A parallel approach to solving the 15 puzzle

Curea Paul-Andrei

1 Abstract

The 15 puzzle is a sliding puzzle that consists of a frame of numbered square tiles in random order with one tile missing.[1]

In this paper I will discuss one method of solving 15 puzzles using IDA* and how the algorithm can be parallelized to obtain faster results as well as comparing times between the Single Process and Multi Process programs to solve some medium difficulty puzzles.

2 The machine

All experiments were run on a *i7-6700HQ* processor having *4 physical cores*, *8 threads* a *base clock speed of 2.6 GHz* and a *max turbo of 3.5 GHz*, the processor is *not overclocked.*

3 The problem

As stated above a 15 puzzle is a frame of 4 by 4 pieces numbered from 1 to 15 and one blank space, one such puzzle looks like this:

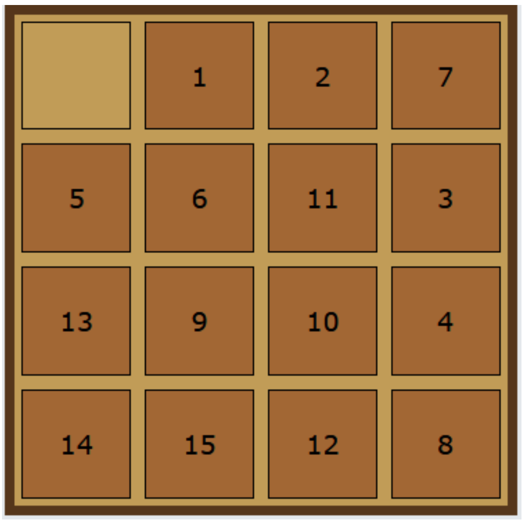


Figure 1: Example Puzzle

The purpose of the puzzle is to put all the numbers in increasing order and the empty space in the lower right corner like so:

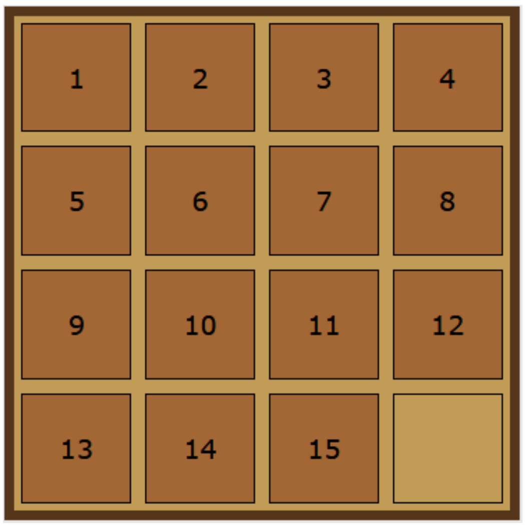


Figure 2: Solved Puzzle

This state of the board must be obtained trough successive swaps of the position of the empty space with a piece that touches it in one of the four cardinal points.

4 The Single Process program

4.1 The goal of the program

We want to solve any random given puzzle in the least amount of moves.

4.2 Some definitions

1. A neighbour of a puzzle is a puzzle that can be obtained by making a valid swap;
2. Two pieces are inverted if piece i is seen in the puzzle before piece j and $i > j$;
3. A $n * n$ puzzle is valid if for an odd n then it has an even number of inversions and for an even n the number of inversions plus the number of the row on which the empty piece is found is odd (the row counts start from 0).

4.3 Finding an algorithm to solve the problem

Since we want to solve the 15 puzzle optimally the first idea to come to mind might be to continuously generate all neighbours of all the current boards until no more new neighbours can be generated, this is a standard method for solving such a problem, to generate a complete solution and then searching trough the generated tree for the shortest path between the initial and final board. This approach is unfeasible at best given that a $n * n$ puzzle has $\frac{(n*n)!}{2}$ valid states, for example the 15 puzzle has 10,461,394,944,000 valid states making it impossible to generate a complete tree of the problem.

Seeing as generating a complete tree of the problem cannot be done the next logical step would be to try an informed search algorithm like A*, but since A* cannot be efficiently optimized in a non-shared memory multi processed program and the sheer size of the problem space would create memory problems for it we will go to IDA*.

Since we will use IDA* we also need to define a function that scores our current puzzle, for this I will use the sum of Manhattan Distances for each piece, where the Manhattan Distance for a piece is the absolute difference between the current coordinates of the piece on the board and the coordinates where the piece should be in the board in the final state.

Now that I have decided to use *IDA** as an algorithm to find the path to the goal node and the sum of Manhattan Distances, noted from here as *Manhattan Distance* or *score* of a board, as the function to determine how close a board is to the goal node we can proceed to write and test the algorithm.

4.4 Making the algorithm parallel

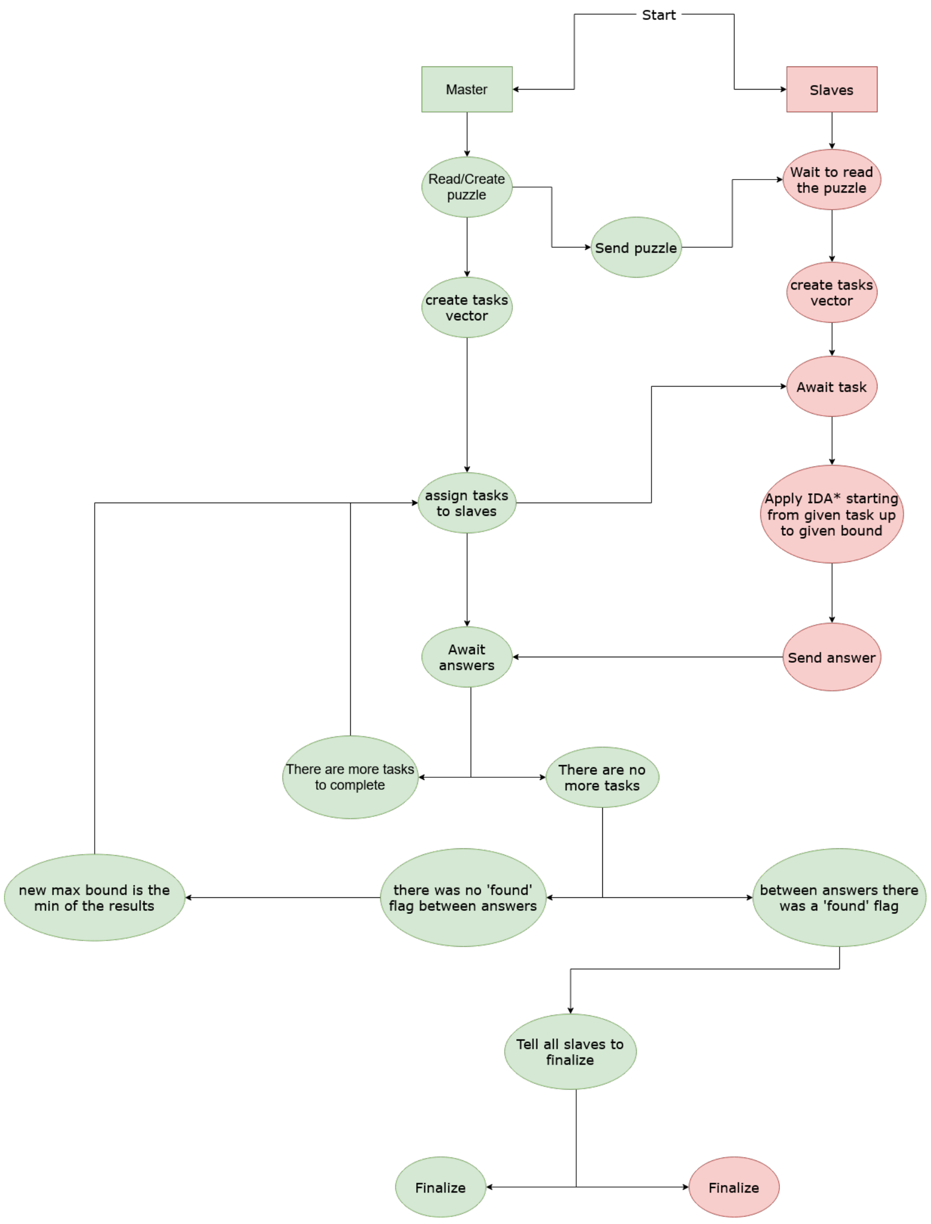
To make the algorithm parallel I applied the following idea: have one central node to give jobs to each slave process, every process, including the master goes down the tree structure starting from the initial node until a level with enough unique nodes is found so that each slave process has at least 1 job to do, those jobs will be kept in a vector, since all processes calculated this vector, and the nodes are determined by a deterministic function we know that everybody has the nodes in the vector in the same order.

Now the master begins to give to each free process the index of the node from where the slave should begin to apply the normal IDA* algorithm, the master first sends a wave of jobs, knowing that the number of jobs is at least as big as the number of processes, after sending this initial wave of jobs the master awaits to receive an answer from any of the processes, the answer contains a flag that determines if the slave found the solution, the job that was completed and a new max limit for that job if the answer was not found, then the master checks if there are any other jobs to give, if there are not then the slave hangs until every other process finishes.

When all processes have finished their assigned job up to the given bound the step above is repeated for a new bound until the solution has been found, if any of the slaves has found a solution then all slaves are signalled that they can call MPI_finalize. This kind of blocking behaviour has the advantage of keeping all slaves working with the same bound but it does come with the downside of the processes having to wait some processes that work on a job that takes more time.

Another variant of this algorithm will be also used where instead of starting the algorithm with a predetermined number of processes we start the algorithm, go down the tree of the problem for a number of levels and then call the parallel algorithm with a number of processes that is equal to the number of unique nodes on that level + 1 or with a divisor of that number + 1 up to a max of 9 processes

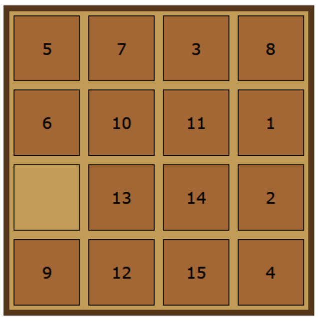
The application flow can be also seen on the next page:


Start
Master
Slaves
Read/Create puzzle
Send puzzle
Wait to read the puzzle
create tasks vector
create tasks vector
Await task
assign tasks to slaves
Apply IDA* starting from given task up to given bound
Await answers
Send answer
There are more tasks to complete
There are no more tasks
new max bound is the min of the results
there was no 'found' flag between answers
between answers there was a 'found' flag
Tell all slaves to finalize
Finalize
Finalize

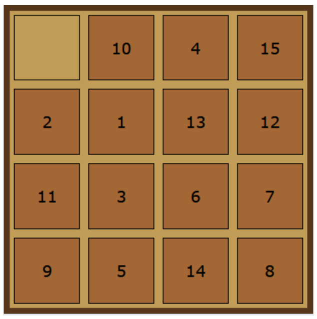
5 Testing the algorithms

5.1 Used puzzles

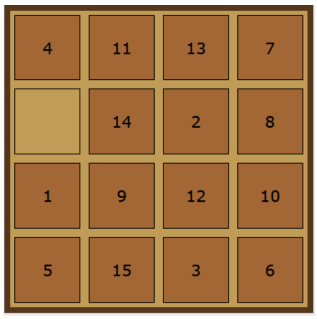
For testing all the algorithms used the following boards are defined:



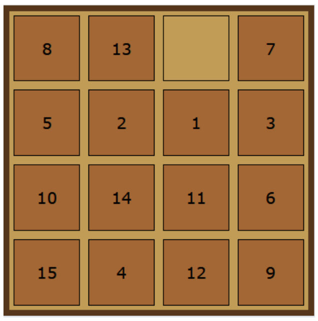
(a)



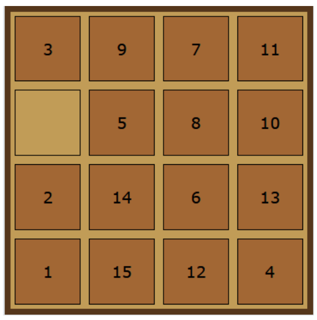
(b)



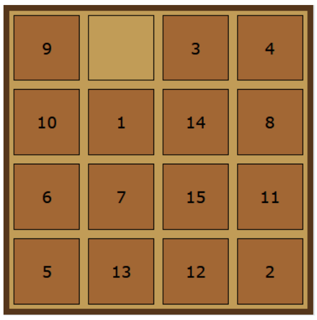
(c)



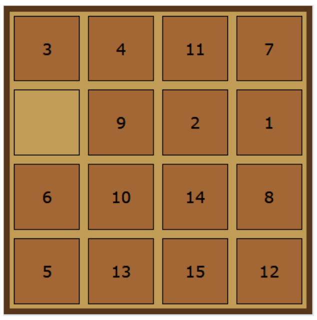
(d)



(e)



(f)



(g)

Figure 3: Puzzles used in experiments

Where

(a) has ID = 1 and can be solved in a minimum of 42 moves

(b) has ID = 2 and can be solved in a minimum of 44 moves

(c) has ID = 3 and can be solved in a minimum of 49 moves

(d) has ID = 4 and can be solved in a minimum of 50 moves

(e) has ID = 5 and can be solved in a minimum of 47 moves

(f) has ID = 6 and can be solved in a minimum of 35 moves

(g) has ID = 7 and can be solved in a minimum of 37 moves

5.2 How testing will be made

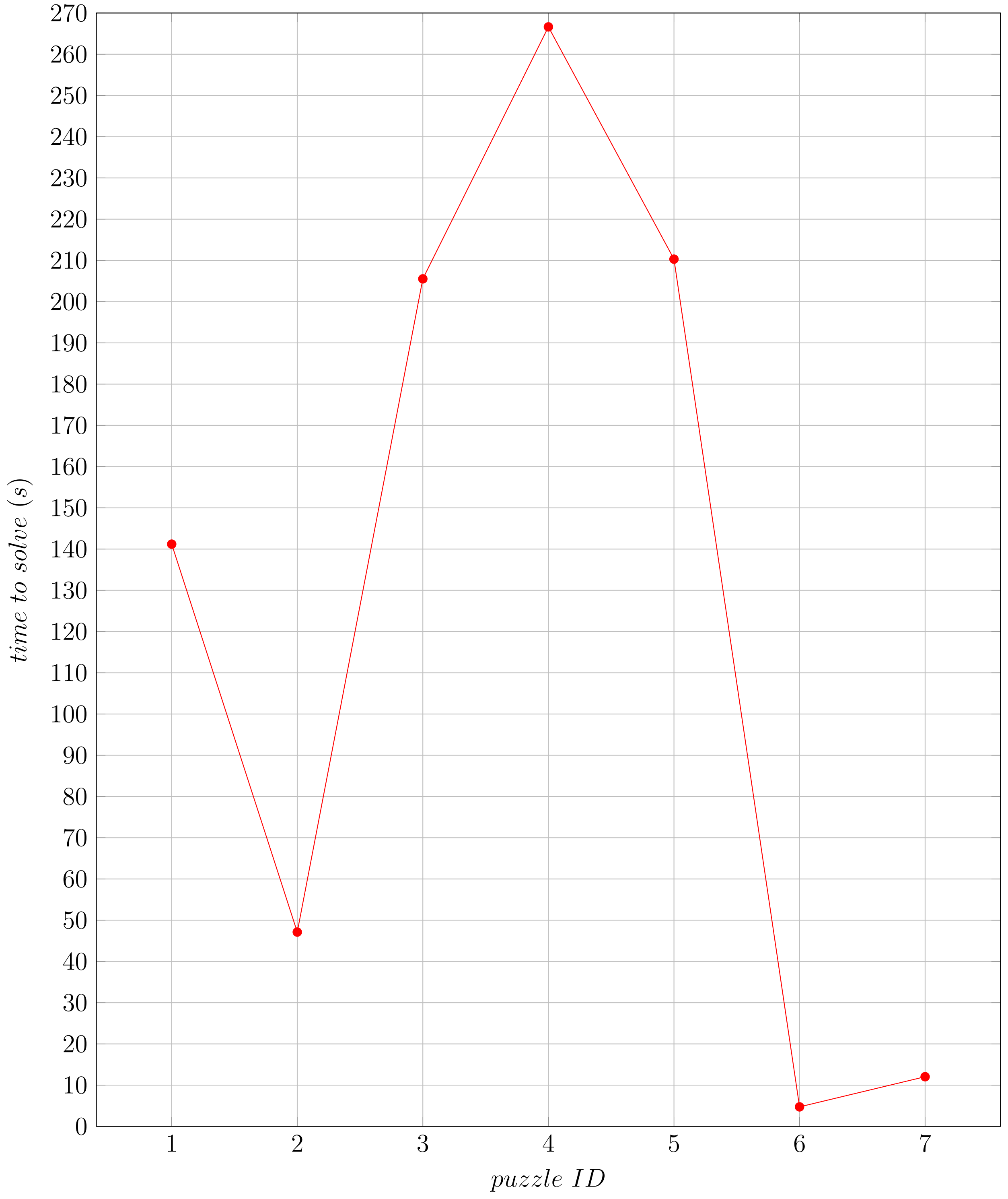
Each of the above puzzles will be given to each algorithm and run 5 times while recording the time it takes to find a solution, an average from those 5 times will be made and this will be compared between algorithms, an xmlx file with all the numbers can be also found with this project.

As for algorithms that will be compared there will be the normal IDA* on a single process, the first method of the parallelized algorithm described above with a number of 3, 4, 5, 8 and 13 processes, and the second method of the parallelized algorithm were I start from a depth of 2, 3, 4 and 5.

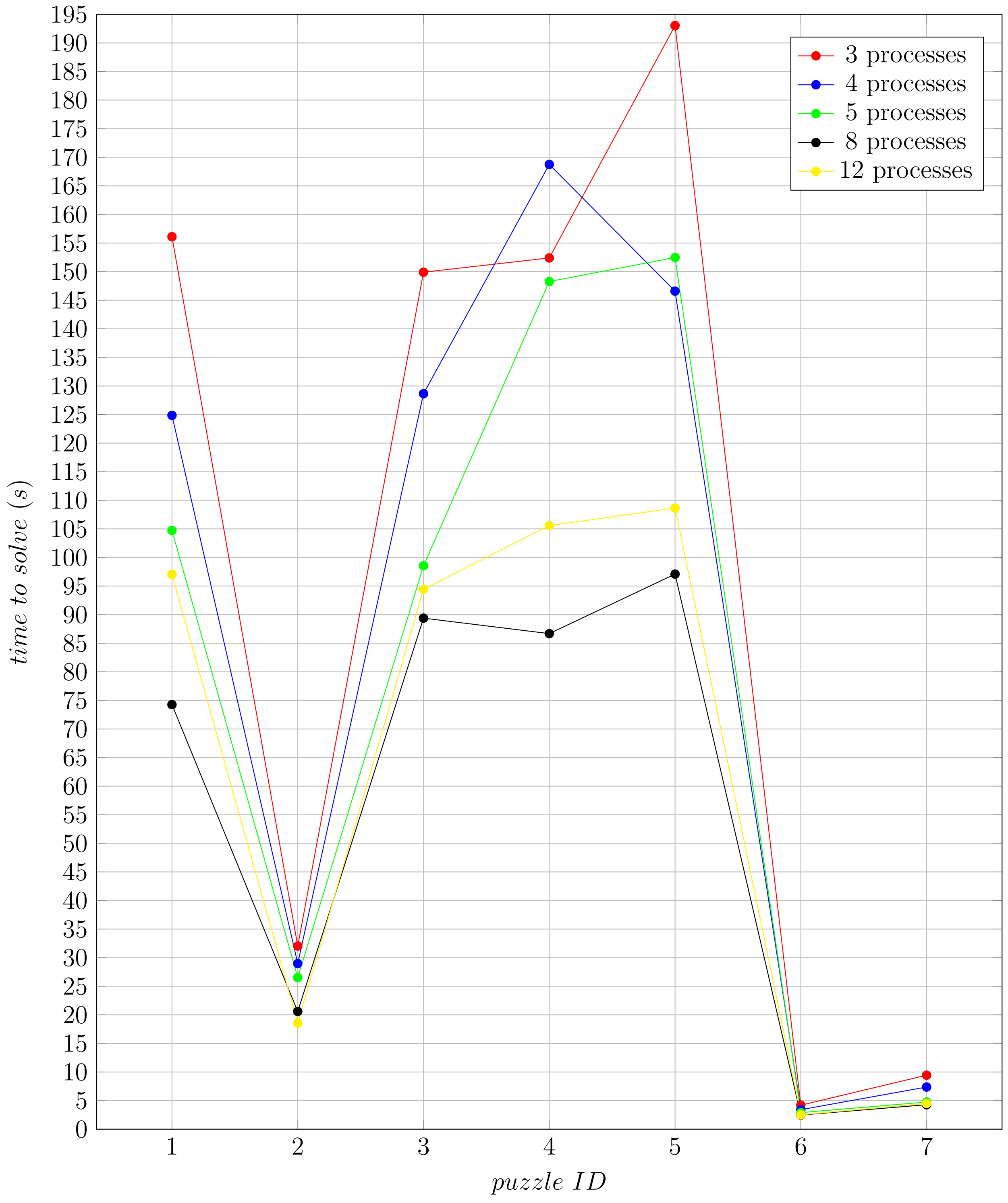
Do remember that since 1 process acts as a master the actual number of processes trying to solve the problem will be n - 1

5.3 Results

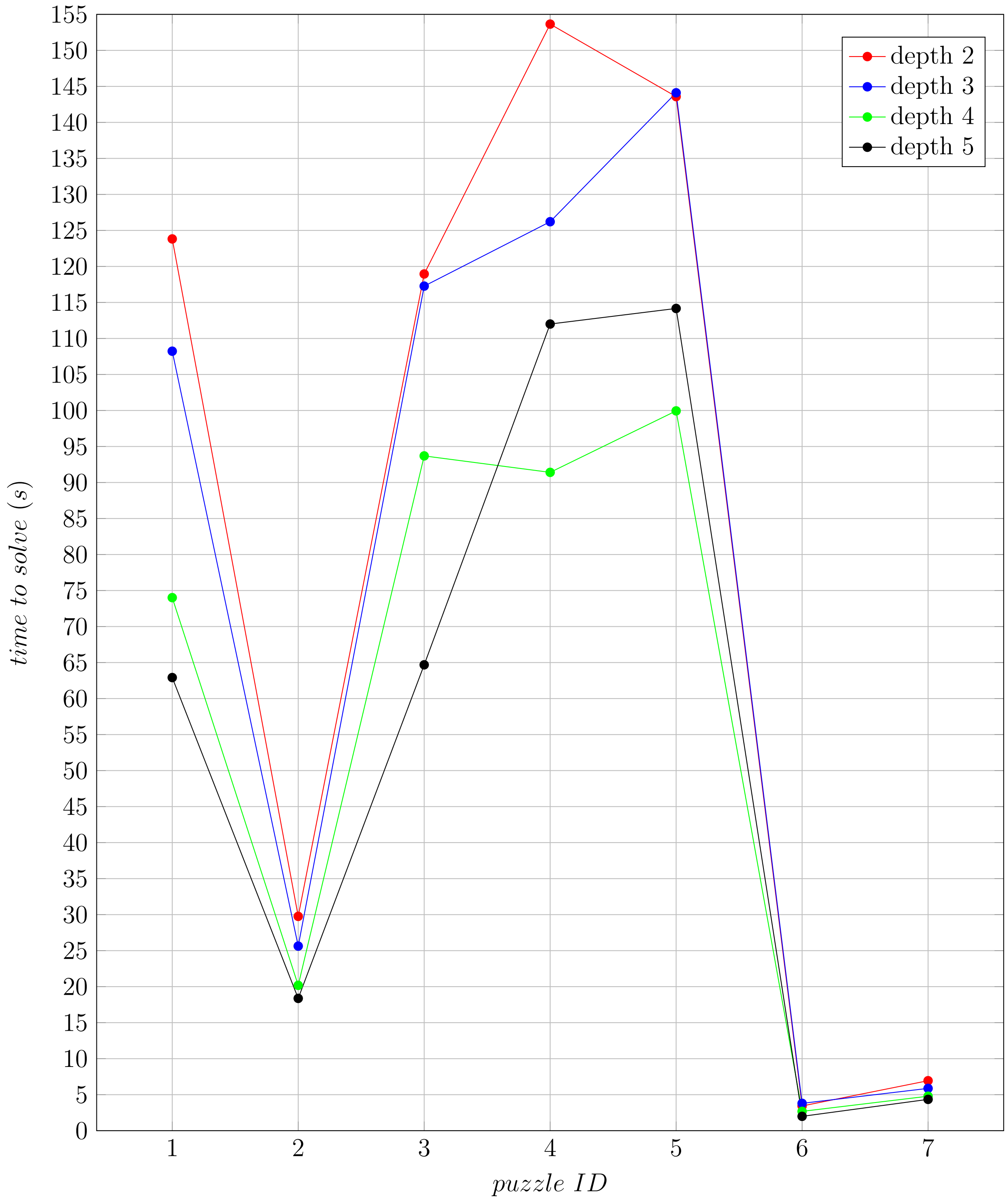
5.4 Single process



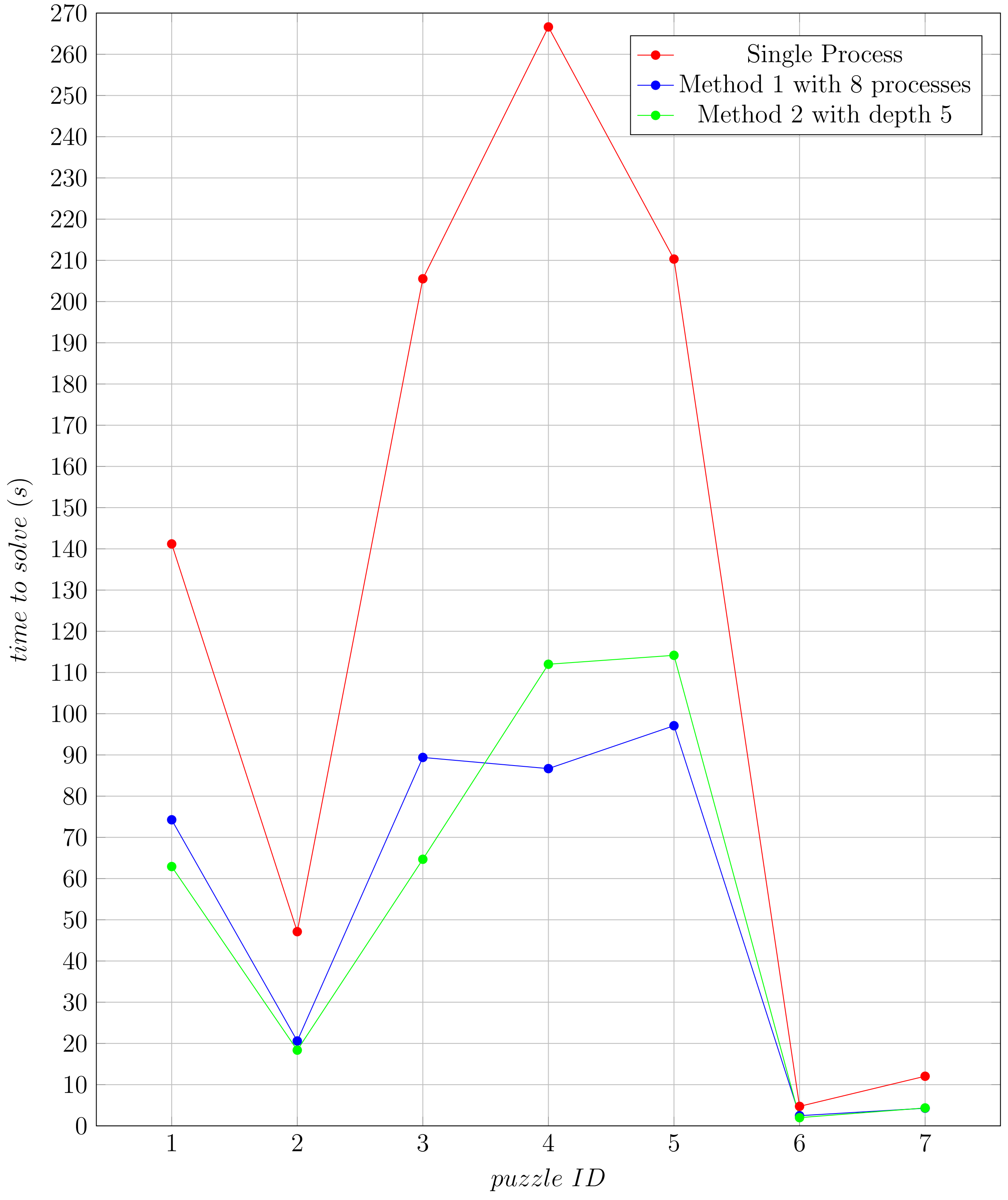
5.5 Multi Process method 1



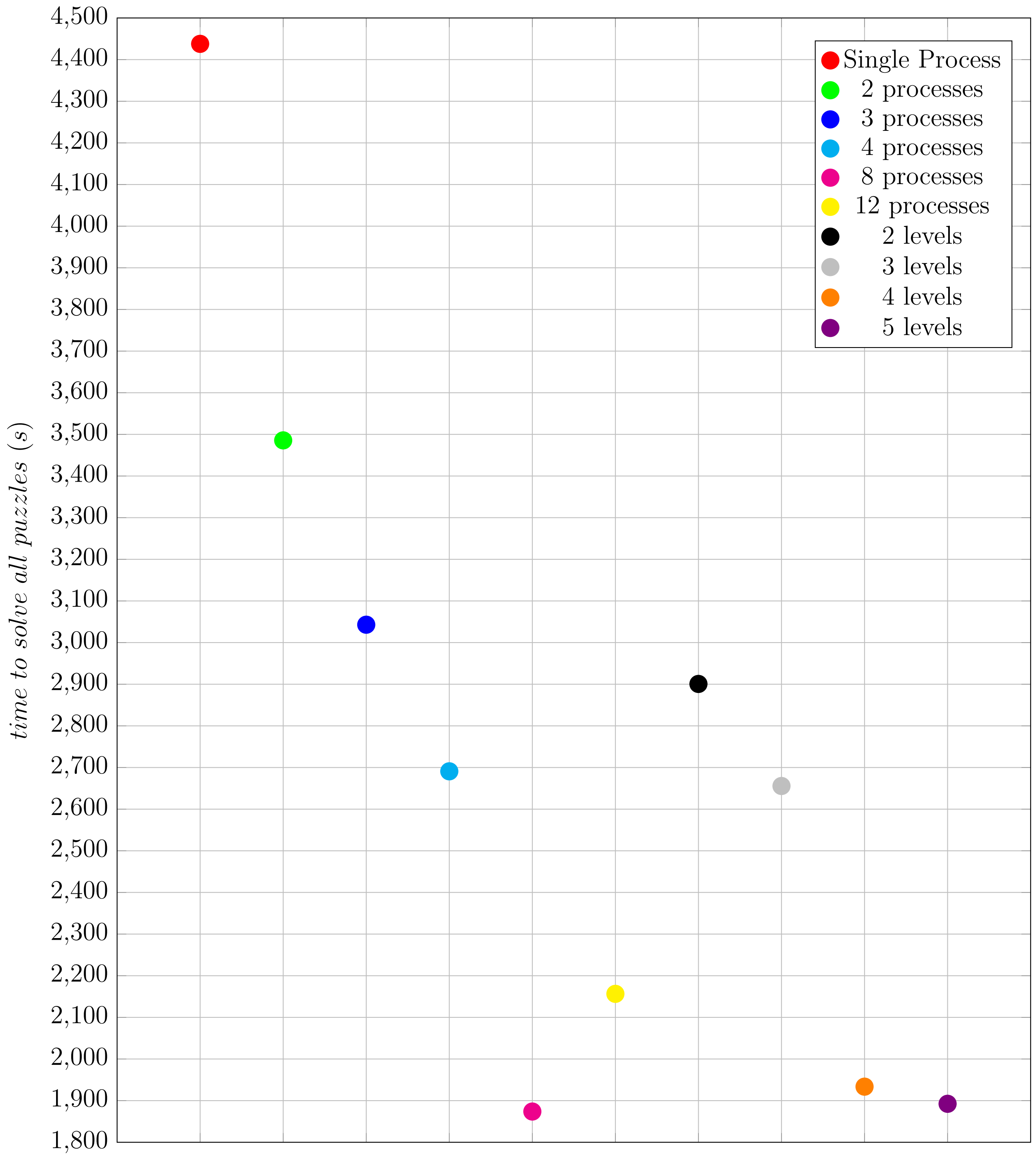
5.6 Multi Process method 2



5.7 Comparing the best of each type



5.7.1 Comparing all methods



5.7.2 Discussing results

As it can be seen from the last plot the best method was the first method of parallelism with 8 processes, with the second method on 5 levels coming on a close second both being about 2.3 times faster than the Single Process algorithm, for the record this instance ended up using 9 processes for all puzzles.

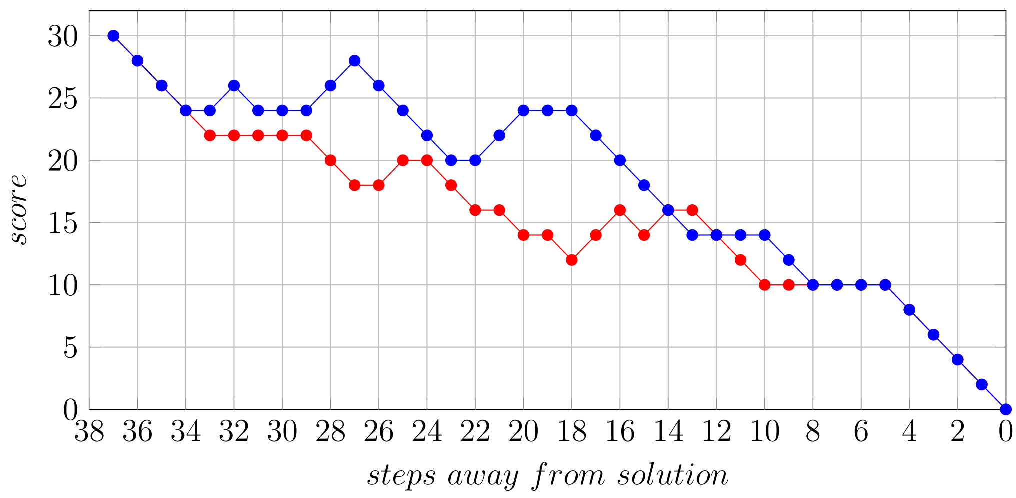
Judging from the above images and numbers it seems that the best performance gains are for bigger puzzles, which makes sense given how the IDA* algorithm works, even between multiprocessing methods the bigger differences are on the bigger puzzles, even if the graphs seem a bit odd it is pretty normal for the lines be intertwined in such a way, this behaviour can be explained by how the scores for puzzles increase and decrease along the path to the goal, see next image, and how sometimes the number of nodes to process is not a multiple of the number of working processes, making processes that have finished their work wait for one/two that just got a new job.

Running all the tests took a total of 07 hours 31 minutes 09 seconds 847 milliseconds or 27069847 milliseconds

6 Possible improvements

There still are some improvements that can be made to the algorithm, some of them might be smaller, dependant on implementation and language, for example more efficient memory allocation/ deallocation and more of this kind of changes that could grant a small increase in performance for all the algorithms.

Some changes that could be applied might need some testing beforehand, for example in IDA* you have to see if a node already is in you current path, I have used a vector and searched trough every element of it but maybe keeping something like a set of buckets and searching only nodes that have identical score to the one we want to verify might give a little boost in speed. We cannot use a binary search in this chase though, theoretically the scores of the puzzle should be decreasing the closer you are to the goal node, but this is not he case for this kind of puzzle, since to make some moves you will end up changing pieces that are on their goal goal position or putting them further away, this can be seen in the next image, it would be ideal to find a heuristic that gives a smooth decreasing line to the goal, but the Manhattan distance end up giving those kind of graphs and since you need to keep nodes ordered by apparition makes finding an efficient search method harder without employing other containers.



Other improvements are method dependent, for example playing around in the second method with the maximum number of processes or with how far we go down the tree initially.

Other possible improvements need to change all algorithms, for example changing the heuristic for scoring puzzles, see Disjoint pattern database heuristics[2].

Or even changing just multiprocessing methods, going from IDA* to ID removes the need to wait for all processes to finish, since the next bound is always easily calculated and does not depend on the results of the current level, in this case non-blocking sends and receives can be used to shave of some waiting, but this method will be useful only if the number of starting nodes is bigger than the number of processes, since ID will make the bound graph a straight increasing line, all processes should finish at about the same time without needing a special mechanism of syncing. This said at the end of the program some waiting still needs to be done, even after finding a solution all processes must wait for those who are calculating nodes at a smaller depth than the one of the current solution, in case a shorter path is found in that sub-graph.

References

[1] Wikipedia, 15 puzzle article

[2] Disjoint Pattern Database Heuristics by R.E. Korf and A. Felner