UPPSALA UNIVERSITET

Parallel and Distributed Programming
Assignment 3

Authors:

Aikaterini Manousidou

Moa Li Lekander

Uppsala

May 17, 2021

Contents

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.

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

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

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

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).

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

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

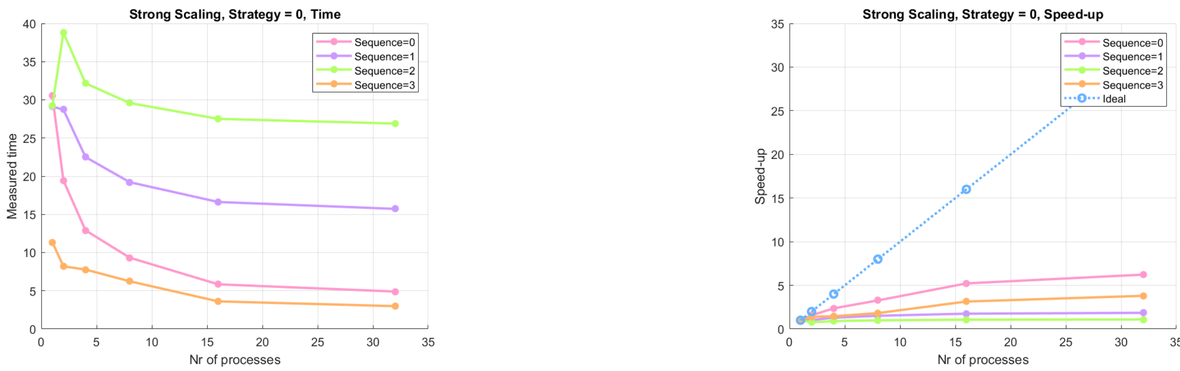
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	200000000	1.000	1.000	1.000	1.000
2	200000000	1.570	1.014	0.753	1.385
4	200000000	2.364	1.294	0.909	1.460
8	200000000	3.271	1.515	0.988	1.820
16	200000000	5.221	1.751	1.062	3.144
32	200000000	6.227	1.852	1.086	3.802

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

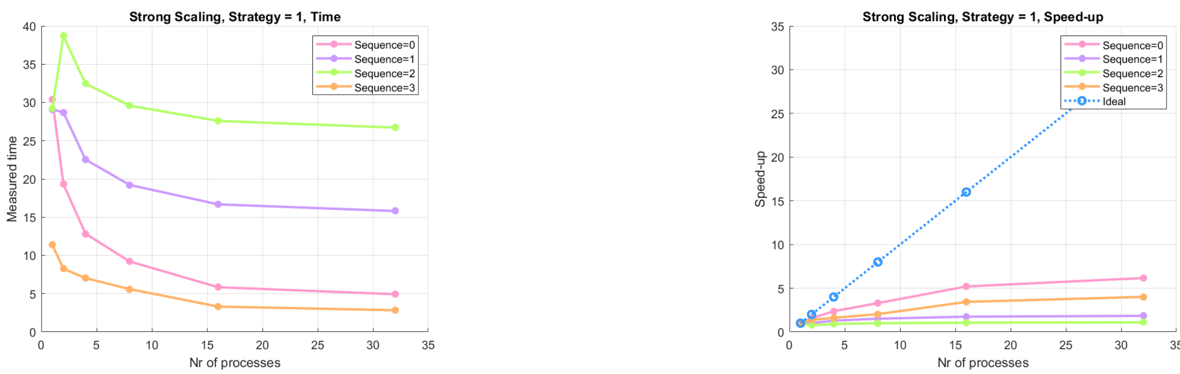
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	200000000	1.000	1.000	1.000	1.000
2	200000000	1.571	1.015	0.761	1.480
4	200000000	2.390	1.295	0.912	1.727
8	200000000	3.318	1.516	0.988	2.230
16	200000000	5.216	1.748	1.065	3.906
32	200000000	6.182	1.850	1.098	4.157



(a) Execution time

(b) Speed-up

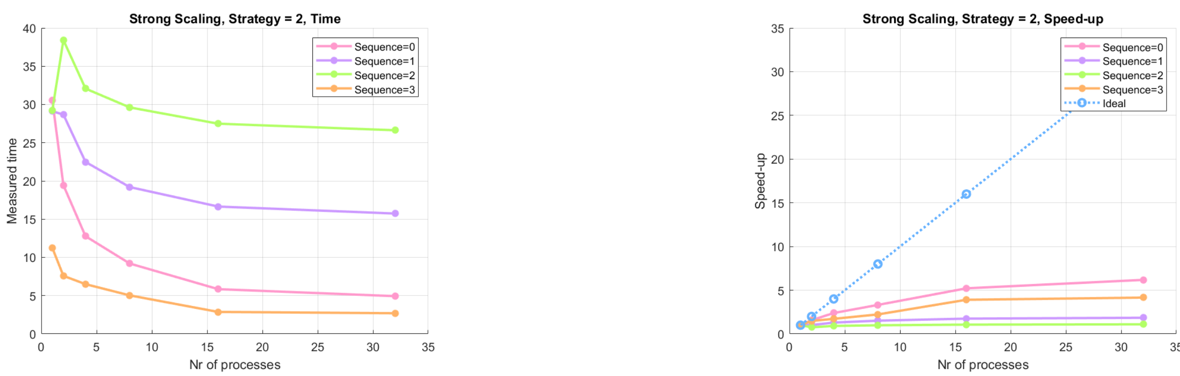
Figure 1: Plots over the strong scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 0.



(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

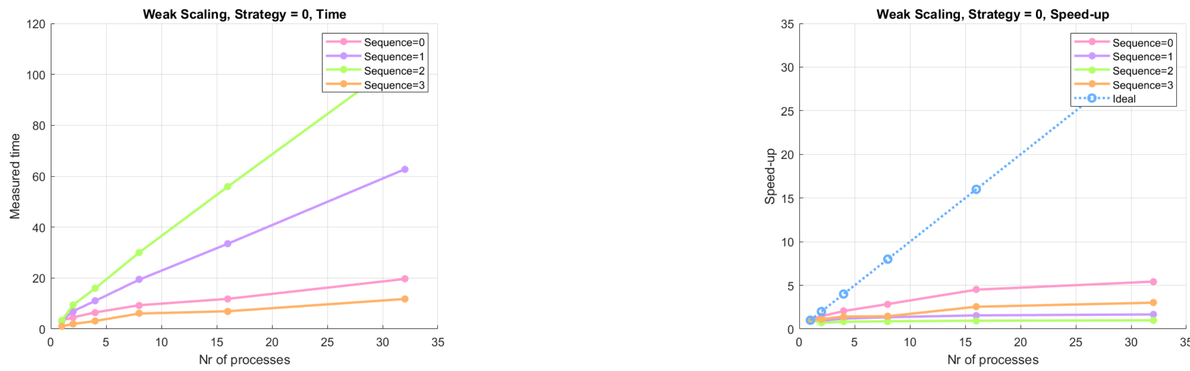
Processes	Problem size	Sequence = 0	Sequence = 1	Sequence = 2	Sequence = 3
1	25000000	1.000	1.000	1.000	1.000
2	50000000	1.461	0.938	0.700	1.135
4	100000000	2.051	1.185	0.830	1.420
8	200000000	2.855	1.347	0.878	1.461
16	400000000	4.501	1.560	0.945	2.547
32	800000000	5.426	1.665	0.991	3.011

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

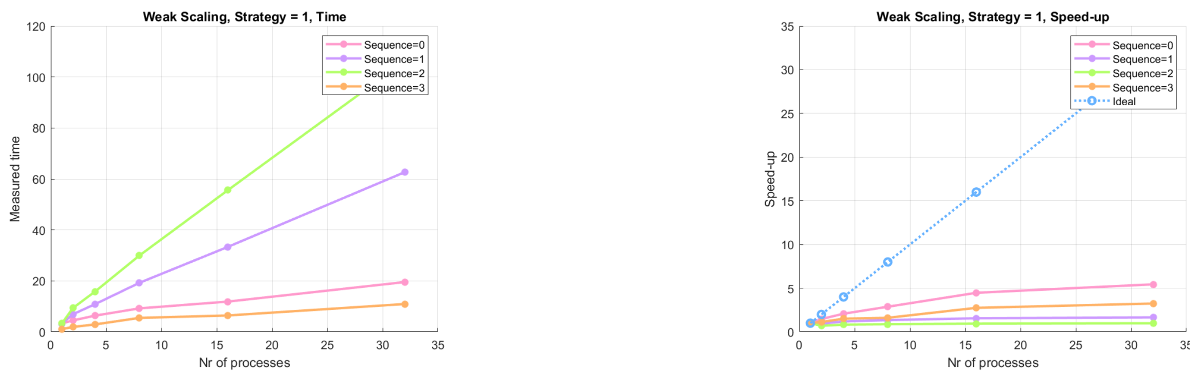
Weak Scaling Speed-up - Strategy = 2

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



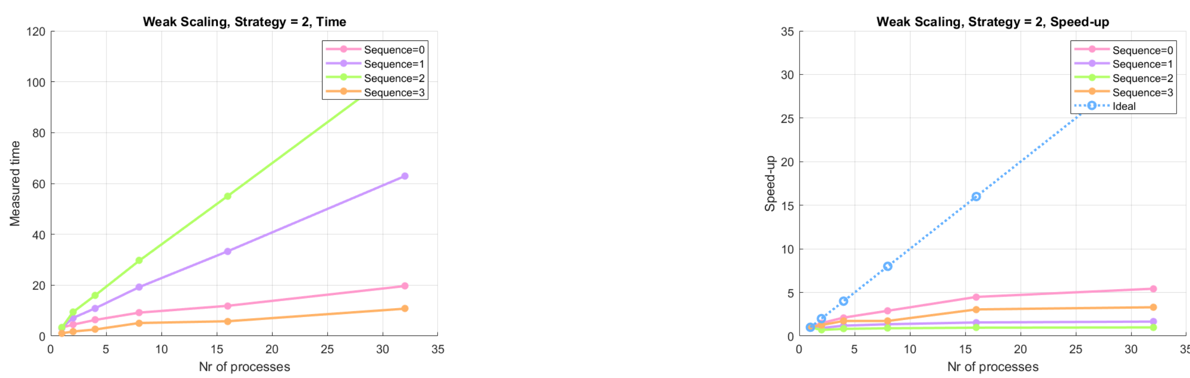
(a) Execution time **(b)** Speed-up

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

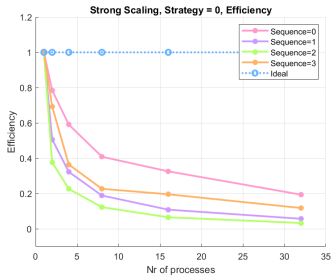
Figure 5: Plots over the weak scaling analysis, including the measured execution time and the calculated speed-up, for Strategy 1.



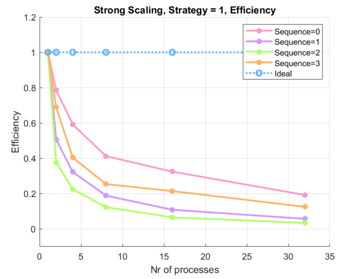
(a) Execution time **(b)** Speed-up

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

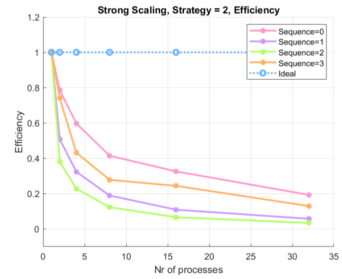
Lastly, the efficiency of the strong and weak scaling experiments was calculated with equation (3). Figure 7 shows the efficiency of the strong scaling analyses while Figure 8 depicts the efficiency of the weak scaling analyses. The Matlab code responsible for generating all the presented plots can be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



(a) Strategy= 0

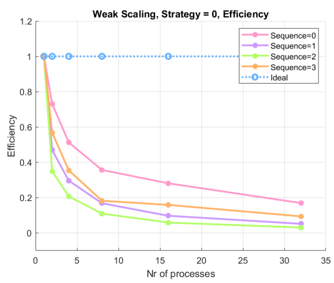


(b) Strategy= 1

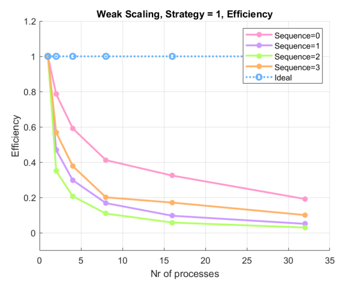


(c) Strategy= 2

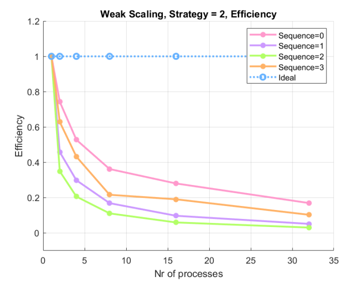
Figure 7: Plots over the efficiency of the strong scaling analyses for Strategies 0, 1 and 2.



(a) Strategy= 0



(b) Strategy= 1



(c) Strategy= 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 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

```c
/*********************************************************************
 * Parallel Quick sort
 * Usage: ./a.out sequence length strategy
 *********************************************************************/
#define PI 3.14159265358979323846
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

int partition(double *data, int left, int right, int pivotIndex);
void quicksort(double *data, int left, int right);
double *merge(double *v1, int n1, double *v2, int n2);
void printList(double* data, int len);
void p_quicksort(double *local_data, int len, MPI_Comm comm, int strategy, int world_rank);
void writeOutput(char* filename, double* data, int n);

int main(int argc, char *argv[]) {
    double start_time, execution_time, max_time, *data, *local_data, pivot, *temp_data;
    MPI_Status status;
    MPI_Request request;
    int size, rank, len, seq, strategy, num_splits, chunk, last_chunk = 0, ...
        uneven_distribution = 0;

    if(argc != 4){
        printf("ERROR! Expected input: quicksort sequence length strategy\n");
    }
    seq=atoi(argv[1]);
    len=atoi(argv[2]);
    strategy=atoi(argv[3]);

    MPI_Init(&argc, &argv);               // Initialize MPI
    MPI_Comm_size(MPI_COMM_WORLD, &size); // Get the number of processors
    MPI_Comm_rank(MPI_COMM_WORLD, &rank); // Get my number

    // Check assumptions
    if((seq < 0) || (seq > 3)){
        if(rank == 0) printf("ERROR! Invalid sequence selection. Use 0, 1 or 2!\n");
        exit(0);
    }
    if((strategy < 0) || (strategy > 2)){
        if(rank == 0) printf("ERROR! Invalid strategy selection. Use 0, 1, 2 or 3!\n");
        exit(0);
    }
    if(len <= 0){
        if(rank == 0) printf("ERROR! Invalid length selection. Please use a positive ...
            integer!\n");
        exit(0);
    }
    num_splits = log(size) / log(2);
    if(pow(2, num_splits) != size){
        if(rank == 0) printf("ERROR! Invalid number of processes.\n");
        exit(0);
    }

    // Fill list
    if(rank==0){
        data = (double *)malloc(len*sizeof(double));
        if(seq == 0){ // Uniform random numbers
```

```
            for (int i = 0; i < len; i++)
                data[i] = drand48();
        }else if(seq == 1){ // Exponential distribution
            double lambda = 10;
            for (int i = 0; i < len; i++)
                data[i] =- lambda*log(1-drand48());
        }else if(seq == 2){ // Normal distribution
            double x, y;
            for (int i = 0;i < len; i++){
                x = drand48(); y = drand48();
                data[i] = sqrt(-2*log(x)) * cos(2*PI*y);
            }
        }else if(seq == 3){
            for(int i = 0; i < len; i++)
                data[i] = len - i;
        }
        // Print initial list
        /*
        printf("Initial list\n");
        printList(data, len);
        */
    }

    // Calculate local_data size and allocate memory
    chunk = len/size;
    if(len % size != 0){
        uneven_distribution = 1;
        if(rank==0){
            last_chunk = len % size;
        }
    }
    local_data = (double *)malloc((chunk+last_chunk)*sizeof(double));

    // Start timer
    start_time = MPI_Wtime();

    // Scatter data to local_data arrays
    MPI_Scatter(&data[0], chunk, MPI_DOUBLE, &local_data[0], chunk, MPI_DOUBLE, 0, ...
        MPI_COMM_WORLD);
    if(uneven_distribution == 1){ // add rest to rank 0
        if(rank == 0){
            int counter = 0;
            for(int i = size*chunk; i < size*chunk+last_chunk; i++){
                local_data[chunk+counter] = data[i];
                counter++;
            }
        }
    }

    // Sort local lists
    quicksort(local_data, 0, chunk+last_chunk-1);

    // Call parallel quicksort
    p_quicksort(local_data, chunk+last_chunk, MPI_COMM_WORLD, strategy, rank);

    // Receive data
    if(rank == 0){
        int* recv_sizes = (int*)malloc(size*sizeof(int));
        int position = 0;
        for(int i = 0; i < size; i++){
            MPI_Probe(i, 200+i, MPI_COMM_WORLD, &status); // fetch status thing for ...
                communication
            MPI_Get_count(&status, MPI_DOUBLE, &recv_sizes[i]); // find number of elements ...
                to receive

```

```
            MPI_Irecv(&data[position], recv_sizes[i], MPI_DOUBLE, i, 200+i, ...
                MPI_COMM_WORLD, &request);
            MPI_Wait(&request, &status);

            position += recv_sizes[i];
        }
        free(recv_sizes);
    }
    // Print final list
    /*
    if(rank == 0) {
        printf("FINAL LIST \n");
        printList(data, len);
    }
    */

    // Compute time
    execution_time = MPI_Wtime()-start_time; // stop timer
    MPI_Reduce(&execution_time, &max_time, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);

    // Display timing results
    if(rank==0){
        printf("%f\n", max_time);
    }

    // Check results
    if(rank == 0){
        int OK = 1;
        for (int i = 0; i < len-1; i++) {
            if(data[i] > data[i+1]) {
                printf("Wrong result: data[%d] = %f, data[%d] = %f\n", i, data[i], i+1, ...
                    data[i+1]);
                OK = 0;
            }
        }
        if (OK) printf("Data sorted correctly!\n");
    }

    // Clean up
    if (rank == 0){
        free(data);
    }
    MPI_Finalize();
    return 0;
}

void p_quicksort(double *local_data, int len, MPI_Comm comm, int strategy, int world_rank){
    int local_size, local_rank;
    double pivot;
    MPI_Request request, request_send, request_recv;
    MPI_Status status;

    MPI_Comm_size(comm, &local_size);
    MPI_Comm_rank(comm, &local_rank);

    // Basecase
    if(local_size == 1){
        MPI_Isend(&local_data[0], len, MPI_DOUBLE, 0, 200+world_rank, MPI_COMM_WORLD, ...
            &request);
        return;
    }

    // Select pivot element according to chosen strategy
    if(strategy == 0){ // median in one process
        if(local_rank == 0){
```

```
            pivot = local_data[len/2];
        }
        MPI_Bcast(&pivot, 1, MPI_DOUBLE, 0, comm);
    } else if(strategy == 1){ // median of all medians
        double *medians;
        int *lengths;

        // Find how many local lists that are NOT empty
        if(local_rank == 0){
            lengths = (int*)malloc(local_size*sizeof(int));
        }
        MPI_Gather(&len, 1, MPI_INT, &lengths[0], 1, MPI_INT, 0, comm);

        if(local_rank == 0){
            int num_medians = 0;
            for(int i = 0; i < local_size; i++){
                if(lengths[i]>0){
                    num_medians++;
                }
            }
            medians = (double *)malloc(local_size*sizeof(double));
        }

        // If local list not empty: send median
        if(len > 0){
            MPI_Isend(&local_data[len/2], 1, MPI_DOUBLE, 0, 500+local_rank, comm, &request);
        }

        // Receive medians
        if(local_rank == 0){
            int counter = 0;
            for(int i = 0; i < local_size; i++){
                if(lengths[i] > 0){
                    MPI_Irecv(&medians[counter], 1, MPI_DOUBLE, i, 500+i, comm, &request);
                    MPI_Wait(&request, &status);
                    counter++;
                }
            }

            // Compute medians of medians
            quicksort(medians, 0, counter-1);
            pivot = medians[counter/2];
            free(medians);
            free(lengths);
        }
        MPI_Bcast(&pivot, 1, MPI_DOUBLE, 0, comm);

    } else if (strategy == 2){ // mean of all medians
        double median = local_data[len/2];
        MPI_Reduce(&median, &pivot, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
        pivot = pivot/local_size;
        MPI_Bcast(&pivot, 1, MPI_DOUBLE, 0, comm);
    }

    // Find split in local list
    int num_smaller_elements = 0;
    for(int i = 0; i < len; i++){
        if(local_data[i] ≤ pivot){
            num_smaller_elements++;
        } else {
            break;
        }
    }
    int num_larger_elements = len - num_smaller_elements;

```

```c
    // EXCHANGE DATA PAIRWISE
    // Split processes in two groups
    int rank_friend; // rank to exchange with
    int group;
    if(local_rank < local_size/2) {
        group = 0; // left part of list
        rank_friend = local_rank + local_size/2;
    } else {
        group = 1; // right part of list
        rank_friend = local_rank - local_size/2;
    }

    // Send elements
    if(group == 0){ // left group
        MPI_Isend(&local_data[num_smaller_elements], num_larger_elements, MPI_DOUBLE, ...
            rank_friend, 100+rank_friend, comm, &request_send);
    } else { // right group
        MPI_Isend(&local_data[0], num_smaller_elements, MPI_DOUBLE, rank_friend, ...
            100+rank_friend, comm, &request_send);
    }

    MPI_Probe(rank_friend, 100+local_rank, comm, &status); // fetch status thing for ...
        communication
    int recv_size;
    MPI_Get_count(&status, MPI_DOUBLE, &recv_size); // find number of elements to receive

    double *temp_data, *old_data;
    int new_len;
    if(group == 0){
        temp_data = (double*)malloc(recv_size*sizeof(double));
        MPI_Irecv(&temp_data[0], recv_size, MPI_DOUBLE, rank_friend, 100+local_rank, comm, ...
            &request_recv);
        MPI_Wait(&request_recv, &status);

        old_data = (double*)malloc(num_smaller_elements*sizeof(double));
        for(int i = 0; i < num_smaller_elements; i++){
            old_data[i] = local_data[i];
        }

        // Merge
        MPI_Wait(&request_send, &status);
        free(local_data);
        local_data = merge(temp_data, recv_size, old_data, num_smaller_elements);
        new_len = recv_size+num_smaller_elements;
    } else {
        temp_data = (double*)malloc(recv_size*sizeof(double));
        MPI_Irecv(&temp_data[0], recv_size, MPI_DOUBLE, rank_friend, 100+local_rank, comm, ...
            &request_recv);
        MPI_Wait(&request_recv, &status);

        old_data = (double*)malloc(num_larger_elements*sizeof(double));

        int counter = 0;
        for(int i = num_smaller_elements; i < len; i++){
            old_data[counter] = local_data[i];
            counter++;
        }
        // Merge
        MPI_Wait(&request_send, &status);
        free(local_data);
        local_data = merge(temp_data, recv_size, old_data, num_larger_elements);
        new_len = recv_size+num_larger_elements;
    }

    // Split comm into two parts
```

```
    MPI_Comm comm_new;
    MPI_Comm_split(comm, group, local_rank, &comm_new);

    // Local clean up
    free(temp_data);
    free(old_data);

    // Recursive call
    p_quicksort(local_data, new_len, comm_new, strategy, world_rank);
}

int partition(double *data, int left, int right, int pivotIndex){
    double pivotValue, temp;
    int storeIndex, i;
    pivotValue = data[pivotIndex];
    temp = data[pivotIndex];
    data[pivotIndex] = data[right];
    data[right] = temp;
    storeIndex = left;
    for (i = left; i < right; i++)
    if (data[i] ≤ pivotValue){
        temp = data[i];
        data[i] = data[storeIndex];
        data[storeIndex] = temp;

        storeIndex = storeIndex + 1;
    }
    temp = data[storeIndex];
    data[storeIndex] = data[right];
    data[right] = temp;
    return storeIndex;
}

void quicksort(double *data, int left, int right){
    int pivotIndex, pivotNewIndex;
    if (right > left){
        pivotIndex = left+(right-left)/2;
        pivotNewIndex = partition(data, left, right, pivotIndex);
        quicksort(data, left, pivotNewIndex - 1);
        quicksort(data, pivotNewIndex + 1, right);
    }
}

double *merge(double *v1, int n1, double *v2, int n2){
    int i,j,k;
    double *result;

    result = (double *)malloc((n1+n2)*sizeof(double));

    i=0; j=0; k=0;
    while(i<n1 && j<n2)
        if(v1[i]<v2[j])
        {
            result[k] = v1[i];
            i++; k++;
        }
        else
        {
            result[k] = v2[j];
            j++; k++;
        }
    if(i==n1)
        while(j<n2)
        {
            result[k] = v2[j];
```

```c
            j++; k++;
        }
    else
        while(i<n1)
        {
            result[k] = v1[i];
            i++; k++;
        }
    return result;
}

void printList(double* data, int len){
    for(int i = 0; i < len; i++){
        printf("%10f ", data[i]);
    }
    printf("\n");
}
```

6.2 Performance Evaluation Code

```
close all;
clear all;

%------------- PDP ASSIGNMENT 3 -------------

%------------- Strong Scaling -------------

cores = [1, 2, 4, 8, 16, 32];

% ------------ STRATEGY = 0 ------------

% Sequence = 0 Strategy = 0
time0_0 = zeros(1,length(cores));
time0_0(1) = 30.525632; % 1 core
time0_0(2) = 19.438958; % 2 cores
time0_0(3) = 12.912413; % 4 cores
time0_0(4) = 9.332403; % 8 cores
time0_0(5) = 5.847092; % 16 cores
time0_0(6) = 4.902279; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup0_0 = zeros(1,length(cores));
efficiency0_0 = zeros(1,length(cores));
t0_0 = zeros(1,length(cores));
for i=1:length(cores)
    speedup0_0(i) = time0_0(1)/time0_0(i);
    efficiency0_0(i) = speedup0_0(i)/cores(i);
    t0_0(i) = time0_0(1)/cores(i);
end

% Sequence = 1 Strategy = 0
time1_0 = zeros(1,length(cores));
time1_0(1) = 29.117637; % 1 core
time1_0(2) = 28.721584; % 2 cores
time1_0(3) = 22.506445; % 4 cores
time1_0(4) = 19.215338; % 8 cores
time1_0(5) = 16.627161; % 16 cores
time1_0(6) = 15.724588; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup1_0 = zeros(1,length(cores));
efficiency1_0 = zeros(1,length(cores));
t1_0 = zeros(1,length(cores));
for i=1:length(cores)
    speedup1_0(i) = time1_0(1)/time1_0(i);
    efficiency1_0(i) = speedup1_0(i)/cores(i);
    t1_0(i) = time1_0(1)/cores(i);
end

% Sequence = 2 Strategy = 0
time2_0 = zeros(1,length(cores));
time2_0(1) = 29.218643; % 1 core
time2_0(2) = 38.792964; % 2 cores
time2_0(3) = 32.160024; % 4 cores
time2_0(4) =  29.581625; % 8 cores
time2_0(5) = 27.506236; % 16 cores
time2_0(6) = 26.894346; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup2_0 = zeros(1,length(cores));
efficiency2_0 = zeros(1,length(cores));
```

```
t2_0 = zeros(1,length(cores));
for i=1:length(cores)
    speedup2_0(i) = time2_0(1)/time2_0(i);
    efficiency2_0(i) = speedup2_0(i)/cores(i);
    t2_0(i) = time2_0(1)/cores(i);
end

% Sequence = 3 Strategy = 0
time3_0 = zeros(1,length(cores));
time3_0(1) = 11.361397; % 1 core
time3_0(2) = 8.201077; % 2 cores
time3_0(3) = 7.781006; % 4 cores
time3_0(4) = 6.243395; % 8 cores
time3_0(5) = 3.614225; % 16 cores
time3_0(6) = 2.988533; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup3_0 = zeros(1,length(cores));
efficiency3_0 = zeros(1,length(cores));
t3_0 = zeros(1,length(cores));
for i=1:length(cores)
    speedup3_0(i) = time3_0(1)/time3_0(i);
    efficiency3_0(i) = speedup3_0(i)/cores(i);
    t3_0(i) = time3_0(1)/cores(i);
end

% Generating figures
figure
hold on
grid on
plot(cores, time0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, time1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, time2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, time3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
%plot(cores, t0_0, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
%plot(cores, t1_0, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
%plot(cores, t2_0, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
%plot(cores, t3_0, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
title('Strong Scaling, Strategy = 0, Time');
xlabel('Nr of processes');
ylabel('Measured time');
hold off

figure
hold on
grid on
plot(cores,speedup0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores,speedup1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores,speedup2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores,speedup3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Strong Scaling, Strategy = 0, Speed-up');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off

% ------------ STRATEGY = 1 ------------

% Sequence = 0 Strategy = 1
time0_1 = zeros(1,length(cores));
time0_1(1) = 30.387830; % 1 core
time0_1(2) = 19.329221; % 2 cores
time0_1(3) = 12.838916; % 4 cores
```

```
time0_1(4) = 9.207917; % 8 cores
time0_1(5) = 5.844428; % 16 cores
time0_1(6) = 4.929357; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup0_1 = zeros(1,length(cores));
efficiency0_1 = zeros(1,length(cores));
t0_1 = zeros(1,length(cores));
for i=1:length(cores)
    speedup0_1(i) = time0_1(1)/time0_1(i);
    efficiency0_1(i) = speedup0_1(i)/cores(i);
    t0_1(i) = time0_1(1)/cores(i);
end

% Sequence = 1 Strategy = 1
time1_1 = zeros(1,length(cores));
time1_1(1) = 29.084993; % 1 core
time1_1(2) = 28.703074; % 2 cores
time1_1(3) = 22.535320; % 4 cores
time1_1(4) = 19.215622; % 8 cores
time1_1(5) = 16.678529; % 16 cores
time1_1(6) = 15.804138; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup1_1 = zeros(1,length(cores));
efficiency1_1 = zeros(1,length(cores));
t1_1 = zeros(1,length(cores));
for i=1:length(cores)
    speedup1_1(i) = time1_1(1)/time1_1(i);
    efficiency1_1(i) = speedup1_1(i)/cores(i);
    t1_1(i) = time1_1(1)/cores(i);
end

% Sequence = 2 Strategy = 1
time2_1 = zeros(1,length(cores));
time2_1(1) = 29.225064; % 1 core
time2_1(2) = 38.682093; % 2 cores
time2_1(3) = 32.456669; % 4 cores
time2_1(4) = 29.577839; % 8 cores
time2_1(5) = 27.584621; % 16 cores
time2_1(6) = 26.715707; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup2_1 = zeros(1,length(cores));
efficiency2_1 = zeros(1,length(cores));
t2_1 = zeros(1,length(cores));
for i=1:length(cores)
    speedup2_1(i) = time2_1(1)/time2_1(i);
    efficiency2_1(i) = speedup2_1(i)/cores(i);
    t2_1(i) = time2_1(1)/cores(i);
end

% Sequence = 3 Strategy = 1
time3_1 = zeros(1,length(cores));
time3_1(1) = 11.379709; % 1 core
time3_1(2) = 8.254606; % 2 cores
time3_1(3) = 7.038642; % 4 cores
time3_1(4) = 5.595951; % 8 cores
time3_1(5) = 3.312614; % 16 cores
time3_1(6) = 2.834814; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup3_1 = zeros(1,length(cores));
efficiency3_1 = zeros(1,length(cores));
t3_1 = zeros(1,length(cores));
```

```
for i=1:length(cores)
    speedup3_1(i) = time3_1(1)/time3_1(i);
    efficiency3_1(i) = speedup3_1(i)/cores(i);
    t3_1(i) = time3_1(1)/cores(i);
end

% Generating figures
figure
hold on
grid on
plot(cores, time0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, time1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, time2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, time3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
%plot(cores, t0_1, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
%plot(cores, t1_1, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
%plot(cores, t2_1, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
%plot(cores, t3_1, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
title('Strong Scaling, Strategy = 1, Time');
xlabel('Nr of processes');
ylabel('Measured time');
hold off

figure
hold on
grid on
plot(cores,speedup0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores,speedup1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores,speedup2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores,speedup3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, cores, ':o', 'Color', [51/255 153/255 255/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Strong Scaling, Strategy = 1, Speed-up');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off



% ------------ STRATEGY = 2 ------------

% Sequence = 0 Strategy = 2
time0_2 = zeros(1,length(cores));
time0_2(1) = 30.534189; % 1 core
time0_2(2) = 19.441526; % 2 cores
time0_2(3) = 12.773972; % 4 cores
time0_2(4) = 9.202893; % 8 cores
time0_2(5) = 5.854132; % 16 cores
time0_2(6) = 4.939196; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup0_2 = zeros(1,length(cores));
efficiency0_2 = zeros(1,length(cores));
t0_2 = zeros(1,length(cores));
for i=1:length(cores)
    speedup0_2(i) = time0_2(1)/time0_2(i);
    efficiency0_2(i) = speedup0_2(i)/cores(i);
    t0_2(i) = time0_2(1)/cores(i);
end

% Sequence = 1 Strategy = 2
time1_2 = zeros(1,length(cores));
time1_2(1) = 29.114526; % 1 core
time1_2(2) = 28.676153; % 2 cores
```

```
time1_2(3) = 22.483128; % 4 cores
time1_2(4) = 19.200803; % 8 cores
time1_2(5) = 16.652579; % 16 cores
time1_2(6) = 15.739615; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup1_2 = zeros(1,length(cores));
efficiency1_2 = zeros(1,length(cores));
t1_2 = zeros(1,length(cores));
for i=1:length(cores)
    speedup1_2(i) = time1_2(1)/time1_2(i);
    efficiency1_2(i) = speedup1_2(i)/cores(i);
    t1_2(i) = time1_2(1)/cores(i);
end

% Sequence = 2 Strategy = 2
time2_2 = zeros(1,length(cores));
time2_2(1) = 29.246983; % 1 core
time2_2(2) = 38.418233; % 2 cores
time2_2(3) = 32.073953; % 4 cores
time2_2(4) = 29.614933; % 8 cores
time2_2(5) = 27.487457; % 16 cores
time2_2(6) = 26.626454; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup2_2 = zeros(1,length(cores));
efficiency2_2 = zeros(1,length(cores));
t2_2 = zeros(1,length(cores));
for i=1:length(cores)
    speedup2_2(i) = time2_2(1)/time2_2(i);
    efficiency2_2(i) = speedup2_2(i)/cores(i);
    t2_2(i) = time2_2(1)/cores(i);
end

% Sequence = 3 Strategy = 2
time3_2 = zeros(1,length(cores));
time3_2(1) = 11.238143; % 1 core
time3_2(2) = 7.593101; % 2 cores
time3_2(3) = 6.507833; % 4 cores
time3_2(4) = 5.040112; % 8 cores
time3_2(5) = 2.877320; % 16 cores
time3_2(6) = 2.703439; % 32 cores

% Calculating speed-up, efficiency and ideal time
speedup3_2 = zeros(1,length(cores));
efficiency3_2 = zeros(1,length(cores));
t3_2 = zeros(1,length(cores));
for i=1:length(cores)
    speedup3_2(i) = time3_2(1)/time3_2(i);
    efficiency3_2(i) = speedup3_2(i)/cores(i);
    t3_2(i) = time3_2(1)/cores(i);
end

% Generating figures
figure
hold on
grid on
plot(cores, time0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, time1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, time2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, time3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
%plot(cores, t0_2, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
%plot(cores, t1_2, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
%plot(cores, t2_2, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
%plot(cores, t3_2, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
```

```
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
title('Strong Scaling, Strategy = 2, Time');
xlabel('Nr of processes');
ylabel('Measured time');
hold off

figure
hold on
grid on
plot(cores,speedup0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores,speedup1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores,speedup2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores,speedup3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Strong Scaling, Strategy = 2, Speed-up');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off

%------------- Weak Scaling -------------

%cores = [1, 2, 4, 8, 16, 32];
%sizes = [125,250,500,1000,2000,4000];


% ------------ STRATEGY = 0 ------------

% Sequence = 0 Strategy = 0
wtime0_0 = zeros(1,length(cores));
wtime0_0(1) = 3.330238; % 1 cores
wtime0_0(2) = 4.558199; % 2 cores
wtime0_0(3) = 6.494247; % 4 cores
wtime0_0(4) = 9.330230; % 8 cores
wtime0_0(5) = 11.837867; % 16 cores
wtime0_0(6) = 19.639124; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup0_0 = zeros(1,length(cores));
wefficiency0_0 = zeros(1,length(cores));
wt0_0 = wtime0_0(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup0_0(i) = wtime0_0(1)/wtime0_0(i)*cores(i);
    wefficiency0_0(i) = wspeedup0_0(i)/cores(i);
end

% Sequence = 1 Strategy = 0
wtime1_0 = zeros(1,length(cores));
wtime1_0(1) = 3.264187; % 1 cores
wtime1_0(2) = 6.956834; % 2 cores
wtime1_0(3) = 11.019184; % 4 cores
wtime1_0(4) = 19.380542; % 8 cores
wtime1_0(5) = 33.474841; % 16 cores
wtime1_0(6) = 62.729792; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup1_0 = zeros(1,length(cores));
wefficiency1_0 = zeros(1,length(cores));
wt1_0 = wtime1_0(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup1_0(i) = wtime1_0(1)/wtime1_0(i)*cores(i);
    wefficiency1_0(i) = wspeedup1_0(i)/cores(i);
end

% Sequence = 2 Strategy = 0
```

```
wtime2_0 = zeros(1,length(cores));
wtime2_0(1) = 3.300334; % 1 cores
wtime2_0(2) = 9.425067; % 2 cores
wtime2_0(3) = 15.910506; % 4 cores
wtime2_0(4) = 30.065450; % 8 cores
wtime2_0(5) = 55.890569; % 16 cores
wtime2_0(6) = 106.613515; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup2_0 = zeros(1,length(cores));
wefficiency2_0 = zeros(1,length(cores));
wt2_0 = wtime2_0(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup2_0(i) = wtime2_0(1)/wtime2_0(i)*cores(i);
    wefficiency2_0(i) = wspeedup2_0(i)/cores(i);
end

% Sequence = 3 Strategy = 0
wtime3_0 = zeros(1,length(cores));
wtime3_0(1) = 1.109118; % 1 cores
wtime3_0(2) = 1.954304; % 2 cores
wtime3_0(3) = 3.123370; % 4 cores
wtime3_0(4) = 6.072183; % 8 cores
wtime3_0(5) = 6.966555; % 16 cores
wtime3_0(6) = 11.786162; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup3_0 = zeros(1,length(cores));
wefficiency3_0 = zeros(1,length(cores));
wt3_0 = wtime3_0(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup3_0(i) = wtime3_0(1)/wtime3_0(i)*cores(i);
    wefficiency3_0(i) = wspeedup3_0(i)/cores(i);
end

% Generating figures
figure
hold on
grid on
plot(cores, wtime0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, wtime1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, wtime2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, wtime3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
%plot(cores, wt0_0, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
%plot(cores, wt1_0, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
%plot(cores, wt2_0, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
%plot(cores, wt3_0, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
title('Weak Scaling, Strategy = 0, Time');
xlabel('Nr of processes');
ylabel('Measured time');
hold off

figure
hold on
grid on
plot(cores,wspeedup0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores,wspeedup1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores,wspeedup2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores,wspeedup3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Weak Scaling, Strategy = 0, Speed-up');
xlabel('Nr of processes');
ylabel('Speed-up');
```

```
hold off

% ------------ STRATEGY = 1 ------------

% Sequence = 0 Strategy = 1
wtime0_1 = zeros(1,length(cores));
wtime0_1(1) = 3.326104; % 1 cores
wtime0_1(2) = 4.510797; % 2 cores
wtime0_1(3) = 6.385795; % 4 cores
wtime0_1(4) = 9.233575; % 8 cores
wtime0_1(5) = 11.891216; % 16 cores
wtime0_1(6) = 19.550342; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup0_1 = zeros(1,length(cores));
wefficiency0_1 = zeros(1,length(cores));
wt0_1 = wtime0_1(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup0_1(i) = wtime0_1(1)/wtime0_1(i)*cores(i);
    wefficiency0_1(i) = speedup0_1(i)/cores(i);
end

% Sequence = 1 Strategy = 1
wtime1_1 = zeros(1,length(cores));
wtime1_1(1) = 3.253699; % 1 cores
wtime1_1(2) = 6.933638; % 2 cores
wtime1_1(3) = 10.888042; % 4 cores
wtime1_1(4) = 19.239228; % 8 cores
wtime1_1(5) = 33.337780; % 16 cores
wtime1_1(6) = 62.664098; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup1_1 = zeros(1,length(cores));
wefficiency1_1 = zeros(1,length(cores));
wt1_1 = wtime1_1(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup1_1(i) = wtime1_1(1)/wtime1_1(i)*cores(i);
    wefficiency1_1(i) = wspeedup1_1(i)/cores(i);
end

% Sequence = 2 Strategy = 1
wtime2_1 = zeros(1,length(cores));
wtime2_1(1) = 3.290981; % 1 cores
wtime2_1(2) = 9.385015; % 2 cores
wtime2_1(3) = 15.825040; % 4 cores
wtime2_1(4) = 29.931524; % 8 cores
wtime2_1(5) = 55.575614; % 16 cores
wtime2_1(6) = 106.233427; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup2_1 = zeros(1,length(cores));
wefficiency2_1 = zeros(1,length(cores));
wt2_1 = wtime2_1(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup2_1(i) = wtime2_1(1)/wtime2_1(i)*cores(i);
    wefficiency2_1(i) = wspeedup2_1(i)/cores(i);
end

% Sequence = 3 Strategy = 1
wtime3_1 = zeros(1,length(cores));
wtime3_1(1) = 1.110164; % 1 cores
wtime3_1(2) = 1.951809; % 2 cores
wtime3_1(3) = 2.932658; % 4 cores
wtime3_1(4) = 5.498014; % 8 cores
wtime3_1(5) = 6.448554; % 16 cores
```

```
wtime3_1(6) = 10.938075; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup3_1 = zeros(1,length(cores));
wefficiency3_1 = zeros(1,length(cores));
wt3_1 = wtime3_1(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup3_1(i) = wtime3_1(1)/wtime3_1(i)*cores(i);
    wefficiency3_1(i) = wspeedup3_1(i)/cores(i);
end

% Generating figures
figure
hold on
grid on
plot(cores, wtime0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, wtime1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, wtime2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, wtime3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
%plot(cores, wt0_1, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
%plot(cores, wt1_1, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
%plot(cores, wt2_1, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
%plot(cores, wt3_1, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
title('Weak Scaling, Strategy = 1, Time');
xlabel('Nr of processes');
ylabel('Measured time');
hold off

figure
hold on
grid on
plot(cores,wspeedup0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores,wspeedup1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores,wspeedup2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores,wspeedup3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Weak Scaling, Strategy = 1, Speed-up');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off



% ------------ STRATEGY = 2 ------------

% Sequence = 0 Strategy = 2
wtime0_2 = zeros(1,length(cores));
wtime0_2(1) = 3.333055; % 1 cores
wtime0_2(2) = 4.487260; % 2 cores
wtime0_2(3) = 6.316947; % 4 cores
wtime0_2(4) = 9.205733; % 8 cores
wtime0_2(5) = 11.887559; % 16 cores
wtime0_2(6) = 19.653326; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup0_2 = zeros(1,length(cores));
wefficiency0_2 = zeros(1,length(cores));
wt0_2 = wtime0_2(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup0_2(i) = wtime0_2(1)/wtime0_2(i)*cores(i);
    wefficiency0_2(i) = wspeedup0_2(i)/cores(i);
end

```

```
% Sequence = 1 Strategy = 2
wtime1_2 = zeros(1,length(cores));
wtime1_2(1) = 3.252186; % 1 cores
wtime1_2(2) = 7.097667; % 2 cores
wtime1_2(3) = 10.894351; % 4 cores
wtime1_2(4) = 19.252760; % 8 cores
wtime1_2(5) = 33.348899; % 16 cores
wtime1_2(6) = 62.891490; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup1_2 = zeros(1,length(cores));
wefficiency1_2 = zeros(1,length(cores));
wt1_2 = wtime1_2(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup1_2(i) = wtime1_2(1)/wtime1_2(i)*cores(i);
    wefficiency1_2(i) = wspeedup1_2(i)/cores(i);
end

% Sequence = 2 Strategy = 2
wtime2_2 = zeros(1,length(cores));
wtime2_2(1) = 3.290253; % 1 cores
wtime2_2(2) = 9.416190; % 2 cores
wtime2_2(3) = 15.958708; % 4 cores
wtime2_2(4) = 29.653036; % 8 cores
wtime2_2(5) = 55.075647; % 16 cores
wtime2_2(6) = 106.591060; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup2_2 = zeros(1,length(cores));
wefficiency2_2 = zeros(1,length(cores));
wt2_2 = wtime2_2(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup2_2(i) = wtime2_2(1)/wtime2_2(i)*cores(i);
    wefficiency2_2(i) = wspeedup2_2(i)/cores(i);
end

% Sequence = 3 Strategy = 2
wtime3_2 = zeros(1,length(cores));
wtime3_2(1) = 1.109987; % 1 cores
wtime3_2(2) = 1.765386; % 2 cores
wtime3_2(3) = 2.568317; % 4 cores
wtime3_2(4) = 5.130037; % 8 cores
wtime3_2(5) = 5.817957; % 16 cores
wtime3_2(6) = 10.754171; % 32 cores

% Calculating speed-up, efficiency and ideal time
wspeedup3_2 = zeros(1,length(cores));
wefficiency3_2 = zeros(1,length(cores));
wt3_2 = wtime3_2(1)*ones(1,length(cores));
for i=1:length(cores)
    wspeedup3_2(i) = wtime3_2(1)/wtime3_2(i)*cores(i);
    wefficiency3_2(i) = wspeedup3_2(i)/cores(i);
end

% Generating figures
figure
hold on
grid on
plot(cores, wtime0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, wtime1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, wtime2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, wtime3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
%plot(cores, wt0_2, ':o', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
%plot(cores, wt1_2, ':o', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
%plot(cores, wt2_2, ':o', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
```

```
%plot(cores, wt3_2, ':o', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3');
title('Weak Scaling, Strategy = 2, Time');
xlabel('Nr of processes');
ylabel('Measured time');
hold off

figure
hold on
grid on
plot(cores,wspeedup0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores,wspeedup1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores,wspeedup2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores,wspeedup3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, cores, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Weak Scaling, Strategy = 2, Speed-up');
xlabel('Nr of processes');
ylabel('Speed-up');
hold off

%------------- Efficiency -------------
e = ones(1,length(cores));

figure
hold on
grid on
plot(cores, efficiency0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, efficiency1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, efficiency2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, efficiency3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Strong Scaling, Strategy = 0, Efficiency');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off

figure
hold on
grid on
plot(cores, efficiency0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, efficiency1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, efficiency2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, efficiency3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Strong Scaling, Strategy = 1, Efficiency');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off

figure
hold on
grid on
plot(cores, efficiency0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, efficiency1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, efficiency2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, efficiency3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Strong Scaling, Strategy = 2, Efficiency');
```

```
xlabel('Nr of processes');
ylabel('Efficiency');
hold off

figure
hold on
grid on
plot(cores, wefficiency0_0, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, wefficiency1_0, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, wefficiency2_0, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, wefficiency3_0, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Weak Scaling, Strategy = 0, Efficiency');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off

figure
hold on
grid on
plot(cores, wefficiency0_1, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, wefficiency1_1, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, wefficiency2_1, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, wefficiency3_1, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Weak Scaling, Strategy = 1, Efficiency');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off

figure
hold on
grid on
plot(cores, wefficiency0_2, '-*', 'Color', [255/255 153/255 204/255], 'LineWidth', 2);
plot(cores, wefficiency1_2, '-*', 'Color', [204/255 153/255 255/255], 'LineWidth', 2);
plot(cores, wefficiency2_2, '-*', 'Color', [178/255 255/255 102/255], 'LineWidth', 2);
plot(cores, wefficiency3_2, '-*', 'Color', [255/255 178/255 102/255], 'LineWidth', 2);
plot(cores, e, ':o', 'Color', [102/255 178/255 255/255], 'LineWidth', 2);
ylim([-0.1 1.2])
legend('Sequence=0','Sequence=1','Sequence=2','Sequence=3','Ideal');
title('Weak Scaling, Strategy = 2, Efficiency');
xlabel('Nr of processes');
ylabel('Efficiency');
hold off
```