Parallel and Distributed Computing

2022/2023

OpenMP Implementation

Group 45

João Gonçalves 85211 Luis Maçorano 96265 Miguel Albuquerque 105828

Approach used for parallelization:

The parallelization focus was solely on the *procedure* function as parallelization for other parts would have a low impact. Since the *weight matrix* and the *min1* and *min2* calculations amount to an insignificant impact on the computational times there was no point in parallelizing them.

Our first approach was to compute every neighbour of the first node and add them to a queue, the *queue_master*, that was shared between threads. Then each thread would fetch a node from this queue, and upon finishing its computations the threads would fetch the remaining nodes of the *queue_master* until it was empty. This revealed to be a very bad implementation, with times getting even slower, since we would be computing all the branches of the node, ignoring the remaining ones still on the *queue_master*. Then, when fetching new nodes from the queue master the computation would have to start from the beginning.

We then decided to divide the nodes between the threads until the *queue_master* was empty before starting the computations. This implementation, while better, still wasn't satisfactory. Finally we upgraded this implementation and, on certain predefined times of the computation, the threads would sync, and all of them would focus on the queue from the node that had the lowest lower_bound.

Decomposition used:

Firstly the nodes were divided equally between threads. Then, when the synchronization was reached, we inserted nodes from the queue of the node with the lowest bound on the queues of the other threads.

We start by discovering which thread had the node with the lowest bound. Then, this thread will be responsible for inserting a node from its queue on each of the other threads' queues. This way all the threads will be performing computations on the region of the tree where the lowest bound was reached, and will allow our implementation to reach a good solution sooner and avoid performing unnecessary computations, eliminating branches with bigger lower bounds.

We later reached the conclusion that performing this synchronization multiple times during our computation would greatly improve our results.

Synchronization concerns:

To avoid racing between threads we inserted critical regions and atomic operations when performing computations on variables that were shared between the threads. Furthermore, when it was necessary to synchronize we had to guarantee that all the threads were at the same stage of the computation. This was achieved through two methods. Firstly we added barriers, to ensure the threads waited for the other to finish their computation. Secondly we added a flag that would start the synchronization when all the threads visited a specific number of nodes. When testing we found out that the optimal value to perform this synchronization was after every 1000 computations. When one of the nodes reached the end of its queue we stopped making this synchronization since the nodes would be stuck on the barriers inserted waiting for the node that had already finished.

Load balancing:

As stated before, to obtain a good balance between the threads we start by dividing them equally. Later, for each synchronization we would insert the optimal nodes in all threads to guarantee that all of them are focused on the right area of the tree and we perform this synchronization multiple times (every 1000 computations) to ensure we reach the end of the queues as fast as we can.

To improve our load balancing, when a thread finished its queue, instead of stopping the synchronization, we could assign it nodes from the queues that are still active to avoid it being stall. This didn't have a major impact on our results, since when a thread finished its queue we were already close to finding the optimal solution.

Another aspect that helped us balance the work between threads was the shared flag to ensure that all threads started the synchronization at the same time, this way avoiding threads being stall waiting for the other to reach the synchronization stage.

Performance results:

The results obtained are from lab3 pc3 and are the mean of 3 runs in each execution.

	Serial		OMP (2 thrds)	OMP (4 thrds)	OMP (8 thrds)	Speed-up (2 thrds)	Speed-up (4 thrds)	Speed-up (8 thrds)
	Old	Optimized						
Gen20-5000	94.1	54.2	31.4	18.3	12.1	1.73	2.97	4.48
Gen22-25000	131.5	105	56.6	35.8	21.5	1.86	2.93	4.88
Gen24-50000	68.4	24.9	16.7	8.8	6	1.49	2.83	4.15
Gen26-50000	55.4	21.7	9.3	4.1	3.4	2.33	5.29	6.38
Gen30-5000	199.3	59.3	33.1	18.5	11.2	1.79	3.21	5.29

Table 1 – Performance results in seconds on PC-3 of Lab-3

Analysing the results, we can conclude that the implementation was very successful, with speedups very close to the theoretical limit. We noticed that our parallel implementation with one thread, i.e. performed serially, far exceeded the results from the previous serial implementation. Our speedups are then calculated using the results from the optimized serial implementation.

The fact that the results for 8 threads are slightly lower can be explained by the fact that the computer may not have enough threads available to perform the computation.

Despite having great results we can notice that the gen26 speedups are not realistic. This is due to the fact that some bug in our code made it not able to reach the best solution and end the computation sooner than expected. Still, the solution obtained was very close to the desired one and we can see that the parallelization results improve in that way that was expected.