend, 100+local_rank, comm, &status); // fetch status thing for ...
269 // communication
270 int recv_size;
271 MPI_Get_count(&status, MPI_DOUBLE, &recv_size); // find number of elements to receive
272
273 double *temp_data, *old_data;
274 int new_len;
275 if(group == 0){
276     temp_data = (double*)malloc(recv_size*sizeof(double));
277     MPI_Irecv(&temp_data[0], recv_size, MPI_DOUBLE, rank_friend, 100+local_rank, comm, ...
278         &request_recv);
279     MPI_Wait(&request_recv, &status);
280
281     old_data = (double*)malloc(num_smaller_elements*sizeof(double));
282     for(int i = 0; i < num_smaller_elements; i++){
283         old_data[i] = local_data[i];
284     }
285
286     // Merge
287     MPI_Wait(&request_send, &status);
288     free(local_data);
289     local_data = merge(temp_data, recv_size, old_data, num_smaller_elements);
290     new_len = recv_size+num_smaller_elements;
291 } else {
292     temp_data = (double*)malloc(recv_size*sizeof(double));
293     MPI_Irecv(&temp_data[0], recv_size, MPI_DOUBLE, rank_friend, 100+local_rank, comm, ...
294         &request_recv);
295     MPI_Wait(&request_recv, &status);
296
297     old_data = (double*)malloc(num_larger_elements*sizeof(double));
298
299     int counter = 0;
300     for(int i = num_smaller_elements; i < len; i++){
301         old_data[counter] = local_data[i];
302         counter++;
303     }
304
305     // Merge
306     MPI_Wait(&request_send, &status);
307     free(local_data);
308     local_data = merge(temp_data, recv_size, old_data, num_larger_elements);
309     new_len = recv_size+num_larger_elements;
310 }
311
312 // Split comm into two parts

```

```

307     MPI_Comm comm_new;
308     MPI_Comm_split(comm, group, local_rank, &comm_new);
309
310     // Local clean up
311     free(temp_data);
312     free(old_data);
313
314     // Recursive call
315     p_quicksort(local_data, new_len, comm_new, strategy, world_rank);
316 }
317
318 int partition(double *data, int left, int right, int pivotIndex){
319     double pivotValue, temp;
320     int storeIndex, i;
321     pivotValue = data[pivotIndex];
322     temp = data[pivotIndex];
323     data[pivotIndex] = data[right];
324     data[right] = temp;
325     storeIndex = left;
326     for (i = left; i < right; i++)
327         if (data[i] <= pivotValue){
328             temp = data[i];
329             data[i] = data[storeIndex];
330             data[storeIndex] = temp;
331
332             storeIndex = storeIndex + 1;
333         }
334     temp = data[storeIndex];
335     data[storeIndex] = data[right];
336     data[right] = temp;
337     return storeIndex;
338 }
339
340 void quicksort(double *data, int left, int right){
341     int pivotIndex, pivotNewIndex;
342     if (right > left){
343         pivotIndex = left+(right-left)/2;
344         pivotNewIndex = partition(data, left, right, pivotIndex);
345         quicksort(data, left, pivotNewIndex - 1);
346         quicksort(data, pivotNewIndex + 1, right);
347     }
348 }
349
350 double *merge(double *v1, int n1, double *v2, int n2){
351     int i, j, k;
352     double *result;
353
354     result = (double *)malloc((n1+n2)*sizeof(double));
355
356     i=0; j=0; k=0;
357     while(i<n1 && j<n2)
358         if(v1[i]<v2[j])
359             {
360                 result[k] = v1[i];
361                 i++; k++;
362             }
363         else
364             {
365                 result[k] = v2[j];
366                 j++; k++;
367             }
368     if(i==n1)
369         while(j<n2)
370             {
371                 result[k] = v2[j];

```

```
372         j++; k++;
373     }
374     else
375         while(i<n1)
376         {
377             result[k] = v1[i];
378             i++; k++;
379         }
380     return result;
381 }
382
383 void printList(double* data, int len){
384     for(int i = 0; i < len; i++){
385         printf("%10f ", data[i]);
386     }
387     printf("\n");
388 }
```

6.2 Performance Evaluation Code

```
1 close all;
2 clear all;
3
4 %----- PDP ASSIGNMENT 3 -----
5
6 %----- Strong Scaling -----
7
8 cores = [1, 2, 4, 8, 16, 32];
9
10 % ----- STRATEGY = 0 -----
11
12 % Sequence = 0 Strategy = 0
13 time0_0 = zeros(1,length(cores));
14 time0_0(1) = 30.525632; % 1 core
15 time0_0(2) = 19.438958; % 2 cores
16 time0_0(3) = 12.912413; % 4 cores
17 time0_0(4) = 9.332403; % 8 cores
18 time0_0(5) = 5.847092; % 16 cores
19 time0_0(6) = 4.902279; % 32 cores
20
21 % Calculating speed-up, efficiency and ideal time
22 speedup0_0 = zeros(1,length(cores));
23 efficiency0_0 = zeros(1,length(cores));
24 t0_0 = zeros(1,length(cores));
25 for i=1:length(cores)
26     speedup0_0(i) = time0_0(1)/time0_0(i);
27     efficiency0_0(i) = speedup0_0(i)/cores(i);
28     t0_0(i) = time0_0(1)/cores(i);
29 end
30
31 % Sequence = 1 Strategy = 0
32 time1_0 = zeros(1,length(cores));
33 time1_0(1) = 29.117637; % 1 core
34 time1_0(2) = 28.721584; % 2 cores
35 time1_0(3) = 22.506445; % 4 cores
36 time1_0(4) = 19.215338; % 8 cores
37 time1_0(5) = 16.627161; % 16 cores
38 time1_0(6) = 15.724588; % 32 cores
39
40 % Calculating speed-up, efficiency and ideal time
41 speedup1_0 = zeros(1,length(cores));
42 efficiency1_0 = zeros(1,length(cores));
43 t1_0 = zeros(1,length(cores));
44 for i=1:length(cores)
45     speedup1_0(i) = time1_0(1)/time1_0(i);
46     efficiency1_0(i) = speedup1_0(i)/cores(i);
47     t1_0(i) = time1_0(1)/cores(i);
48 end
49
50 % Sequence = 2 Strategy = 0
51 time2_0 = zeros(1,length(cores));
52 time2_0(1) = 29.218643; % 1 core
53 time2_0(2) = 38.792964; % 2 cores
54 time2_0(3) = 32.160024; % 4 cores
55 time2_0(4) = 29.581625; % 8 cores
56 time2_0(5) = 27.506236; % 16 cores
57 time2_0(6) = 26.894346; % 32 cores
58
59 % Calculating speed-up, efficiency and ideal time
60 speedup2_0 = zeros(1,length(cores));
61 efficiency2_0 = zeros(1,length(cores));
```

```

62 t2_0 = zeros(1,length(cores));
63 for i=1:length(cores)
64     speedup2_0(i) = time2_0(1)/time2_0(i);
65     efficiency2_0(i) = speedup2_0(i)/cores(i);
66     t2_0(i) = time2_0(1)/cores(i);
67 end
68
69 % Sequence = 3 Strategy = 0
70 time3_0 = zeros(1,length(cores));
71 time3_0(1) = 11.361397; % 1 core
72 time3_0(2) = 8.201077; % 2 cores
73 time3_0(3) = 7.781006; % 4 cores
74 time3_0(4) = 6.243395; % 8 cores
75 time3_0(5) = 3.614225; % 16 cores
76 time3_0(6) = 2.988533; % 32 cores
77
78 % Calculating speed-up, efficiency and ideal time
79 speedup3_0 = zeros(1,length(cores));
80 efficiency3_0 = zeros(1,length(cores));
81 t3_0 = zeros(1,length(cores));
82 for i=1:length(cores)
83     speedup3_0(i) = time3_0(1)/time3_0(i);
84     efficiency3_0(i) = speedup3_0(i)/cores(i);
85     t3_0(i) = time3_0(1)/cores(i);
86 end
87
88 % Generating figures
89 figure
90 hold on
91 grid on
92 plot(cores, time0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
93 plot(cores, time1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
94 plot(cores, time2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
95 plot(cores, time3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
96 %plot(cores, t0_0, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
97 %plot(cores, t1_0, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
98 %plot(cores, t2_0, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
99 %plot(cores, t3_0, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
100 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
101 title('Strong Scaling, Strategy = 0, Time');
102 xlabel('Nr of processes');
103 ylabel('Measured time');
104 hold off
105
106 figure
107 hold on
108 grid on
109 plot(cores,speedup0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
110 plot(cores,speedup1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
111 plot(cores,speedup2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
112 plot(cores,speedup3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
113 plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
114 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
115 title('Strong Scaling, Strategy = 0, Speed-up');
116 xlabel('Nr of processes');
117 ylabel('Speed-up');
118 hold off
119
120 % ----- STRATEGY = 1 -----
121
122 % Sequence = 0 Strategy = 1
123 time0_1 = zeros(1,length(cores));
124 time0_1(1) = 30.387830; % 1 core
125 time0_1(2) = 19.329221; % 2 cores
126 time0_1(3) = 12.838916; % 4 cores

```

```

127 time0_1(4) = 9.207917; % 8 cores
128 time0_1(5) = 5.844428; % 16 cores
129 time0_1(6) = 4.929357; % 32 cores
130
131 % Calculating speed-up, efficiency and ideal time
132 speedup0_1 = zeros(1,length(cores));
133 efficiency0_1 = zeros(1,length(cores));
134 t0_1 = zeros(1,length(cores));
135 for i=1:length(cores)
136     speedup0_1(i) = time0_1(1)/time0_1(i);
137     efficiency0_1(i) = speedup0_1(i)/cores(i);
138     t0_1(i) = time0_1(1)/cores(i);
139 end
140
141 % Sequence = 1 Strategy = 1
142 time1_1 = zeros(1,length(cores));
143 time1_1(1) = 29.084993; % 1 core
144 time1_1(2) = 28.703074; % 2 cores
145 time1_1(3) = 22.535320; % 4 cores
146 time1_1(4) = 19.215622; % 8 cores
147 time1_1(5) = 16.678529; % 16 cores
148 time1_1(6) = 15.804138; % 32 cores
149
150 % Calculating speed-up, efficiency and ideal time
151 speedup1_1 = zeros(1,length(cores));
152 efficiency1_1 = zeros(1,length(cores));
153 t1_1 = zeros(1,length(cores));
154 for i=1:length(cores)
155     speedup1_1(i) = time1_1(1)/time1_1(i);
156     efficiency1_1(i) = speedup1_1(i)/cores(i);
157     t1_1(i) = time1_1(1)/cores(i);
158 end
159
160 % Sequence = 2 Strategy = 1
161 time2_1 = zeros(1,length(cores));
162 time2_1(1) = 29.225064; % 1 core
163 time2_1(2) = 38.682093; % 2 cores
164 time2_1(3) = 32.456669; % 4 cores
165 time2_1(4) = 29.577839; % 8 cores
166 time2_1(5) = 27.584621; % 16 cores
167 time2_1(6) = 26.715707; % 32 cores
168
169 % Calculating speed-up, efficiency and ideal time
170 speedup2_1 = zeros(1,length(cores));
171 efficiency2_1 = zeros(1,length(cores));
172 t2_1 = zeros(1,length(cores));
173 for i=1:length(cores)
174     speedup2_1(i) = time2_1(1)/time2_1(i);
175     efficiency2_1(i) = speedup2_1(i)/cores(i);
176     t2_1(i) = time2_1(1)/cores(i);
177 end
178
179 % Sequence = 3 Strategy = 1
180 time3_1 = zeros(1,length(cores));
181 time3_1(1) = 11.379709; % 1 core
182 time3_1(2) = 8.254606; % 2 cores
183 time3_1(3) = 7.038642; % 4 cores
184 time3_1(4) = 5.595951; % 8 cores
185 time3_1(5) = 3.312614; % 16 cores
186 time3_1(6) = 2.834814; % 32 cores
187
188 % Calculating speed-up, efficiency and ideal time
189 speedup3_1 = zeros(1,length(cores));
190 efficiency3_1 = zeros(1,length(cores));
191 t3_1 = zeros(1,length(cores));

```

```

192 for i=1:length(cores)
193     speedup3_1(i) = time3_1(1)/time3_1(i);
194     efficiency3_1(i) = speedup3_1(i)/cores(i);
195     t3_1(i) = time3_1(1)/cores(i);
196 end
197
198 % Generating figures
199 figure
200 hold on
201 grid on
202 plot(cores, time0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
203 plot(cores, time1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
204 plot(cores, time2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
205 plot(cores, time3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
206 %plot(cores, t0_1, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
207 %plot(cores, t1_1, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
208 %plot(cores, t2_1, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
209 %plot(cores, t3_1, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
210 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
211 title('Strong Scaling, Strategy = 1, Time');
212 xlabel('Nr of processes');
213 ylabel('Measured time');
214 hold off
215
216 figure
217 hold on
218 grid on
219 plot(cores,speedup0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
220 plot(cores,speedup1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
221 plot(cores,speedup2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
222 plot(cores,speedup3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
223 plot(cores, cores, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
224 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
225 title('Strong Scaling, Strategy = 1, Speed-up');
226 xlabel('Nr of processes');
227 ylabel('Speed-up');
228 hold off
229
230
231
232 % ----- STRATEGY = 2 -----
233
234 % Sequence = 0 Strategy = 2
235 time0_2 = zeros(1,length(cores));
236 time0_2(1) = 30.534189; % 1 core
237 time0_2(2) = 19.441526; % 2 cores
238 time0_2(3) = 12.773972; % 4 cores
239 time0_2(4) = 9.202893; % 8 cores
240 time0_2(5) = 5.854132; % 16 cores
241 time0_2(6) = 4.939196; % 32 cores
242
243 % Calculating speed-up, efficiency and ideal time
244 speedup0_2 = zeros(1,length(cores));
245 efficiency0_2 = zeros(1,length(cores));
246 t0_2 = zeros(1,length(cores));
247 for i=1:length(cores)
248     speedup0_2(i) = time0_2(1)/time0_2(i);
249     efficiency0_2(i) = speedup0_2(i)/cores(i);
250     t0_2(i) = time0_2(1)/cores(i);
251 end
252
253 % Sequence = 1 Strategy = 2
254 time1_2 = zeros(1,length(cores));
255 time1_2(1) = 29.114526; % 1 core
256 time1_2(2) = 28.676153; % 2 cores

```

```

257 time1_2(3) = 22.483128; % 4 cores
258 time1_2(4) = 19.200803; % 8 cores
259 time1_2(5) = 16.652579; % 16 cores
260 time1_2(6) = 15.739615; % 32 cores
261
262 % Calculating speed-up, efficiency and ideal time
263 speedup1_2 = zeros(1,length(cores));
264 efficiency1_2 = zeros(1,length(cores));
265 t1_2 = zeros(1,length(cores));
266 for i=1:length(cores)
267     speedup1_2(i) = time1_2(1)/time1_2(i);
268     efficiency1_2(i) = speedup1_2(i)/cores(i);
269     t1_2(i) = time1_2(1)/cores(i);
270 end
271
272 % Sequence = 2 Strategy = 2
273 time2_2 = zeros(1,length(cores));
274 time2_2(1) = 29.246983; % 1 core
275 time2_2(2) = 38.418233; % 2 cores
276 time2_2(3) = 32.073953; % 4 cores
277 time2_2(4) = 29.614933; % 8 cores
278 time2_2(5) = 27.487457; % 16 cores
279 time2_2(6) = 26.626454; % 32 cores
280
281 % Calculating speed-up, efficiency and ideal time
282 speedup2_2 = zeros(1,length(cores));
283 efficiency2_2 = zeros(1,length(cores));
284 t2_2 = zeros(1,length(cores));
285 for i=1:length(cores)
286     speedup2_2(i) = time2_2(1)/time2_2(i);
287     efficiency2_2(i) = speedup2_2(i)/cores(i);
288     t2_2(i) = time2_2(1)/cores(i);
289 end
290
291 % Sequence = 3 Strategy = 2
292 time3_2 = zeros(1,length(cores));
293 time3_2(1) = 11.238143; % 1 core
294 time3_2(2) = 7.593101; % 2 cores
295 time3_2(3) = 6.507833; % 4 cores
296 time3_2(4) = 5.040112; % 8 cores
297 time3_2(5) = 2.877320; % 16 cores
298 time3_2(6) = 2.703439; % 32 cores
299
300 % Calculating speed-up, efficiency and ideal time
301 speedup3_2 = zeros(1,length(cores));
302 efficiency3_2 = zeros(1,length(cores));
303 t3_2 = zeros(1,length(cores));
304 for i=1:length(cores)
305     speedup3_2(i) = time3_2(1)/time3_2(i);
306     efficiency3_2(i) = speedup3_2(i)/cores(i);
307     t3_2(i) = time3_2(1)/cores(i);
308 end
309
310 % Generating figures
311 figure
312 hold on
313 grid on
314 plot(cores, time0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
315 plot(cores, time1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
316 plot(cores, time2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
317 plot(cores, time3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
318 %plot(cores, t0_2, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
319 %plot(cores, t1_2, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
320 %plot(cores, t2_2, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
321 %plot(cores, t3_2, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);

```



```

322 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
323 title('Strong Scaling, Strategy = 2, Time');
324 xlabel('Nr of processes');
325 ylabel('Measured time');
326 hold off
327
328 figure
329 hold on
330 grid on
331 plot(cores,speedup0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
332 plot(cores,speedup1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
333 plot(cores,speedup2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
334 plot(cores,speedup3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
335 plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
336 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
337 title('Strong Scaling, Strategy = 2, Speed-up');
338 xlabel('Nr of processes');
339 ylabel('Speed-up');
340 hold off
341
342 %----- Weak Scaling -----
343
344 %cores = [1, 2, 4, 8, 16, 32];
345 %sizes = [125,250,500,1000,2000,4000];
346
347
348 % ----- STRATEGY = 0 -----
349
350 % Sequence = 0 Strategy = 0
351 wtime0_0 = zeros(1,length(cores));
352 wtime0_0(1) = 3.330238; % 1 cores
353 wtime0_0(2) = 4.558199; % 2 cores
354 wtime0_0(3) = 6.494247; % 4 cores
355 wtime0_0(4) = 9.330230; % 8 cores
356 wtime0_0(5) = 11.837867; % 16 cores
357 wtime0_0(6) = 19.639124; % 32 cores
358
359 % Calculating speed-up, efficiency and ideal time
360 wspeedup0_0 = zeros(1,length(cores));
361 wefficiency0_0 = zeros(1,length(cores));
362 wt0_0 = wtime0_0(1)*ones(1,length(cores));
363 for i=1:length(cores)
364     wspeedup0_0(i) = wtime0_0(1)/wtime0_0(i)*cores(i);
365     wefficiency0_0(i) = wspeedup0_0(i)/cores(i);
366 end
367
368 % Sequence = 1 Strategy = 0
369 wtime1_0 = zeros(1,length(cores));
370 wtime1_0(1) = 3.264187; % 1 cores
371 wtime1_0(2) = 6.956834; % 2 cores
372 wtime1_0(3) = 11.019184; % 4 cores
373 wtime1_0(4) = 19.380542; % 8 cores
374 wtime1_0(5) = 33.474841; % 16 cores
375 wtime1_0(6) = 62.729792; % 32 cores
376
377 % Calculating speed-up, efficiency and ideal time
378 wspeedup1_0 = zeros(1,length(cores));
379 wefficiency1_0 = zeros(1,length(cores));
380 wt1_0 = wtime1_0(1)*ones(1,length(cores));
381 for i=1:length(cores)
382     wspeedup1_0(i) = wtime1_0(1)/wtime1_0(i)*cores(i);
383     wefficiency1_0(i) = wspeedup1_0(i)/cores(i);
384 end
385
386 % Sequence = 2 Strategy = 0

```

```

387 wtime2_0 = zeros(1,length(cores));
388 wtime2_0(1) = 3.300334; % 1 cores
389 wtime2_0(2) = 9.425067; % 2 cores
390 wtime2_0(3) = 15.910506; % 4 cores
391 wtime2_0(4) = 30.065450; % 8 cores
392 wtime2_0(5) = 55.890569; % 16 cores
393 wtime2_0(6) = 106.613515; % 32 cores
394
395 % Calculating speed-up, efficiency and ideal time
396 wspeedup2_0 = zeros(1,length(cores));
397 wefficiency2_0 = zeros(1,length(cores));
398 wt2_0 = wtime2_0(1)*ones(1,length(cores));
399 for i=1:length(cores)
400     wspeedup2_0(i) = wtime2_0(1)/wtime2_0(i)*cores(i);
401     wefficiency2_0(i) = wspeedup2_0(i)/cores(i);
402 end
403
404 % Sequence = 3 Strategy = 0
405 wtime3_0 = zeros(1,length(cores));
406 wtime3_0(1) = 1.109118; % 1 cores
407 wtime3_0(2) = 1.954304; % 2 cores
408 wtime3_0(3) = 3.123370; % 4 cores
409 wtime3_0(4) = 6.072183; % 8 cores
410 wtime3_0(5) = 6.966555; % 16 cores
411 wtime3_0(6) = 11.786162; % 32 cores
412
413 % Calculating speed-up, efficiency and ideal time
414 wspeedup3_0 = zeros(1,length(cores));
415 wefficiency3_0 = zeros(1,length(cores));
416 wt3_0 = wtime3_0(1)*ones(1,length(cores));
417 for i=1:length(cores)
418     wspeedup3_0(i) = wtime3_0(1)/wtime3_0(i)*cores(i);
419     wefficiency3_0(i) = wspeedup3_0(i)/cores(i);
420 end
421
422 % Generating figures
423 figure
424 hold on
425 grid on
426 plot(cores, wtime0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
427 plot(cores, wtime1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
428 plot(cores, wtime2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
429 plot(cores, wtime3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
430 %plot(cores, wt0_0, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
431 %plot(cores, wt1_0, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
432 %plot(cores, wt2_0, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
433 %plot(cores, wt3_0, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
434 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
435 title('Weak Scaling, Strategy = 0, Time');
436 xlabel('Nr of processes');
437 ylabel('Measured time');
438 hold off
439
440 figure
441 hold on
442 grid on
443 plot(cores,wspeedup0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
444 plot(cores,wspeedup1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
445 plot(cores,wspeedup2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
446 plot(cores,wspeedup3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
447 plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
448 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
449 title('Weak Scaling, Strategy = 0, Speed-up');
450 xlabel('Nr of processes');
451 ylabel('Speed-up');

```

```

452 hold off
453
454 % ----- STRATEGY = 1 -----
455
456 % Sequence = 0 Strategy = 1
457 wtime0_1 = zeros(1,length(cores));
458 wtime0_1(1) = 3.326104; % 1 cores
459 wtime0_1(2) = 4.510797; % 2 cores
460 wtime0_1(3) = 6.385795; % 4 cores
461 wtime0_1(4) = 9.233575; % 8 cores
462 wtime0_1(5) = 11.891216; % 16 cores
463 wtime0_1(6) = 19.550342; % 32 cores
464
465 % Calculating speed-up, efficiency and ideal time
466 wspeedup0_1 = zeros(1,length(cores));
467 wefficiency0_1 = zeros(1,length(cores));
468 wt0_1 = wtime0_1(1)*ones(1,length(cores));
469 for i=1:length(cores)
470     wspeedup0_1(i) = wtime0_1(1)/wtime0_1(i)*cores(i);
471     wefficiency0_1(i) = speedup0_1(i)/cores(i);
472 end
473
474 % Sequence = 1 Strategy = 1
475 wtime1_1 = zeros(1,length(cores));
476 wtime1_1(1) = 3.253699; % 1 cores
477 wtime1_1(2) = 6.933638; % 2 cores
478 wtime1_1(3) = 10.888042; % 4 cores
479 wtime1_1(4) = 19.239228; % 8 cores
480 wtime1_1(5) = 33.337780; % 16 cores
481 wtime1_1(6) = 62.664098; % 32 cores
482
483 % Calculating speed-up, efficiency and ideal time
484 wspeedup1_1 = zeros(1,length(cores));
485 wefficiency1_1 = zeros(1,length(cores));
486 wt1_1 = wtime1_1(1)*ones(1,length(cores));
487 for i=1:length(cores)
488     wspeedup1_1(i) = wtime1_1(1)/wtime1_1(i)*cores(i);
489     wefficiency1_1(i) = wspeedup1_1(i)/cores(i);
490 end
491
492 % Sequence = 2 Strategy = 1
493 wtime2_1 = zeros(1,length(cores));
494 wtime2_1(1) = 3.290981; % 1 cores
495 wtime2_1(2) = 9.385015; % 2 cores
496 wtime2_1(3) = 15.825040; % 4 cores
497 wtime2_1(4) = 29.931524; % 8 cores
498 wtime2_1(5) = 55.575614; % 16 cores
499 wtime2_1(6) = 106.233427; % 32 cores
500
501 % Calculating speed-up, efficiency and ideal time
502 wspeedup2_1 = zeros(1,length(cores));
503 wefficiency2_1 = zeros(1,length(cores));
504 wt2_1 = wtime2_1(1)*ones(1,length(cores));
505 for i=1:length(cores)
506     wspeedup2_1(i) = wtime2_1(1)/wtime2_1(i)*cores(i);
507     wefficiency2_1(i) = wspeedup2_1(i)/cores(i);
508 end
509
510 % Sequence = 3 Strategy = 1
511 wtime3_1 = zeros(1,length(cores));
512 wtime3_1(1) = 1.110164; % 1 cores
513 wtime3_1(2) = 1.951809; % 2 cores
514 wtime3_1(3) = 2.932658; % 4 cores
515 wtime3_1(4) = 5.498014; % 8 cores
516 wtime3_1(5) = 6.448554; % 16 cores

```

```

517 wtime3_1(6) = 10.938075; % 32 cores
518
519 % Calculating speed-up, efficiency and ideal time
520 wspeedup3_1 = zeros(1,length(cores));
521 wefficiency3_1 = zeros(1,length(cores));
522 wt3_1 = wtime3_1(1)*ones(1,length(cores));
523 for i=1:length(cores)
524     wspeedup3_1(i) = wtime3_1(1)/wtime3_1(i)*cores(i);
525     wefficiency3_1(i) = wspeedup3_1(i)/cores(i);
526 end
527
528 % Generating figures
529 figure
530 hold on
531 grid on
532 plot(cores, wtime0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
533 plot(cores, wtime1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
534 plot(cores, wtime2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
535 plot(cores, wtime3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
536 %plot(cores, wt0_1, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
537 %plot(cores, wt1_1, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
538 %plot(cores, wt2_1, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
539 %plot(cores, wt3_1, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
540 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
541 title('Weak Scaling, Strategy = 1, Time');
542 xlabel('Nr of processes');
543 ylabel('Measured time');
544 hold off
545
546 figure
547 hold on
548 grid on
549 plot(cores,wspeedup0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
550 plot(cores,wspeedup1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
551 plot(cores,wspeedup2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
552 plot(cores,wspeedup3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
553 plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
554 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
555 title('Weak Scaling, Strategy = 1, Speed-up');
556 xlabel('Nr of processes');
557 ylabel('Speed-up');
558 hold off
559
560
561
562 % ----- STRATEGY = 2 -----
563
564 % Sequence = 0 Strategy = 2
565 wtime0_2 = zeros(1,length(cores));
566 wtime0_2(1) = 3.333055; % 1 cores
567 wtime0_2(2) = 4.487260; % 2 cores
568 wtime0_2(3) = 6.316947; % 4 cores
569 wtime0_2(4) = 9.205733; % 8 cores
570 wtime0_2(5) = 11.887559; % 16 cores
571 wtime0_2(6) = 19.653326; % 32 cores
572
573 % Calculating speed-up, efficiency and ideal time
574 wspeedup0_2 = zeros(1,length(cores));
575 wefficiency0_2 = zeros(1,length(cores));
576 wt0_2 = wtime0_2(1)*ones(1,length(cores));
577 for i=1:length(cores)
578     wspeedup0_2(i) = wtime0_2(1)/wtime0_2(i)*cores(i);
579     wefficiency0_2(i) = wspeedup0_2(i)/cores(i);
580 end
581

```

```

582 % Sequence = 1 Strategy = 2
583 wtime1_2 = zeros(1,length(cores));
584 wtime1_2(1) = 3.252186; % 1 cores
585 wtime1_2(2) = 7.097667; % 2 cores
586 wtime1_2(3) = 10.894351; % 4 cores
587 wtime1_2(4) = 19.252760; % 8 cores
588 wtime1_2(5) = 33.348899; % 16 cores
589 wtime1_2(6) = 62.891490; % 32 cores
590
591 % Calculating speed-up, efficiency and ideal time
592 wspeedup1_2 = zeros(1,length(cores));
593 wefficiency1_2 = zeros(1,length(cores));
594 wt1_2 = wtime1_2(1)*ones(1,length(cores));
595 for i=1:length(cores)
596     wspeedup1_2(i) = wtime1_2(1)/wtime1_2(i)*cores(i);
597     wefficiency1_2(i) = wspeedup1_2(i)/cores(i);
598 end
599
600 % Sequence = 2 Strategy = 2
601 wtime2_2 = zeros(1,length(cores));
602 wtime2_2(1) = 3.290253; % 1 cores
603 wtime2_2(2) = 9.416190; % 2 cores
604 wtime2_2(3) = 15.958708; % 4 cores
605 wtime2_2(4) = 29.653036; % 8 cores
606 wtime2_2(5) = 55.075647; % 16 cores
607 wtime2_2(6) = 106.591060; % 32 cores
608
609 % Calculating speed-up, efficiency and ideal time
610 wspeedup2_2 = zeros(1,length(cores));
611 wefficiency2_2 = zeros(1,length(cores));
612 wt2_2 = wtime2_2(1)*ones(1,length(cores));
613 for i=1:length(cores)
614     wspeedup2_2(i) = wtime2_2(1)/wtime2_2(i)*cores(i);
615     wefficiency2_2(i) = wspeedup2_2(i)/cores(i);
616 end
617
618 % Sequence = 3 Strategy = 2
619 wtime3_2 = zeros(1,length(cores));
620 wtime3_2(1) = 1.109987; % 1 cores
621 wtime3_2(2) = 1.765386; % 2 cores
622 wtime3_2(3) = 2.568317; % 4 cores
623 wtime3_2(4) = 5.130037; % 8 cores
624 wtime3_2(5) = 5.817957; % 16 cores
625 wtime3_2(6) = 10.754171; % 32 cores
626
627 % Calculating speed-up, efficiency and ideal time
628 wspeedup3_2 = zeros(1,length(cores));
629 wefficiency3_2 = zeros(1,length(cores));
630 wt3_2 = wtime3_2(1)*ones(1,length(cores));
631 for i=1:length(cores)
632     wspeedup3_2(i) = wtime3_2(1)/wtime3_2(i)*cores(i);
633     wefficiency3_2(i) = wspeedup3_2(i)/cores(i);
634 end
635
636 % Generating figures
637 figure
638 hold on
639 grid on
640 plot(cores, wtime0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
641 plot(cores, wtime1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
642 plot(cores, wtime2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
643 plot(cores, wtime3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
644 %plot(cores, wt0_2, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
645 %plot(cores, wt1_2, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
646 %plot(cores, wt2_2, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);

```

```

647 %plot(cores, wt3_2, 'o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
648 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
649 title('Weak Scaling, Strategy = 2, Time');
650 xlabel('Nr of processes');
651 ylabel('Measured time');
652 hold off
653
654 figure
655 hold on
656 grid on
657 plot(cores,wspeedup0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
658 plot(cores,wspeedup1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
659 plot(cores,wspeedup2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
660 plot(cores,wspeedup3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
661 plot(cores, cores, 'o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
662 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
663 title('Weak Scaling, Strategy = 2, Speed-up');
664 xlabel('Nr of processes');
665 ylabel('Speed-up');
666 hold off
667
668 %----- Efficiency -----
669 e = ones(1,length(cores));
670
671 figure
672 hold on
673 grid on
674 plot(cores, efficiency0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
675 plot(cores, efficiency1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
676 plot(cores, efficiency2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
677 plot(cores, efficiency3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
678 plot(cores, e, 'o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
679 ylim([-0.1 1.2])
680 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
681 title('Strong Scaling, Strategy = 0, Efficiency');
682 xlabel('Nr of processes');
683 ylabel('Efficiency');
684 hold off
685
686 figure
687 hold on
688 grid on
689 plot(cores, efficiency0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
690 plot(cores, efficiency1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
691 plot(cores, efficiency2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
692 plot(cores, efficiency3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
693 plot(cores, e, 'o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
694 ylim([-0.1 1.2])
695 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
696 title('Strong Scaling, Strategy = 1, Efficiency');
697 xlabel('Nr of processes');
698 ylabel('Efficiency');
699 hold off
700
701 figure
702 hold on
703 grid on
704 plot(cores, efficiency0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
705 plot(cores, efficiency1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
706 plot(cores, efficiency2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
707 plot(cores, efficiency3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
708 plot(cores, e, 'o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
709 ylim([-0.1 1.2])
710 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
711 title('Strong Scaling, Strategy = 2, Efficiency');

```

```

712 xlabel('Nr of processes');
713 ylabel('Efficiency');
714 hold off
715
716 figure
717 hold on
718 grid on
719 plot(cores, wefficiency0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
720 plot(cores, wefficiency1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
721 plot(cores, wefficiency2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
722 plot(cores, wefficiency3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
723 plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
724 ylim([-0.1 1.2])
725 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
726 title('Weak Scaling, Strategy = 0, Efficiency');
727 xlabel('Nr of processes');
728 ylabel('Efficiency');
729 hold off
730
731 figure
732 hold on
733 grid on
734 plot(cores, wefficiency0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
735 plot(cores, wefficiency1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
736 plot(cores, wefficiency2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
737 plot(cores, wefficiency3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
738 plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
739 ylim([-0.1 1.2])
740 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
741 title('Weak Scaling, Strategy = 1, Efficiency');
742 xlabel('Nr of processes');
743 ylabel('Efficiency');
744 hold off
745
746 figure
747 hold on
748 grid on
749 plot(cores, wefficiency0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
750 plot(cores, wefficiency1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
751 plot(cores, wefficiency2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
752 plot(cores, wefficiency3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
753 plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
754 ylim([-0.1 1.2])
755 legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
756 title('Weak Scaling, Strategy = 2, Efficiency');
757 xlabel('Nr of processes');
758 ylabel('Efficiency');
759 hold off

```