Homework 4

José Javier González Ortiz

November 11, 2015

1 Algorithm

Given the problem, the algorithm for the serial version was quite straightforward. Using *Depth First Search* via a stack data structure we just compute the paths and update a global bound with the best path that allows us to prune subtrees when they are bigger than the bound.

In order to parallelize this problem in a Manager-Worker fashion we can define a clear approach. First, all the threads help create the lookup table for the distances and once they are done, they wait for the Manager to create some initial work. After the work is created they all act as workers in DFS searches with a common bound. From this point on the manager task is a shared role. Using a shared variable with bits for each processor we can track if any of the processors is idle and put a task in the global queue for them to fetch.

This way the manager work is distributed among the processors and we can maximize computing power.

We have only to determine the way the initial jobs are going to be created. We established to approaches, BFS until there is a significant amount of initial work per thread or DFS until a bound is found.

2 Analysis

The analysis for this problem is not trivial since we can explore multiple paths at once and find best bounds in parallel, we can significantly speedup the algorithm. However, even if we had the best bound from the start there is a unavoidable amount of work associated with pruning that will happen as in serial. Namely, that if we have to explore 100 nodes to discard them, adding p processors will only allow us to get a SpeedUp of p.

What it is more, since the exploration and pruning won't be self balanced, the processors will have to spend time sending and receiving tasks to keep the work balanced. Therefore we should a expect an overhead created by the task handling. At the same time, since the shared bound may need to be changed this could also involve some extra overhead. Being a shared variable the change has to be done in mutual exclusion or otherwise we may incur into correctness issues.

Finally we should determine how does the communication grow as as function of p. Namely, how many times in average does a processor need to return work to the global queue to keep all the processors busy. If this grows significantly faster than the processors, we will spend more time keeping the work balanced than doing actual useful work.

3 Implementation

For the implementation the following approach has been followed. First a master thread initializes the time and some shared variables. Next, all the coordinates and distances are computed in parallel, this does not save much time but it was easy enough to do with `pragma omp for`. After that step we can generate the initial amount of work with the master thread. In this step two variations were tried:

a) **BFS** - The master thread generates an initial amount of work by doing a Breath First Search on the tree. Specifically since $n < 32$ for all the test cases, $C(C-1)(C-2)$ has been a good amount of initial jobs and it means the master thread is exploring the first three levels.

b) **Directed DFS** - Another more esoteric approach has been doing a DFS until a bound is found and then start parallelizing from there. To improve such bound, the DFS Search always follows the minimum distance branch. Finally to improve the initial paths explored, the DFS Stack order is reversed and the threads start pruning.

After the initial amount of work has been set up the threads start exploring via DFS in parallel. For each path to explore, they compare the accumulated cost to the global shared bound and only explore it if is less than that. Moreover, if they see that any other processor is idle by looking at a shared variable, then they send that work to the global queue that the idle processors are constantly watching. All the operations over the global queue are done in mutual exclusion to preserve correctness. Finally, when a better bound is found the processor also in mutual exclusion updates the global bound to enhance the pruning.

Once the idle variable is zero for all processors and the global queue is empty the program finishes by outputting the best path found.

Finally, since we are creating a great amount of initial jobs $C(C-1)(C-2)\big|_{C=15} = 2730$ which is still small compared to the average amount of nodes explored, $N = 127065766$, but quite big compared to the processor sizes in hang. Therefore, we can try removing the load balancing to compare the behaviors.

4 Results

The DFS approach had good results for $n = 4$ but scaled quite badly for bigger values due to the huge communication involved. Therefore, only two implementations were throughly measured, BFS with load balancing and BFS without load balancing.

The results from the timings are shown in the Table 4.1. Part 4.1a contains the values for balanced workload and Part 4.1b for unbalanced workload. We can see a surprising result. While for $n = 4$ the balanced algorithm behaves slightly better than the unbalanced one, for $n = 16, 32$ the unbalanced one does reasonably better.

p	average	min	max
1	29.68331	29.49489	29.83359
4	8.04715	8.01577	8.06586
16	2.83875	2.64807	3.09577
32	1.93877	1.72996	2.08387

(a) Results for a balanced workload

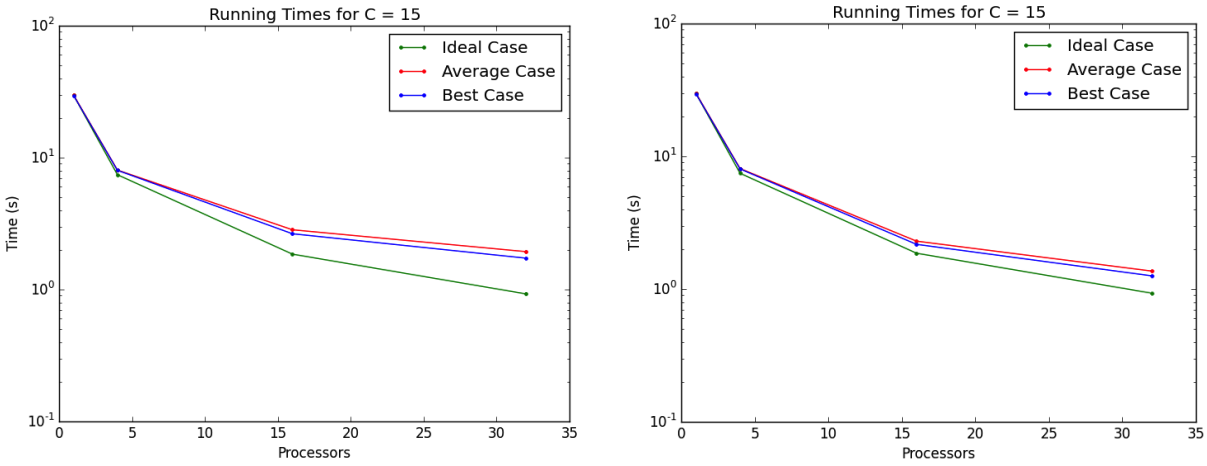
p	average	min	max
1	29.80083	29.60831	30.05029
4	8.08582	8.03430	8.12880
16	2.29715	2.17243	2.34336
32	1.36655	1.25793	1.52740

(b) Results for an unbalanced workload

Table 4.1: Results in seconds for different number of processors

To ease the understanding of the data we can plot different relations between the data. In figures 4.1a and 4.1b we can see the timing results in a log scale for balanced and unbalanced workload. Defining $\text{SerTime}(n) = \text{ParTime}(n, 1)$ and calculating the ideal parallel time as $\text{SerTime}/p$ we get some interesting results.

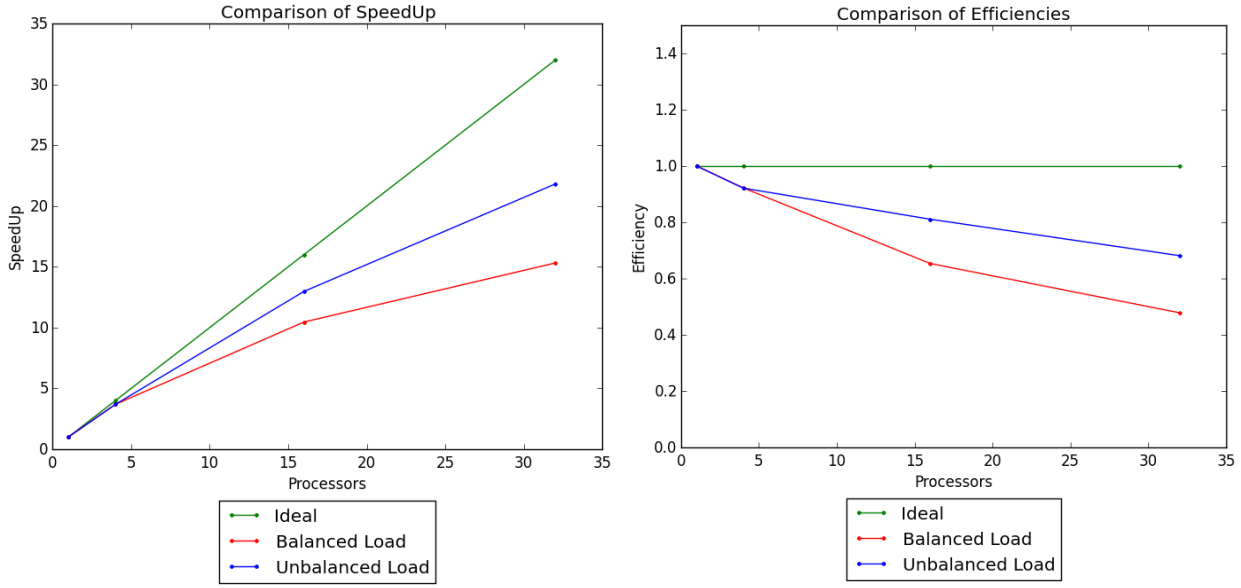
As we can see the curves match up quite nicely, being the minimum time better than the average one, which seems logical. The deviation for greater p seems larger, however we are using a log scale so in fact it is not as big of a difference. Finally, we can see that for unbalanced workload the timings are a bit better than for a balanced workload with higher number of processors.



(a) Balanced Workload (b) Unbalanced Workload

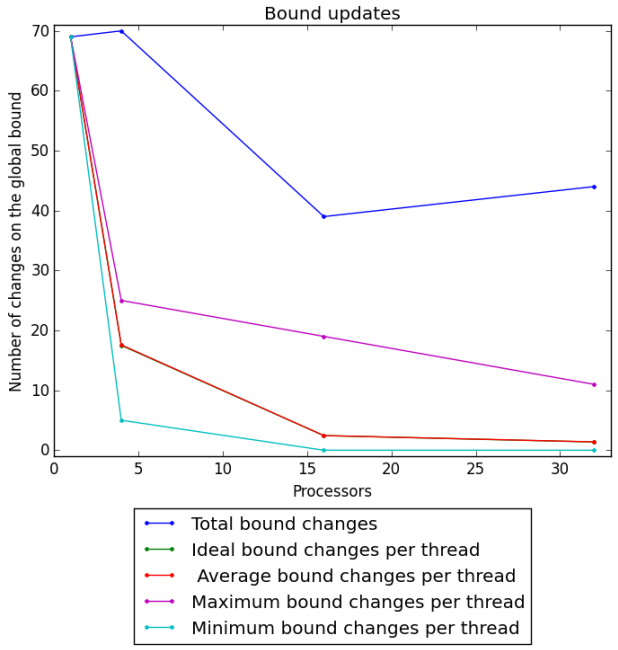
Figure 4.1: Logarithmic plot of times for different number of processors

Now, to analyze both SpeedUp and Efficiency, we can see the plots shown in Figures 4.2a and 4.2b, in which we can find again the ideal case, for balanced and unbalanced node exploration. The same patterns appear here. We can appreciate that the parallelization is good enough to render a quasi linear speedup which in a load balancing algorithm seems good enough. However, bigger data sizes and processor numbers should be ran in order to get definitive results.

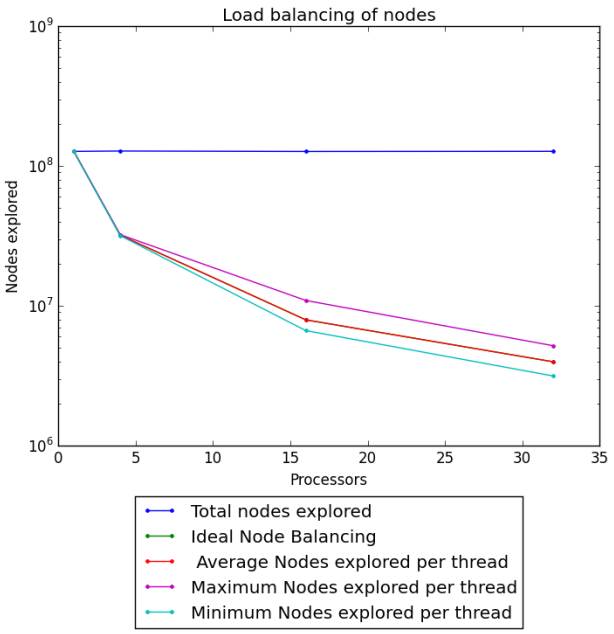


(a) Speed-Up (b) Efficiency

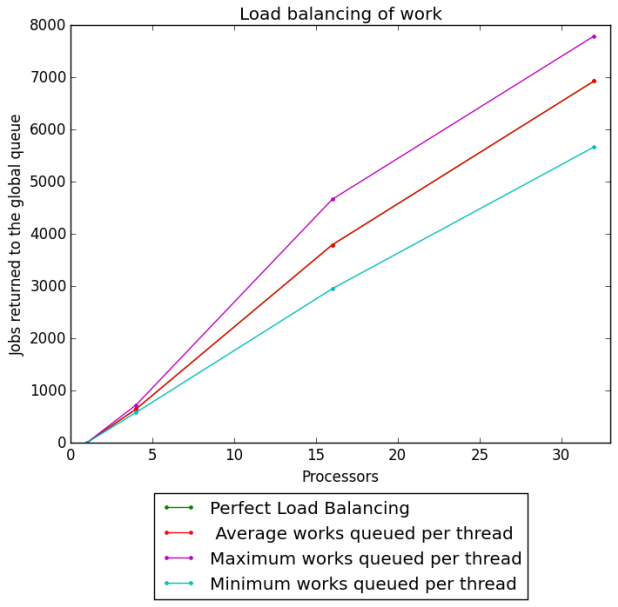
Figure 4.2: Speed-Up and Efficiency plots for the ideal case, Balanced and Unbalanced Workload



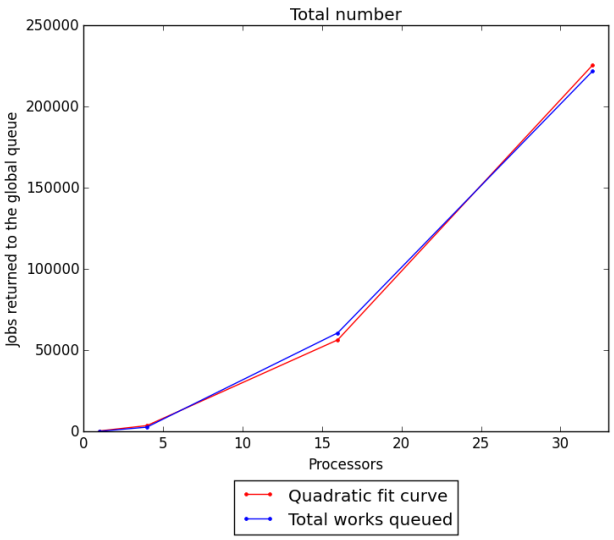
(a) Changes over the global bound



(b) Nodes explored per thread



(c) Pushes into the global queue per proc.



(d) Total number of pushes into the global queue

Figure 4.3: Workload balancing measurements for different number of processors

In order to analyze why the balanced workload algorithm is behaving like that, a number of measurements variables have been set up in order to keep track on the behavior for the load balancing. The variables that have been recorded were the following

1. **Changes on the global bound** - Since no matter what the algorithm we are forcing the best bound update to be done in mutual exclusion, we would want to estimate how often does it occur in order to determine if it can be slowing down the program. As Figure 4.3a shows, the number of updates is ridiculously low. For all of the runs, the number of changes was lower than 100, and since the number of explored nodes $N = 127065766$ is 7 orders of magnitude greater than the number of changes this should not affect the running times.

2. **Nodes explores per thread** - This is a good estimate of how much work each processor is taking per execution. The sum of the explored nodes should not exceed the serial number of explored nodes ($N = 127065766$) and ideally for p processors we should expect N/p explored nodes per processors. This is shown in Figure 4.3b. Since we are plotting logarithmically, the results shown look good. The load is being balanced and no processor is doing a excessive amount of work.

3. **Job pushes to the global queue** - The previous variable measured the *worker* load balancing while this one is going to measure the way the *manager role* is being balanced through the processors, since as we said earlier we are using a distributed manager scheme.

   In Figure 4.3c we can appreciate two facts, the first is that the number of pushes into the queue per processor grows as the number of processors increases. This is logical, since having more processors implies that more dead-ends will produce more empty stacks and therefore more idle processors than need work to be sent to them. The second fact is that the workload seems balanced, although is not as good balanced as the path exploring task, we still get an average coincidental to the ideal.

   Finally the biggest clue about the surprising speed of the unbalanced approach comes from the analysis of Figure 4.3d. As we can see the total amount of pushes into the queue grows as a quadratic function, more specifically $F(p) \approx 220p^2$. This implies that the communication overhead complexity is $\Theta(p^2)$ and therefore we will spend more time balancing the work than just doing the work in an unbalanced fashion. This explains the results obtained before.

As an end note just to remark that the unbalanced algorithm could do better in this case since we initialized the queue with three levels of recursion which usually is a big enough number for a realistic number of processors. Namely, had we chosen to initialize the queue with p nodes, the unbalanced algorithm would have behaved much worse than the balanced one. Also, it has to be taken into account the fact that we are only using a strict set of data points, which coincidentally have distances that almost self balance the problem. Given a more complex set of data points, probably the scaling for the unbalanced algorithm could be worse than for the balanced algorithm.

5 Correctness

Finally, we need to check that the algorithm produced the correct results. For all the different runs with varying number of processors, the algorithm always outputted the correct answer, guaranteeing us the correctness of the implementation. In Figure 5.1 we can see the visualization on the solution. (Note that although euclidean distances are plotted, those are not the distances that have been used in the algorithm)

$$\begin{aligned} \text{path} &= [0, 13, 9, 1, 8, 14, 7, 2, 3, 4, 11, 6, 10, 12, 5] \\ \text{cost} &= 782.989290 \end{aligned} \tag{5.1}$$

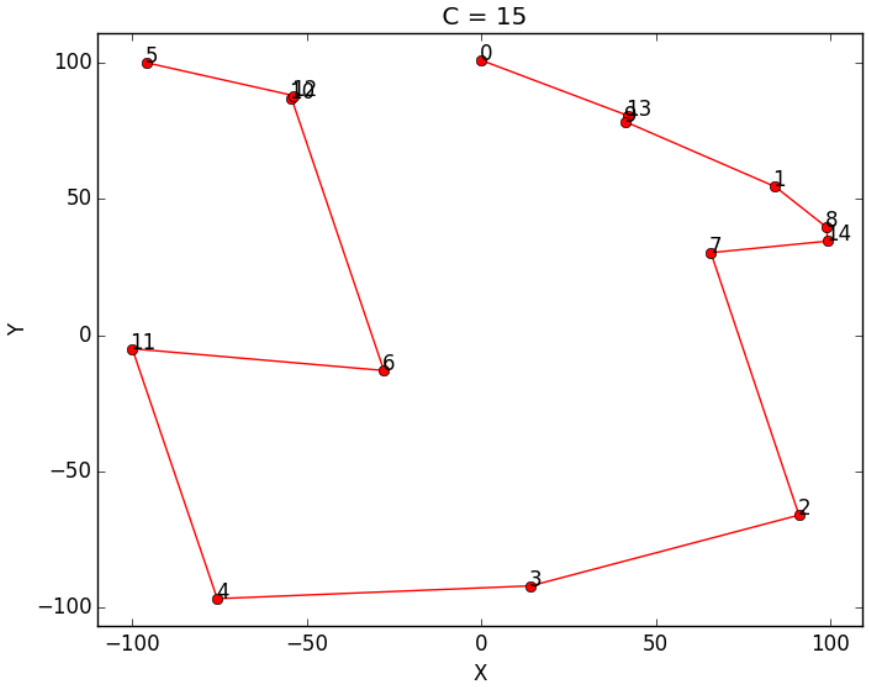


Figure 5.1: Shortest Path