MYE023 – Parallel Systems and Programming 2019-20

Program set # 2 (MPI)

Christoforos KarvelisAM: 2989 E-mail:
cse02989@cse.uoi.gr,christoph.karv@yahoo.gr

The 2nd set of exercises concerns the parallel programming with the model of transmitting messages through the MPI standard. The parallelization of 2 applications is requested: 1. multiplication of tables and 2. calculation of the number of prime numbers as well as the largest prime number.

All measurements were made in the following systems:

The compilation and execution of the programs took place on this machine:

Computer name	opti3060ws03
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

The following machines were used as nodes to share processes:

Computer name	opti3060ws05
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

Computer name	opti3060ws06
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

Computer name	opti3060ws07
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

Computer name	opti3060ws08
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

Computer name	opti3060ws09
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

Computer name	opti3060ws10
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

Computer name	opti3060ws11
Processor	Intel i3-8300
Many nuclei	4
Translator	gcc v7.5.0

The following commands were used to compile and run the programs:

```
$ mpicc filename.c

$ mpirun -np N -hostfile mynodes filename.c

$ mpirun -np N filename.c
```

Exercise 1

The problem

In this exercise it is requested to parallelize the calculation of the number of prime numbers as well as the largest prime number in an interval [0, N] using MPI. This must be done in such a way that each process undertakes to control a subset of numbers. Finally, it is requested to run the same program for 8, 16 and 32 processes at 2, 4 and 8 nodes respectively and to time each case.

Parallelization method

The serial program from the course website was used. MPI calls were used for parallelization. The loop

... for (i = 0; i <(n-1) / 2; ++ i) {....

was divided into each process as follows:

... for (i = myid; i <(n-1)/2; i += nproc) {{

so that each process finds primes in a separate subset of numbers. The calculation results of each process are sent to the process with id = 0 by calling:

MPI_Reduce (&lastprime, &lastprime_0, 1, MPI_LONG, MPI_MAX, 0, MPI_COMM_WORLD);

with which the maximum calculated lastprime is sent to process 0 and with the call:

MPI_Reduce (&count, &count_0, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

which sends the sum of all the calculated counts in process 0. Because the count is initialized to 1 for each process the final count will have nproc - 1 (for process 0) redundant added 1s that must be subtracted from count_0. Finally, process 0 undertakes to calculate the final times and to display the final results of the calculations.

Experimental results - measurements

The programs were run on the system mentioned in the introduction and the timing was done with the **MPI_Wtime()** function. 8, 16 and 32 processes were used which were performed in 2, 4 and 8 nodes respectively while each node undertook 4 out of 4 processes. Each case was executed 4 times and the total execution time, the time of the calculations of a process and finally by their subtraction the time of the overhead of the parallelism as well as their averages were calculated.

Total time:

Numerous processes	1st execution	2nd execution	3rd execution	4th execution	Average
8	1.688300	1.684032	1.687195	1.680142	1.68491725
16	2.445837	2.157956	2.116524	2.037644	2.18949025
32	1.217872	1.327689	1.257681	1.302303	1.27638625

Time for the overheads of the parallel:

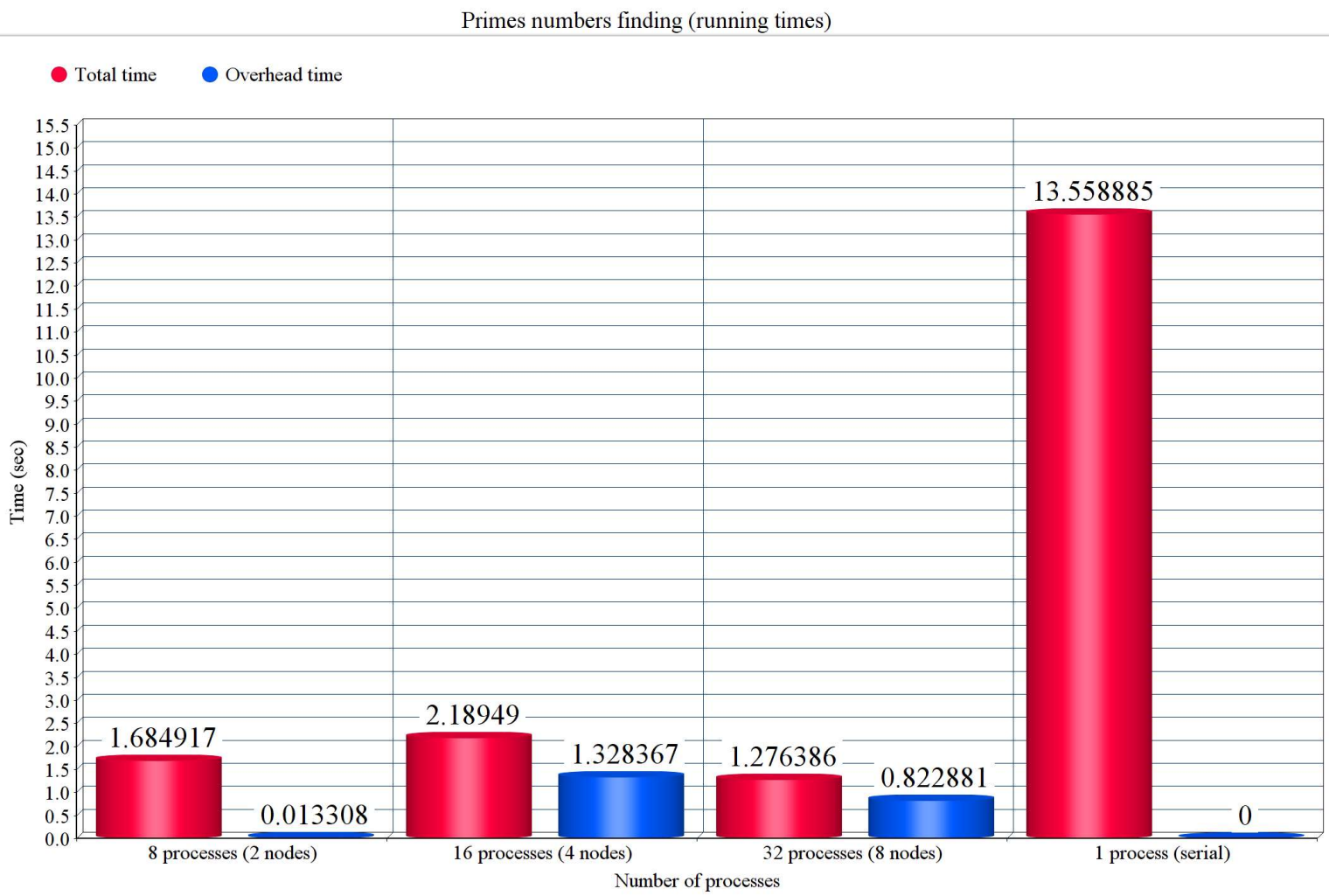
Numerous processes	1st execution	2nd execution	3rd execution	4th execution	Average
8	0.001662	0.016016	0.021687	0.013868	0.01330825
16	1.579796	1.307202	1.253149	1.173319	1.3283665
32	0.769243	0.852729	0.809435	0.860115	0.8228805

Time for the serial:

1st execution	2nd execution	3rd execution	4th execution	Average

13.558659	13.561592	13.557027	13.558262	13.558885

Based on theresults, the following graph is obtained:



Comments

Based on the results we see that as the number of processes increases the total execution time generally decreases, while the time for the overheads of the parallelism is increased for a large number of processes. All of the above is rather expected because for many processes we have a better division of labor as each process takes up less work and therefore less total execution time. On the contrary many processes lead to many communications and communications are known to be "enemy" of speed, so we have large overheads.

Exercise 2

The problem

In this exercise we are asked to parallelize the multiplication of arrays using MPI. This should be done by parallelizing the outer loop (which iterates through the rows). This parallelization is also known as the strip-partitioning method where each process undertakes the calculation of a "strip" of continuous rows of the result.

Parallelization method

The serial program from the course website was used. MPI calls were used for parallelization. The outer loop:

... for (i = 0; i <N; i ++) {...

in each process as follows:

... for (i = 0; i <WORK; i ++) {...

where WORK = N / (number of processes) so that in each process the the product of the WORK rows of matrix A with matrix B is calculated. Matrices A and B are read by files from process 0 and become known to the rest by calling:

MPI_Scatter (A, WORK * N, MPI_INT, A_sep, WORK * N, MPI_INT, 0, MPI_COMM_WORLD);

which distributes in each process WORK rows of matrice A and calling:

MPI_Bcast (B, N * N, MPI_INT, 0, MPI_COMM_WORLD);

which sends B to all processes. The results of each process are stored in mypart which is a pointer in consecutive memory locations where it is initialized with:

int * mypart = (int *) malloc (WORK * N * sizeof (int));

and can be used as a two-dimensional array in the following way:

for (i = 0; i <WORK; i ++) {for (j = 0; j <N; j ++) {mypart [i * N + j];

}

}

Finally mypart is sent by each process and stored in the correct positions of C in process 0 via:

MPI_Gather (mypart, WORK * N, MPI_INT, C, WORK * N, MPI_INT, 0, MPI_COMM_WORLD);

while process 0 undertakes to write the results found in C in a similar file.

Experimental results - measurements

The programs were run on the system mentioned in the introduction and the timing was done with the **MPI_Wtime()** function. 8, 16 and 32 processes were used which were performed in 2, 4 and 8 nodes respectively while each node undertook 4 out of 4 processes. Each case was executed 4 times and the total execution time, the time of calculations of a process and finally by their subtraction the time of the overhead of the parallelism as well as their averages were calculated.

Total time:

Numerous processes	1st execution	2nd execution	3rd execution	4th execution	Average
8	1.049278	1.040455	0.886349	1.013151	0.99730825
16	4.145759	4.317118	5.939499	5.941920	5.086074
32	11.125068	14.944103	13.250519	11.125068	12.6111895

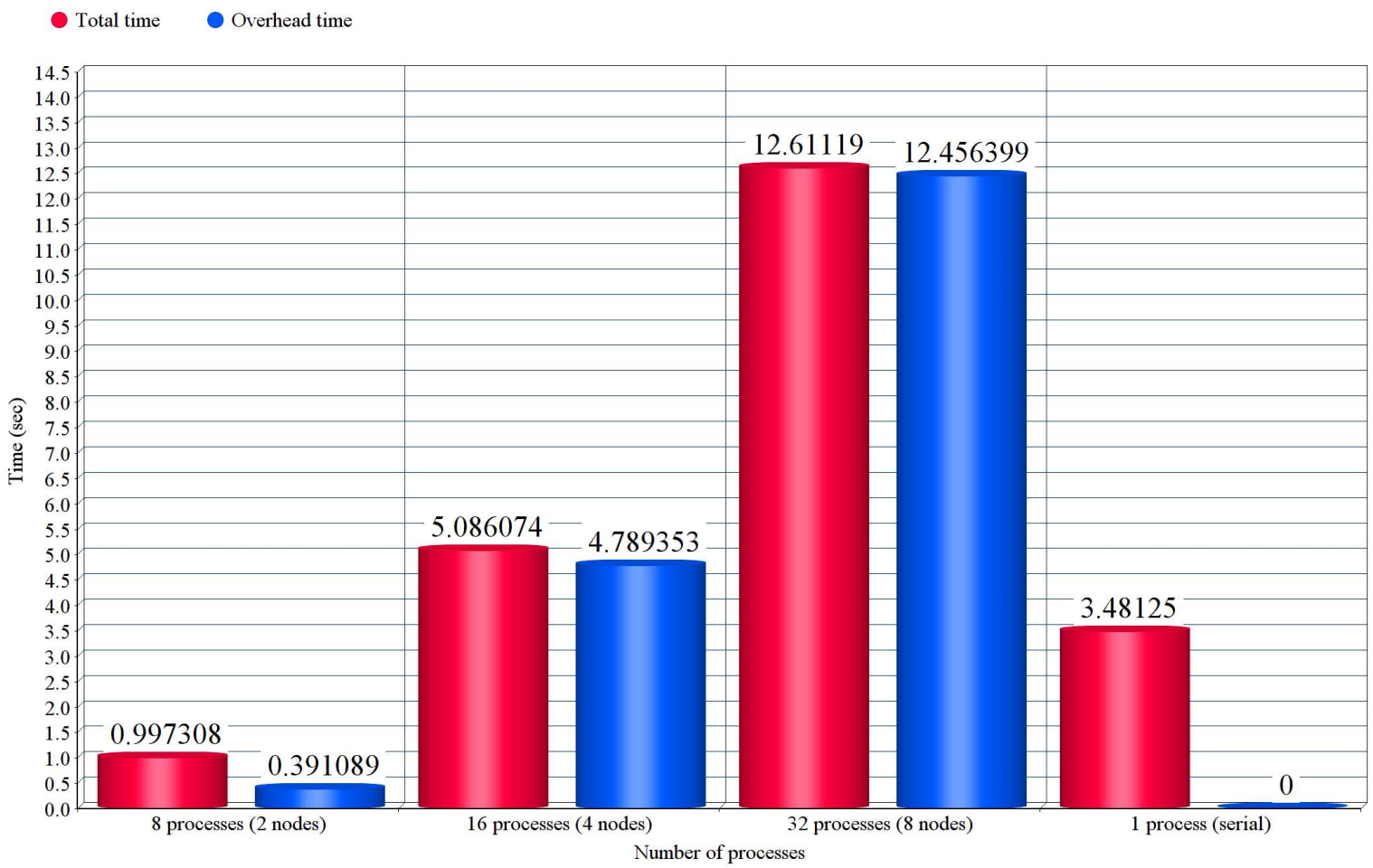
Time for the overheads of the parallel:

Numerous processes	1st execution	2nd execution	3rd execution	4th execution	Average
8	0.420495	0.457546	0.289322	0.396994	0.39108925
16	3.855927	4.018763	5.649437	5.633286	4.78935325
32	10.968047	14.793103	13.096397	10.968047	12.4563985

Rime for the serial:

1st execution	2nd execution	3rd execution	4th execution	Average
3.480173	3.480621	3.482184	3.482020	3.4812495

Based on the results, the following graph is obtained:



Comments

Based on the results we see that for 8 processes we have the best total execution time and the shortest time for the overheads of the parallelism. As we increase the number of the processes the time for the overheads increases rapidly resulting in an increase in the total execution time. All of the above are rather expected because from the graph we see that the execution time of the shared work by each process is quite short as most of the total time consists of the time spent on the overheads. So it makes sense, as we discussed in the previous experiment, that multiple processes lead to high costs in communication time between them and therefore we end up with a large increase in total execution time.