Heat Diffusion over 2D Surface Group 1 Report

Jacinta Sousa, and João Bordalo

Abstract—The present paper presents the steps followed to improve the performance of a slow sequential program for heat diffusion simulation over a 2D surface. The goal was to reduce its execution time leveraging a distributed memory architecture with message passing communication. We tried different communication patterns, primitives and number of working nodes. In the end we saw that this approach can have benefits on execution time. We found some logical optimizations to have little effect on execution time.

Index Terms—Stencil Pattern, Parallel Programming, Performance, MPI, Message Passing, Distributed Memory

1 Introduction

The provided code, simulates heat diffusion over a 2D surface and runs sequentially. As the surface considered grows, the program takes longer to execute, since the number of calculations and runs through the surface matrix that represents the heat of each point in a certain time step rises. With that problem at hand, the approach planned to solve it consists of taking advantage of the possibility to parallelize the computations using multiple working nodes communicating via message passing over a distributed memory architecture.

2 Methods

We worked from the original C program, using it as a baseline. The method employed followed an identify-change-test-benchmark policy. To this end, we would identify a possible problem with our current version or possible optimization and make a fixing change. After changing the implementation, we assured correctness of the program by comparing its output with that of the original version. To assess the effectiveness of the changes made to the program, we benchmarked and profiled it, to obtain average execution times.

3 Work Distribution and Communication

To distribute work, we split the matrix, so that each working node gets a fraction of its rows. At the beginning, each node calculates how many rows it will have to compute.

In our implementation, there are two communication points:

- Border Sharing - Upon performing one step of computation, the nodes need to communicate among each other to share the borders of their fraction of the matrix. The last row that node n does needs to be shared with node $n+1$, since $n+1$ needs to read it to complete its computations. The same applies to the first row $n+1$ computes, which must be shared with node n. In sum, the communication in this step consists of each node sending one row to its previous and next nodes, if existent.
- Outputting the Matrix - When we wish to output the matrix to a file, each node must send the part of the matrix it computed to the master, for writing and outputting the final result.

We implemented two types of communication pattern.

In the first implementation, every node does an equal amount of work, with node 0 doing any extra row if the matrix size isn't divisible by the number of workers. Node 0 is also responsible for all the outputs and for writing the matrix to the output file.

We also experimented with a master-worker approach, where node 0 functions as the master and doesn't do any work. It is still responsible for outputting the matrix and other useful information. In this approach, node 1 performs the extra computations if needed.

All results can be found in section 4 and were obtained with the same program parameters, $nx = ny = 200$ and $h = 0.005$.

4 Experiments

The original C program, which we call the *sequential* version is used as a baseline.

Our first implementation evenly divided the work by all the nodes, including node 0. To communicate, it used the primitives `MPI_Send` and `MPI_Recv` for both border sharing and intermediate results communication. We experimented with different communication primitives and patterns. First, we changed the communication of intermediate results from the `MPI_Send` and `MPI_Recv` primitives to `MPI_Gather`. This improved the code but didn't seem to have any impact on performance. For this reason, we chose not to benchmark the `MPI_Send` and `MPI_Recv` version.

If the matrix doesn't divide evenly between all workers, we have node 0 perform some extra computations. `MPI_Gather` doesn't allow for different nodes to send different amounts of data so, to tackle this issue, we changed `MPI_Gather` to `MPI_Gatherv`. With this communication

- *Jacinta Sousa, 55075, and João Bordalo, 55697, are Master's students in the Department of Computer Science, NOVA School of Science and Technology, Lisbon.*

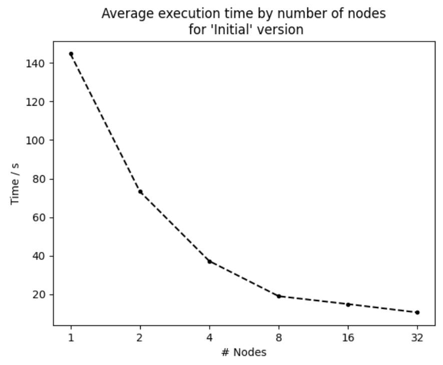

Fig. 1. Average execution times per number of nodes for our initial version, where every node performs computations.

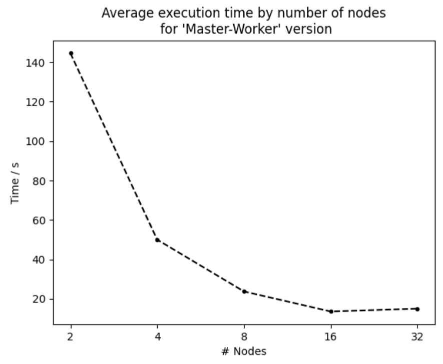

Fig. 2. Average execution times per number of nodes for the master-worker pattern, where $n-1$ nodes perform computations.

pattern, we don't need node 0 to send the information since it is the master and already has the data it computed in memory. The gather primitives don't allow this so we instead work around by setting the amount of sent and received data to 0. This is only possible with the use of `MPI_Gatherv` so we changed to that.

At this point we're ready to benchmark this version as our initial version.

Figure 1 shows how the average execution time varies with an increasing number of nodes. We can see that doubling the number of nodes reduces execution time by almost half with some overhead due to communication.

4.1 Master-Worker

For our Master-Worker implementation, node 0 acts as the master and doesn't perform any computations, merely handling output. The communication primitive for sharing the computation results with the master is still `MPI_Gatherv`. In this approach, node 1, the first worker, is the one to perform the extra computations when needed. We benchmarked this version to see if there would be any difference between the two communication patterns.

Figure 2 shows how the average execution time varies with an increasing number of nodes for the Master-Worker pattern. We can see that, although increasing the number of nodes tends to reduce execution time, there's an increase in execution time when comparing 16 to 32 nodes.

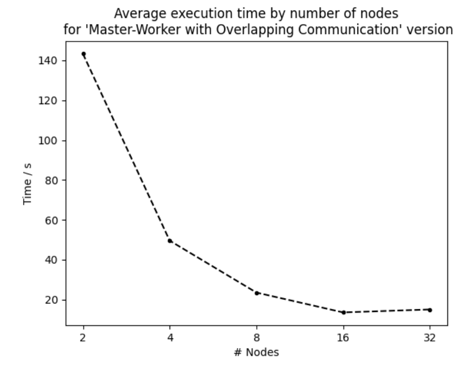

Fig. 3. Average execution times per number of nodes for the master-worker pattern with overlapping computation and communication, where $n-1$ nodes perform computations.

4.2 Overlapping Communication and Computation

With the Master-Worker pattern, there's a clear separation between the nodes which are performing computations and the master which is outputting the file to disk. So as to not delay computation while outputting the matrix to a file, we experimented with overlapping communication and computation. To achieve this, we used the `MPI_Igatherv` primitive, which is non-blocking. To ensure correctness, we called `MPI_Wait` on the master node before writing the file and on the worker nodes before overwriting the positions of memory which could still be sending. We were careful to manage the pointers in such a way as to allow for this asynchronicity.

Figure 3 shows how the average execution time varies with an increasing number of nodes for the Master-Worker pattern with Overlapping Communication and Computation. It follows the same pattern as in Figure 2, decreasing initially but increasing when jumping from 16 to 32 nodes.

4.3 Discussion

The results from figure 1 aren't directly comparable to figures 2 and 3. To compare results we would need to compare a result with n nodes on figure 1 with one with $n+1$ nodes in figures 2 and 3, since the master doesn't do work.

We can see that the two communication patterns perform similarly, with no significant difference between execution times. However, by looking at the results in table 1, we can see that, although the Master-Worker pattern uses only 15 worker nodes, it actually obtained better results than the initial version with 16 worker nodes. During small empirical tests, where we actually matched the number of nodes between the two versions, we saw no difference that was relevant to report.

This was according to our expectations.

TABLE 1
Execution times in seconds for Initial implementation and Master-Worker.

	Mean	Max	Min
Initial version with 16 nodes	14.971	15.099	14.922
Master-Worker with 16 nodes	**13.564**	13.673	13.478

The overlapping, shown in section 4.2, also seemed to have little to no effect on execution times when compared to the version without overlapping. As seen in table 2, the version with overlapping obtained roughly the same execution times. This was not according to our expectations. We expected to see at least a slight decrease in execution times, since the file could be outputted at the same time as other computations were being performed.

TABLE 2
Execution times in seconds for Master-Worker with and without overlapping.

	Mean	Max	Min
Master-Worker with 16 nodes	13.564	13.673	13.478
Master-Worker with 16 nodes and Overlapping Communication and Computation	**13.563**	13.715	13.509

For reference, table 3 shows all the obtained results.

5 Conclusion

By analyzing the results, one can see that distributing the work between various nodes has benefits and can improve the execution time of sequential programs. It shows good efficiency by roughly halving the execution time when doubling the number of nodes, with some overhead due to communication.

We didn't find optimizations to have much effect on execution times. As can be seen in section 4, the plots for execution times are very similar.

Our fastest version was our initial version, using `MPI_Gatherv` and scaled up to 32 nodes. We had a final speedup of $S = \frac{144.746}{10.647} = 13.595$.

Some possible further optimizations include:

1) Reducing the amount of memory each node needs, a problem which we didn't tackle since it wouldn't improve computation time
2) Scaling up the problem size and benchmarking it at scale

TABLE 3
Execution times in seconds for every version implemented.

	Mean	Max	Min
Sequential	144.746	144.932	144.515
Initial version			
1 node	144.703	144.793	144.516
2 nodes	73.351	73.492	73.252
4 nodes	37.295	37.400	37.220
8 nodes	19.061	19.116	18.983
16 nodes	14.971	15.099	14.922
32 nodes	**10.647**	10.864	10.554
Master-Worker			
2 nodes	144.580	144.693	144.479
4 nodes	50.009	50.121	49.845
8 nodes	23.737	23.786	23.699
16 nodes	13.564	13.673	13.478
32 nodes	15.016	15.134	14.869
Master-Worker with Overlapping			
2 nodes	143.285	143.374	143.155
4 nodes	49.560	49.723	49.468
8 nodes	23.520	23.570	23.429
16 nodes	13.563	13.715	13.509
32 nodes	15.058	15.197	14.941

References

[1] HPC Tutorials. Retrieved December 2022, from https://hpc-tutorials.llnl.gov/
[2] Open MPI v4.1.4 documentation. Retrieved December 2022, from https://www.open-mpi.org/doc/current/