93698 – David Duque
96201 – Fábio Dias
96219 – Gonçalo Midões
Group 12
Professor Manuel Ribeiro

MPI Parallelization

Parallel and Distributed Computing
April 14, 2023

I. Parallelization Approach

The TSP program was parallelized using MPI, a widely used programming model for parallel computing that enables efficient communication between different processors in a distributed system.

To achieve parallelism, a task decomposition approach was employed. Each processor handles a set of different nodes, where each node represents a possible tour of the TSP problem, completed or to be completed. This decomposition allows processors to work on different parts of the TSP problem in parallel, potentially reducing the overall runtime of the program.

Additionally, dynamic load balancing was achieved during program execution, allowing finished processors to receive tasks reassigned from other processors without restarting the entire computation or remaining idle for too long.

After an initial implementation, it was quickly determined that having all processors inform each other is not scalable. As the number of processors increases, communication also increases, resulting in slower programs even when using asynchronous communications.

Based on these findings, a second approach was adopted using a token mechanism. The token moves along all processors as if they are in a ring, holding information about stopped processors and the best cost found so far. This mechanism ensures good scalability, with only two communications required per processor: one incoming from the back and one outgoing to the next processor in the ring.

In the final implementation, the program also randomly redirects the top of its queue after a certain number of Pops. This feature ensures that no lousy search tree grows too large and that every processor actively contributes to finding the better solution.

II. Software Implementation

In this version of the project, a preamble was added similar to the OpenMP version, where the neighbors of the root of the TSP problem (node 0) were searched based on the rank of the task (Id_{tasks}) and the total number of tasks (N_{tasks}). Furthermore, there are no shared variables except for the traveling token. Additionally, task 0 is responsible for initializing the token ring and finalizing all task execution. The preamble algorithm can be seen in Alg. 1.

After the preamble, tasks enter an infinite while loop that breaks only when they receive a token with a specific ending flag. Message-receiving is always performed synchronously after a positive MPI_Iprobe test. This implementation enables tasks to check for messages to receive in a non-blocking way and reduces the time spent blocked when receiving them. This process can be seen in Alg. 2.

Algorithm 1 TSP Branch and Bound MPI Version - Preamble.

```
1:  procedure TSPBB($Distances, N, BestTourCost, Id_{tasks}, N_{tasks}$)
2:      for $city = Id_{tasks}$ to $N - 1$ do          ▷ $city$ is incremented by $N_{tasks}$ each loop
3:          if $city$ is neighbor of 0 then
4:              (...)
5:              $queue.push(new)$
6:          end if
7:      end for
8:      if $Id_{tasks} = 0$ then
9:          MPI_Send($token$)          ▷ Sent to next $Id_{tasks}$, in this case $Id_{tasks} = 1$
10:     end if
11:     (...)
12: end procedure
```

Sending the token is also being performed synchronously. Aside from the probing mechanism, the node sending is the only asynchronous communication. Node sending occurs in two cases: when the number of Pops reaches a certain threshold and when selecting a stopped task from the token to redirect some work.

The threshold for sending a node to a random task was empirically determined. With fewer parallel tasks, this scheme introduces significant overhead if done too frequently. However, without this mechanism, the search trees may diverge to non-relevant problems in large-scale problems simply because it takes a long time to find the first solution and start pruning. The threshold values settled in 20000 Pops for $N_{tasks} \in]1; 15]$ and 7500 for $N_{tasks} \in [16; +\infty]$. A more in-depth analysis of the results will provide a better explanation for these values.

Algorithm 2 TSP Branch and Bound Parallel Version - Message handling.

```
1:  procedure TSPBB($Distances, N, BestTourCost, Id_{tasks}, N_{tasks}$)
2:      (...)
3:      while $true$ do
4:          $flag \leftarrow$ MPI_Iprobe($status$)
5:          if $flag = true$ then
6:              if $status$.MPI_TAG $= token$ then
7:                  MPI_Recv($token$)
8:                  $token.read(PausedIndex, OverallBestTourCost, StoppingTag)$
9:                  $token.write(Paused, BestTourCost, StoppingTag)$          ▷ Only update if need
10:                 MPI_Send($token$)          ▷ Sent to next $Id_{tasks}$
11:             else if $status$.MPI_TAG $= node$ then
12:                 MPI_Recv($node$)
13:                 $queue.push(node)$
14:             end if
15:             break if $StoppingTag$ is activated
16:         end if
17:         (...)
18:     end while
19:     return BestTour, BestTourCost
20: end procedure
```

Like the OpenMP version, load balancing is addressed by sending nodes to tasks that have stopped their execution. As previously mentioned, information about the stopped tasks is circulated through the ring. When a working task receives the token, it checks for stopped tasks. If the active task has not already sent nodes to a given stopped task, it stores its index, changes the stopped flag, and proceeds to send the token. This system allows a working task to send nodes progressively to all stopped tasks.

Additionally, it is worth noting that in both cases mentioned, the top node of the sending task queue is the one that is sent. This ensures a more proactive search throughout the possible solution tree, leaving less promising search paths behind.

To conclude the program execution, a comparison is performed to ensure that the output displays the solution with the lowest cost. This conclusion is achieved by collecting the final results from all running processes using the MPI_Allgather function. Then, the process with the lowest cost is determined by comparing the results, and only the process with the lowest cost prints the final solution. If there are two tasks with a possible best solution, only the task with the lower rank prints the solution, ensuring that only one answer is printed.

III. Results

The results obtained from the experiment can be observed in Figure 1. The data shows that there was an overall improvement in the code's speed when compared to the serial version. However, the increase in the number of processing units did not result in a linear improvement in test times. This is mainly due to the complexity of the problem not benefiting greatly from the added processing units. In fact, the performance even decreased when more processes were added, as the increased communication between processes led to a higher communication overhead without significant benefits.

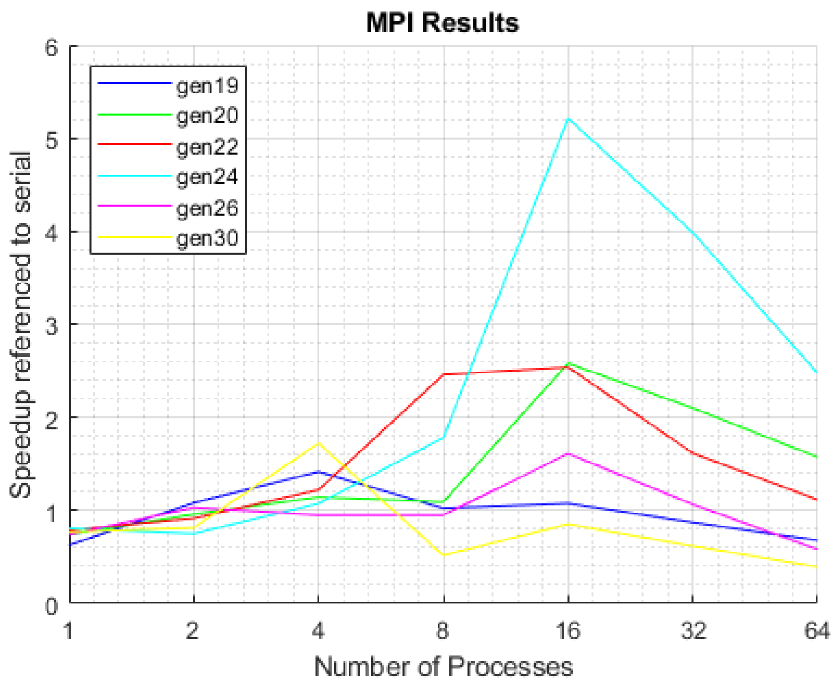

Fig. 1 – Results for the MPI implementation.

As mentioned earlier, random node sending can help balance workloads but can also introduce significant overhead due to excessive communication. Thus, for a high number of tasks, the advantages of randomly sending nodes become less effective, and the implementation can resemble the scalability issues of the initial version of the project. As shown in Figure 1, this strategy has improved every test from 8 to 16 processes, but a decrease in performance is observed for a higher number of tasks.

It is worth noting that, except for test $gen19$, the larger problems ($gen26$ and $gen30$) have the worst performance. This issue can be attributed to the fact that, for these problems, the number of nodes created is $\mathcal{O}(N!)$ if every city is connected to every other city. As N increases, this number grows significantly, which can result in the problem taking too much time to find a good answer or behaving like a breadth-first search algorithm.

A possible improvement could be achieved by determining the optimal number of Pops after which a node should be sent randomly. Ideally, this study would result in an expression that is a function of both N and N_{tasks}. Due to time limitations, the number of node sending, as explained in Section II., was used statically.

Upon analyzing the values in Table 1, it is evident that the characteristic follows a semi-Gaussian shape, peaking at 16 processes.

Tab. 1 – Speedup analysis.

Speedup by thread	1	2	4	8	16	32	64
Max	0.8077	1.0828	1.7221	2.4583	5.2116	3.9850	2.473
Mean	0.7466	0.9262	1.2548	1.3046	2.3118	1.7094	1.1366
Min	0.6296	0.7523	0.9507	0.5191	0.8533	0.6183	0.3951

The MPI implementation did not surpass the average performance of the OpenMP implementation. However, the analysis of the test results indicated possibilities for improvement. In conclusion, although the results were not optimal, there is still potential for enhancing the MPI implementation and exploring the benefits of a mixed implementation utilizing both OpenMP and MPI.