New Self-Scheduling Schemes for Internet-Based Grids of Computers

Javier Díaz, Sebastián Reyes, Alfonso Niño, and Camelia Muñoz-Caro

Universidad de Castilla-La Mancha, Escuela Superior de Informática,
Paseo de la Universidad 4, 13071, Ciudad Real, Spain,
{javier.diaz,sebastian.reyes,alfonso.nino,camelia.munoz}@uclm.es

Abstract. Self-scheduling algorithms are useful tools for achieving load balance in computational systems, in particular, in computational Grids. In this work, we introduce two families of self-scheduling algorithms. The first is defined by methods looking for an explicit form for the chunks distribution function. The second corresponds to methods focusing in derive an explicit form for the rate of variation of the chunks distribution function. From the first family, we select a Quadratic Self-Scheduling (QSS) algorithm. From the second, a new Exponential Self-Scheduling (ESS) algorithm is introduced. QSS and ESS are optimized for working in an Internet-based Grid of Computers involving resources from Spain and Mexico. Performance tests show that QSS and ESS outperform previous self-scheduling algorithms. QSS is found slightly more efficient than ESS.

1 Introduction

Workload distribution is an important issue in distributed systems. Efficient allocation of the workload among the processors is necessary to optimize the use of the resources. This is specially important when the computational resources are not only distributed, but also heterogeneous, as in a computational Grid [1]. A computational Grid is a hardware and software infrastructure providing dependable, consistent, and pervasive access to resources among different administrative domains. In this form we allow the sharing of the resources in a unified way, maximizing their use. A Grid can be used to perform large-scale runs of distributed applications. An ideal case to be run in Grid is that with a large number of independent tasks. This is the case of parameter sweep problems. To minimize the overall computing time, a correct assignment of tasks is needed, so that computer loads and communication overheads are well balanced. In this context, scheduling algorithms are needed. Different scheduling strategies have been developed along the years (for the classical taxonomy see [2]). In particular, dynamic self-scheduling algorithms are extensively used in practical applications. Some examples are: the TREE-PUZZLE package [3] used for quartet-based maximum-likelihood phylogenetic analysis, the MCell Simulator [4] [5] used for simulating simultaneous diffusion and chemical reactions of molecules in complex 3-D spaces, the Nexus Proxy [6] which relays on TCP communications to provide a communication mechanism beyond a firewall, or the OpenMP parallel programming model that implements self-scheduling to schedule the parallel code [7] [8]. In addition, self-scheduling is applied in real-time computing to minimize the runtime scheduling overhead [9].

Self-Scheduling algorithms represent adaptive schemes where tasks are allocated in run-time. These algorithms were initially developed to solve parallel loop scheduling problems in homogeneous memory-shared systems, see for instance [10]. Here, the loop iterations have not interdependencies, and they can be scheduled as independent tasks. Despite their origin, these algorithms have been tested successfully in distributed memory multiprocessor systems and heterogeneous clusters [11–21]. In addition, some works on the performance of Grid connected clusters of computers have been reported [22], [23], [24].

In this work, we present two approaches to the development of new self-scheduling algorithms. These algorithms are oriented to the scheduling of tasks at the application level in an Internet-based Grid of computers. We consider one method from each of the two families of methods presented. The methods are optimized for a Grid of heterogeneous computers. Comparison with prior self-scheduling algorithms is also presented.

The rest of this paper is organized as follows. Section 2 presents an overview of the field of self-scheduling algorithms. In section 3, we introduce the two new families of self-scheduling algorithms, presenting one method from each. Section 4 details the methodology used for the optimization of the algorithms and for the comparison with well-established self-scheduling schemes. Finally, section 5 presents and interprets the results about the optimization process and the comparative study.

2 Background

Originally, self-scheduling was an automatic loop-scheduling method to assign loop iterations to idle processors. Here, an important issue is the number of iterations or independent tasks to assign to each processor. Due to the processors heterogeneity, and their distributed nature, we have two complementary effects, load balance and communication overhead. The theoretical work of Kruskal and Weiss [25] shows that two extreme scheduling models correspond to the optimization of one of these factors. On one hand, we have static chunking. This method assigns a task to each idle processor, and achieves good load balance but may cause too much communication and scheduling overhead. On the other hand, Chunk Self-Scheduling (CSS) [26], which divides the total number of tasks among the available processors, assigning the resulting number of tasks (the chunk) to each processor. Here, we achieve low overhead, but load imbalance will be important if the tasks use different computing times. The practical self-scheduling schemes look to find an equilibrium between load balance and overhead. In this context, Kruskal and Weiss [25] showed that an appropriate strategy is to use a number of chunks higher than the number of processors, assigning a decreasing number of tasks to each processor as it becomes free. On this basis, several self-scheduling algorithms have been proposed.

Guided Self-Scheduling (GSS) dynamically changes the number of tasks assigned to each processor [27]. More specifically, the chunk size is determined by dividing the number of remaining tasks by the total number of processors. In this form, the chunk size decreases in each chunk, achieving good load balancing and reducing the scheduling overhead. However, sometimes GSS may assign too many tasks in the early

chunks to the processors, and the remaining tasks may not be sufficient to guarantee a good load balance.

Factoring Self-Scheduling (FSS) results from a statistical analysis, where the computing time in each processor is considered an independent random variable. Here, the tasks are scheduled in batches (or stages) of P chunks of equal size, where P is the number of processors [28]. The chunk size is determined by dividing the number of remaining tasks by the product of the number of processors and a parameter (α). This parameter ensures that the total number of tasks per batch is a fixed ratio of those remaining, *i. e.*, only a subset of the remaining tasks (usually half) is scheduled in each batch. This algorithm provides better workload balance than GSS when task execution times vary widely and unpredictably. The overhead of FSS is not significantly greater than that of GSS.

Another approach is Trapezoid Self-Scheduling (TSS). TSS uses a linear decreasing chunk function [29]. In this form, the algorithm tries to reduce the scheduling overhead, since the function is simple and minimizes the need for synchronization. In addition, the decreasing chunk size balances the workload. TSS assigns F and L tasks to the first and last chunk, respectively. F and L are parameters adjustable by the programmer. Tzen and Ni [29] proposed F=I/2P and L=1, being I the total number of tasks and P the available processors.

Self-scheduling algorithms are derived for homogeneous systems but, in principle, they can be applied to heterogeneous ones. However, they could be not enough flexible (they have not enough degrees of freedom) to adapt efficiently to a heterogeneous environment. Different algorithms have been proposed for this case, such as Affinity Scheduling (AS) [30]. AS combines static and dynamic scheduling strategies. Other approaches introduce additional degrees of freedom in the model. Previously, we proposed a new self-scheduling algorithm called Quadratic Self-Scheduling (QSS) [24], [21]. This algorithm is based on a quadratic form for the chunk function. Therefore, we have three degrees of freedom, which provide higher adaptability to distributed heterogeneous systems.

A different approach is that of Weighted Factoring (WF). WF uses a variant of factoring self-scheduling, which incorporates weight factors for each processor [31]. These factors are experimentally determined from the relative computing power of the processors. In WF the processor weights remain constant throughout the parallel execution. Banicescu and Liu [32] proposed a method called Adaptive Factoring (AF). AF adjust the processors weights according to timing information reflecting variations in the computation power of the slaves. Chronopoulos et al. extended the TSS algorithm, proposing the Distributed TSS (DTSS) in [33]. In this algorithm, chunks sizes are weighted by the relative power of the server processors and the number of processes in their run-queue, at the moment of requesting work from the master. Ciorba at el. extended in [34] the DTSS algorithm, proposing DTSS+SP to handle loops with dependencies on heterogeneous clusters via synchronization points (SPs). Also, there are self-scheduling algorithms based on the history of task timing results, like SAS [35]. This algorithm attempts to balance dynamically the application workload in the presence of a fluctuating exogenous load. SAS assigns chunks according to the history of previous timing results, the history of all incoming tasks, and the current number of

processes in the run-queue. Another approach, based in guided self-scheduling, is Gap-Aware Self-Scheduling (GAS) [36]. Here, a temporal gap factor for each processor is calculated, measuring its availability. Depending on these factors the algorithm modifies the chunk size calculated by GSS. Finally, RAS [11] is an algorithm that using a master-slaves model estimates the computational performance of the slaves. The tasks are distributed among the slave processors depending on the performance estimates. However, the master can redistribute tasks when a load imbalance is detected.

3 New Self-Scheduling Schemes

This section presents two different families of Self-Scheduling methods. Both of them uses the chunk distribution function, C(t). This functions gives the chunk size as a function of the t-chunk. The first family is based in the chunks distribution function, whereas the second focuses on the slope of C(t).

3.1 Methods based in the chunks distribution function

In this case, we look for an explicit form for C(t), which we can consider given by a function f(t), the target function. From a general point of view, we can expand C(t) in a Taylor series as a function of t as shown in equation (1),

$$C(t) = f(t_0) + \left|\frac{df(t)}{dt}\right|_0 (t-t_0) + \frac{1}{2}\left|\frac{d^2f(t)}{dt^2}\right|_0 (t-t_0)^2 + \ldots \quad (1)$$

Depending on the size of the expansion we have methods of different order. Thus, for C(t) constant we have pure self-scheduling or chunk self-scheduling. For the linear approach we have TSS. Finally, for the quadratic case a new method called Quadratic Self-Scheduling (QSS) has been proposed recently [21], [24]. In QSS, C(t) is given by equation (2)

$$C(t) = a + bt + ct^2 \quad (2)$$

where t represent the t-th chunk assigned to a processor. To apply the QSS algorithm we need the a, b and c coefficients of equation (2). Thus, we can select three reference points (C(t), t) and solve for the resulting system of equations. In the studies presented until now, the two first points are selected as in the TSS linear case, that is, (I/2P, 0) and (1, N). Here, I is the number of tasks, P the number of processors and N the total number of chunks. The third point is selected as ($C_{N/2}$, N/2), where N/2 is half the number of chunks for the quadratic case. $C_{N/2}$ can be any value in the interval [C_0, C_N]. Solving for a, b and c, we obtain,

$$\begin{aligned} a &= C_0 \\ b &= (4C_{N/2} - C_N - 3C_0)/N \\ c &= (2C_0 + 2C_N - 4C_{N/2})/N^2 \end{aligned} \quad (3)$$

where N is defined [24] by,

$$N = 6I/(4C_{N/2} + C_N + C_0) \tag{4}$$

The $C_{N/2}$ value is given by,

$$C_{N/2} = \frac{C_N + C_0}{\delta} \tag{5}$$

and since C_0 and C_N are fixed, the $C_{N/2}$ value determines the slope of equation (2) at a given point. Therefore, depending on the δ value, the slope of the quadratic function for a t value is higher or smaller than that of the linear case (TSS algorithm), which corresponds to δ =2. In the present work, we keep the size of the initial chunk as I/2P, which is the optimal value determined with TSS by Tzen and Ni [29]. However, the size of the last chunk and the δ value are optimized for the execution environment.

3.2 Methods based in the slope of the chunks distribution function

Now, the starting point is the slope of the chunks distribution function, C(t). Thus, we are selecting the rate of variation of C(t) as a function of t. Therefore, if the slope is given by a decreasing function, f(t), we have the general expression,

$$\frac{dC(t)}{dt} = f(t) \tag{6}$$

Equation (6) defines a differential equation. After integration we will have an explicit functional form for C(t) as a function of t.

A first approach is to consider that the slope (negative) is proportional to the chunk size, equation (7). Here, k is a parameter and t represents the t-th chunk assigned to a processor.

$$\frac{dC(t)}{dt} = -kC(t) \tag{7}$$

Equation (7) can be integrated by separation of variables yielding equation (8),

$$C(t) = \left(\frac{I}{2P}\right) e^{-kt} \tag{8}$$

where we have used C(0)=I/2P, as proposed by Tzen and Ni [29], I is the total number of chunks and P is the total number of processors. Equation (7) defines a new self-scheduling method that we call Exponential Self-Scheduling (ESS). In this method, k is a parameter that will be optimized for our working environment.

4 Methodology

Our aim is to compare, at the application level, the working of the optimized QSS and ESS against other self-scheduling algorithms in an Internet-based Grid of computers.

Thus, apart from QSS and ESS we select chunk self-scheduling, guided self-scheduling, factoring self-scheduling and trapezoid self-scheduling.

The tests have been carried out in an Internet-based Grid of computers formed by three clusters. Two of them, Tales and Hermes, are located in the Universidad de Castilla-La Mancha at Ciudad Real (Spain). These clusters consist of twelve Pentium IV with CPU frequencies between 2.4 and 3.0 GHz and 1GB of physical memory. The third cluster (Popocatepetl) is in the Universidad de Puebla (Mexico). This last is formed by nine 64-bit Opteron biprocessors with 1.66 GHz CPU frequency and 1 GB of physical memory. Each cluster uses a 100-Mbps Fast Ethernet network. Connection between clusters is achieved through Internet. The Grid uses Globus as basic middleware in its 4.03 flavour [37]. To allocate the tasks on the Grid we will use the 5.0 version of the GridWay metascheduler [38].

The different parameters of the self-scheduling algorithms have been configured as follows. In Chunk Self-Scheduling, the chunk size is given by the quotient of tasks to processors. On the other hand, Guided Self-Scheduling determines the chunk size at each step using the number of tasks and available processors. In Factoring Self-Scheduling, we have fixed the α parameter to 2, the suboptimal value obtained in the original statistical analysis leading to the FSS method [28]. In Trapezoid Self-Scheduling we used the values proposed by Tzen and Ni [29] for the initial and final chunk sizes. In QSS we have selected C_0 as the initial chunk size of TSS. The $C_{N/2}$ and C_N parameters will be experimentally determined in a two-dimensional (2D) study. Finally, in ESS we have parameter k. This parameter will be experimentally determined for our environment.

In the tests, we will consider sets of tasks without any predefined relationship among their durations. Thus, we build an application that uses an amount of computing time stochastically determined. The application performs several times a product of square matrices. In each product, the size of the matrices is randomly determined in run time between 300 and 1000. Sets of several hundred products (tasks) are used for allocation in the Grid by the different self-scheduling algorithms. To reduce the effect of network latency we do not allocate each matrix product independently. Instead, each chunk size determines the number of matrix products (tasks). Thus, each chunk defines a chore or job in the system. When running the tests, the system has been dedicated to this task, in order workload and communication overhead to be the only factors to consider.

In the performance study, we will use 20 processors of our Grid distributed 8 from Hermes, 8 from Tales and 4 from Popocatepetl. We consider a total of 2804 independent tasks. The different tests have been performed three times, in order to obtain an average. The time used in the study corresponds to this average. As performance index we will use the speedup relative to the worst case found.

5 Results and Discussion

Firstly, we have performed the 2D study to obtain the optimal values of $C_{N/2}$ and C_N parameters for the QSS algorithm. The different values of $C_{N/2}$ are obtained by modifiying parameter δ, see equation (5). The δ values range from three to seven in increments of one. We show previosuly [24] that values smaller or higher than these affect negatively

the efficiency of the algorithm. The values of the parameter C_N will range from one to eight in increments of one.

The results for QSS are graphically represented in Figure 1. The global minimum is found for $\delta = 3$ and $C_N = 2$. Figure 1 shows a valley along all the δ values for $C_N = 1 - 3$. It is interesting to indicate that even for the worst combination of δ and C_N QSS is more efficient that the rest of the algorithms, except ESS.

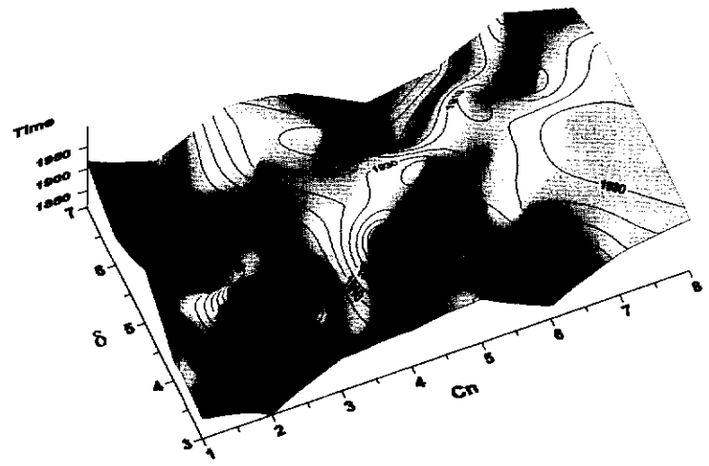

Fig. 1. Performance of the QSS algorithm as a function of δ and C_N parameters. Temporal data are reported in seconds. Interval between contour lines is 10 seconds.

Respect to ESS, the k parameter has been also optimized. We have selected values from 0.010 to 0.024, in increments of 0.001. Smaller or larger values do not provide any significant information. This is easy to explain, since for small k values the exponential approaches one, and the chunk size is constant. On the other hand, for large k values the exponential approaches zero. This leads to very small chunk sizes. This fact maximizes the communication overhead.

Figure 2 represents graphically the results of the different tests. This Figure exhibits a generally negative slope until k equals 0.017, where the minimum is found. From here a generally increasing trend is manifested. The minimum found in Figure 2 corresponds to the point where communication overhead compensates load balance in our system. Therefore, we select this point, k=0.017, to compare ESS with the other algorithms.

After calibration of QSS and ESS, we compare with the rest of self-scheduling algorithms with the test described in the Methodology section. Table 1 collects the time used in the test (average values) and standard deviation for the different algorithms. CSS is found to give the poorest performance. This is what we can expect due to the static nature (and therefore the lack of adaptability) of CSS. Thus, using CSS as reference, we

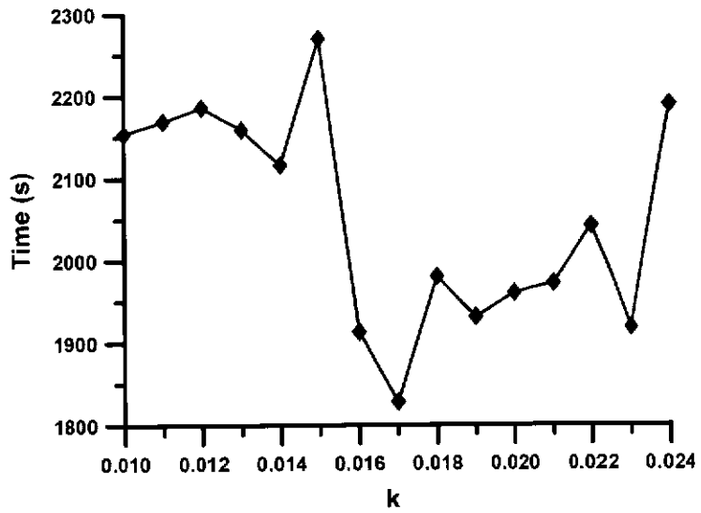

Fig. 2. Performance of the ESS algorithm as a function of the k parameter. Temporal data are reported in seconds. Black diamonds mark the experimental data.

compare the relative performance using speedups. Here, the speedup (S) is defined as the quotient between the time used by CSS and the time used by each algorithm. Figure 3 collects the speedups for the different algorithms.

Table 1. Average total time values used by the different self-scheduling algorithms in the test performed.Standard deviation is included. Temporal data in seconds.

	CSS	GSS	TSS	FSS	QSS	ESS
Time:	2910	2065	2068	2232	1811	1828
σ	57.79	118.93	132.88	106.07	48.52	39.00

Figure 3 shows that the performance variation is QSS≥ESS>GSS≥TSS>FSS>CSS. QSS outperforms CSS, GSS, TSS, FSS and slightly ESS. The better performance of QSS can be attributed to the higher adaptability of the model (due to its three degrees of freedom) to the heterogeneous environment and conditions used in the test. It is remarkable than ESS, with only two parameters has a similar performance that QSS. Therefore, this new algorithm is not only simple, but seems to be almost as efficient as QSS.

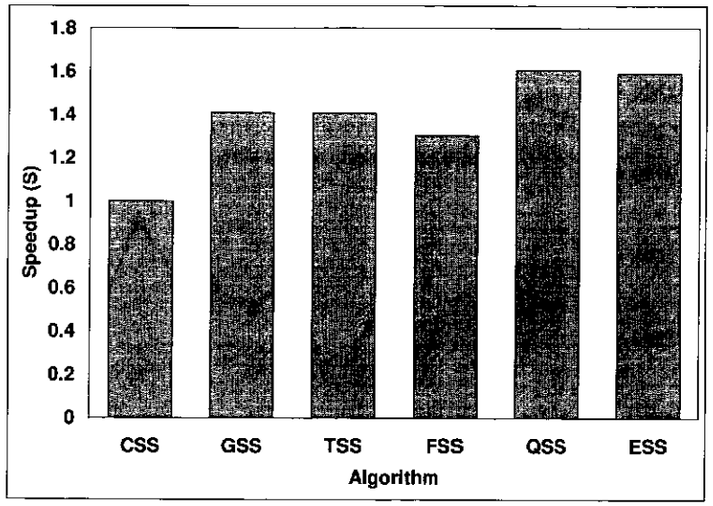

Fig. 3. Speedup (S), respect to CSS, for the considered self-scheduling algorithms in the tests performed.

As expected, CSS presents the worst behaviour because its use of very large chunks produces a larger load imbalance than the other algorithms. As shown previously [24] GSS performs better than FSS. FSS is an algorithm developed assuming all the computing times are independent, but equal, random variables. Therefore, FSS is a good approach only in a homogeneous environment. The poor results of FSS are due to the use of stages where all processors are assigned equal chunk sizes. In an heterogeneous system, where the processors have different performance, this fact results in load imbalance, as in CSS. On the other hand, TSS shows better performance than FSS because the decreasing chunk sizes used allow to obtain a better load balance in heterogeneous environments. GSS do better than TSS and FSS because its small chunks (in the last allocations) permit to compensate better the load imbalance.

6 Acknowledgements

This work has been cofinanced by FEDER funds and the Consejería de Educación y Ciencia de la Junta de Comunidades de Castilla-La Mancha (grant # PBI05-009). The Universidad de Castilla-La Mancha is acknowledged. The authors wish also to thank the Facultad de Ciencias Químicas and the Laboratorio de Química Teórica of the Universidad Autónoma de Puebla (Mexico), for the use of the Popocatepetl cluster. We also wish to thank the Distributed Systems Architecture Group of the Universidad Complutense de Madrid for their help in the configuration and use of GridWay.

References

1. I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure, *Morgan Kauffman Publishers*, 1999, San Francisco.
2. T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing, *IEEE Transactions on Software Engineering*, Vol. 14, No. 2, 1988, pp. 141-154.
3. H. A. Schmidt, K. Strimmer, M. Vingron, and A. von Haeseler. TREE-PUZZLE: maximum likelihood phylogenetic analysis using quartets and parallel computing, *Bioinformatics*, Vol. 18, No. 3, 2002, pp. 502-504.
4. H. Casanova, T. M. Bartol, J. Stiles, F. Berman. Distributing MCell Simulations on the Grid, *Intl. Journal of High Performance Computing Applications*, Vol. 15, No. 3, August 2001, pp. 243-257.
5. H. Casanova, T. Bartol, F. Berman, A. Birnbaum, J. Dongarra, M. Ellisman, M. Faerman, E. Gockay, M. Miller, G. Obertelli, S. Pomerantz, T. Sejnowski, J. Stiles, R. Wolski. The Virtual Instrument: Support for Grid-enabled Scientific Simulations, *Intl. Journal of High Performance Computing Applications*, Vol. 18, No 1, February 2004, pp. 3-17.
6. Y. Tanaka, M. Sato, M. Hirano, H. Nakada, S. Sekiguchi. Performance Evaluation of a Firewall-Compliant Globus-Based Wide-Area Cluster System, *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC'00)*, 2000.
7. E. Ayguade, B. Blainey, A. Duran, J. Labarta, F. Martinez, X. Martorell, and R. Silvera. Is the schedule clause really necessary in OpenMP?, *International Workshop on OpenMP Applications and Tools*, June 2003, pp. 147-159.
8. Chapter 10 of Fortran Programming Guide. http //docs.sun.com/source/819-3685/10_parallel.html. Last access: April, 10th 2007.
9. M. Maode, H. Babak. A Fault-tolerant Strategy for Real-time Task Scheduling on Multiprocessor System, *Proceedings of the 1996 International Symposium on Parallel Architectures, Algorithms and Networks*, 1996, p. 544.
10. D.J. Lilja. Exploiting the Parallelism Available in Loops, *IEEE Computer*, Vol. 27, No. 2, 1994, pp. 13-26.
11. Y. Kee and S. Ha. A Robust Dynamic Load-Balancing Scheme for Data Parallel Application on Message Passing Architecture, *Internation Conf. on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, July 1998, pp. 974-980.
12. Chao-Tung Yang and Shun-Chyi Chang. A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters, *The Ninth Workshop on Compiler Techniques for High-Performance Computing*, Academia Sinica, Taiwan, March 2003.
13. T. Philip and C.R. Das. Evaluation of Loop Scheduling Algorithms on Distributed Memory Systems, *Proc. of Intl. Conf. on Parallel and Distributed Computing Systems*, Washinton DC, October 1997.
14. T.H. Kim and J.M. Purtilo. Load Balancing for Parallel Loops in Workstation Clusters, *Proc. of Intl. Conference on Parallel Processing*, Vol. 3, 1996, pp. 182-189.
15. B. Hamidzadeh, D.J. Lilja and Y. Atif. Dynamic Scheduling Techniques for Heterogeneous Computing Systems, *Concurrency: Practice and Experience*, Vol. 7, 1995, pp. 633-652.
16. F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks, *In Proc. of Supercomputing '96*, Pittsburgh, November 1996.
17. F. Berman, High-performance schedulers, in: I. Foster, C. Kesselman (Eds.). The Grid: Blueprint for a New Computing Infrastructure, *Morgan-Kaufmann*, 1999, pp. 279-309.
18. A.T. Chronopoulos, M. Benche, D. Grosu, R. Andonie. A Class of Loop Self-Scheduling for Heterogeneous Clusters, *Proceeding of the 2001 IEEE International Conference on Cluster Computing*, Newport Beach, USA, October 2001, pp. 282-294.

19. A.T. Chronopoulos, S. Penmatsa, N. Yu. Scalable Loop Self-Scheduling Schemes for Heterogeneous Clusters, *Proceeding of the 2002 IEEE International Conference on Cluster Computing*, Chicago, USA, September 2002.
20. A.T. Chronopoulos, S. Penmatsa, J. Xu and S. Ali. Distributed Loop Scheduling Schemes for Heterogeneous Computer Systems, *Concurrency and Computation: Practice and Experience*, Vol. 18, Issue 7, 2006, pp. 771-785.
21. J. Díaz, S. Reyes, A. Niño and C. Muñoz-Caro. Un Algoritmo Autoplanificador Cuadrático para *Clusters* Heterogéneos de Computadores. *XVII Jornadas de Paralelismo*, Albacete, Spain, September 2006.
22. P. J. Sokolowski and D. Grosu and C. Xu. Analysis of Performance Behaviors of Grid Connected Clusters, in: M. Ould-khaoua and G. Min (Eds.). Performance Evaluation of Parallel And Distributed Systems, *Nova Science Publishers*, 2006.
23. J. Herrera and E. Huedo and R. S. Montero and I. M. Llorente. Ejecución Distribuida de Bucles en Grids Computacionales, 3ª *Reunión para la Red Temática en Grid Middleware*, Logroño, Spain, September 2005.
24. J. Díaz, S. Reyes, A. Niño and C. Muñoz-Caro. A Quadratic Self-Scheduling Algorithm for Heterogeneous Distributed Computing Systems, *In Proceedings of the 5th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar '06)*, Barcelona, Spain, September 2006.
25. C.P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors, *IEEE Trans. Software Engineering*, Vol. SE-11, No. 10, Oct. 1985, pp. 1001-1016.
26. P. Tang and P. C. Yew. Processor Self-Scheduling for Multiple Nested Parallel Loops, *In Proceedings of the 1986 International Conference on Parallel Processing*, 1986, pp. 528-535.
27. C. D. Polychronopoulos and D. Kuck. Guided Self-Scheduling: a Practical Scheduling Scheme for Parallel Supercomputers, *IEEE Trans. on Computers*, Vol. 36, December 1987, pp. 1425-1439.
28. S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A Method for Scheduling Parallel Loops, *Comm. ACM*, Vol. 35, No. 8, August 1992, pp. 90-101.
29. T. H. Tzen and L. M. Ni. Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, No. 1, January 1993, pp. 87-98.
30. E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5, No. 4, April 1994, pp. 379-400.
31. S. Flynn Hummel, J. Schmidt, R. N. Uma, and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring, *in Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures,* Padua, Italy, June 1996, pp. 318-328.
32. I. Banicescu and Z. Liu. Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes, *Proc. of the High Performance Computing Symposium 2000*, Washington, USA, 2000, pp. 122-129.
33. A. T. Chronopoulos, R. Andonie, M.Benche and D. Grosu. A Class of Distributed Self-Scheduling Schemes for Heterogeneous Clusters, *Proc. of the 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001)*, Newport Beach, CA USA, 2001.
34. F. M. Ciorba, T. Andronikos, I. Riakiotakis, A. T. Chronopoulos and G. Papakonstantinou. Dynamic Muti Phase Scheduling for Heterogeneous Clusters, *Proc. of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006)*, Rhodes, Greece, 2006.
35. I. Riakiotakis, F. M. Ciorba, T. Andronikos and G. Papakonstantinou. Self-Adapting Scheduling for Tasks with Dependencies in Stochastic Environments, *In Proceedings of the 5th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar '06)*, Barcelona, Spain, September 2006.

36. A. Kejariwal and A. Nicolau and C. D. Polychronopoulos. An Efficient Approach for Self-Scheduling Parallel Loops on Multiprogrammed Parallel Computers, *The 18th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Hawthorne, USA, October 2005.
37. I. Foster. Globus Toolkit Version 4: Software for Service-Oriented Systems, *IFIP International Conference on Network and Parallel Computing*, Springer-Verlag LNCS 3779, 2005, pp. 2-13.
38. R.S. Montero, E. Huedo and I.M. Llorente. The GridWay Approach for Job Submission and Management on Grids, *iAstro Workshop on Distributed Processing, Transfer, Retrieval, Fusion and Display of Images and Signals: High Resolution and Low Resolution in Data and Information Grids*, Granada, Spain, February 2003.