Solving the PFSP's Benchmarks using Grid: An Experimental Study
Samia Kouki and Mohamed Jemni
Pages 1 – 8
DOI: http://dx.doi.org/10.21700/ijcis.2017.xxx

Solving PFSP 's Benchmarks Using Grid: An Experimental Study

[1]Samia Kouki **[2]Mohamed Jemni**
Laboratory LaTICE, National Graduate School of Engineer of Tunis
University of Tunis, Tunisia
[1]samia.kouki@esstt.rnu.tn
[2]mohamed.jemni@alecso.org.tn

Abstract: *The Permutation Flow Shop Problem (PFSP) is one of the most known problems of the combinatorial optimization problems. Its resolution by exact methods is NP-hard and many instances are still not resolved until now. In previous works, we proposed a first parallel algorithm based on the paralelisation of the branch and bound core of the resolution algorithm then an improvement of this algorithm by introducing a load balancing mechanism of the tasks allocated to different processors participating in the execution of the application and, therefore, minimizing the whole execution time. In this paper, we will present briefly these two algorithms, then we will present an exhaustive experimental study of them using the famous benchmarks of Taillard and the Ftrench Grid environment GRID5000. We will prove clearly the benefit of the parallel processing and how the load balancing method improves significantly the running time of the application.*

Keywords: *Grid computing, Permutation Flow Shop problem, Branch and Bound, load balancing.*

Received: *July 30, 2016* | **Revised**: *December 10, 2016* | **Accepted**: *Janaury 05, 2017*

1. Introduction

In this paper, we are considering the permutation flow shop scheduling problem (PFSP). It is a well known problem of combinatorial optimization and has many applications in the industrial and production fields. The PFSP is a particular scheduling problem of jobs with a strict order to satisfy by all operations performed on all jobs.

In the PFSP, we are given a set of N jobs $\{J_1, J_2, \ldots, J_N\}$ to run on a set of M successive machines $\{M_1, M_2, \ldots, M_M\}$ in the same order, where job j takes $p_{i,j}$ units of time to be performed by the machine i. Considering the makespan criterion, in the PFSP, we have to give an optimal schedule for N jobs on M machines, which minimized the total completion time of the schedule (makespan). Thus, some hypotheses must be considered for solving the PFSP:

- Each job must go through all the machines in exactly the same order and the job order must be the same on each machine.
- Each machine can process at most one job at any point of time and each job may be processed on at most one machine at any time.
- The objective is to find a schedule minimizing the maximal job completion time.

This problem is strongly NP-hard whenever $M \geq 3$ [13] and even the significant increase in computer power can generate only a very small increase in the size of instances to solve. In particular, exact solving methods and algorithms, for hard PFSP data instances, have an exponential computational complexity. According to the notation presented in [4] by Pinedo this problem is denoted $F|prmu|C_{max}$.

Two main not mutually conflicting approaches which can solve hard data instances of the PFSP in a reasonable time, are approximate methods and parallel methods. Approximate methods give the approximate solution of the problem, and require a personalized adjustment of the method to the problem to solve. As presented in our paper [5], we introduced a parametric study of a parallel Genetic Algorithm (GA) to solve the PFSP. An extensive experimental study was presented and we proved the efficiency of the proposed settings of the GA in finding better quality solutions for the Taillard [2] benchmarks. Besides, approximate methods can be more efficient when handset with parallel solving methods.

Exact methods give optimal solutions for combinatorial optimization problems. The Branch and Bound (B&B) is one of the most known method used to solve the PFSP, but for only small size instances. Parallel computing is a suitable alternative

to solve such kind of problems and may resolve many instances that are not yet solved.
In this context, we present in this work an experimental study considering two parallel algorithms (*GAUUB* and *GALB*) which were presented in our previous publications [11, 12]. We proposed these two algorithms to solve hard instances of the literature using the powerful parallel techniques and environments.

The rest of the paper is structured as follows. Next section presents details of our previous works. Section 3 provides a detailed description of our distributed algorithm called GALB, in which we present the principal of the algorithm and an example of its execution. Section 4, is dedicated to present an experimental study reporting a comparative evaluation of the performance of both GAUUB and GALB algorithms using best known benchmarks. Finally, some concluding remarks are provided.

2. Our Previous Works

In previous papers, we proposed two parallel branch and bound algorithms to solve a selected set of instances from the Taillard benchmarks. The *GAUUB* algorithm which was presented in [11], is a parallel algorithm based on updating the upper bound value in order to make the algorithm converging rapidly to the optimal solution. Our parallelization strategy was based on dividing the PFSP into many sub-problems to be performed independently and simultaneously by different processors.

The experimentation of *GAUUB* algorithm, as presented in [9, 10, 11], shows unfortunately that when several nodes are pruned from the first level of the search tree, the use of parallelism is not very significant. However, when the processors share the workload that is normally assigned to a single processor, we can achieve reasonable running times for most instances. Based on such observations, we propose another algorithm called Grid Algorithm with Load Balancing (*GALB*) [9, 10], in order to correct the unbalancing load of processors in the *GAUUB* algorithm.

In fact, the *GALB* algorithm is an improvement of our previous algorithm (*GAUUB*) by establishing an efficient load balancing technique between all available processors. This algorithm is based on the master/slave paradigm and is composed of two main steps. The first step is done by the master processor and is performed in serial until a level L of the search tree.
The goal of this step is to generate a large amount of work (many nodes to explore) to distribute among the slaves processors and then to guarantee a load balancing between all processors. The master assigns the unexplored nodes to the processors, ordered by the ranks of processors. Then all processors execute a local *B&B*.
Furthermore, the second step is accomplished in parallel, by all the available processors (slaves). Each processor will give the new solution to the master in order to update the global Upper bound.
Once a processor completes the task assigned to it, it requests another task from the master and so on. Indeed, in the algorithm *GALB*, we note that the processors finish at almost the same time and therefore, all processors participate in solving the problem during almost the same duration. The use of the *GALB* algorithm gives two advantages, first distributing the load across the processors and second reducing the execution time, which may allow us to solve more complicated and hard instances of data.

3. Our Distributed Algorithm GALB

3.1 Principal

There is a vast literature on load balancing for optimizing the performance of distributed and parallel systems. In fact, achieving moderate CPU time in solving hard and exponential problems such as optimization problems can be reached by equalizing the loads of available resources. In this context, our *GALB* algorithm is based on an efficient technique to distribute the load to the available processors with a more equitable manner [6, 7, 8]. Whatever, our previous algorithm (*GAUUB*) has already yielded good results as reported in our papers [11, 12], it cannot give good results for big sizes instances.

The quality of a *B&B* algorithm applied to a specific problem (i.e. PFSP in this work) depends on several parameters and configurations. Namely, the branching rule, the lower bounding, the search strategy and the termination condition. Indeed, these criteria have a strong impact on the quality of produced solutions and the CPU time needed to achieve them. The branching rule consists to divide the set of feasible solutions into subsets. The lower bounding is an algorithm which computes a lower bound (*LB*) in each node of the search tree (node: sub problem generated by the branching scheme). A stronger bound eliminates relatively more nodes of the search tree. However, if its computational requirements turn excessively large, it may become advantageous to search through larger parts of the tree, using a weaker but more quickly computable bound. The third component is the search strategy that selects a node from which to branch. For instance, three main selection strategies are: Depth first, Best first and Breadth first.

Termination conditions are the same for the basic B&B algorithm (all the branches of the search tree have been explored or pruned).
The first part of our algorithm involves performing the classical *B&B* algorithm sequentially, while considering the level L as the last level of the search tree. This initial part of the algorithm produces a large number of nodes that will be distributed to slaves.

(1) Initialization Step:
Initially, the master has the parameters of the problem to solve; the number of jobs (*N*) the number of machines (*M*), the processing times of all the jobs in all the machines (p_{ij}) and the initial Upper Bound (*UB*).
Based on the parameters mentioned later, the master performs the *B&B* algorithm until reaching the level *L* of the search tree. The value of the parameter *L* (equal to 5) was established experimentally in preliminary runs on some Taillard benchmarks.
It should be noted that, in this algorithm, we used the strategy of Depth first search. This strategy Depth first search (DFS) starts at the root and explores as far as possible along each branch before backtracking (see Figure 1).

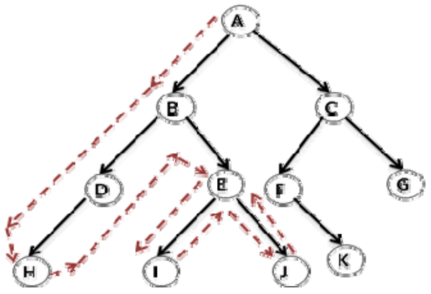

Figure 1. The Depth First Searh (DFS)

It is assumed, in our algorithm, that the master processor executes the backtrack function at the level *L*, before reaching the last level of the search tree.
In this step of our algorithm, we have, first, to prune the unneeded nodes of the search tree and to avoid the costs of communication between the different processors, second, to generate a large amount of work (many nodes to explore) to distribute among the slaves processors and therefore to warranty a load balancing between all processors.
The nodes generated in the first sequential step are stored in a particular data structure that facilitates its dispatching through the slaves processors. The master assigns the unexplored nodes to the processors, ordered by their ranks.

(2) Distributed step (performed by slaves)
The second step of our algorithm is performed by all available slaves.
All slaves execute a *B&B* algorithm to explore the assigned node. As soon as a processor finds a feasible or optimal solution, it communicates it to the master. The master checks whether the solution is better than the current solution (New Upper Bound (*NUB*) < Current Upper Bound (*CUB*)) and updates it. After that, the master affects the next node of the list to the free processor. And so on, until the list will be empty and all the workers become free.
Using this strategy, the processors finish their work at nearly the same time.

3.2 Example of Solving Instance using GALB

To describe in details the main steps of our *GALB* algorithm, we present in this section an example of Taillard instances (Tail074 with 100 jobs and 10 machines) performed by *GALB* using 5 processors. We present as follows the manually execution of this instance.

(1) 1st step
This step is performed in sequential until reaching the level 5 of the search tree. The number of tasks obtained in this level is equal to 74.
For more details, we present the rest of this step by following 6 different stages. Figure 2 shows the progress of the first sequential step performed by the master processor. We do not present all the search tree branches (the more we advance in the levels of the search tree, the more the number of branches increases). Once at level 5, we do the backtracking step to ensure that all nodes of the tree have been explored. In this step, level 5 is considered as the last level of the *B&B* search tree. All generated solutions by this sequential step will be used by all available processors in the next parallel step. In our example (instance Tail047), many nodes have already been pruned before the 5th level of the tree. It may therefore penalize the active processors that will continue alone solving the instance until the last level of the tree. This problem can be solved if the load of all processors is distributed fairly among all available processors.

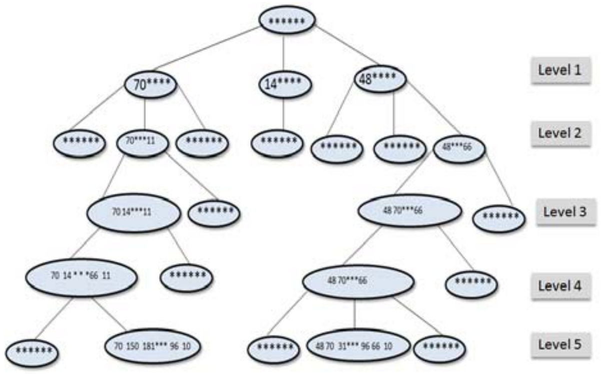

Figure 2: Sequential step of resolution of Tail047

(2) 2nd step: store resulting nodes of the sequential step in a list *L* (Figure 3).
In this stage we store all the nodes that were generated during the first sequential step done by the master processor. The nodes are classified in a list (*list-jobs*) and affected to slave processors in order to be explored at the last level of the local search tree. In this example, the first step has generated 74 different

tasks that will be distributed among all different slave processors.

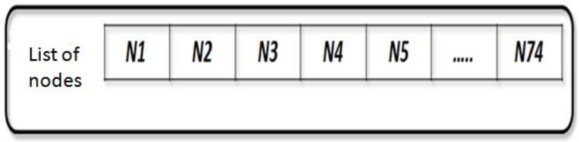

Figure 3: List of nodes to be explored

(3) 3rd step: Assign nodes to slave processors according to their rank (Figure 4).
The objective of this step is to distribute a node to each slave processor to explore. The assignment of different nodes to different processors is done according to their ranks.

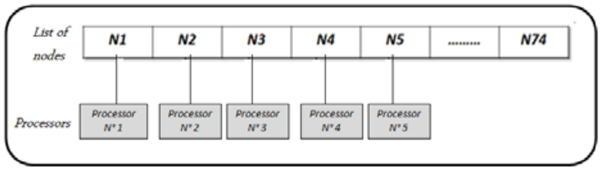

Figure 4: Assign nodes to processors

(4) 4th step: each processor applies locally the algorithm *B&B* on the node assigned to it (Figure 5).
During this stage, each processor applies locally the algorithm *B&B* on the node to explore. This is a parallel scanning of the nodes of the search tree generated during the previous step. The parallel exploration of the set of nodes is done by all available processors independently since there is no new updates of the upper bound.

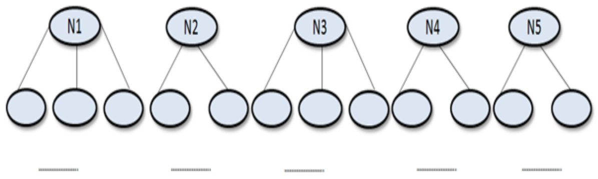

Figure 5: Assigning nodes to processors

(5) 5th step: Explore the node by a slave processor (Figure 6).
This is a step that depends on the completion of the exploration of a node by a processor.
In Figure 6, we can notice that once the processor *N°2* finished solving the node number 2 (N_2), it asks the master processor for a new task to solve.

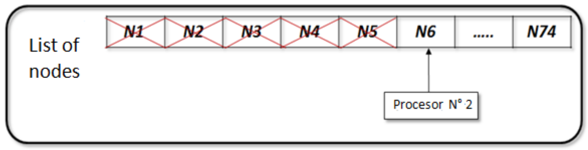

Figure 6: Assigning node N_6 to processor *N°2*

(6) 6td step: Similarly, the processor *N°2* applies the algorithm *B&B* locally to the new node N_6. The algorithm stops when all the nodes of the list *L* (*list-jobs*) will be explored and all processors become inactive.

4. Experimental Study

In this section, we present the main results we obtained from an exhaustive experimentation study we conducted on the French Grid platform *GRID 5000* [14]. We performed our experimental tests on the well-known set of Taillard benchmark instances [2]. These instances are known to be very hard and many of them are still unsolved. Our objective is to present a complete comparative study of the two algorithms previously presented in previous section *GAUUB* and *GALB* solving a set of instances from the Taillard benchmarks.
Remark here, that in our paper [4], we presented some short results we obtained from our earliest experimentations concerning *GAUUB* versus *GALB*. Our aim has been to illustrate the importance of load balancing improving the performance of the algorithm. In this paper, we present a complete experimental study. We proceeded progressively; we vary the instances and for every instance, we run series of tests using different number of processors. We implemented our algorithm with C and we used MPI library (Massage Passing Interface) [3] in order to ensure communication between processors. The instances we used are: 20X5, 20X10, 50X5, 50X10, 100X10, 200X10 and 200X20, the number of processor varies from 10 to 200. We choose these instances of data because they still include some unsolved problems. For instance, all instances with 5 machines (xX5) have been almost solved to optimality. For the 20 machines instances (xX20), some instances have been solved by *GALB* algorithm.
In the next subsections, we present a comparative study of *GAUUB* and *GALB* following CPU Time, scalability and load balancing.

4.1 Running Time

(1) 20 X10 instances

This subsection presents processing times of the 20X10 instances using different number of processors varying from 10 to 200.
From Table 1, we can remark that for few 20X10 instances, CPU time with 10 processors, given by *GAUUB* are slightly better than those given by GALB (see Tail011 and Tail012 in Table 1). Furthermore, when the number of processors increases, the execution time decreases significantly with *GALB*.
We specify that all run times are in seconds.

Table 1. *GAUUB* Vs *GALB* : solving 20X10 instances

	10 Processors	
Instance	***GAUUB***	***GALB***
Tail011	315	373
Tail012	15	9
Tail013	42	50
Tail014	17	9
Tail015	11	6
Tail016	0	1
Tail017	unsolved	7598
Tail018	unsolved	23
Tail019	~ 0	~ 0
Tail020	85	35

We note however that two instances (Tail017 and Tail018) among 10, were resolved by *GALB* and not by *GAUUB.*

(2) 50 X10 instances

In Figure 7, we present the CPU times for solving 50x10 instances with both *GAUUB* and *GALB* using 50 processors.

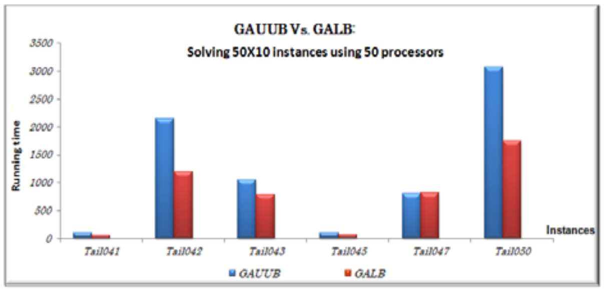

Figure 7. *GAUUB* Vs *GALB* : solving 50X10 instances using 50 processors.

The Tail050 instance was resolved to optimality, only by *GALB*. In Figure 7, we see more clearly that *GALB* outperforms *GAUUB* in resolving all 50X10 instances. This is thanks to the efficient load balancing strategy used by the *GALB* algorithm.

(3) 100 X10 instances

This subsection is devoted to compare the execution times for 100x10 instances.

Table 2. *GAUUB* Vs *GALB* solving 100X10 instances

	10 Processors		50 Processors		100 Processors	
Instance	***GAUUB***	***GALB***	***GAUUB***	***GALB***	***GAUUB***	***GALB***
Tail073	unsolved	unsolved	unsolved	9125	unsolved	4850
Tail075	14	10	12	10	8	5
Tail077	unsolved	8789	unsolved	3620	unsolved	1580
Tail080	2354	1531	984	957	358	736

In Table 2, we note that the Tail073 instance could not be solved with *GAUUB* even with 100 processors, whereas using only 50 processors, *GALB* could solve it to optimality. Similarly, Tail077 instance, could not be solved with *GAUUB* (even with 100 processors), however using *GALB*, we have reached optimal solution with only 10 processors.

In Figure 8, we present CPU times on the instance Tail080 (10x10) performed by both *GAUUB* and *GALB*, by varying the number of processors from 10 to 50 then to 100.

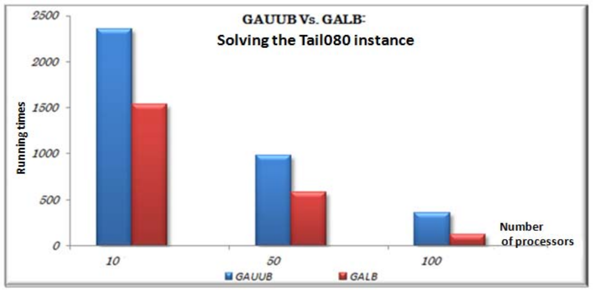

Figure 8. *GAUUB* Vs *GALB* : solving the Tail080 instance

We notice that all execution times of Tail080 given by *GALB* are better than those of *GAUUB* regardless of the number of processors used.

(4) 200X10 benchmark instances

This subsection is dedicated to running time's comparative study of 200x10 instances given by *GAUUB* and *GALB*, using, 10, 50, 100 and 200 processors.

In Figure 9, we see that, using 10 processors, *GALB* gives always better runtime than *GAUUB*. For example, for Tail098 instance, the running time given by *GALB* are significantly reduced compared with those given by *GAUUB* (less than the half).

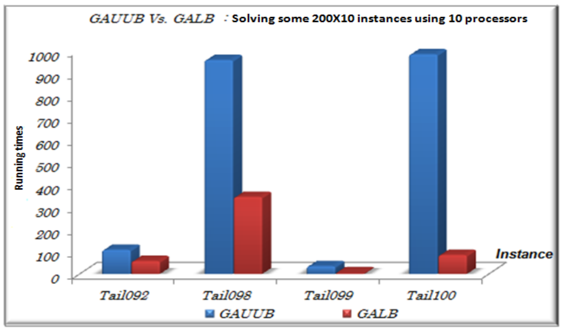

Figure 9. *GAUUB* Vs *GALB* : solving some 200X10 instances using 10 processors

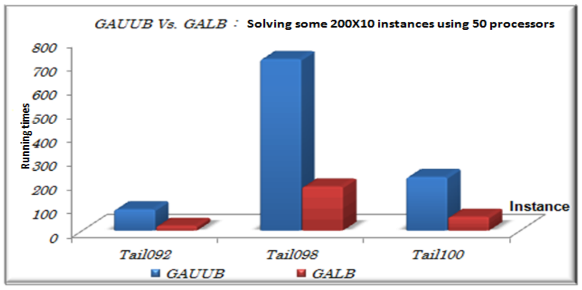

Figure 10. *GAUUB* Vs *GALB* : solving some 200X10 instances using 50 processors

Using 50 processors (see Figure 10), CPU time given by *GALB* divided by three, in average, those given by *GAUUB*. These results confirm the importance of providing the same workload to all available processors in order to reach more quickly the optimal solution (*GALB* principal).

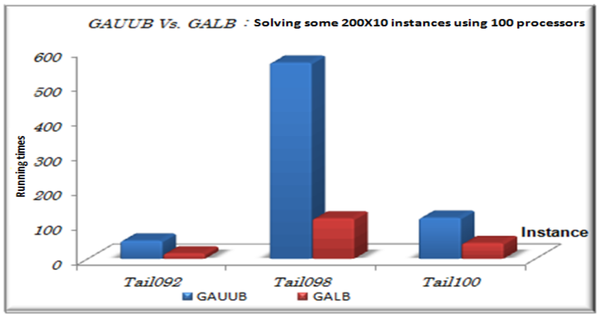

Figure 11. *GAUUB* Vs *GALB* : solving some 200X10 instances using 100 processors

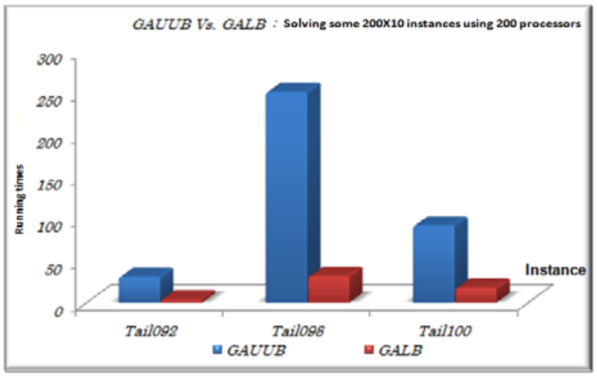

Figure 12. *GAUUB* Vs *GALB* : solving some 200X10 instances using 200 processors

Indeed, *GALB* ensures good regular updates to the upper bound which are resulting on a dynamic distribution workload among different processors. Both Figure 11 and 12 confirm the efficiency of *GALB* comparing to *GAUUB* when the number of processors increases (from 10 to 100, then to 200 processors).

For the Tail098 instance (Figure 12), we can notice that the execution time on *GAUUB* is about 3 times the CPU time given by *GALB*. Similarly, for the two instances (Tail100 and Tail092), we can notice that CPU times given by *GALB* are always better than those by *GAUUB*.

(5) 200X20 benchmark instances

This subsection compares the performance of *GAUUB GALB* to solve some 200 jobs and 20 machines instances. Only two instances among 10 were resolved by *GAUUB* (Tail105 and Tail106). We present in Figures 13, 14, the execution time for these two instances given by both *GAUUB* and *GALB* by varying the number of processors from 10, 50, 100 to 200.

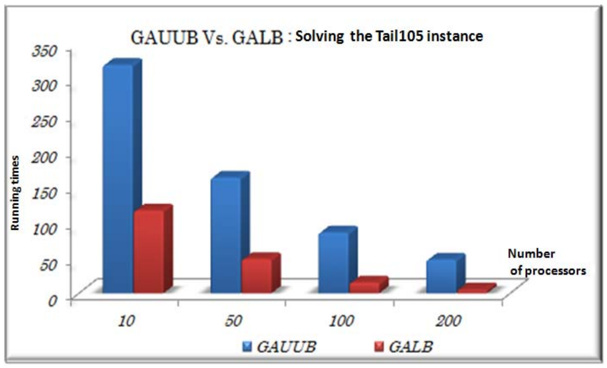

Figure 13. GAUUB Vs GALB : solving Tail105 instance

We notice from Figure 13 and 14 a clear difference between CPU times given by *GALB* and those given by *GAUUB* to solve Tail105 and Tail106 instances.

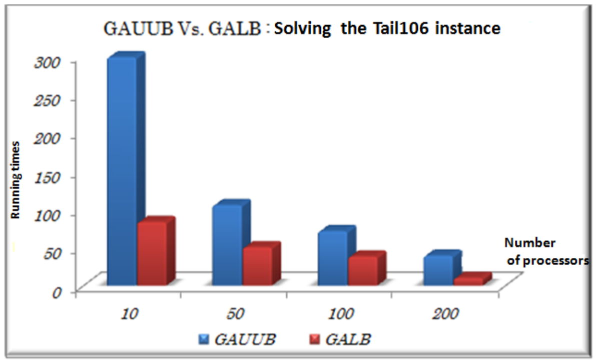

Figure 14. *GAUUB* Vs *GALB* : solving Tail106 instance

4.2 Scalability

We can conclude from the previous experimental study that only *GALB* is scalable. Indeed, with *GALB* we can vary the number of processors, while ensuring good accelerations and efficiencies. However, for *GAUUB* algorithm, the number of processors to use depends a lot on the size of the instance to solve and cannot exceed the number of tasks to be scheduled. This constraint on the number of processors makes *GAUUB* limited in terms of scalability and the possibility to move to large scale parallelism.

4.3 Load Balancing

In this section, we present a comparative study that focuses on load balancing of both *GAUUB* and *GALB* algorithms. In order to compare the performance of the two algorithms, we performed our experiments in the same parallel environment.

(1) 20X10 benchmark instances

We present in Figure 15, the distribution load of 10 processors of *GALB* in solving the instance Tail017. We note that this instance has not been resolved by *GAUUB* with 20 processors. As presented in Figure 14, we can remark that all processors (9 slave processors) have completed their work almost at the same time and the execution time of the processors varies between 7450 and 7600 seconds.

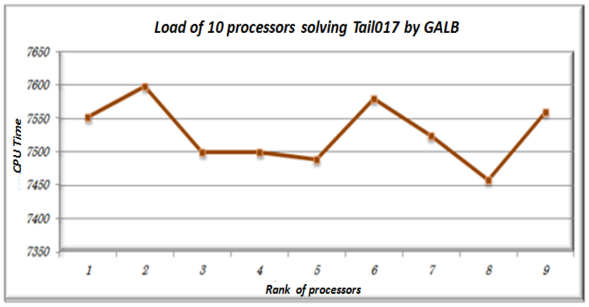

Figure 15. Load of 10 processors solving Tail017 by *GALB*

(2) 50X10 benchmark instances

In this subsection, we present a comparison of load distribution of 50 processors of both *GAUUB* and *GALB* in solving the instance Tail050. This instance was resolved with *GAUUB* (see Figure 16) after 3065 seconds, however *GALB* (see Figure 17) spends only 1748 seconds to reach the optimal solution. Figure 16 shows that with *GAUUB* only 5 among 50 processors remain active until the end of the execution of the instance Tail050. However, the rest of processors become inactive after a few seconds or have running time equal to zero (processor 2, processor 4, processor 6, processor 7, 8 processor, etc ..).

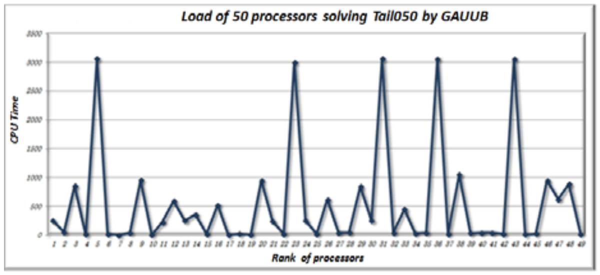

Figure 16. Load of processors solving Tail050 by GAUUB

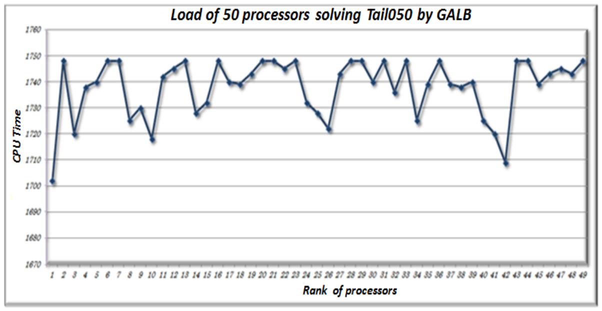

Figure 17. Load of 50 processors solving Tail050 by *GALB*

(3) 200X10 benchmark instances

To compare both *GAUUB* and *GALB* load balancing performances, we present in this subsection the workload relative to the two algorithms to solve some 200x10 instances. We present here only the execution of the Tail091 instance (see Figure 18 Figure 19) using 50 processors.

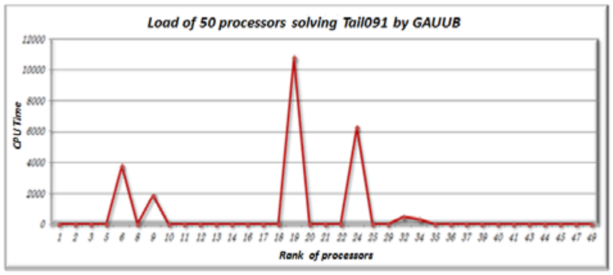

Figure 18. Load of 50 processors solving Tail091 by *GAUUB*

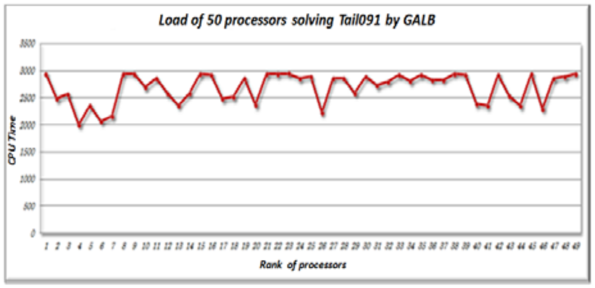

Figure 19. Load of 50 processors solving Tail091 by *GALB*

We notice the same results as we did for previous presented instances. For *GALB*, we have usually a better load balancing of different processors and, therefore, better execution time. Indeed, for *GALB*, when a processor completes the exploration of its portion of the search tree, he asked the master for new tasks to explore. This leads to a better distribution of the workload between processors and improves the performance of *GALB*. However for *GAUUB*, since no load balancing strategy has been adapted, many processors may become inactive in the early levels of the search tree and only one or a few processors achieve the exploration of the tree until reaching optimal solution.

5. Conclusion and Future Work

In this paper, we presented an exhaustive experimentation study of two parallel algorithms to solve the PFSP. The first one called *GAUUB* is based on dividing the PFSP into many sub-problems to be performed independently and simultaneously by different processors. In the second algorithm, called *GALB*, we introduced a load balancing approach in order to make all processor participating in the execution of instances finishing at almost the same time. Also, compared to *GAUUB*, *GALB* divides the search tree into a very big number of sub-problems in order to obtain fine granularity parallel tasks. When a processor finishes the execution of one task, it asks the master to have another task to execute and so on. All the empirical study has been carried on the French grid *GRID 5000*. We achieved a huge number of experiments by running the two algorithms applied to a big number of instances of Taillard benchmarks. All experimentations proved that *GALB* is better than *GAUUB* in term of execution time and Scalability thanks to its load balancing approach. *GALB* allowed to obtain very good performance and also to resolve new instances not resolved before thanks to the computing power of grid and parallel computing i.e. Tail108, Tail109 and Tail110 (200 jobs with 20 machines).

Furthermore, due to its NP-hardness, it is still expensive or sometimes impossible for some large size instances of PFSP to achieve optimal solutions even when we use *GALB* algorithm. Thus, in our future research directions, we will consider approximate methods such as metaheuristics algorithms (Genetic algorithm, Mimetic algorithm) to find good quality solutions for hard data instances.

References

[1] B. Le Cun, T.G. Crainic and C. Roucairol, “Parallel combinatorial optimization, chapter Parallel Branch and-Bound algorithms,” Wiley, John & Sons incorporated, 2006.

[2] E. Taillard : Scheduling instances (2008), http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html,

[3] LAM/MPI parallel computing. Online document at http://www.lam-mpi.org/, last visited on June 30, 2011.

[4] M. Pinedo, Scheduling: Theory, Algorithms and Systems, Springer, Fourth Edition, , 686 pages, 2012.

[5] S. Kouki, M. Guenaoui, M. Jemni, A genetic algorithm for the Permutation Flow Shop Problem : a parametric study, in Proceedings of the International Symposium on Networks, Computers and Communications, IEEE, Hammamet, Tunisia, May 2016.

[6] S. Kouki, M. Jemni and T. Ladhari, A parallel algorithm for solving the PFSP with load balancing on the grid: An empirical study, in *Proceedings of the 5th International Conference on Information & Communication Technology and Accessibility (ICTA'15)*, Marrakech, Morocco, December 2015.

[7] S. Kouki, M. Jemni, and T. Ladhari, Scalable Distributed Branch and Bound for the Permutation Flow Shop Problem, in *Proceedings of the 10th international conference on P2P, parallel, grid, cloud and internet computing 3PGCIC,* Compiegne, France, October 2013, pp. 503-508.

[8] S. Kouki, M. Jemni, and T. Ladhari, A distributed algorithm for the Permutation Flow Shop Problem-An empirical analysis, In *Proceedings of the International Conference on Parallel Computing (Parco 2013),* Munich, Germany, September 2013, pp. 451-460.

[9] S. Kouki, M. Jemni, and T. Ladhari, A Load Balanced Distributed Algorithm to Solve the Permutation Flow Shop Problem Using the Grid, in *Proceedings of the 15th IEEE International Conference on Computational Science and Engineering*, Paphos, Cyprus, December 2012, pp. 146 - 153.

[10] S. Kouki, M. Jemni, and T. Ladhari, New Load Balancing Strategy for Solving Permutation Flow Shop Problem Using Grid Computing, in *proceedings of the 12th International Conference on Parallel Problem Solving From Nature,(PPSN'12)*, LNCS, Springer, Taormina, Italy, September 2012.

[11] S. Kouki, M. Jemni, and T. Ladhari, Solving the Permutation Flow Shop Problem with Makespan Criterion using Grids, *International Journal of Grid and Distributed Computing*, vol. 4, n° 2, pp. 53-64, 2011.

[12] S. Kouki, M. Jemni, and T. Ladhari, Deployment of Solving Permutation Flow Shop Scheduling Problem on the Grid, in *Proceedings of the International Conference on Grid and Distributed Computing (GDC'10)* Jeju Island, Korea, LNCS, Springer, December 2010, pp. 95-104.

[13] S. M. Johnson, Optimal two- and three-stage production schedules with setup times included, *Naval Research Logistics Quarterly*, Vol. 8, pp. 1-61, 1954.

[14] www.grid5000.fr