Basics and Maths in MPI

February 22, 2022

1 Exercise 1

Write a program in C (ask1.c) that implements parallel programming with Open MPI. Your program should

1. Display on the screen the number of tasks participating in the execution
2. Display on the screen the message: «Hello. I am the master node. » from the master task. Afterwards this message will be sent (by the master task) to all tasks.
3. As soon as the other tasks receive the above message, they will respond by displaying on the screen the message «Hello. This is node *.» where * is their id. After that, they send the particular message to the main task.
4. The main task will confirm that all the tasks have been answered with an appropriate message on the screen (e.g., «* nodes replied back to master node.» where * the number of tasks that sent their message).
5. The main task will count the execution time of the broadcast and will display it on the screen.
6. Add a table in your report, which will show the number of tasks and the corresponding execution time on your computer.

1.1 Solution

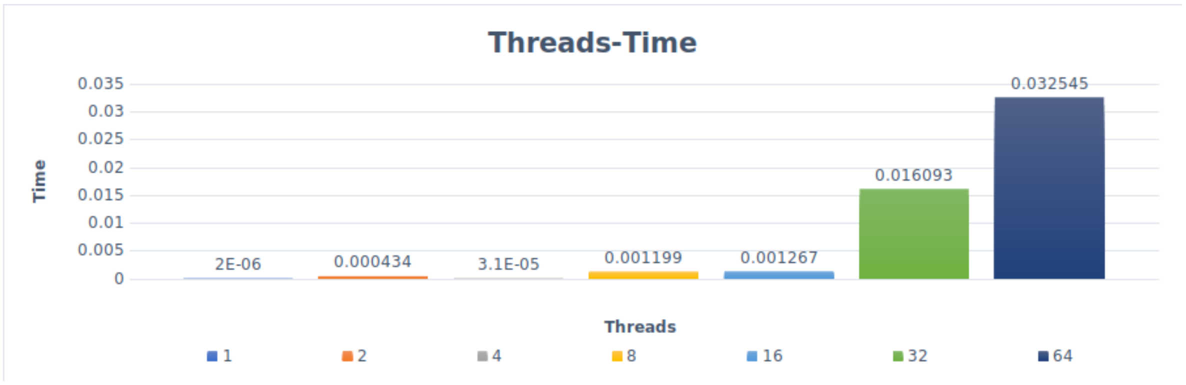
The master thread prints the requested information on the screen using some simple MPI API calls and then transmits the message to its children. Each child receives the message from master, prints a new message on the screen and sends its response back to the master thread. Finally, the master thread receives and counts all children responses. The function calls used to solve the problems above is described below.

1. **MPI_Comm_size(...)**: Stores the total number of threads participating in the execution in the variable passed as a parameter to the function.
2. **MPI_Send(...)**: Function that sends a message with a certain length of bytes to process i from current process in the defined communication world. In our case the master thread, with rank 0, sends messages to all children.
3. **MPI_Recv(...)**: Function that receives a message of a certain length and stores it into a buffer. In our case the receive function used on the children side is allowed to receive from any source since only the master thread actually is the only source. On the other hand, the server only receives from its children.
4. **MPI_Wtime(...)**: Function that returns the current time and is used to calculate computation and communication latency.

1.1.1 Table and Chart

As it can be observed from the graph below when the number of processes is increased the total execution time also increases. This increase in execution time can be attributed to the cost of communication between processes.

Processes	1	2	4	8	16	32	64
Time	0.000002	0.000434	0.000031	0.001199	0.001267	0.016093	0.032545



2 Exercise 2

Write a program in C (pi.c) that implements parallel programming with MPI and calculates an approximation of pi, using the formula

$$pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

Use Simpson's rule to approach the integral, i.e.,
if $x_i = i * dx$, i=0,1,...,N, $dx = \frac{1}{N}$, a partition of [0,1] with N intervals (N + 1 points)

$$pi = \int_0^1 \frac{4}{1+x^2}\,dx \approx \frac{dx}{3}(y + 4y_1 + 2y_2 + ... + 2y_{N-2} + 4y_{N-1} + y_N)$$

where $y_i = \frac{4}{1+x_i^2}$, i = 0,1,2, ..., N.

Your program should compute correctly the above sum by:

1. The main task should compute all the x i points and distributes them appropriately to the other tasks
2. The sum will break into nt (= number of tasks) partial sums and each one for a particular task.
3. The assignment of the terms of the sum to the tasks will be done in blocks according to the number of tasks. That is, the task with id = 0, will add the terms with K = 0, 1, 2,..., ((N + 1) / nt) -1, where nt the number of tasks, the task with id = 1 will add them terms with K = (N + 1) / nt,..., (2 (N + 1) / nt)-1, etc. The total sum will be computed by the main task and will be displayed on the screen accordingly.
4. Measure the execution time of calculations, the communication time and calculate the number of operations per sec. This message should appear on the screen: pi is approximately * , computations time = *, communication time = *, number of tasks = *, FLOPS = *
5. Add a table in your report that will contain all the above information. Hint: FLOPS = number of operations per sec. Use N+1 = $10^8, 10^{10}, 10^{12}, 10^{15}\ldots$ and nt = 1, 2, 4, 8, 16, 32 ....

Hint: FLOPS = number of operations per sec. Use N+1 = $10^8, 10^{10}, 10^{12}, 10^{15}\ldots$ and nt = 1, 2, 4, 8, 16, 32 ....

2.1 Solution

The master thread creates an array of different x_i items. This array is scattered across all processes and stored into a thread-private x_i array. Each thread calculates the partial sum based on its thread-private x_i array and its rank. When all partial sums are calculated the result is reduced into a single variable which is accessed by the master thread. The MPI API calls used and their usage can be seen below.

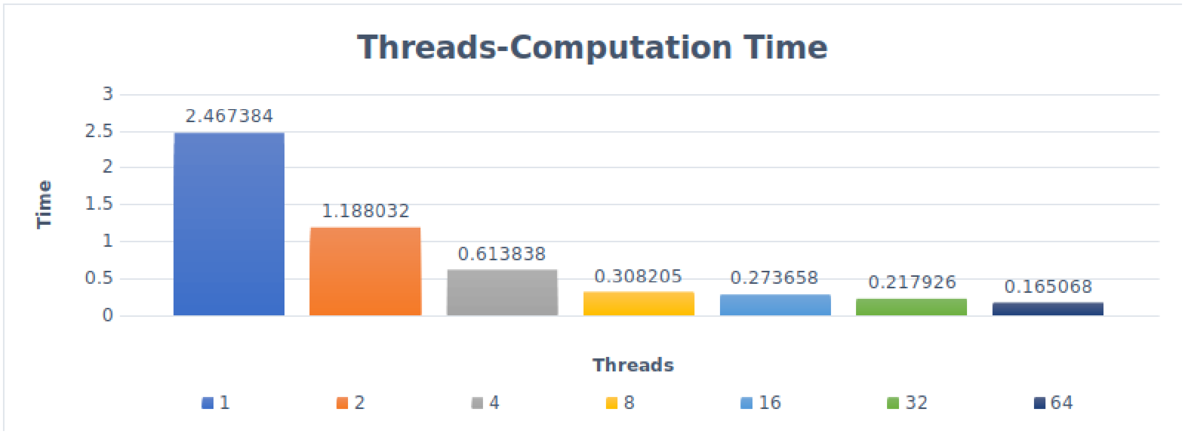
1. **MPI_Scatter(...)**: The x_i array is partitioned and its chunks are scattered to all available processes in the communication world.
2. **MPI_Reduce(...)**: The partial sums calculated by each thread are reduced into a single sum on the master thread.

2.2 Tables and Charts

2.2.1 Computation

As the number of processes increases the total computation time decreases since the workload is distributed to multiple processes that run in parallel.

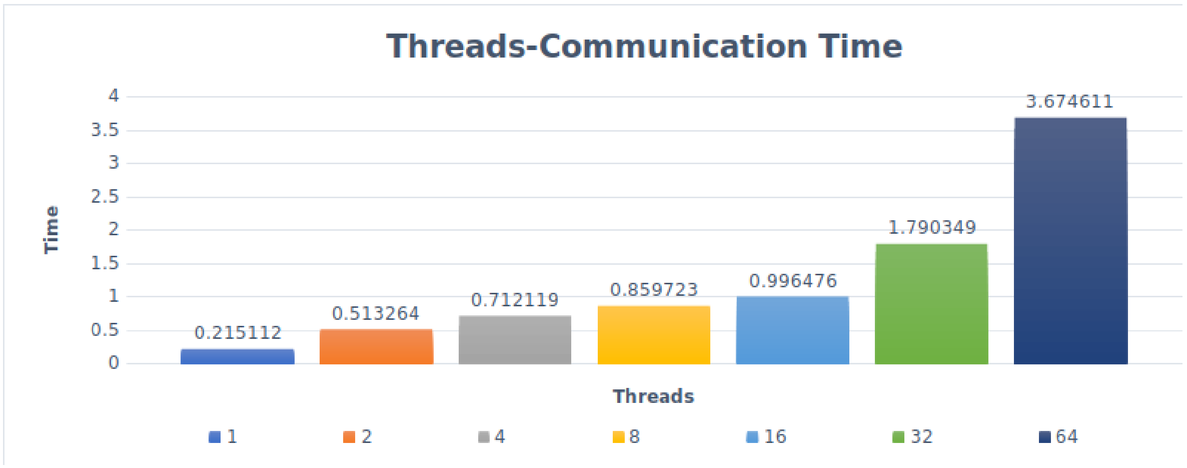
Processes	1	2	4	8	16	32	64
Time	2.467384	1.188032	0.613838	0.308205	0.273658	0.217926	0.165068



2.2.2 Communication

As the number of processes increases the total communication time also increases since more processes must communicate with each other. Latency due to increase in scattering and reduction times plays a huge part on the increase of the communication time.

Processes	1	2	4	8	16	32	64
Time	0.215112	0.513264	0.712119	0.859723	0.996476	1.790349	3.674611



3 Exercise 3

Write a program in C (jacobi.c) using MPI, which implements the Jacobi method for system solution where N is the dimension of the system

$$Ax = b \Leftrightarrow \begin{bmatrix} 2 & -1 & \\ -1 & \ddots & -1 \\ & -1 & 2 \end{bmatrix} x = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ N+1 \end{bmatrix}$$

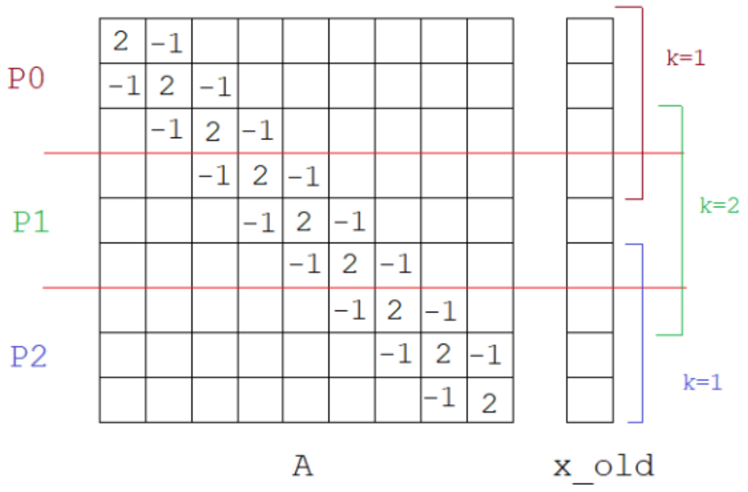
1. Runs for M iterations
2. In each iteration it should calculate the new approach of the solution, xnew, (and display on the screen) the remainder of the approach, i.e. || b - A xnew || 2 , and the difference of the last 2 approaches, i.e. || xold - xnew || 2 , with appropriate comments, e.g.: iter = *, residual = *, difference = *
3. Also display the execution time.
4. Run your program for different number of threads (1, 2, 4, 8, 16, 32, 64) and add a table to your report, which will show the number of threads and the corresponding execution time on your computer.

3.1 Solution

Given a matrix A and a result b the Jacobi method is applied. The Jacobi method is an iterative process and a new solution in each step is calculated with the method bellow.

$$X_{new} = D^{-1} * (b + (L + U) * X_{old})$$

Since the matrix is tridiagonal the complexity of Jacobi drops from O(N^2) to O(N). Each process takes N/P rows both from the matrix A and the result b. In each iteration each thread now takes $N/P + K$ from the vector X_{old} where K is equal to the rank of the process. The partitioning can be seen in the figure below.



The result vector b is scattered from master thread to all its children. In the beginning of each iteration the contents of X_{old} is also scattered to all processes. Each process computes its residual and difference based on the sub-matrix assigned to it and the result is reduced with a summation operation into the master thread. Finally, we gather all results to the X_{new} vector and we assign X_{old} to be equal to X_{new}. The additional MPI API calls used are described bellow.

1. **MPI_Gather(...)**: Information from all processes is gathered back to a certain process. In our case all processes send their data buffer to master process.

2. **MPI_Scatterv(...)**: A specified data buffer is transmitted to all processes and thus all processes view the same instance of the data buffer.

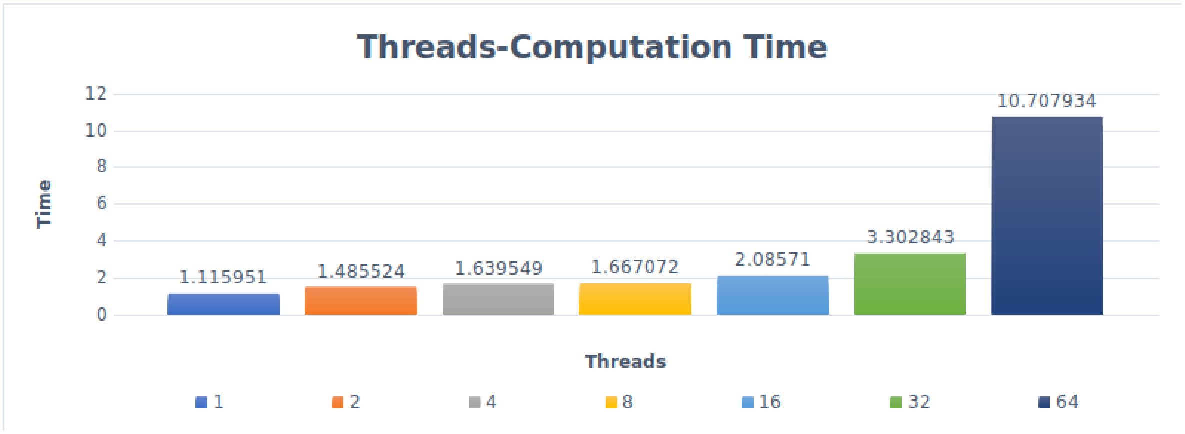
3.1.1 Tables and Charts

For explicitly defined dimensions $N = 10^8$ and a total of 40 iterations the tables and the corresponding charts is as follows.

3.1.2 Computation

Contrary to the example above when the number of processes increases the total computation time also increases. The results here might seem counter-intuitive. The increase in computation time can be attributed to high latency memory operations, high rate of cache misses and the latency due to conservation of cache coherence.

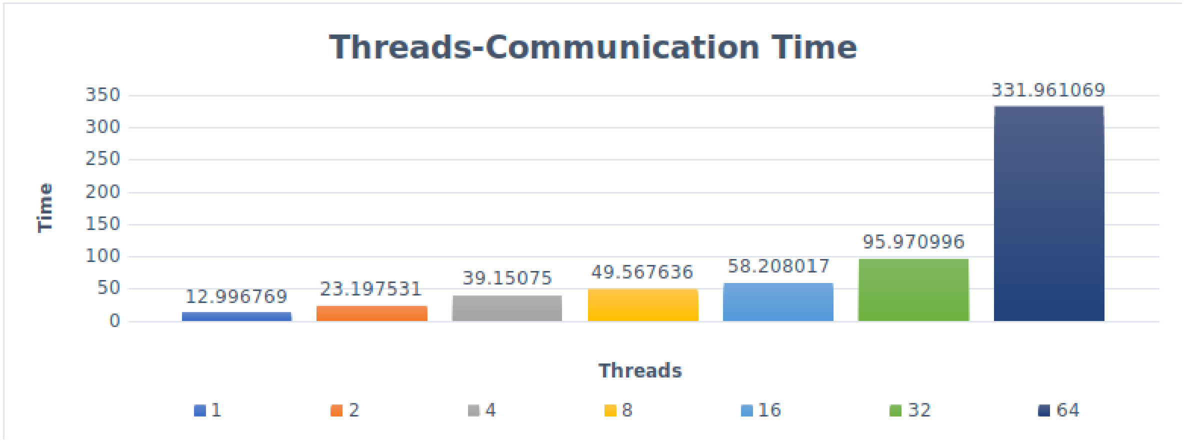
Processes	1	2	4	8	16	32	64
Time	1.115951	1.485524	1.639549	1.667072	2.085710	3.302843	10.707934



3.1.3 Communication

Just like the exercise above as the number of processes increases the total communication time also increases since more processes must communicate with each other.

Processes	1	2	4	8	16	32	64
Time	12.996769	23.197531	39.150750	49.567636	58.208017	95.970996	331.961069



4 Contact Us

1. Στολτίδης Αλέξανδρος (2824)
2. Νικόλαος Κουτσούκης (2907)
3. Λάντζος Στέργιος (2789)