Università
di Genova

MSc Computer Science
Data Science and Engineering Curriculum

MPI assignment

High Performance Computing

Roberto Di Via

January, 2023

Contents

1 Introduction

The goal of this homework is to parallelize the following program, which calculates an approximation of pi value. The original program sequentially computes the sum of a series of terms, each of which is calculated using a loop over a fixed number of intervals. By implementing parallelization, I hope to improve the performance of the program and reduce the time required to compute it.

1.1 How to compile and execute the program

First of all, I have to connect to the INFN Cluster. This cluster contains 8 nodes with each an Intel® Xeon Phi™ 7210 as CPU. This CPU has 64 cores and supports hyperthreading with 256 logical cores. To log in I use the command:

```
ssh -CYJ rodivia@linuxge.ge.infn.it  rodivia@hpcocapie09.ge.infn.it
```

1.1.1 MPI version:

To compile the MPI program I use the Intel C compiler **mpiicc** in this way:

```
mpiicc mpi_parallel.c -o parallel
```

To execute it I use the command **mpirun** with the following flags:

-hostfile	Specify a file containing a list of hostnames where MPI processes should be run.
-np	Specify the number of processes to run.
-perhost	Specify the number of processes to run on each host.

So the command line to execute the program in a distributed way is:

```
mpirun -hostfile ~/mpi_hosts.txt -np <n>  -perhost <n> ./parallel
```

While to execute the program on a single machine is:

```
mpirun -np <n> ./parallel
```

1.1.2 MPI + OMP version:

For the parallelized MPI + OMP program, it's like the MPI version but first I have to define the number of threads for the OMP directives. I define it with:

```
export OMP_NUM_THREADS=<n>
```

where $< n >$ corresponds to the number of threads I want to use.

Then, to compile I use:

```
mpiicc -D _OMP -qopenmp mpi_omp_parallel.c -o parallel
```

where:
-D _OMP is used to define my OMP code.
-qopenmp is used to generate multi-threaded code based on OpenMP.

2 Analysis of the program

The program begins defining a constant INTERVALS which determines the number of intervals to use to calculate an approximation of the mathematical constant pi. Then initialize the variables sum and dx to 0 and 1/INTERVALS, respectively. After that, there is a loop over the intervals, where it calculates each term of the series using the current value of i with the following formula:

$$\sum_{i=1}^{N} \frac{4}{1+x_i^2}\Delta \cong \pi$$

$$where \quad \Delta = 1/N \quad and \quad x_i = (i-0.5)\Delta$$

The value of the term is added to the running total stored in sum. After the loop finishes, the program calculates the approximation of pi by multiplying dx by sum and storing the result in the variable pi. Finally, the program prints the computed value of pi, the true value of pi, and the elapsed time for the computation.

```
#define PI25DT 3.14159265358979323846264 3
#define INTERVALS 100000000000

int main(int argc, char **argv) {
    ...
    sum = 0.0;
    dx = 1.0 / (double) intervals;

    for (i = 1; i <= intervals; i++) {
        x = dx * ((double) (i - 0.5));
        f = 4.0 / (1.0 + x*x);
        sum = sum + f;
    }

    pi = dx*sum;
    ...
    printf("Computed PI %.24f\n", pi);
    printf("The true PI %.24f\n\n", PI25DT);
    printf("Elapsed time (s) = %.2lf\n", time2);

    return 0;
}
```

3 Parallelization with MPI

In the original version of the program, the loop over the intervals is executed sequentially by a single process. This can be slow if the number of intervals is very large since it requires a lot of computation to be done by a single process.

To make the program faster, I can use MPI to parallelize the loop over the intervals by dividing the work among multiple processes and multiple machines. Each process calculates the sum of a subset of the intervals, and then the partial sums are combined together.

To do this, I first initialize MPI and get the rank and size of the current process. The rank of a process is its unique identifier within the group of processes, and the size is the total number of processes.

```
1    int rank, size;
2    MPI_Init(&argc, &argv);
3    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
4    MPI_Comm_size(MPI_COMM_WORLD, &size);
```

Next, we calculate the start and end indices for the loop based on the rank and size of the current process. For example, if we have 4 processes and the current process has rank 2, then the start index for the loop will be 2 * (INTERVALS/4) + 1. If we have 8 intervals, the start index will be 2*(8/4)+1 = 5, and the end index will be 5 + (INTERVALS/4) - 1, so 5+(8/4)-1 = 6. This means that the process with rank 2 will calculate the sum of the intervals from 5 to 6, that's correct.

```
1    // Divide the work among the processes
2    long int intervals_per_process = intervals / size;
3    long int start = rank * intervals_per_process + 1;
4    long int end = start + intervals_per_process - 1;
5
6    // Perform the iterations assigned to the current process
7    for (i = start; i <= end; i++) {
8        x = dx * ((double) (i - 0.5)); // Center of each rectangle
9        f = 4.0 / (1.0 + x*x); // Area of each rectangle
10       sum = sum + f;
11   }
```

After the loop finishes, each process has a partial sum that needs to be combined into a global sum. I use the MPI_Reduce to perform this reduction, with the process of rank 0 being the root process that receives the final result. The final result is then multiplied by dx and stored in the variable pi.

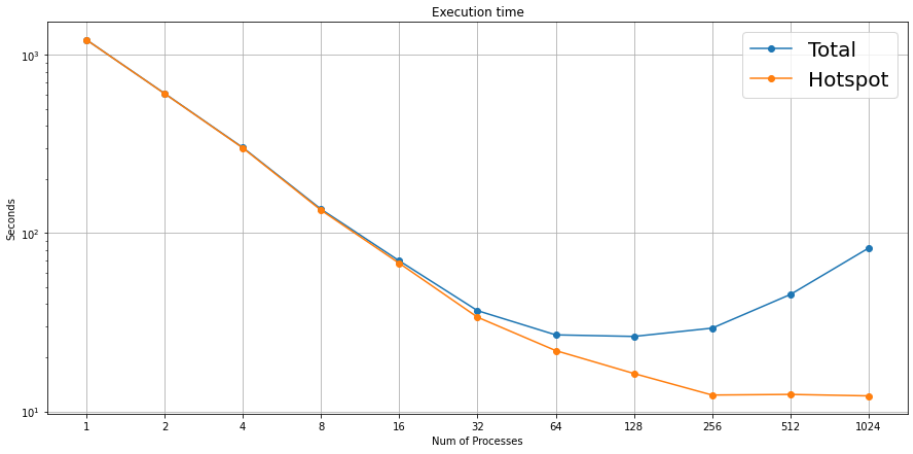
```
    double global_sum;
    MPI_Reduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Finally, the process of rank 0 prints the computed value of pi, the true value of pi, and the elapsed time for the computation. I also use the MPI Finalize function to clean up all MPI states and terminate them.

```
    if (rank == 0) {
        pi = dx * global_sum;

        end_time = MPI_Wtime();
        double elapsed_time = end_time - start_time;

        printf("Computed PI %.24f\n", pi);
        printf("The true PI %.24f\n\n", PI25DT);
        printf("Elapsed time (s) = %.2lf\n", elapsed_time);
    }

    MPI_Finalize();
```

3.1 Execution time on a single machine

In my initial experiment to assess the impact of MPI on performance, I measured the execution time of the program on a single machine. I found that as the number of processes increased, the hotspot execution time decreased logarithmically. However, the overall program execution time only improved up to 64 processes. Beyond that, it started to increase due to the overhead of communication between processes and the limited resources available on the machine.



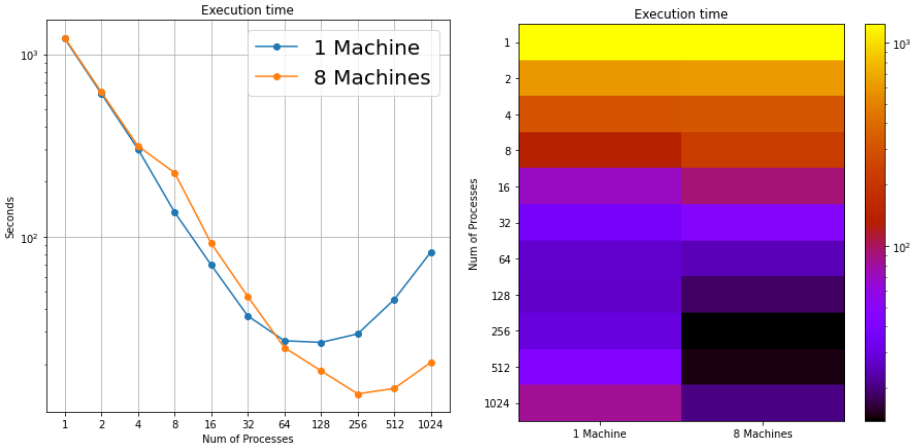
3.2 Scalability

Scalability refers to a program's ability to take advantage of multiple processors or cores to improve its performance. A program that scales well will run faster on a larger number of processors or cores, while a program that does not scale well will not see much improvement in performance with additional processors or cores.

To determine the scalability of the parallelized MPI program, I measured the execution time using a different number of processes, both on a single machine and distributed on multiple machines (in this case 8). The results that I achieved are shown in the table below.

Processes / Hosts	1	8
1	1218.33	1230.89
2	608.38	619.51
4	302.3	314.1
8	136.34	224.
16	69.95	91.96
32	36.75	46.93
64	26.87	24.65
128	26.33	18.4
256	29.32	13.75
512	45.28	14.73
1024	82.47	20.48

Finally, I visualized the scalability of the program by plotting the results.



The line chart and heatmap show the behaviour of the program's execution time with an increasing number of processes. It can be observed that for the program running on a single machine, the execution time decreases until 64

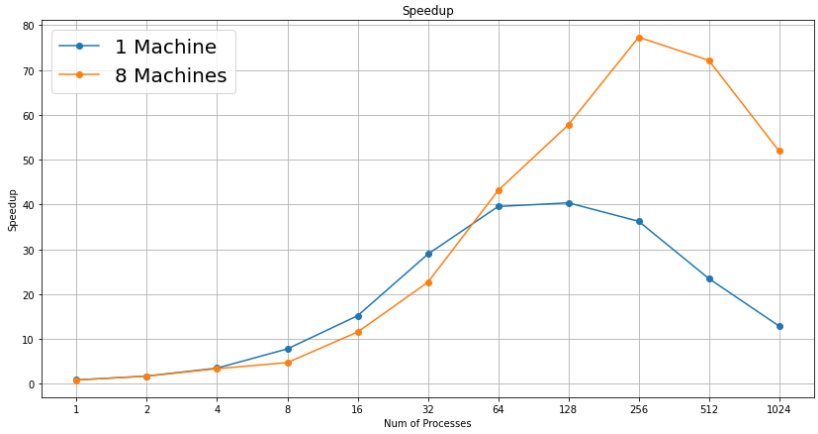
processes are utilized, after which it begins to increase. On the other hand, when the program is running in a distributed environment, the execution time decreases until 256 processes are used. The increasing execution time after 64 processes on a single machine and 256 processes in a distributed environment may be due to the overhead of communication between the processes and the limitations of the available resources.

3.3 Speedup

Speedup is a measure of how much faster a parallelized program can run compared to the same program running on a single processor. There are several reasons why it is useful to calculate the speedup of a parallel computing system. By calculating it, we can compare the relative performance of different parallel programs. We can determine the optimal configuration under different configurations (in this case using different numbers of threads and processes), or we can determine how well the system is utilizing its resources and so identify inefficiencies.

The speedup in parallel computing can be defined as:

$$Speedup = \frac{Time\ on\ single\ processor}{Time\ on\ parallel\ system}$$



The program runs faster when multiple machines are used (distributed version) than just one machine (single machine version). The performance of the program on a single machine improves up to 64 processes, but then decreases. This is because the machine's resources become limited and the communication between the processes causes overhead. On the other hand, the performance of the distributed version continues to improve until 256 processes, but then also decreases because of communication overhead and limited resources. Overall, using multiple machines allows for more processing power and helps to reduce the impact of resource limitations.

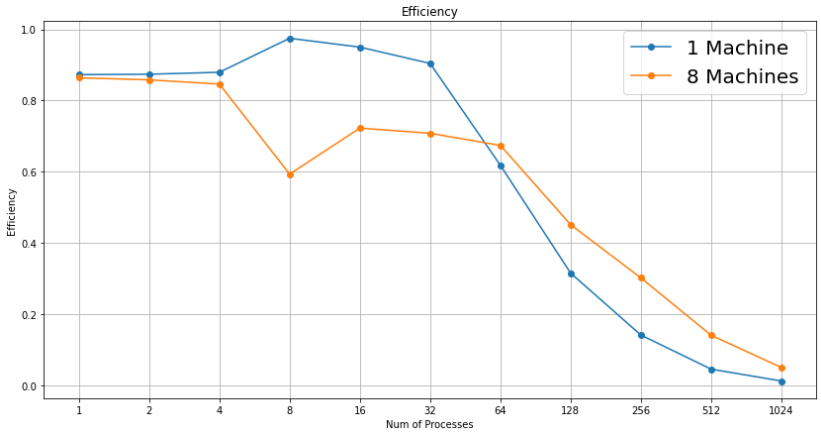
3.4 Efficiency

The efficiency of a parallelized program depends on several factors, including the number of processors or cores available, the nature of the tasks being performed, and the communication and synchronization overhead between the different processes or threads. In general, a parallelized program can be more efficient than a single-threaded program because it can take advantage of multiple processors or cores to perform tasks simultaneously. However, if the overhead of communication and synchronization between the processes or threads is high, it may decrease the overall efficiency of the program.

The efficiency (E) of a parallelized program is defined as:

$$E = \frac{Speedup}{Number\ of\ Processors}$$

Where *Speedup* is the ratio of the execution time of the program on a single processor to the execution time of the program on multiple processors.



The program running on one machine shows a decline in efficiency after using 32 processes, while the same program running on multiple machines shows a decline after using 64 processes. The reason for this is that as more processes are used, the communication between them becomes a more significant challenge and slows down the program. Distributing the program to run on multiple machines can help improve its efficiency because it allows for more processing power. However, it also increases the need for communication between the processes, leading to a decline in efficiency after a certain point. Overall, distributing the program is still a better choice because it provides more resources for computation and reduces the impact of limited resources on a single machine.

4 Parallelization with MPI + OMP

In this version of the program, I use both MPI and OMP to parallelize the computation of the approximation of pi.

The program begins by initializing MPI, determining the rank, size and then calculating the start-end indices for the loop, as in the only MPI version.

Next, we use the OMP directive "#pragma omp parallel for" to parallelize the loop over the intervals. This directive creates a team of threads that will execute the loop in parallel. The reduction clause specifies that the variable sum should be used to store a running total of the loop iterations, and the private clause specifies that each thread should have its own copies of the variables x and f. The variable i is private by default.

```
#ifdef _OMP
    #pragma omp parallel for private(x, f) reduction(+:sum)
#endif
for (i = start; i <= end; i++) {
    x = dx * ((double) (i - 0.5));
    f = 4.0 / (1.0 + x*x);
    sum = sum + f;
}
```

After the loop finishes, each process has a partial sum that needs to be combined into a global sum. To perform this reduction I use the MPI_Reduce, with the process of rank 0 being the root process that receives the final result. The final result is then multiplied by dx and stored in the variable pi.

4.1 Scalability

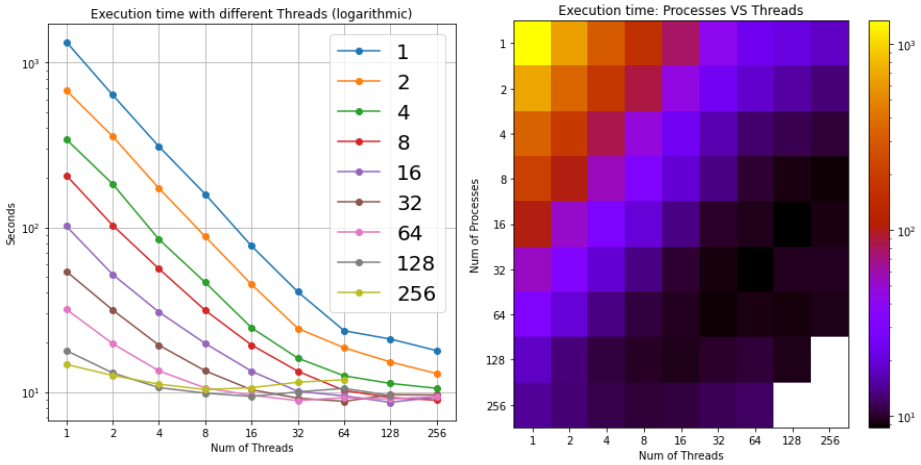
To determine the scalability of the OMP + MPI version I carried out experiments with a different number of threads and a different number of processes. The results of MPI experiments indicate that distributing processes across multiple machines can lead to improved outcomes, particularly as the number of processes utilized increases. Therefore, for our next experiments, I will distribute the processes across 8 machines. The resulting data are displayed in the table below.

Execution time on 8 Machine									
Processes / Threads	1	2	4	8	16	32	64	128	256
1	1335.15	636.63	309.55	158.97	77.05	40.57	23.56	20.97	17.84
2	678.36	356.65	172.79	87.88	44.99	24.32	18.56	15.23	12.95
4	342.15	183.34	84.54	46.29	24.55	16.05	12.55	11.28	10.53
8	206.73	102.89	56.02	31.32	19.29	13.37	10.21	9.27	8.92
16	101.57	51.76	30.59	19.73	13.38	10.1	9.48	**8.67**	9.33
32	54.01	31.5	19.25	13.44	10.32	9.18	8.78	9.64	9.6
64	31.78	19.66	13.44	10.58	9.57	8.87	9.25	9.08	9.4
128	17.85	13.07	10.64	9.84	9.41	10.01	10.53	9.52	x
256	14.72	12.61	11.17	10.36	10.62	11.48	11.88	x	x

In the tables, some cells are represented by the symbol 'x', this is because, in that particular configuration, the program gives the following error:

```
=====================================================================
 BAD TERMINATION OF ONE OF YOUR APPLICATION PROCESSES
 RANK 1 PID 20066 RUNNING AT node01
 KILLED BY SIGNAL: 9 (Killed)
=====================================================================
```

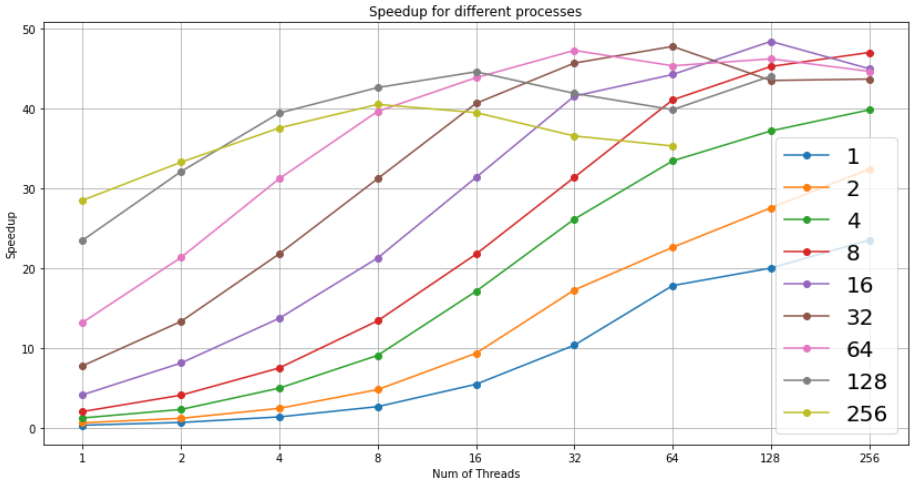
This happens because, in my MPI and OMP version, I have to be careful about the number of threads and processes I initialize. Each process will have its own memory space, and each thread will also have its own stack. Therefore, if I'm using too many processes and threads at the same time, the program may run out of memory because each thread and process will require a certain amount of memory to run. To optimize the performance of my parallel program and avoid running out of memory, I have to find the right balance between the number of threads and processes and the available resources on the machine.



The logarithmic line-chart demonstrates that as the number of processes and threads increases, the execution time decreases. However, determining the optimal configuration can be challenging. To address this, a 2D representation of the program's execution time is provided to effectively visualize the effect of changing the number of threads and processes. The colorbar on the right side of the heatmap corresponds to the execution time, with different shades of color representing different execution times. Thanks to this representation it's easy to identify patterns in the execution time as we change the number of threads and processes used. In this case, the best configuration is obtained using 32 processes and 64 threads or 16 processes and 128 threads.

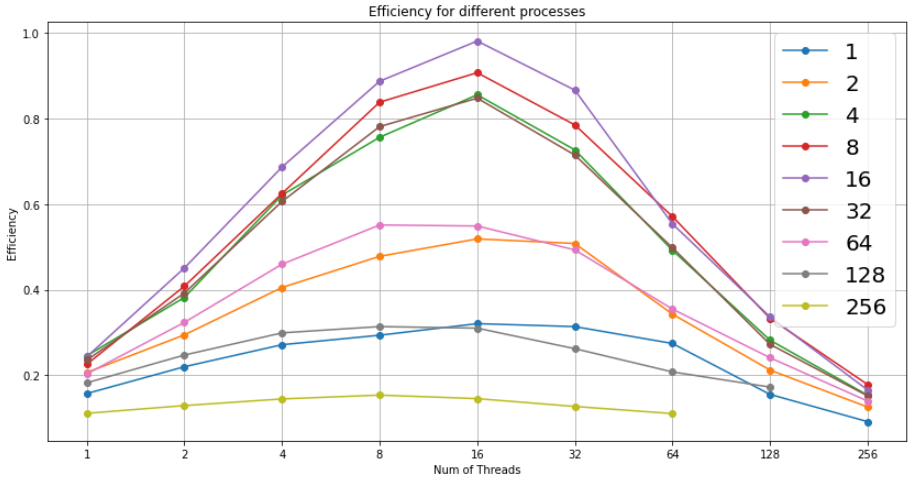
4.2 Speedup

The line-chart below shows that the optimal speedup is achieved by utilizing either 16 processes and 128 threads or 32 processes and 64 threads, as we previously noted.



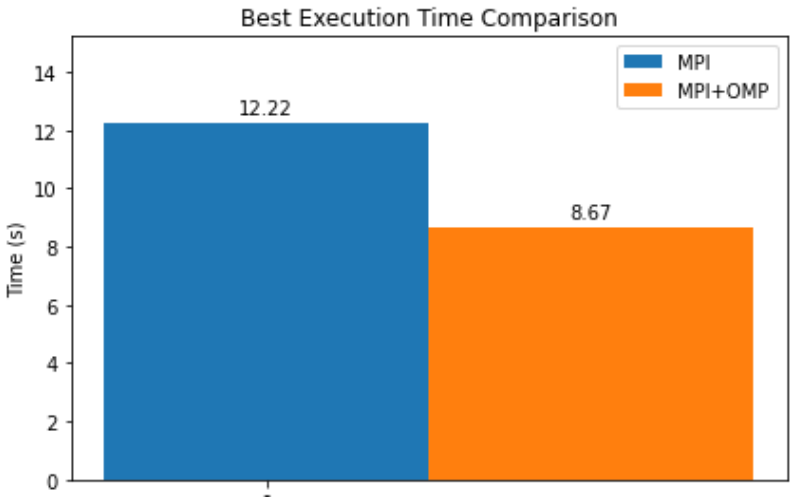
4.3 Efficiency

The efficiency of the MPI+OMP version appears to increase until 16 threads are used, particularly for the combinations of 8, 16, and 32 processes. However, after 16 threads, the efficiency tends to decline for almost all process combinations. This suggests that beyond this point may result in overhead communication and so in a decrease in performance.



5 Conclusions

In this assignment, the goal was to parallelize a C program that computes the value of pi using both MPI and a combination of MPI and OMP, in order to improve its performance. The implementation of a reduction in the MPI version and a parallel for with a reduction clause in the MPI + OpenMP version was able to achieve this goal, by effectively distributing the computational workload among multiple processors. The bar-chart below shows a comparison between the two versions of the program.



To evaluate the performance of the parallel program, a series of experiments were conducted by varying the configurations of the number of threads and processes. The execution times were recorded and presented through a tabular format and different charts, providing a comprehensive comparison of the program's performance under various configurations. Additionally, the speedup and efficiency of the program were calculated to gain a deeper understanding of its performance.

The results of the experiments revealed that parallelization significantly improved the performance of the program and that distributing the processes across 8 machines rather than 1 can result too in improved performance.

However, it was also discovered that increasing the number of threads and processes beyond a certain point resulted in a depletion of available memory and overhead communication, leading to a decrease in performance. This highlights the importance of considering the number of threads and processes used (according to the architecture) when parallelizing a program, in order to achieve optimal performance.