University of Illinois at Chicago

Parallel Processing

ECE 566

A.Y. 2017 - 2018

Assignment 4

CIPOLLETTA Antonio 655452970
GUALCO Andrea 651262476

Contents

1 Introduction

The main goal of this assignment is to implement a parallel algorithm to solve optimally the TSP problem , *Travelling Salesman Problem*.
There are many different ways to solve this problem. For example two algorithms as Best-First Search and Depth-First Search can be used to accomplish the task.
Among all the possibilities, it has been decided to implement a `Branch and Bound Depth-First Search`. This is done in order to prune some path that for sure will not lead to a better solution than the one already found. The worst case complexity of the problem is still $O(n!)$, but in major input instances the running time is largely reduced.
A dynamic load balancing is implemented to divide the workload among all the processors. In particular three different policies, `Random Polling, Asynchronous Round Robin and Global Round Robin`, have been tried in order to make a comparison. The specific policy can be selected at compile-time since in order to avoid additional overhead.
The `Dijkstra Termination Algorithm` is used to recognize that all the processors are out of work.
Multiple experiments have been run using four graphs, testing the performance changing the number of processors and the different policies.

2 Solution adopted

2.1 TSP library

In order to test the program and do the experiments, some instances of TSP graphs have been chosen. The TSP files have been retrieved on-line[1][2][3].
It has been performed a pre-processing since the developed program takes as input a graph described through an adjacency matrix.
If the graph from the library is not described as a matrix, but as node coordinates, it is necessary to compute the euclidean distance between all the nodes and create the adjacency matrix.

2.2 Data structure used

The main data structures adopted are:

- `Stack`. In order to store the untried alternatives during the exploration of the graph. The stack has been implemented using a linked list. Each element of the stack is a path.

- `Path`. A path is represented through a vector of nodes. Each path stores both the cost of the path and the estimation of the cost to reach a solution.

[1]http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsplib.html
[2]https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html
[3]http://www.math.uwaterloo.ca/tsp/data/index.html

2.3 Depth-First Search algorithm - Branch & Bound

2.3.1 Depth-First Search

The Depth-First Search algorithm starts exploring as deep as possible until it has to backtrack because neither a solution is found nor from the current path is not possible to reach an hamiltonian path.
The branch & bound method is used to prune some paths. This method is applied because the TSP is a NP problem therefore also for small instances it takes a long time to do a complete exploration. Pruning some paths does not affect the final result, but it may reduce the execution time.
The branch & bound method is based on the availability of an estimation of the cost to reach a solution from the current path that is a lower bound on the actual cost. The pruning is done as follows:

1. the actual cost is the sum of the cost of edges from the root to the actual node
2. then the estimated cost is the sum over the unvisited nodes on the minimum and second minimum of that node divided by two.
3. in practice the estimated cost is updated
   - if the explored node is one of the child of the root node the estimated cost will be computed as
     $$est\_cost_P = est\_cost - \frac{\text{min edge cost of the root} + \text{min edge cost of the node}}{2}$$
     where P is the node explored on the path
   - if the explored node is not one of the child of the root node the estimated cost will be computed as P is the node explored on the path
     $$est\_cost_P = est\_cost - \frac{\text{min edge cost of the node} + \text{second min edge cost of the previous}}{2}$$
4. the estimated value is summed to the actual value to obtain a total cost
5. if the total value is greater equal to the actual best solution the path explored is not added to the stack and the node is labeled as visited so that the solution in not more explored in the future.

2.3.2 Load balancing

In order to solve the TSP problem optimally and reducing the execution time, it is possible to parallelize the program.
The parallelization is done using a dynamic load balancing approach.
The workflow of the program is the following:

1. An initial phase where all the processor have an empty stack of path, so they will enter in a condition where they start sending work request. First the root processor starts working and then it sends the white token to the other processor.

2. Once a processor send a request of work, the request is first verified. If it is rejected the processor returns to send work requests, otherwise if it accepted the processor that has received the request splits its stack, serializes it and sends it to the destination processor, that receives the data, de-serializes it. Now the processor has not the stack empty and can start working.

3. The work consists in pop a path from the stack, exploring the non visited node and computing the total cost (see pruning above). During the work if a path of dimension n is found the actual best solution is checked and it is returned the new one only if it less than the actual. The new solution is broadcast to all the other processors.
   The work is stopped periodically to check if there are work requests and in case the requests are served.

4. Each processor before starting working again, first check if there are broadcast of the best solution to receive and then check if there are incoming work request arrived from other processors.

5. During all the process the token continue passing through the processor when the processor that is holding the token becomes idle. Only in the case where all the processors has finished their job the token will return white to the root. In this case the termination signal is sent to all the processor and the TSP solution is returned

The work request policy is decided by macro in the code. There are 3 possible policies:

- random polling
- synchronous round robin
- asynchronous round robin

In figure 1 it is shown the work-flow of the of the program

2.3.3 Termination

In order to determine if the work is finished and to terminate the communication between the processor, it is used the Dijkstra's algorithm.
The color used are: white, black and green.
At the beginning all the processor are white. If a processor communicates with one with a rank higher the next step the token sent will be white and the processor remains white. On the contrary if the processor communicates with a one with lower rank the token sent will be black and the color of the processor will be set to black.
This means that once that the token returns to the root, if the token is white, all the processor has completed their work and they can be terminated. If the token received by the root is black, there is still work pending, so it is necessary to continue the work and the token will

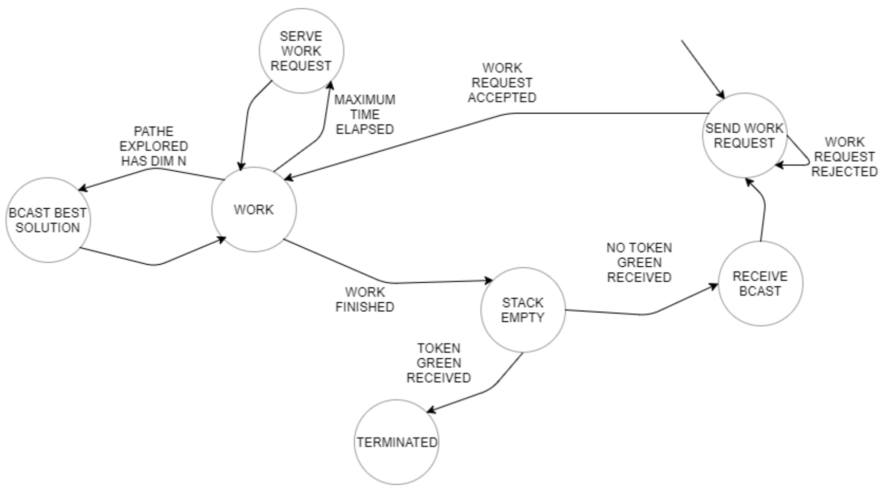

Figure 1: Work-flow of the program

be passed again till it returns white to the root.
When the token returns white to the root, it is sent a token green to all the processor as a termination signal.

3 Formulation

It has been used the default communicator MPI_Comm_World since there is no need to have a particular virtual topology in this kind of problem.
The adjacency matrix of dimension $n \times n$ has been read from the file in the root processor. Then the root performs the broadcast of the matrix to all the processor.
The function `MPI_Bcast` has been used both to send the number of nodes in the graph and the adjacency matrix to all the other processors.
The `MPI_ISend` is used for the asynchronous communication in the request of work mechanism and also in the termination detection algorithm.
On the contrary the stack split mechanism is implemented using the `MPI_Send` and `MPI_Recv` since the processors involved are already synchronized through the request mechanism. In order to receive the asynchronous message in a non blocking way the function `MPI_Probe` has been used to test the availability of a message.
The function `MPI_Comm_rank` is used to retrieve the rank of the processor.
The function `MPI_Finalize` is used to close the MPI environment.

4 Parameter ranges

The input parameters of the programs are:

- the filename of a TSP file which contains the adjacent matrix;
- p that is the number of physical processor used for the topology.

Many experiments can be done in this case. All the combination of the n, p has been investigated.
The ranges of the values for the TSP problem experiments are:

- 4 file `.tsp` (`bays29, fri26, gr17, p15`) taken from the library suggested;
- $p = 4, 8, 16, 32, 64, 128$

The `.tsp` file has to be formatted in this way:

- in the first row there is the dimension of the adjacency matrix
- then there is the adjacency matrix. The edges the connect the node to itself have to be value -1.

5 Timing measurements

In order to have a quantitative measurement of the performance for each simulation a time measurement has been done. Since the cluster is a shared machine and due to the high sources of variability the following approach has been adopted.
In each source code the main part has been identified, i.e. DFS solution of the TSP. This part has been executed 20 times with a time measurement for each iteration.
At the end the average time and the deviation $\sqrt{\frac{\sum_{i=1}^{20}(time[i]-average\ time)^2}{19}}$ has been computed and used to evaluate the simulation.

6 Results and analysis

NOTE: There are some points outside the trend curve in all the graphs. Since the CLUSTER is a shared machine is not easy to perform good measurement of time even if an average between multiple running is performed, for this reason the *outsiders* are not considered for the comments.

6.1 Analysis

The analysis performed are based on:

- Statistics extracted during the execution of the program.
- Time meaurements.

6.1.1 Parallel Run Time Analysis

In the following pictures are shown for each of the graph used as input instance the value of the parallel run time changing the number of processors. In the same graph are also shown the value for all the three policies.

Except for the Fri26 [4] the synchronous policy lead to a greater overhead compared to the other policies. The contention to access the dispatcher processor and also the more regular reqeust pattern increase the overhead.
Only when the size of the graph starts increasing, the availability of more computing resources reduces the parallel run time. Otherwise if the size of the graph is small the overhead due to communication and contention in case of the synchronous RR becomes dominant compared to the amount of work that can be split among the processors.

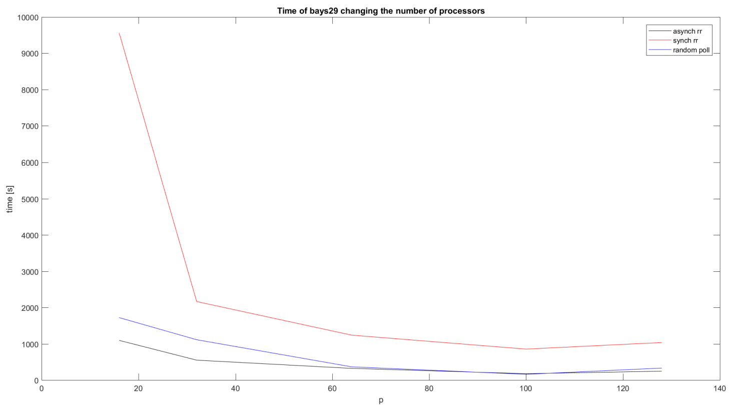

Figure 2: Bays29 timing results

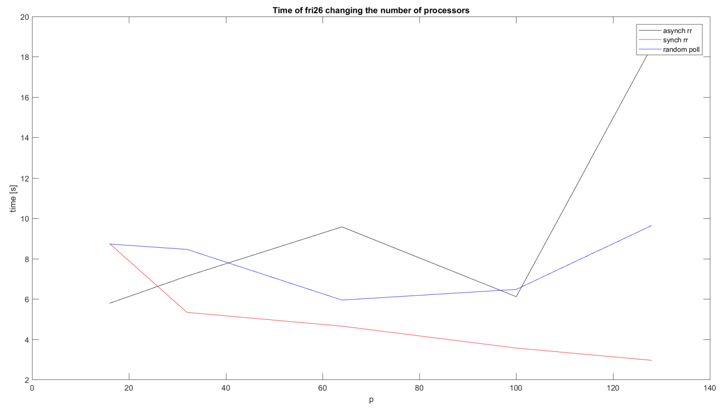

Figure 3: Fri26 timing results

[4] where as can be noticed from the speedup graphs reported below there is a superlinear speedup due to the pruning function and that a very good solution is early found

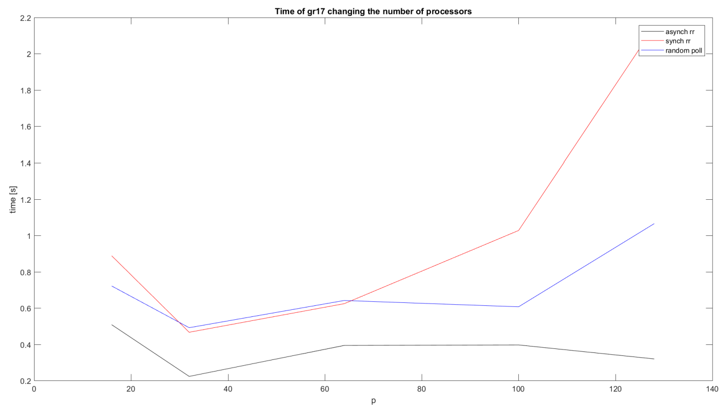

Figure 4: Gr17 timing results

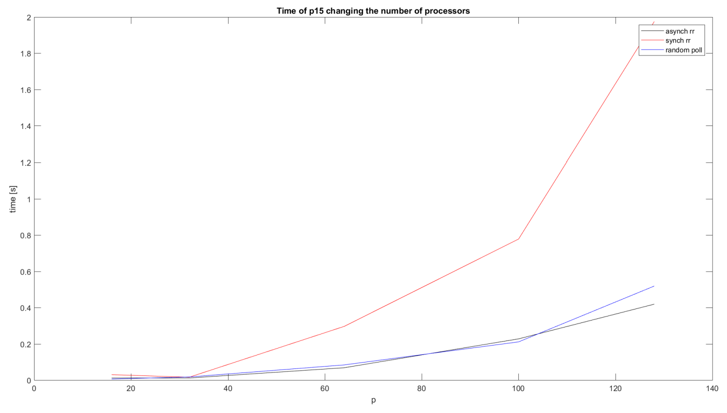

Figure 5: P15 timing results

6.1.2 Speedup and Efficiency Analysis

Globally the speedup for the smallest graph p15 is pratically negligible due to the small amount of work compared to the overhead. The efficiency in the plot is always increasing proceding in the graph dimension axis and decreasing proceding in the number of processors axis. To keep the efficiency constant the direction of moving in the plot is the diagonal therefore incresing both the number of nodes and the number of processors. This can be seen in the plot looking for flat region. The speedup and also the efficiency are pretty similar for random polling and asynchronous round robin except a slightly inversion of the role in terms of performance of fri26 and bayes29.

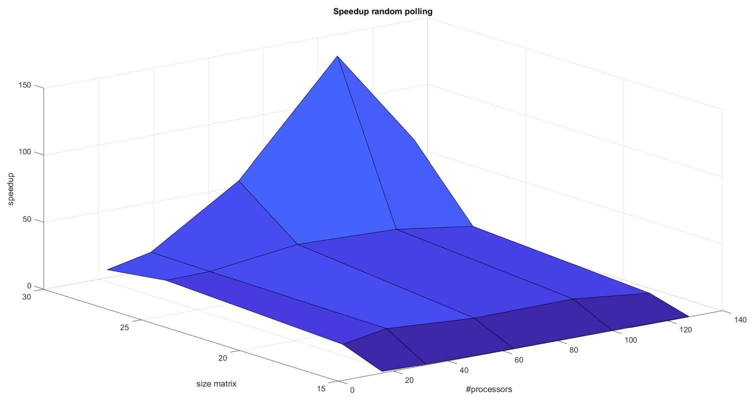

Figure 6: Speedup random polling

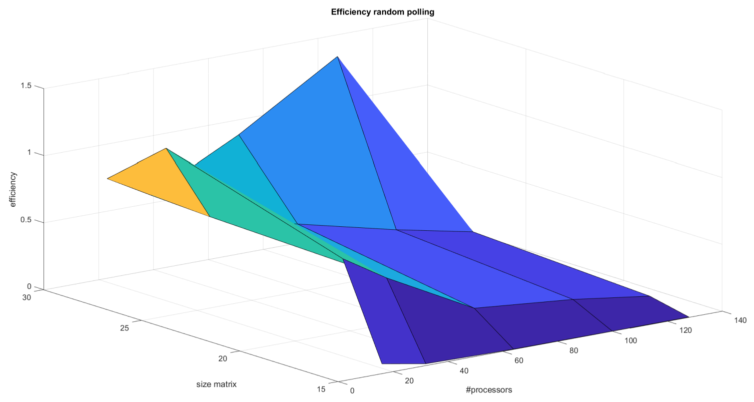

Figure 7: Efficiency random polling

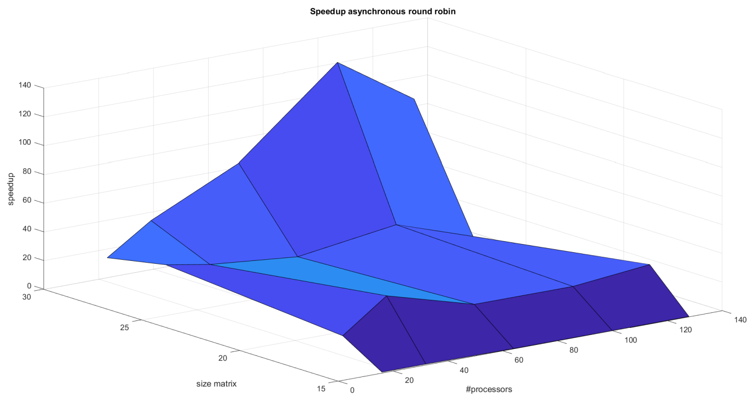

Figure 8: Speedup asynchronous round robin

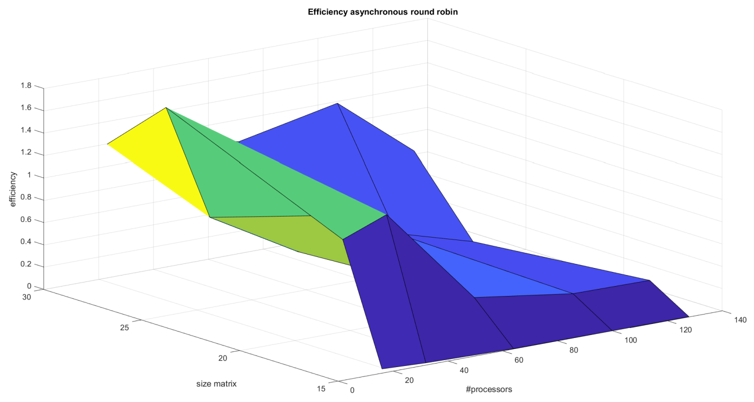

Figure 9: Efficiency asynchronous round robin

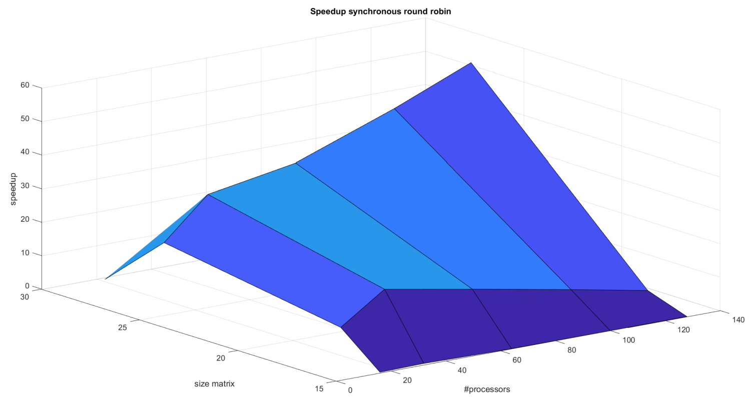

Figure 10: Speedup synchronous round robin

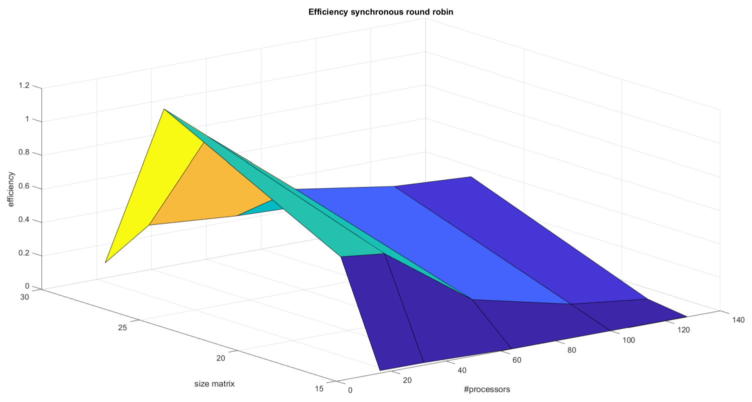

Figure 11: Efficiency synchronous round robin

NB: in the x axis of the histograms, the 1 correspond to the `bays29`, 2 to `fri26`, 3 to `gr17` and 4 to `p15`

6.1.3 Percentage of work

The following stats show the average and the deviation of the percentage of working time for different number of processors and policies. When the number of processors is high there is much more variability of the working time meaning that the load is not very well balanced and this is reasonable since the overhead in terms of requests is greater.

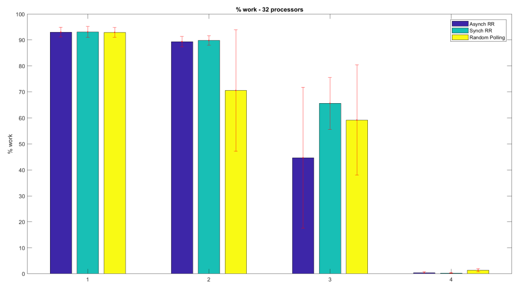

Figure 12: Percentage of work on 32 processors

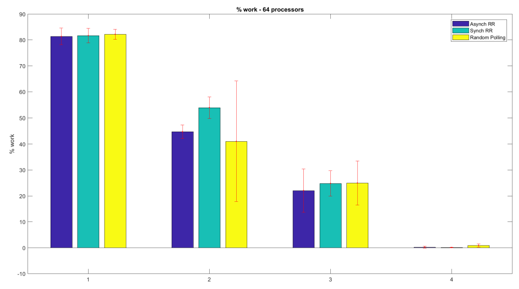

Figure 13: Percentage of work on 64 processors

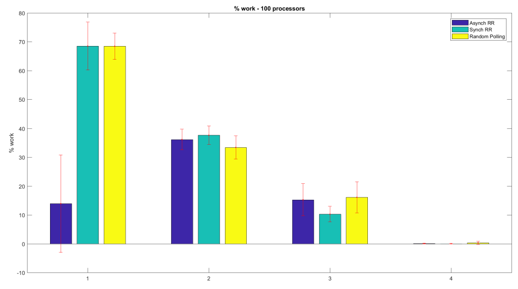

Figure 14: Percentage of work on 100 processors

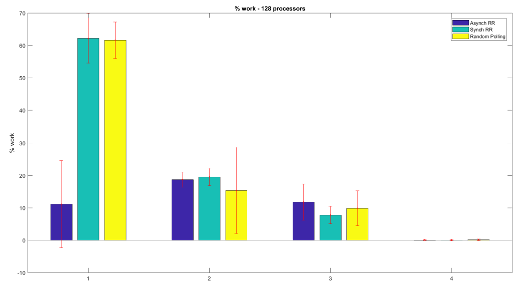

Figure 15: Percentage of work on 128 processors

6.1.4 Best solutions broadcast

This stats show the percentage of received best solution cost that are actually better than the one stored locally. This gives an idea on how much the value of the best solution cost is synchronized among the processors.

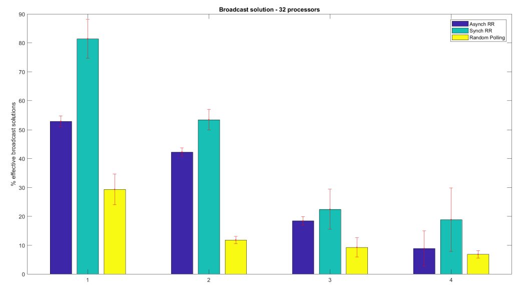

Figure 16: Broadcast best solutions on 32 processors

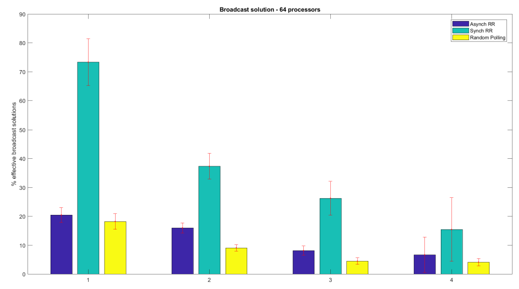

Figure 17: Broadcast best solutions on 64 processors

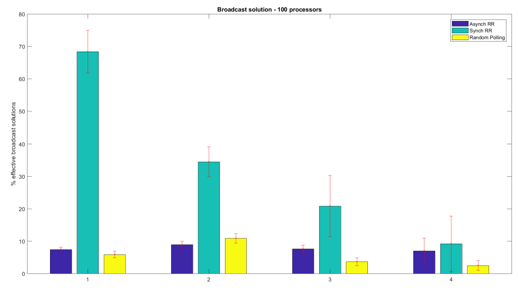

Figure 18: Broadcast best solutions on 100 processors

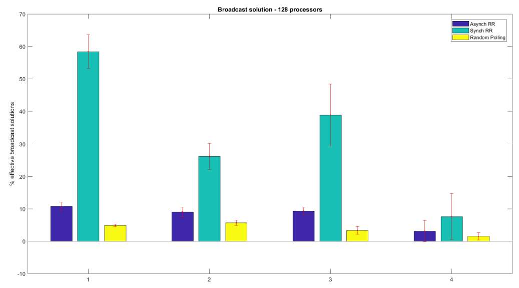

Figure 19: Broadcast best solutions on 128 processors

6.1.5 Received request satisfied

This stats show the percentage of request received that a single processor was able to satisfy over all the one received. Clearly with big graphs the percentage are higher because each processor has more work that can be split. The two Round Robin seems to be more efficient than the random polling in every case. But the variance of the synchronous round robin is greater of the one of asynchronous showing that in our cases is more effective for the balancing of the load.

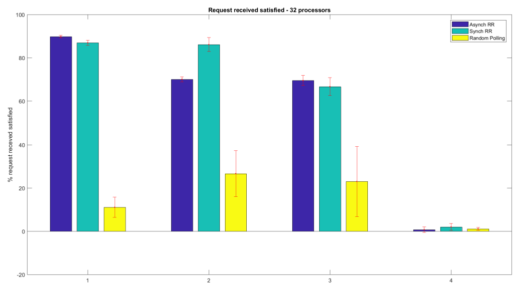

Figure 20: Received request satisfied on 32 processors

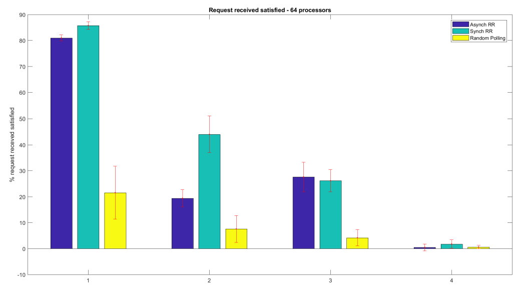

Figure 21: Received request satisfied on 64 processors

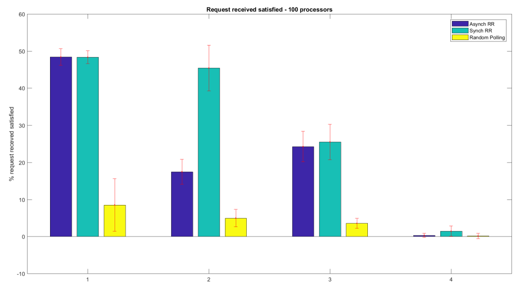

Figure 22: Received request satisfied on 100 processors

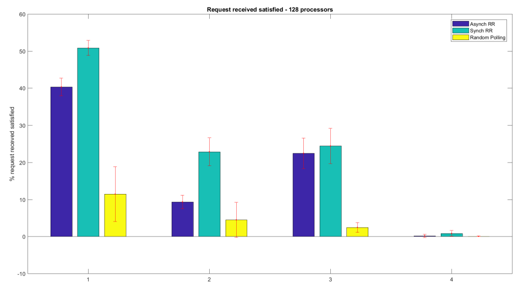

Figure 23: Received request satisfied on 128 processors

6.1.6 Sent request satisfied

This stats show the percentage of request of work that a processors sent and have been satisfied. The comments are very similar to the one of the previous section.

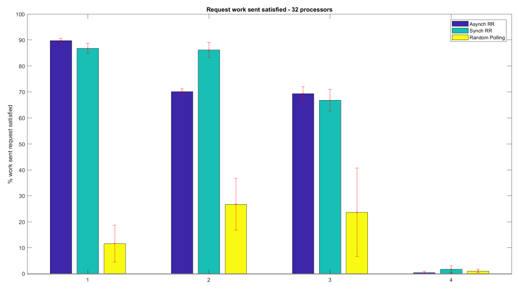

Figure 24: Sent request satisfied on 32 processors

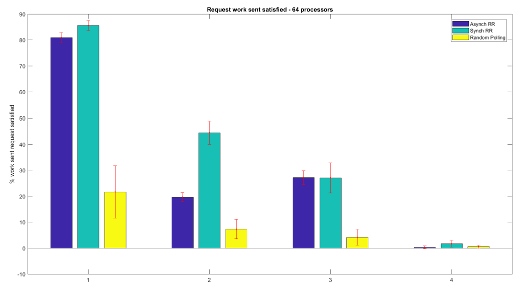

Figure 25: Sent request satisfied on 64 processors

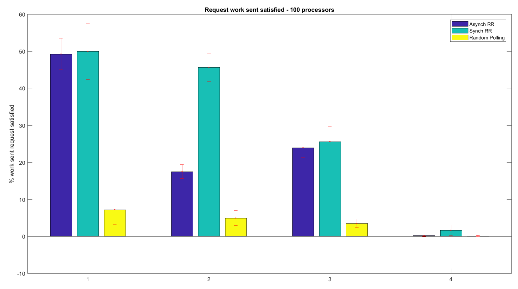

Figure 26: Sent request satisfied on 100 processors

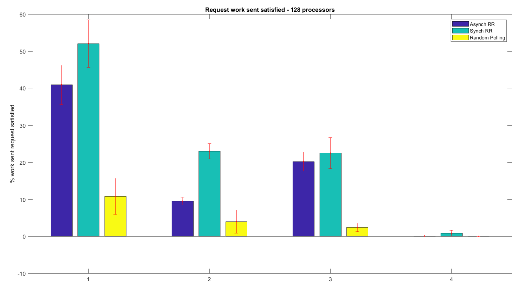

Figure 27: Sent request satisfied on 128 processors

6.1.7 Token recived

This stats show the average number of token received. This gives an idea about the overhead due to the termination detection algoritm that is linear in the number of processors. The asynchronous round robin policy in case of Fri26 shows an anomaly where the number of token exchanged is very high compared to the other cases.

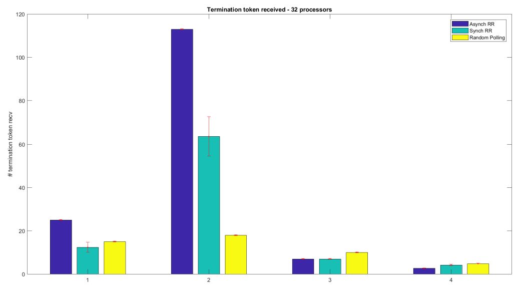

Figure 28: Token recived on 32 processors

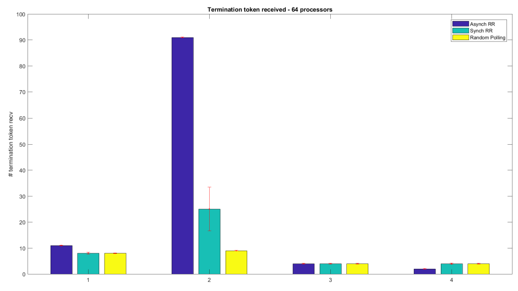

Figure 29: Token recived on 64 processors

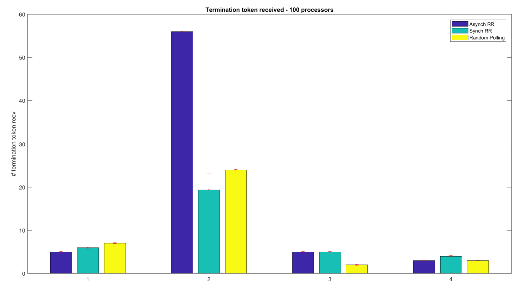

Figure 30: Token recived on 100 processors

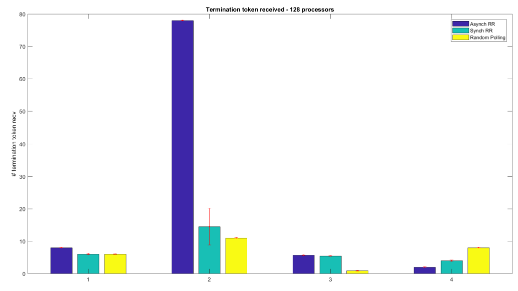

Figure 31: Token recived on 128 processors

6.1.8 Total number of reqeusts sent and requested

This stats show the average number of total request of work that are sent and received during the execution of the network. This stats give a rough idea on the overhead due to communication. As expected from the parallel run time analysis in case of random polling policy the number of requests is much higher. This means that before finding the *right donor* each processor have to send a lot of request. The uniformity given by the random function is not useful since the exploration space is not uniform due to the bounding function.

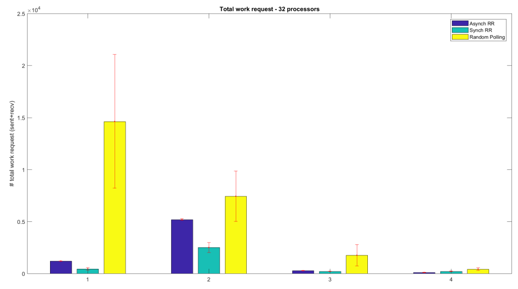

Figure 32: Total work on 32 processors

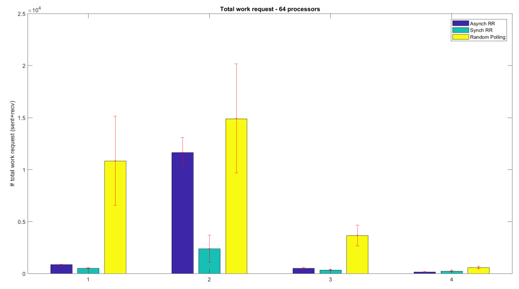

Figure 33: Total work on 64 processors

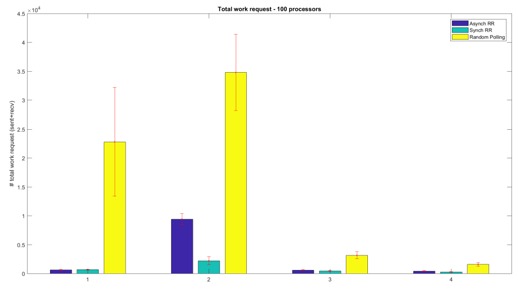

Figure 34: Total work on 100 processors

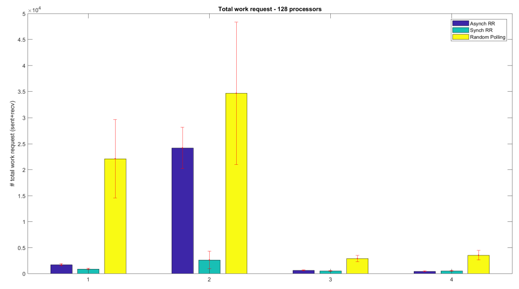

Figure 35: Total work on 128 processors

7 Lessons

This assignment has been particular interesting since the problem to solve wasn't trivial.
The TSP problem is common problem that can be solve in many different way even if using only one algorithm (i.e. DFS).
The implementation, as happened in the previous assignments, helped a lot to understand how the algorithms work.
Moreover the assignment has also been useful to learn more functions call of MPI library and their behavior.