A list scheduling algorithm for scheduling multi-user jobs on clusters

J. Barbosa[1] and A.P. Monteiro[1,2]

[1]Universidade do Porto, Faculdade de Engenharia,
Departamento de Engenharia Informática
[2]INEB - Instituto de Engenharia Biomédica, Lab. Sinal e Imagem
Rua Dr. Roberto Frias, 4200-465 Porto, Portugal
e-mail: {jbarbosa,apm}@fe.up.pt

Abstract. This paper addresses the problem of scheduling multi-user jobs on clusters, both homogeneous and heterogeneous. A user job is composed by a set of dependent tasks and it is described by a direct acyclic graph (DAG). The aim is to maximize the resource usage by allowing a floating mapping of processors to a given job, instead of the common mapping approach that assigns a fixed set of processors to a user for a period of time. The simulation results show a better cluster usage. The scheduling algorithm minimizes the total length of the schedule (*makespan*) of a given set of parallel jobs, whose priorities are represented in a DAG. The algorithm is presented as producing static schedules although it can be adapted to a dynamic behavior as discussed in the paper.
Keywords: static and dynamic scheduling, parallel task, list scheduling, cluster computing.

1 Introduction

The aim of the work herein presented is to improve the performance of clusters in the processing of applications (or jobs) composed by a set of dependent tasks. The common scheduling approach is to consider a fixed number of available processors to schedule the set of tasks [13, 14, 17, 18, 20] which on a multi-user environment corresponds to fix the number of processors available for each user. The presented model is based on a former model [2] to schedule DAGs of dependent parallel tasks. A parallel task, also called malleable task, is a task that can be executed on any number of processors with its execution time being a function of the processors alloted to it [7, 12, 15].

The target computer platform is a cluster, either homogeneous or heterogeneous, with a dedicated network. Such clusters can be private clusters of some organization but also they can be the nodes of a Grid infrastructure which to have a good performance requires, at least, that the end clusters have also a good performance.

The cluster schedulers usually allow that users specify the number of processors required to run their jobs, which imposes a static allocation of the cluster

nodes and a non-unified cluster management based only on user requests. Users try to allocate as much capacity as possible and there is not a global management. The proposed algorithm intends to optimize the cluster utilization by allowing different number of processors to be used along the processing of the tasks of a given user job. The algorithm is non-preemptive and achieves the goal by considering different number of processors to process the tasks of a given job. The DAG to schedule has two levels of detail. There is a master DAG, that establishes priorities among user jobs, and there is a DAG for each job. Figure 1 exemplifies a typical DAG. The scheduler input is the global DAG and all ready tasks are considered to schedule at a given time.

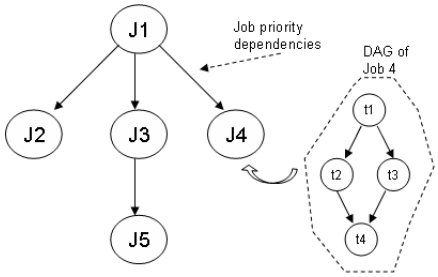


Fig. 1. A DAG composed by jobs of different users where job dependencies are created by priority policies; each job is described by a DAG of dependent tasks

The common approach to schedule DAG's is the task parallel paradigm, which assigns one task to one processor. The scheduling consists on the distribution of the DAG nodes among the machine nodes, so that the *makespan* is minimum [1, 11, 18–20]. Here it is considered the parallel task model where each task can execute in more than one processor but one processor only participates in the execution of a task at any given time [7, 12, 21].

The remaining of the paper is organized as follows: section 2 defines the scheduling problem and revises related work in scheduling parallel and non-parallel tasks. Section 3 presents the computational model and the methodology used in this paper. Section 4 presents the list scheduling algorithm proposed in this paper. Finally, section 5 presents results and section 6 conclusions.

2 Problem definition and related work

The problem addressed in this paper is the scheduling of a parallel application represented by a directed acyclic graph (DAG) on a distributed memory computer (i.e. a cluster). A DAG $G = (V, E)$, where V is the set of v nodes of the graph, representing the tasks to be processed, and E is the set of e edges, representing precedence among tasks and communication costs. For each node v_i it is defined a schedule start-time ($ST(v_i)$) and a finish-time ($FT(v_i)$), being

the schedule length given by $max_i\{FT(v_i)\}$. Therefore, the goal of scheduling is to minimize $max_i\{FT(v_i)\}$ [13].

The above definition is valid either for homogeneous or heterogeneous machines and either for parallel tasks (executed on several processors) and non-parallel tasks (executed on one processor). The existing work on scheduling parallel tasks deals almost exclusively on homogeneous computers, and either dependent or independent tasks. The problem is known as NP-Complete so that several authors proposed polynomial approximation schemes [6, 7, 12, 15, 16, 21].

The problem studied here considers the scheduling of general task dependency graphs and both homogeneous and heterogeneous clusters. Tasks are considered parallel and non-monotonic, this is, the execution time of task i, $t_{i,p}$, is considered to be non-monotonic so that there is a number p of processors for which $t_{i,p} < t_{i,p-1}$ and $t_{i,p} < t_{i,p+1}$. Mainly for heterogeneous clusters connected by a standard network it was shown that, due to communication constraints and task granularity, leaving processors in the idle state can reduce the processing time [3–5]. The solution proposed is based on the list scheduling technique used for non-parallel tasks [13, 18–20].

DAG scheduling is commonly addressed as a non-parallel task problem [13, 14, 17, 18, 20], therefore the algorithm proposed in this paper is compared to the Heterogeneous Earliest-Finish-Time (HEFT) algorithm [20]. The authors compared several scheduling algorithms for heterogeneous computing and conclude that HEFT is the best one for scheduling DAG's on those systems. HEFT comprises two phases: first, there is a task prioritizing phase and second, a processor selection phase that selects the processor that minimizes the task finish time. It implements an insertion based policy which considers the possibility of inserting a task in an earliest idle time slot between two already scheduled tasks on a processor. The aim of comparing to HEFT is to show that the parallel-task approach can improve significantly the performance of a cluster, heterogeneous or not, in scheduling DAGs, with a scheduling algorithm of the same time complexity as HEFT, which is $O(v^2 \times P)$ for a DAG of v tasks and a P processor machine.

The former techniques are all static approaches of the mapping problem that assume static conditions for a given period of time. A dynamic approach intends to be more flexible concerning the availability of information about tasks arrival time and machine availability. Dynamic mapping of tasks is usually addressed as an independent task scheduling [9] problem. This approach can be applied here at the job level because these are independent and our master DAG is also based on job priorities and job deadlines. The dynamic scheduling can be applied with our scheduling algorithm in the following way: a dynamic policy like [9] specifies the DAG for a given scheduling instant and our algorithm scheduled tasks based on that DAG. The DAG is updated when new jobs arrive and a schedule instant happens when there are tasks ready to schedule. However, in this paper dynamic scheduling is not considered, because it requires more research and also it would obfuscate the main comparison that is to show that a parallel task scheduling achieves better performance than a non-parallel one on a cluster.

3 Computational model

The computational platform considered is a distributed memory machine composed by P processors of possibly different processing capacities (heterogeneous cluster), connected by a switched network. It supports simultaneous communications between different pairs of machines. It is assumed that the application is represented by a DAG and the execution time of the tasks can be estimated at compile time or before starting the execution. The communications required to complete a task are included in the computation time as a function of the processors p used by that task. The inter-task communication is defined as a function of the computational time of the sender task and it is represented by the edges weight in the DAG.

The computational model that supports the estimation of the processing time, for each task, is based on the processing capacity S_i of processor i ($i \in [1, P]$) measured in $Mflop/s$, the network latency T_L, and the bandwidth ω measured in $Mbit/s$. The total computation time is obtained by summing the time spent communicating, T_{comm}, and the time spent in parallel operations, $T_{parallel}$. The time required to transmit a message of b elements is $T_{comm} = T_L + b\omega^{-1}$. The time required to compute the pure parallel part of the code, without any sequential part or synchronization time, on p processors is $T_{parallel} = f(n)/\sum_{i=1}^{p} S_i$. The numerator $f(n)$ is the cost function of the algorithm, measured in floating point operations, depending on problem size n.

As an example, for a matrix multiplication of (n, n) matrices, using the algorithm described in [10], the number of floating point operations is estimated to be $f(n) = 2n^3$. The total amount of data required to be transmitted in order to complete the algorithm on a grid of processors $P = r \times c$ is $n^2(r-1)$ across rows of processors and $n^2(c-1)$ across columns of processors, resulting in the total of $n^2(r+c-2)$ data elements. If the broadcast over a column or a row of processors is considered sequential, then they are transformed in $(r-1)$ and $(c-1)$ messages, respectively.

Finally, the time function for the matrix multiplication algorithm is given by:

$$T = T_{comm} + T_{parallel} = \frac{n^2(r+c-2)}{w} + T_L + \frac{2n^3}{\sum_{i=1}^{p} S_i} \tag{1}$$

This expression is computed for $p = 1$ to P and the number of processors that minimize the processing time is determined. The computation of the best processor grid for linear algebra kernels, on a heterogeneous machine, was discussed in [4].

4 Scheduling algorithm

The scheduling algorithm is divided in two steps: first, a construction of a master DAG where each node is a user job and each edge represents a priority of one job over another, as shown in Figure 1; and second, a list scheduling

algorithm, based on [2], that schedules the master DAG, this is tasks of all jobs in a unique DAG.

The master DAG is created based on job priorities and deadlines [9]. Here it will be assumed that the master DAG is already defined and available to be scheduled (first step). In the second step the algorithm ensures that user reservation policy is not compromised such that, for example, if a user has reserved 20% of the cluster, their jobs will be sechedule accordingly. The difference for the fixed capacity schedule is that if the user does not need that capacity at a given time, it will be available for other users. Figure 2 exemplifies the master DAG construction. If user 1 has reserved part of the machine, his tasks are put in parallel, at the top level. The tasks of other users are organized either in parallel or sequentially according to the prioritizing policy [9].

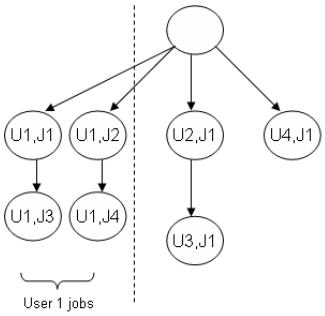


Fig. 2. Master DAG example; tasks of user 1 are organized separately to guarantee the reservation policy

The master DAG have artificial nodes in order to impose priorities among jobs with zero processing time. Jobs are independent so that the edges have zero communication costs. Communications are only considered inside each job.

The scheduling algorithm applied to the global DAG is a list scheduling technique [13] which consists in the following steps: a) determine the available tasks to schedule, b) define a priority to them and c) until all tasks are scheduled, select the task with higher priority and assign it to the processor that allows the earliest start-time. For parallel tasks the last step selects not one processor but several processors that allow the earliest start-time [2]. Note that at this step we refer to tasks that result from all jobs.

Two frequently used attributes to define the tasks priorities are the *t-level* (top-level) and the *b-level* (bottom-level). The *t-level* of a node n_i is defined as the length of the longest path from an entry node to n_i (excluding n_i). The *b-level* of a node n_i is the length of the longest path from n_i to an exit node. The nodes along the DAG with higher *b-level* belong to the critical path.

The execution time $t_{i,p}$ is considered to be non-monotonic so that there is a number p of processors for which $t_{i,p} < t_{i,p-1}$ and $t_{i,p} < t_{i,p+1}$. Let $t^*_{i,p}$ be

the minimum processing time of task i on the heterogeneous machine, which is achieved when the fastest p processors are used. Other combination of p processors will result in less computational capacity and consequently more processing time. The specific best processor layout should be a parameter of the tasks so that it can be considered in the optimal ($t^*_{i,p}$) processing time computation. From this definition we can estimate a lower bound for the *makespan* which is the sum of the minimum processing time of the tasks on the critical path: $t_\infty = \sum_i t^*_{i,p}$, which is the time required to solve all tasks assuming an unbounded number of processors [21], and in this case it means that any task has the cluster fully available for it.

The expected *makespan* is higher because not all concurrent tasks can execute on the fastest processors which may change dynamically the critical path. Lower priority tasks, after being scheduled, can be transformed in critical path tasks if the capacity of the machine is lower than the required capacity to obtain $t^*_{i,p}$ for all concurrent tasks. Therefore, the algorithm [2] evaluates dynamically the *b-level* of the tasks being scheduled and makes scheduling corrections in order to reduce the maximum *b-level* of those tasks.

The processing capacity required to achieve $t^*_{i,p}$ for task i considers the fastest processors and is defined as $S^*_i = \sum_{j=1}^{p} S_j$. It is obvious that if slower processors are used, the capacity that achieves minimum time for task i is $S'_i < S^*_i$, resulting $t'_i > t^*_i$. Since more processors are required to obtain S^*_i they would imply more communications and consequently more processing time; therefore, the minimum processing time achievable will be certainly higher than the estimated t^*_i.

Algorithm1.

1. `while tasks` $\neq \oslash$
2. `  Compute the set of ready tasks`
3. `  For each User k with limit>0`
3. `    For each ready task i`
4. `      Compute the optimal capacity` S^*_i
5. `    if` $\sum_i S^*_i > S_{limit}$
6. `      For each ready task i`
7. $S'_i = (S_{limit} / \sum_j S^*_j) S^*_i$
8. `      else`
9. `        For each ready task i` $S'_i = S^*_i$
10. $S_{max} = S_{max} - \sum_i S'_i$
11. `  For each ready task i`
12. `    Compute the optimal capacity` S^*_i
13. `  if` $\sum_i S^*_i > S_{max}$
14. `    For each ready task i`
15. $S'_i = (S_{max} / \sum_j S^*_j) S^*_i$
16. `  else`
17. `    For each ready task i` $S'_i = S^*_i$

Algorithm 1 is based on [2]. $S_{max} = \sum_{i=1}^{P} S_i$ is the total capacity of the homogeneous or heterogeneous machine computed as the sum of the individual capacity of each node. The while cycle at line 1 refers to all tasks of the DAG. In line 2 the

ready tasks are those for which the predecessors have finished processing. From line 3 to 9 the algorithm determines the computational capacity that minimizes each ready tasks S_i^* for the users that have reserved a slice of the cluster (represented by S_{limit}), according to the computational model and by assuming that the fastest processors are used. The number of processors is not important here and it is not registered. Then if the user limit S_{limit} is exceeded, the capacity assigned to each tasks is limited to the relative weight of each task. On line 10 the capacity left to other users that have no cluster reservation is computed. From line 11 to 17 the algorithm computes the same as from line 3 to 9 for the remaining tasks. The time complexity of algorithm 1 is $O(v \times P + v)$ since each task is only computed once: step 4 and 12 are $O(v \times P)$; and step 6 and 14 are $O(v)$.

Algorithm2.

```
1.   Compute tl
2.   while ready tasks ≠ ⊘
3.      Select the minimum tl
4.      Select processors that allow tl
5.      while  S_i < S'_i and t_{i,p} < t_{i,p-1}
            add processor
6.   Compute bl
7.   while true
8.      Select task k with highest bl
9.      Select task r with minimum bl
10.     if r has been maximum
            then break
11.     Reduce one processor to task r
12.     Assign it to task k
13.     Re-evaluate processing time
            of tasks r and k
14.     Re-evaluate bl of tasks r and k
```

Algorithm 2 is the second part of the scheduling algorithm. Here *t-level* and *b-level* were replaced by `tl` and `bl` respectively. From line 1 to 5 the algorithm schedules all ready tasks trying to assign to them the processing capacity determined before in Algorithm 1. The selected processors allow the tasks to start on their earliest start-time, but it also verifies if starting later, with more processors, they can finish at an earlier time. The processing time needs to be tested since in general the processors used are not the fastest ones and consequently the minimum processing time is achieved with less processing capacity, although higher than $t_{i,p}^*$.

From line 6 to 14 the algorithm tries to correct the last schedule by assigning more processors to the tasks that have higher *b-level*. The computation of *b-level* on line 6 and 14 uses $t_{i,p}^*$ for the tasks on following levels (not processed yet). For tasks of the current level, the time computed on line 5 and 13 are respectively used. The algorithm stops if the task with minimum *b-level* has been maximum during the minimization procedure. At line 11 the computation time of tasks r

and k are re-evaluated considering the new set of processors assigned which have resulted from the transference of one processor from the set of r to the set of k.

The time complexity of *b-level* and *t-level* is $O(e + v)$ [13]. The time complexity of steps 3 to 5 is $O(v \times P)$ since it is executed once for each task and combined up to P processors. Steps 7 to 14 are $O(v^2 \times P)$ since each task can be compared to the others (v^2) and each of those cases is combined up to P processors. The resulting time complexity of both algorithms is $O(v^2 \times P)$.

The algorithm can be applied at the beginning of the computation and generates all the scheduling, resulting in a static scheduling. But if Algorithm 1 and 2 are executed every time that new ready tasks are available, this is, with the feedback of the computation and eventually with new jobs that may have arrived, it will produce a dynamic scheduling that takes into account the availability of the nodes (some may go off), the new jobs submitted and the expiration of user reservations. In fact for a cluster only a dynamic behavior will be useful.

5 Results and discussion

In this section the evaluation of the scheduling algorithm proposed in this paper and a comparison to the HEFT [20], a reference algorithm of the related work section, is presented. The results shown below are obtained from a simulation setup but based on measures taken in the target cluster. The procedure to estimate computation and communication times were presented and analyzed before [4].

5.1 Parallel machine

Although both scheduling algorithms were designed to work on heterogeneous machines, the machine considered here is homogeneous in order to have an unbiased comparison of the algorithms behavior. In fact a homogeneous computer is a particular case of the general heterogeneous paradigm.

The machine considered is composed by 20 processors, connected by a $100Mbit$ switched Ethernet. The processors are Pentium IV at 3.2 GHz and 1 GB of RAM with an individual capacity of $404Mflops$. The main characteristic of the network is that it allows simultaneous communications between different pairs of machines. For parallel tasks this is an important characteristic because to complete a task the involved processors (group) have to exchange data. In the general case, when accounting for the amount of communication involved, we need to ensure that inside the group there is no communication conflicts. Otherwise, it would be very difficult to synchronize communications, in different parallel tasks, to avoid conflicts.

5.2 DAGs and tasks

There were used three DAGs, one with 10 tasks obtained from [20] and shown in Figure 3, for direct comparison, and two other DAGs of 30 and 90

tasks. These last two DAGs were generated based on the algorithm presented in [8] which can be resumed as follows: there are Na nodes with no predecessors and only successors, with ids ranging from 1 to Na; Nb nodes with both predecessors and successors, with ids ranging from Na+1 to Na+Nb; Nc nodes with only predecessors and ids ranging from Na+Nb+1 to Na+Nb+Nc. Here we considered Na=Nc=4 and Nb equal to the remaining nodes. The minimum and maximum out node degree is 2 and 5 respectively. We also make all edges pointing from smaller id nodes to larger id nodes.

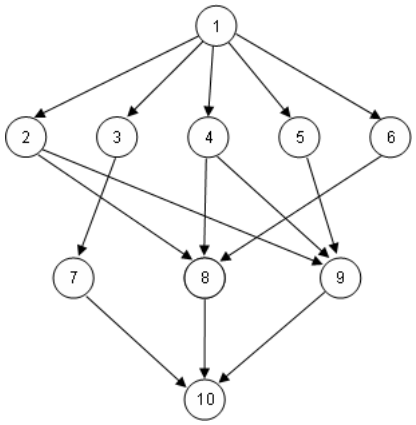


Fig. 3. Sample DAG with 10 nodes obtained from [20]

The structure of the randomly generated DAG [8] can represent a collection of jobs from one or several users that are organized in a master DAG as expressed on section 4, and representing real applications. In this paper the tasks that form all DAGs are linear algebra kernels namely tridiagonal factorization (TRD), matrix Q computation, QR iteration and correlation (C). The size of each task is randomly generated in the interval $[100, 400]$. The processing times estimated in the scheduling algorithm are based on real values measured on the target processors. Table 1 shows the relative computation and intra-task communication weight of the tasks. The DAG edges are assigned a inter-task communication cost of 30% of the computation time of the precedent task (computation to communication ratio of 0.3).

5.3 Limitations of the non-parallel task scheduling

First, it is shown that a non-parallel approach does not take advantage of the capacity available mainly due to the serialization of tasks in the same processor in order to reduce inter-task communications. The scheduling resulted from these algorithms, like HEFT, uses few processors because this results in an optimization of the scheduling length. Figure 4-a) shows the computational load

Task type	TRD	Q	QR	C
Task relative computational weight	1	0.82	2	3
Task relative communication weight	1	0.125	0.25	0.50

Table 1. Relative computational and intra-communication weights of tasks

for a computer with 20 processors. It can be seen that for the 10 node DAG, of Figure 3, only 3 processors have significant load; for the 30 node DAG only 4 processors and with very different loads; and for the 90 node DAG, 5 processors are idle and other 5 have very low load, this is, half of the machine is idle almost of the time. This behavior shows that if a user reserves a set of nodes for a given period of time, it is not guaranteed that the machine is well used even if the user has heavy DAGs to execute.

A parallel task scheduling, on the other hand, achieves better load distribution and consequently better machine usage. Figure 4-b) shows the load distribution for the same DAGs. Although this approach requires data redistribution between tasks and intra-task communications, it reduces the scheduling length as shown in table 2.

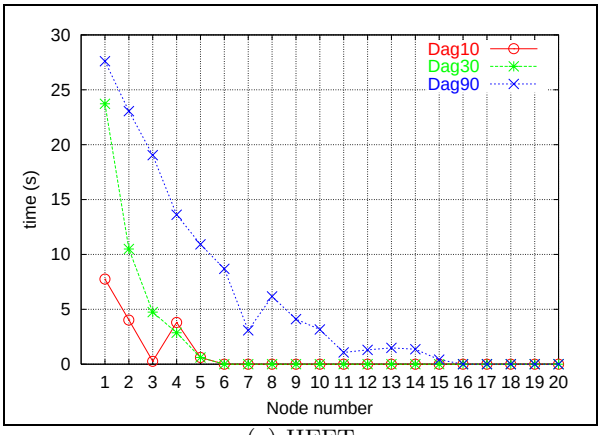
Algorithm	DAG10	DAG30	DAG90
HEFT (s)	7.87	23.85	27.64
Parallel task scheduling (s)	4.02	14.31	21.41

Table 2. Schedule length obtained with HEFT and Parallel Task scheduling

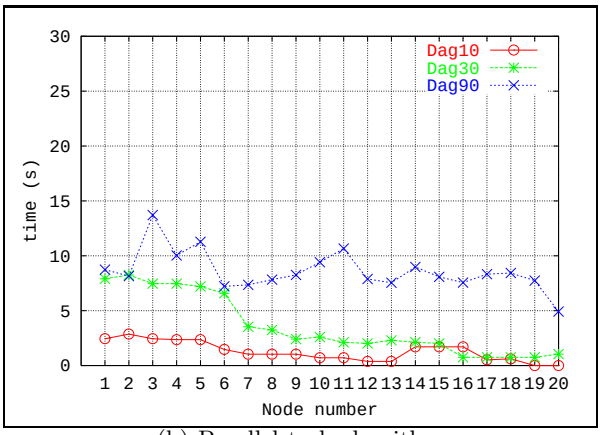
5.4 Scheduling with user reservation of machine nodes

In point 5.3 it was shown the advantages of the parallel task scheduling. Figure 4 shows a better load balance for this algorithm but not a perfect one. A perfect scheduling would result in the same load for all processors assigned to a user. This may not be achievable due to DAG restrictions. Therefore, the alternative proposed with algorithms 1 and 2 is to make a flexible management of the nodes assigned to a user so that if at any given point the user cannot use that processing capacity, it will be available for jobs of other users. Instead of assigning processors it is assigned processing capacity so that along a DAG execution different processors can be used with the restriction of having, together, the same processing capacity. This strategy is straightforward applied for the heterogeneous case.

We distinguish two situations that are: a) the user tasks are independent or the DAG executed allows an efficient usage of the processors reserved with few



(a) HEFT



(b) Parallel task algorithm

Fig. 4. Load distribution obtained with HEFT and parallel task scheduling algorithm

idle periods; and b) the user DAG imposes some idle periods in the processors reserved. In the first situation, the algorithm proposed here does not bring any advantage, apart from the task parallel scheduling that may improve the schedule as shown in point 5.3. It is for the second hypothesis that the usage of the machine as a whole can be improved and consequently the schedule length of the jobs in general. Next, it will be shown with a situation example how the scheduling algorithm improves the global performance.

Consider that a user reserves 12 nodes of a 20 node computer for a period of time. However, user1 runs a job with 10 nodes that finishes before the reserved period. Another user wants to run a job with 30 nodes which would use the remaining 8 nodes. Table 3 resumes the schedule length obtained when an exclusive usage is imposed and when a flexible usage is allowed with the global management as proposed here.

user	Schedule length (s)	
	Exclusive use	Global Management
User1 (12 nodes)	106.59	93.60
User2 (8 nodes)	251.78	210.44

Table 3. Schedule length obtained with exclusive use of nodes and a global management

In this case even user1 that consumed less computing power than the one reserved was able to reduce the schedule length of the job. This is due to the utilization of 13 processors in one given moment of the processing. This happened due to rounding effects that resulted in the assignment of one more processor to user1. The algorithm assigns computing power, but in shanks equivalent to the computing power of the powerful node available. What was expected was to obtain the same processing time as in the case of exclusive usage. The other user reduces substantially the schedule length because when user1 does not use the reserved power it is assigned to the other job. Figure 5 shows the assignment of nodes to the tasks of user1 and user2 jobs when using the global management. It can be seen that one task of user1 is executed on processor number 20 and that before the end of the user1 job, user2's job uses processors in the set of the first 12. This is because task restrictions in user1 DAG left several nodes idle. The gaps between tasks of a given user are due to inter-task communications. The gap is proportional to the edges arriving a node, because the algorithm sums those communications.

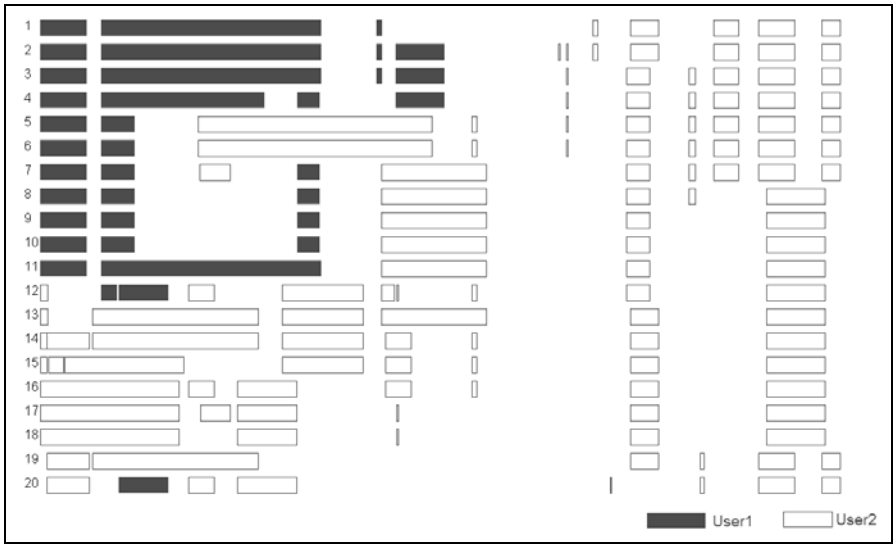


Fig. 5. Mapping of tasks to computer nodes with the global management; node number in the vertical axis and time in the horizontal axis

6 Conclusions

The scheduling algorithm presented in this paper can improve the cluster utilization and the response time once we allow a variable computing power (number of processors) assigned for a job. Although the results presented are for a homogeneous cluster, the algorithm was designed for heterogeneous machines. To overcame the heterogeneity of the machine, the algorithm starts by computing the amount of capacity in $Mflops$, instead of number of processors, that minimizes the processing time of each task. For that it uses the fastest processors and determines the minimum processing time that each task can achieve in that machine. Then, the algorithm joins processors until the maximum capacity required for each task is achieved, independently of the number of processors, but restricted by the maximum capacity available.

The algorithm proposed does not require a fixed subdivision of processors. When scheduling a set of ready tasks the machine is viewed has a whole, independently of the groups of processors formed in the last level, thus allowing a better use of the machine and consequently achieving improvements in processing time.

It was demonstrated that, when scheduling DAGs, a non-parallel task scheduling has limitations to efficiently use a set of processors assigned to a job.

Acknowledgments

The work presented was partially done in the scope of the project Segmentation, Tracking and Motion Analysis of Deformable (2D/3D) Objects using Physical Principles, with reference POSC/EEA-SRI/55386/2004, financially supported by FCT-Fundação para a Ciência e Tecnologia from Portugal.

References

1. A. K. Amoura, E. Bampis, and J.-C. König. Scheduling algorithms for parallel gaussian elimination with communication costs. *IEEE Transactions on Parallel and Distributed Systems*, 9(7):679–686, July 1998.
2. J. Barbosa, C. Morais, R. Nobrega, and A.P. Monteiro. Static scheduling of dependent parallel tasks on heterogeneous clusters. In *Heteropar'05*, pages 1–8. IEEE Computer Society, 2005.
3. J. Barbosa and A.J. Padilha. Algorithm-dependent method to determine the optimal number of computers in parallel virtual machines. In *VECPAR'98*, volume 1573. Springer-Verlag LNCS, 1998.
4. J. Barbosa, J. Tavares, and A.J. Padilha. Linear algebra algorithms in a heterogeneous cluster of personal computers. In *Proceedings of 9th Heterogeneous Computing Workshop*, pages 147–159. IEEE CS Press, May 2000.
5. F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing 96*, 1996.
6. J. Blazewicz, P. Dell'Olmo, M. Drozdowski, and P. Maczka. Scheduling multiprocessor tasks on parallel processors with limited availability. *European journal of Operational Research*, (149):377–389, 2003.

7. J. Blazewicz, M. Machowiak, J. Weglarz, M. Kovalyov, and D. Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan. *Annals of Operations Research*, (129):65–80, 2004.
8. Sameer Shivle et al. Mapping of subtasks with multiple versions in a heterogeneous ad hoc grid environment. In *Heteropar'04*. IEEE Computer Society, 2004.
9. Jong-Kook Kim *et al.* Dynamically mapping tasks with priorities and multiple deadlines in a heterogeneous environment. *Journal of Parallel and Distributed Computing*, 67:154–169, 2007.
10. R. Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. Technical Report CS-95-286, University of Tennessee, Knoxville, 1995.
11. A. Gerasoulis and T. Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, pages 686–701, June 1993.
12. K. Jansen. Scheduling malleable parallel tasks: An asymptotic fully polynomial time approximation scheme. *Algorithmica*, 39:59–81, 2004.
13. Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, December 1999.
14. Y. Kwok and I. Ahmad. On multiprocessor task scheduling using efficient state space search approaches. *Journal of Parallel and Distributed Computing*, 65:1515–1532, 2005.
15. R. Lepère, G. Mounié, and D. Trystram. An approximation algorithm for scheduling trees of malleable tasks. *European journal of Operational Research*, (142):242–249, 2002.
16. Oh-Heum and K-Y Chwa. Scheduling parallel tasks with individual deadlines. *Theoretical Computer Science*, 215:209–223, 1999.
17. Gyung-Leen Park. Performance evaluation of a list scheduling algorithm in distributed memory multiprocessor systems. *Future Generation Computer Systems*, (20):249–256, 2004.
18. B. Shirazi, M. Wang, and G. Pathak. Analysis and evaluation of heuristic methods for static task scheduling. *Journal of Parallel and Distributing Computing*, 10:222–232, 1990.
19. O. Sinnen and L. Sousa. List scheduling: extension for contention awareness and evaluation of node priorities for heterogeneous cluster architectures. *Parallel Computing*, (30):81–101, 2004.
20. H. Topcuoglu, S. Hariri, and M.-Y Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–274, March 2002.
21. Denis Trystram. Scheduling parallel applications using malleable tasks on clusters. In *15th International Conference on Parallel and Distributed Processing Symposium*, 2001.