Nishant Saurabh

Parallel Programming Practicals
Java Assignment

Vrije University, Amsterdam
Student Id : 2205542
VU Id : nsh210

Goal of Assignment :

The assignment was to write a parallel implementation for **15 puzzle game** to determine the shortest number of slides possible, and also showing the number of ways to solve the puzzle using **Ibis** system on **DAS4**.

Introduction to Ibis :

The Ibis and Ibis portability layer library has many examples inside the documentation, which gave us initial insights about working of Ibis API. The master - worker communication algorithm is an interesting one to understand the asynchronous behavior of API's , which we also used in our parallel implementation.

It is important to know, when an ibis instance is created, it joins ibis and completes job. Further same ibis instance is elected as server, which in turn blocks the program execution. Now in this case, master should always have information about the total number of ibis instances which has joined ibis pool. Ibis portability layer provides us with an API, **getPoolSize()**, that allows master to know, how many Ibis instances or workers have joined the pool. After the work is done, master should send an upcall to all ibis instances that were in the pool. We also tried to learn about registry based Ibis functions like **joinedIbises()** in order to join the pool. There were some other registry upcalls to learn about, for eg: **left()**, and **died()**. Although we didn't use all these but it was interesting to learn about them. This assignment also gave the opportunity to learn about the events flag of ipl-server, so as to have good understanding about the scenarios related to about the working of ipl instances.

Initial Approach : Task Distribution Process :

In master - worker scenario, using bound based work distribution, master will compute board with initial distance as bound and distribute it to workers. Further, the boards are generated with fixed bound and is distributed to different workers, and when they are finished with the execution, results are obtained from the workers . It works in the pattern as shown in the figure below. The master computes distance on randomly generated board sets it as initial bound and generates boards to be distributed to all the workers. The workers in turn follows the depth first search algorithm to generate more boards and prunes those boards, whose distance exceeds the bound. As shown in figure 1, after the Master distributes the task to workers, the workers further generate more children, in form of different boards. If we consider, figure 1 below, Master distributes the task to three workers with generated boards with a fixed bound. Further workers generate the different boards . In the absence of any worker, Master will compute all the 5 works generated by itself, but in case of parallelizing it, workers receive boards from the master and recursively generate more boards till the result is obtained or there are no new boards, with distance less than bound. If all three workers couldnt find the solution, the bound is increased by the master and the process is repeated until the solution is found. We distribute the task in a way that, Master has to do less work and the workers won't be idle.

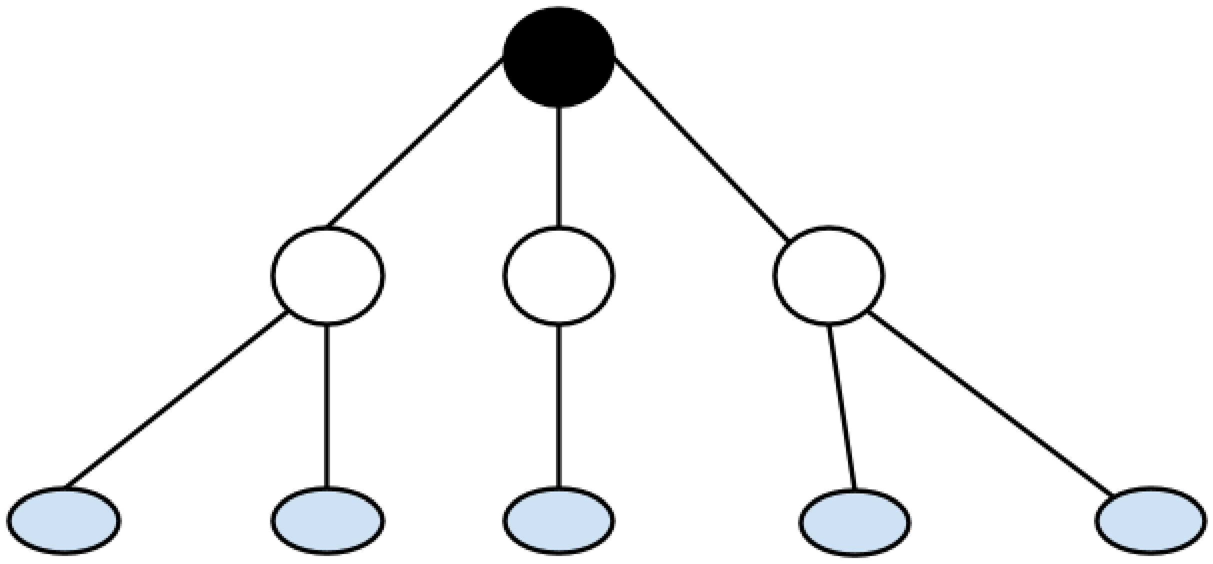

Figure 1 : Task Distribution

Final Approach :

Analysed Reason for the Lower performance in Initial Approach:

In the initial approach the reason for lower performance, was basically due to two reasons:
1. There was a barrier in the algorithm, which was depth first search based, such that Master waits for all the workers to compute task. As majority of workers will be waiting at the barrier for the slowest worker to complete its computation, hence there was imbalance in task distribution.

2. On measuring communication time, we found out that ratio of computation to communication time is almost 1:3 , Hence somehow we needed to devise a way to reduce communication between master and workers.

IMPLEMENTATION - Final Approach

Master has to generate tasks and give it to workers in master-worker scenario. Here master has a board with initial distance. In earlier approach, we did reshuffle the initial board in four ways(upwards, downwards , rightwards and leftwards) and created four tasks. It was creating number of tasks as pool size and give it to workers. Once the workers were finished computation, and if they didnt find solution, they would return back to master and ask for more work. This process continued for a lot number of times until the solution is found and communication time increased.

Hence we took the approach of creating and assigning more boards at a time by master to workers, not depending just on pool size but on basis of depth level and number of jobs. Firstly, we define the depth level till which master would create the boards. The depth 1 is creating four boards from initial board in four ways. Similarly at depth level 2 , out of those boards, 32 more boards will be created. Once master is done creating boards till a certain level, we sorted each board and on basis of distance and stored it inside array list. We did this, as boards with less distance have more probability of finding a solution.

Now, master would give more number of boards to workers at once, which can be done by assigning number of jobs to be given at once. The problem here is that, since the boards are in sorted order of their distance , there is a probability that master might end up assigning boards with less distance only to some workers and bigger distance boards only to another set of workers. This would create imbalance in task distribution. So we decide to distribute the tasks as such that each worker gets somehow similar ratio of less and larger distance boards.

Lets say we have pool size of 4, and each worker has to get 9 jobs at once. Suppose master has created boards till depth level 2, creating 36 boards. Then task distribution has to be in such a way that

1st worker would get boards (1,5,9,13,17,21,25,29,33) at once. Similarly other workers would be assigned 9 tasks or boards each. After they compute, the result , they would return back to master . if solution is found then, master will send a message to all workers to terminate.

In experiments and results section, we show the speedup and execution time obtained for initial version and further the analysis of improvement in performance in accordance to different depth levels and some larger problems.

IPL Communication features used for parallel Implementation :

The parallel based implementation of 15 puzzle game using ibis, was a tree based paradigm, hence master-worker approach of ibis seems most feasible to implement.

In IPL, Communication is established with respect to ReceivePorts and SendPorts of a certain port type. The significant issue is that receive ports and send ports should have same port type to communicate. We implemented master - worker algorithm, using explicit receive and upcalls for communication, between different processor nodes. Ibis instances can register callback function, using upcalls, which is further used by Ipl to collect message requests. The server is enabled to execute different request messages without waiting for the previous requests. The function handling the upcall can inform the master, about the result, and master informs all the workers after the work is completely done using the same upcall. The workers upcall mechanism can also be handled by the same function.

The important aspect is, an upcall messages are sent to all ibis instances within the pool by the server, after work is completed. But, it is to be checked, whether the upcall message was sent by the master or not. As all the ibis instances, which are acting as workers within pool, should be able to terminate their execution.

Experiments Result Analysis :

The performance of Ibis implementation is benchmarked on DAS4 VU cluster.
The table 1 below shows the execution time for Board length 3 for three type of Depth levels and number of Jobs given by master to worker. In case of depth 1/ Job 1, it can be observed that there is not much improvement in the execution time on increasing the workers. The reason being, master is creating just tasks till first depth level and only one job to be assigned to each worker. Hence worker comes back to master everytime to fetch a task, until it doesnt find the solution. This increases the communication time. As we increase the depth level for creation of number of tasks for the master, the execution time improves and is best found in the case of Depth level 6 and Number of Jobs 9. Jobs represent the number of tasks to be asssigned to

each worker at a time. The figure 1 below shows the execution time graph for different depth level and number of Jobs to be assigned as we discussed above.

Board Length = 103

Number of Processors	(Depth 1/ Job 1) Time in milliseconds	(Depth 4/Job 6) Time in milliseconds	(Depth 6/ Job 9) Time in Milliseconds
Sequential	8296	8264	8264
2	8324	7447	3467
4	8233	2640	1395
8	7728	2567	827
16	5678	2233	825

Table 1

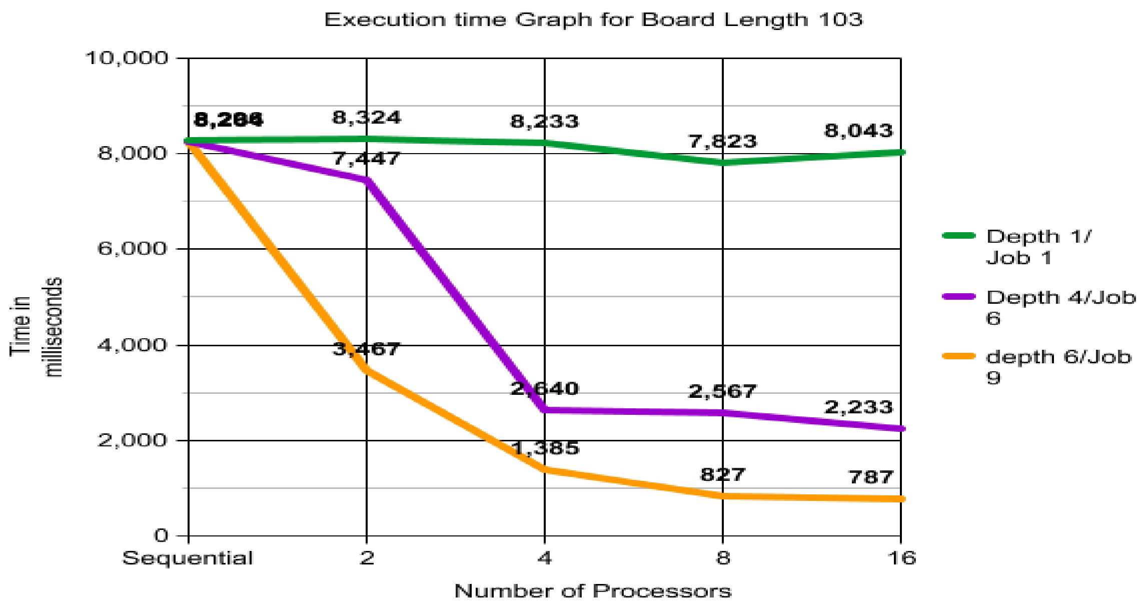

Figure 1

We also test our implementation for higher board length 105 and 107 , but just for depth level 6 and Jobs 9. As it can be observed from Table 2 and figure 3, that execution time improves considerably for board length 105 and 107 on 2,4,8 processors, but from 8 to 16, it doesnt show improvement. One of the reasons for this could be that it reaches the minimum time required to compute those board lengths.

Board Length = 105

Number of processors	(Depth 6/ Job 9) Time in milliseconds
Sequential	15811
2	10130
4	4475
8	3176
16	3113

Table 2

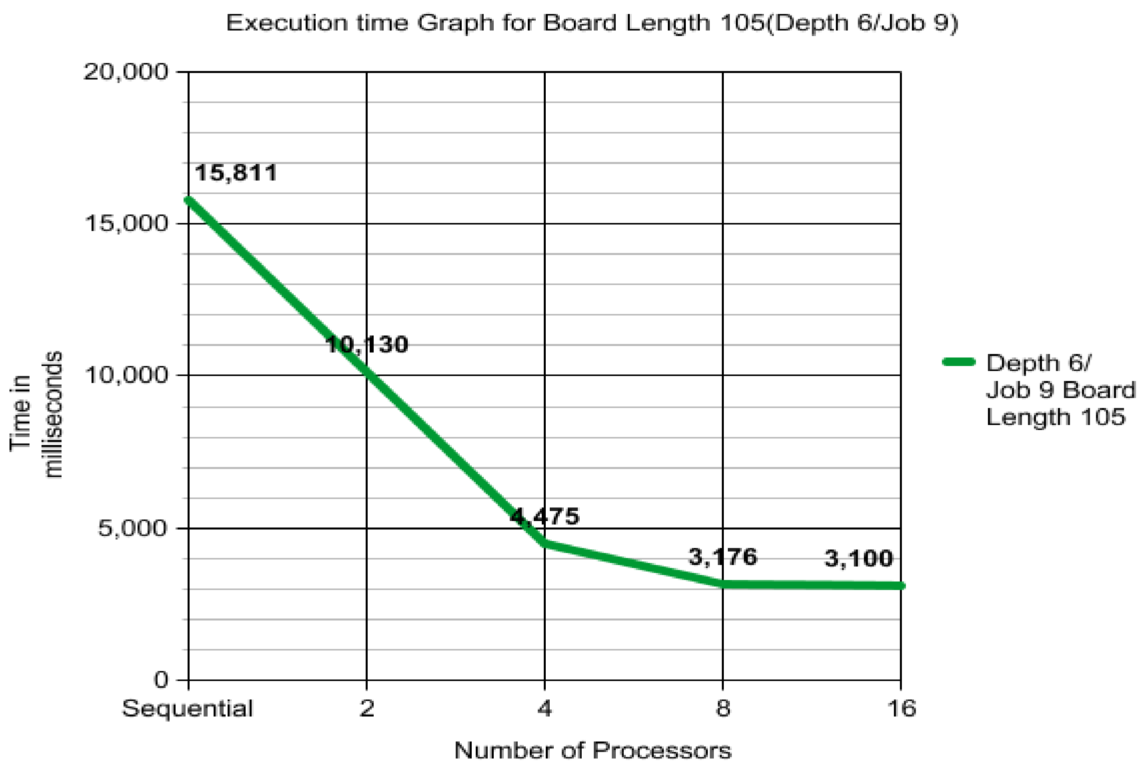

Figure 2

Similarly, we compute for Board Length 107 as well, where execution time improves considerably.

Board Length = 107

Number of processors	(Depth 6/ Job 9) Time in milliseconds
Sequential	80447
2	40969
4	19209
8	10215
16	10033

Table 3

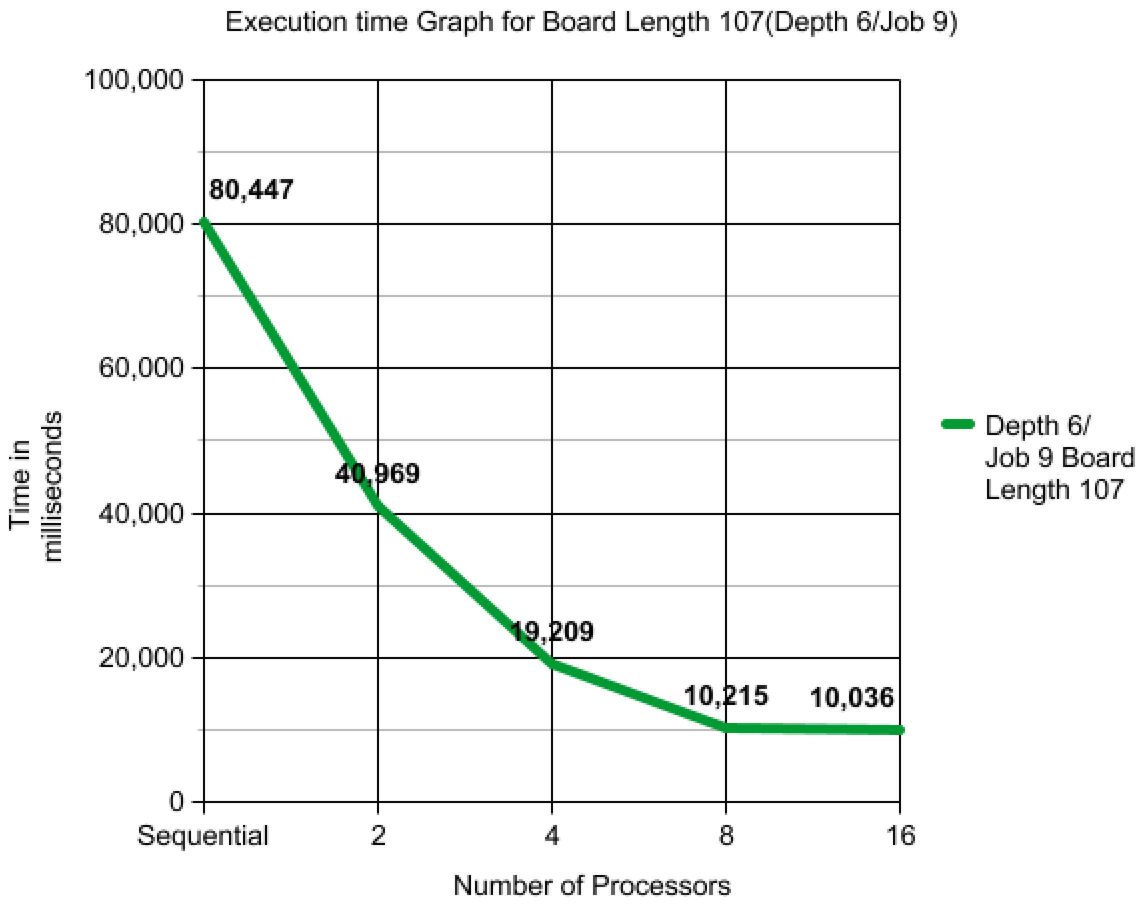

Figure 3

The table 4 shows the speedup obtained for Board lengths 103, 105 and 107. for depth level 6 and Number of Jobs assigned to workers at once as 9.The speedup graph for different board length, on 2,4 8 processors achieves considerable performance, but in case of 16 processors, it has very slight increase as compared to 16 processors. The reason for this is in case of 16 processors, for the larger problems, and on creating tasks for greater depth level and variance in number of Jobs assigned to each worker, might help to extend the performance level as evident with other number of workers.

Speedup : Depth 6/ Job 9 different board length

Number of processors	**Board Length 103**	**Board Length 105**	**Board Length 107**
2	**2.38**	**1.56**	**1.96**
4	**5.92**	**3.53**	**4.18**
8	**9.99**	**4.97**	**7.87**
16	**10.01**	**5.10**	**8.0162**

Table 4

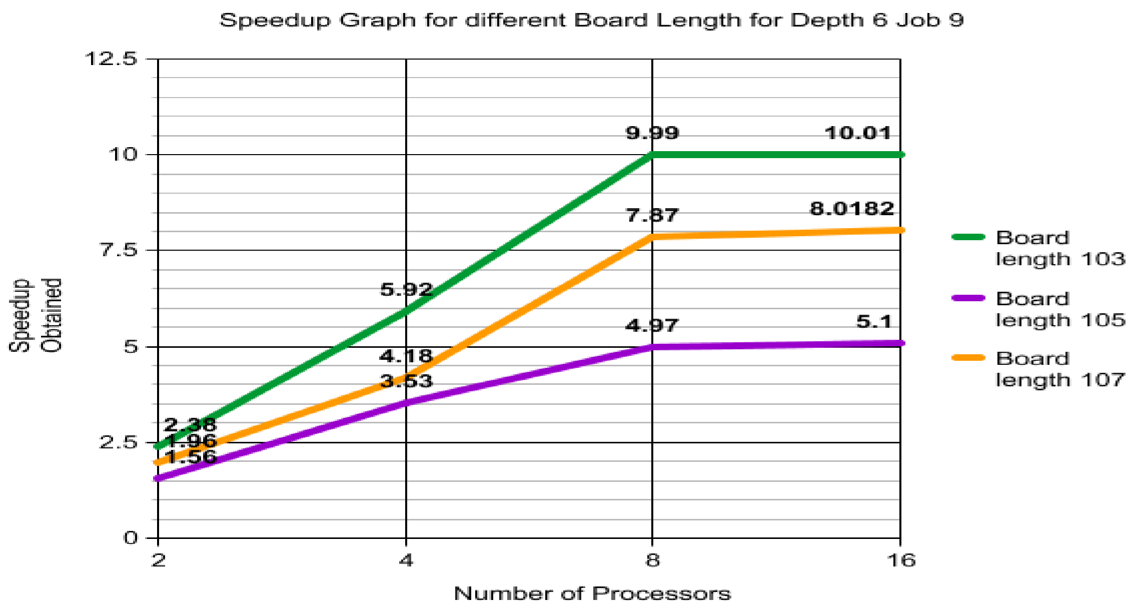

Figure 5

Conclusion :

In accordance to the results shown above, where we tried different depth levels of the board to be generated by the master and different number of Jobs assigned at once, can improve the performance level. The reason is as we discussed earlier, there are few reasons, which resulted in lower performance of earlier version :

1. The task imbalance: where every worker was not getting same intensity of tasks. hence some workers finished earlier and the barrier prevented them from getting more tasks and has to wait for the slowest workers to complete. This increased the waiting time.

2. If we give small jobs to each worker , then after executing it, they would return back to the master again and again, whic increased the execution time as ratio of communication to computation time got higher.

These all performance issues were sorted out, firstly creating more jobs, to a greater depth level, then sorting the boards according to their increasing order of their length. and finally distributing each worker more amount of job , taking care of the fact that each worker has almost similar number of smaller and bigger length boards. As we also showed results for larger problems, where considerable performance was achieved.