MPI implementation of TSP

Group 15
93230 Catarina Bento
95600 João Pereira
95618 Leonor Barreiros

1 Approach used for parallelization

Our program begins with a sequential part (redundant computation), during which every task expands the search tree until there are 100000 nodes in their initial priority queue. If the problem is small enough, it will ultimately be solved during this part. Then, the nodes are distributed interleaving between the tasks. If the nodes weren't distributed in an interleaving manner, the first priority queue would have all nodes with the lowest lower-bound, which would automatically give the other threads worse sub-problems to explore. Then, in a for loop, each task verifies if the nodes in their initial priority queue are to be processed by them, and if not, the node is freed. So, each task will then begin the parallel part of the program with $100000/n\_tasks$ nodes (primitive tasks) in their new priority queue.

Note that we decided to opt for a redundant part, instead of having only one process compute this part and distribute nodes between the others, because we concluded that the redundant computation time was quicker than the communication time.

After that, we enter the parallel phase of our program. Each task is responsible for exploring its own part of the tree; however, when a task finishes exploring its own nodes, they send a message to the other tasks, signaling that its queue is empty. Then, when a task that still has nodes receives that message, it sends some nodes to the empty queue task. Finally, for every 500000 iterations, the tasks execute an asynchronous AllReduce regarding the best tour cost value.

The parallel part ends when all tasks are idle. After that, each task will send to the rank 0 task their best tour cost and best tour.

2 Decomposition used

Our decomposition consists of dividing the search problem between the tasks, i.e., each task will explore a sub-tree. This is ensured by the sequential part of the code, where each task assigns themselves nodes in an interleaving manner. More concretely, our decomposition consists of assigning the processing of each node to a different task.

However, if a task's queue is empty, it will request nodes from the other threads, in order to keep working.

3 Synchronization concerns (and why)

Our solution runs without any explicit synchronization. All collective communications are asynchronous (AllReduce is actually Iallreduce) and we simply check periodically if they've been completed.

As for point-to-point communication, SENDing is synchronous because the sender only blocks itself until the buffer has been sent to the network or the internal buffer (so that doesn't imply any synchronization). Although we use the synchronous Recv, we only do so after an asynchronous Iprobe, so that also ends up not using any explicit synchronization.

In summary, that means our processes run independently of each other, without needing to wait (explicitly) for each other.

Note that that's not the case in the end, when every process communicates its solution to process with $rank0$. There, all processes will have to have terminated their search for the solution to be shared. However, this is a very small fraction of the program.

4 Load Balancing

Our load balancing strategy has 2 components: one that evens out the cost of all process's sub-problems (so that no one is exploring bad solutions); and one that allows for work-sharing so that no process is idle for long while others still have work.

Periodically, processes perform an AllReduce on their local *best_tour_cost*. This ensures that all processes are searching similarly interesting parts of the tree, since they will clear all nodes that exceed the goal *best_tour_cost*.

Also, when a process becomes idle, it sends an IDLE message to all other processes. Each process that receives an IDLE message from another process (and has sufficient nodes) will send a WORK message containing 1 node.

5 Performance results (vs expected)

The main goals of a distributed memory application are to improve performance and scalability. However, these can be conditioned by several factors including the number of processors used, the input problem's size, and the implementation's efficiency.

Regarding our implementation, although we achieved very satisfactory results until $n\_tasks = 16$, we never achieved the desired linear (or even super-linear, since this is a search problem) speedup. Furthermore, when analyzing the speedup values obtained (Table 2) we can see how the input problem's size has a negative impact on the smaller tests, leading to significantly slower execution times. This can be easily explained by the big overhead that is introduced by the communication and synchronization (essentially in the final stage of the algorithm where all processes perform blocking sends to send their results to the process with rank 0 and this process must block until it receives messages from all other processes) between processes.

#processes	ex1	ex2	gen10	gen15	gen19	gen20	gen22	gen24	gen26	gen30
1	0.0	0.0	0.0	0.2	2.1	112.8	173.0	80.2	81.2	-
2	0.0	0.0	0.0	0.2	1.2	72.3	87.0	39.2	45.9	129.3
4	0.0	0.0	0.0	0.2	0.8	40.4	46.8	32.4	25.3	81.9
8	0.0	0.0	0.0	0.5	0.8	23.1	25.0	15.5	14.5	35.6
16	0.0	0.0	0.0	1.5	1.7	13.9	13.5	30.1	10.4	23.4
32	0.0	0.0	0.0	19.1	4.3	17.4	11.9	24.3	11.8	19.8
64	0.0	0.0	0.0	18.7	18.6	29.5	28.0	27.5	57.4	55.2

Table 1: Execution times when running MPI version on input data (for 1, 2, 4, 8, 16, 32, and, 64 processes).

#processes	ex1	ex2	gen10	gen15	gen19	gen20	gen22	gen24	gen26	gen30
4	1	1	1	0.5	1.88	1.98	2.38	1.63	2.21	2.30
8	1	1	1	0.2	1.88	3.46	4.45	3.41	3.85	5.30
16	1	1	1	0.07	0.88	5.76	8.24	1.76	5.37	8.06
32	1	1	1	0.01	0.35	4.60	9.35	2.18	4.73	9.53

Table 2: Speedup values for the MPI version when comparing with the serial one (for 4, 8, 16, and, 32).

We calculated our implementation efficiency using the following expression:

$$\text{Efficiency} = \frac{\text{Sequential Time}}{\text{Processors Used} \times \text{Parallel Time}} = \frac{\text{Speedup}}{\text{Processors Used}}$$

The results can be found in Table 3. As expected, the efficiency decreases with the number of processes. This is because, as p increases, the communication and redundant computation overhead also increases.

#processes	ex1	ex2	gen10	gen15	gen19	gen20	gen22	gen24	gen26	gen30
4	0.25	0.25	0.25	0.13	0.47	0.50	0.59	0.41	0.55	0.58
8	0.13	0.13	0.13	0.03	0.23	0.43	0.56	0.43	0.48	0.66
16	0.06	0.06	0.06	0.00	0.06	0.36	0.52	0.11	0.34	0.50
32	0.03	0.03	0.03	0.00	0.01	0.14	0.29	0.07	0.15	0.30

Table 3: Efficiency values for the MPI version when comparing with the serial one (for 4, 8, 16, and, 32).

The scalability value is given by the following function:

$$\frac{\mathrm{M}(f(p))}{p}$$

In our case:

$$\frac{\mathrm{M}(C \times p^2 \times n^{(1-n)})}{p} = Cp^3$$

Our program has poor scalability: which is visible experimentally as well as by the scalability function. Furthermore, the value obtained means that memory usage per processor must be cubed (cubic scalability) to maintain efficiency.