2006 IEEE International Conference on Cluster Computing (CLUSTER06)

A Quadratic Self-Scheduling Algorithm for Heterogeneous Distributed Computing Systems

J. Díaz, S. Reyes, A. Niño and C. Muñoz-Caro
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Paseo Universidad 4, 13071, Ciudad Real, Spain
{javier.diaz,sebastian.reyes,alfonso.nino,camelia.munoz}@uclm.es

Abstract

Scheduling algorithms play an important role in heterogeneous computing systems. Development of new scheduling strategies is an active research field. In this context, we present a general formulation of the self-scheduling problem, deriving a new, quadratic, self-scheduling algorithm. Initial tests comparing the performance of the new algorithm against well-established ones are carried out. Thus, working at the application level, we allocate sets of several thousand tasks in an Internet-based Grid of computers that involves a transatlantic connection. In all the tests, the new algorithm performs better than the previous ones.

1. Introduction

An essential issue in distributed high-performance computing is how to allocate efficiently the workload among the processors. This is specially important when the computational resources are not only distributed, but also heterogeneous, as in a computational Grid [8]. A computational Grid, or simply a Grid, is a hardware and software infrastructure providing dependable, consistent, and pervasive access to resources among different administrative domains. The objective is to enable the sharing of these resources in a unified way, maximizing their use. A Grid can be used effectively to support very large-scale runs of distributed applications. An ideal case to be run in Grid is that with many large independent tasks. This case arises naturally in parameter sweep problems. A correct assignment of tasks, so that computer loads and communication overheads are well balanced, is the way to minimize the overall computing time. This problem belongs to the active research topic of the development and analysis of scheduling algorithms.

Different scheduling strategies have been developed along the years (for the classical taxonomy see [3]). Among them, dynamic self-scheduling algorithms (adaptive schemes where tasks are allocated in run-time) are extensively used in practical applications. Self-scheduling algorithms were initially proposed to solve parallel loop scheduling problems in homogeneous memory-shared systems, see for instance [17]. Here, the loop iterations have not interdependencies, and they can be scheduled as independent tasks. Notwithstanding their original formulation, these algorithms have also been tested successfully in distributed memory multiprocessor systems and heterogeneous clusters [13, 24, 19, 15, 9, 2, 1, 4, 6, 5]. In addition, some works on the performance of Grid connected clusters of computers have been reported [21], [10]. However, self-scheduling algorithms have not been extensively tested and compared in Grid systems.

Our focus in this work is the scheduling of tasks at the application level, in an Internet-based Grid of computers. We propose here a general treatment of the scheduling problem. Using this treatment, a new self-scheduling algorithm is presented, corresponding to a quadratic form. We have done some preliminary performance tests in a heterogeneous Grid, comparing with prior well-established algorithms.

The remainder of this paper is organized as follows. In section 2, self-scheduling algorithms are reviewed. Section 3, presents a general treatment of the scheduling problem, developing a new self-scheduling algorithm. In section 4, we describe the methodology used for the experimental tests described in the next section. Section 5, presents and discusses the results of a comparative analysis of the proposed scheduling algorithm against previous self-scheduling schemes. Finally, in section 6 we present the main conclusions of this paper and the perspectives for future work.

2. Background

When dealing with Doall loops or sets of data independent tasks, two complementary effects bound the scheduling problem: load balance and communication overhead. The seminal theoretical work of Kruskal and Weiss [16] shows that two extreme scheduling models correspond to the optimization of one of these factors. On one hand, static chunking, that assigns a task to each free processor, achieves good load balance but maximizes overhead. On the other hand, Chunk Self-Scheduling (CSS), which divides the total number of tasks among the available processors assigning the resulting number of tasks (the chunk) to each processor. This algorithm achieves low overhead, but load imbalance will be important if the tasks use different computing times. The practical self-scheduling schemes devised up to now try to find an equilibrium between load balance and overhead. In this context, Kruskal and Weiss [16] 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. Here, we summarize three of them.

2.1. Guided Self-Scheduling (GSS). This algorithm dynamically changes the number of tasks assigned to each processor [20]. 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 calculation, 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.

2.2. Factoring Self-Scheduling (FSS). This algorithm 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 [12]. 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.

2.3. Trapezoid Self-Scheduling (TSS). This algorithm uses a linear decreasing chunk function [23]. 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 [23] proposed F=I/2P and L=1, being I the total number of tasks and P the available processors.

Other approaches have also been proposed, such as Affinity Scheduling (AS) that combines static and dynamic scheduling strategies [18].

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, which introduce additional degrees of freedom in the model. Thus, Weighted Factoring (WF) uses a variant of factoring self-scheduling, which incorporates weight factors for each processor [11]. These factors are experimentally determined from the relative computing power of the processors. Another approach, based in guided self-scheduling, is Gap-Aware Self-Scheduling (GAS) [14]. 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. Another algorithm is RAS [13] 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, but the master can redistribute tasks when a load imbalance is detected.

3. Our Approach

A general treatment of the scheduling problem can be devised in the following form. Consider, using the nomenclature of Tzen and Ni [23], an arbitrary parallel system composed by P processors and a total of I tasks to schedule. As in reference [23], we define a chore or job as a unit of work assigned to a processor (*i.e.*, a block or chunk, composed by a certain number of tasks). Let C(t) be the chunk function giving the chunk size for the t-th chore, with t defined in the interval [0, N]. Here, the index of the first chunk is 0, whereas the last one corresponds to N. Therefore, the total number of chores (allocated chunks) is N+1. The main strategy is to assign each chore to the next free processor in the system.

Two conditions must be fulfilled for any form of C(t). First, the exigency of a decreasing chunk size requires,

$$\frac{dC(t)}{dt} < 0 \tag{1}$$

and, second, the fact that the number of tasks I is fixed translates in,

$$I = \int_0^N C(t)dt \tag{2}$$

Let us consider C(t) to be an unspecified function of t: C(t)=f(t). This function can be proposed ad hoc or it can be derived from a set of assumptions. As examples of these cases we have guided self-scheduling [20] and factoring self-scheduling [12], respectively. In guided self-scheduling each chore has a chunk size given by dividing the remaining jobs among the number of processors. The resulting f(t) function resembles an exponential decay. On the other hand, in factoring self-scheduling f(t) is obtained modelling the execution time of P chores as a Pth order statistics of P independent (but identical) random variables corresponding to the execution time of a chore on each processor.

In this work, we introduce another approach to the general determination of C(t). C(t) can be expressed by a Taylor expansion of f(t),

$$C(t) = f(t_0) + \left|\frac{df(t)}{dt}\right|_0 (t-t_0) + \frac{1}{2}\left|\frac{d^2f(t)}{dt^2}\right|_0 (t-t_0)^2 + \dots \quad (3)$$

or in a more compact form,

$$C(t) = a + bt + ct^2 + \dots \quad (4)$$

Limiting the expansion of equation (4) to the constant term, we have either the pure self-scheduling (a=1) or the chunk self-scheduling (a=I/P) algorithm. A less approximate model is obtained retaining up to the linear term. This case corresponds to the trapezoid self-scheduling algorithm. In this linear case, we can obtain the C(t) function with a couple of (C(t), t) points. The first and last points used in the TSS algorithm are (I/2P, 0) and (1, N), respectively. These points were chosen in the original version of TSS [23] as conservative values selected to keep between bounds the load balance. Using these points, we have

$$C(t) = C_0 + \frac{(C_N - C_0)}{N} t \quad (5)$$

where $C_N = 1$ and $C_0 = I/2P$. This result is equivalent to that derived in reference [23]. The N value is obtained from the condition given in equation (2),

$$N = \frac{2I}{(C_0 + C_N)} \quad (6)$$

A more flexible model is obtained if we retain up to the quadratic term in equation (4),

$$C(t) = a + bt + ct^2 \quad (7)$$

Equation (7) defines a new scheduling algorithm that we will call Quadratic Self-Scheduling (QSS). To apply the QSS algorithm we need the a, b and c coefficients of equation (7). Thus, we must select three reference points (C(t), t) and solve the resulting system of equations. The two first points are selected as in the linear case. The third one is selected as ($C_{N/2}$, N/2), where N/2 is half the number of chores 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,

$$\begin{aligned} a &= C_0 \\ b &= (4C_{N/2} - C_N - 3C_0)/N \\ c &= (2C_0 + 2C_N - 4C_{N/2})/N^2 \end{aligned} \quad (8)$$

and from equation (2),

$$N = 6I/(4C_{N/2} + C_N + C_0) \quad (9)$$

Considering that the slope is given by

$$\frac{dC(t)}{dt} = b + 2ct \quad (10)$$

and since C_0 and C_N are fixed, the $C_{N/2}$ value determines if the slope of equation (7) is higher or smaller than that of equation (5) for a given point, t. Equation (10) shows that in the QSS model not only the chunk size, but also its variation over time are configurable factors.

The quadratic model reduces to the linear (TSS) when c=0. This condition leads to,

$$c = 0 \Rightarrow C_{N/2} = \frac{C_0 + C_N}{2} = \frac{2P + I}{4P} \quad (11)$$

where the last term is obtained after substitution of the C_0 and C_N values. Therefore, respect to the TSS linear algorithm two cases exist,

$$\begin{aligned} Case\ a &\Rightarrow C_{N/2} > \frac{C_0 + C_N}{2} \\ Case\ b &\Rightarrow C_{N/2} < \frac{C_0 + C_N}{2} \end{aligned} \quad (12)$$

The two cases are compared with the linear model in Figure 1. We observe for the case a) that $N_q < N_l$, where the q and l subscripts are used to distinguish the quadratic from the linear model, respectively. This smaller number of chores in the QSS algorithm translates in a smaller communication overhead to the expense of higher initial chunk sizes, see Figure 1. On the other hand, it is possible to invert this behaviour, as shown by the dashed line in Figure 1 (case b). Here, chunk sizes are smaller than in the linear case (almost all the way) to the expense of a higher number of chores, $N'_q > N_l$. In this case, we have a better load balance, but the communication overhead increases. Therefore, the QSS algorithm can be tuned to optimize the ratio load balance to overhead by selecting an appropriate value of the $C_{N/2}$ coefficient.

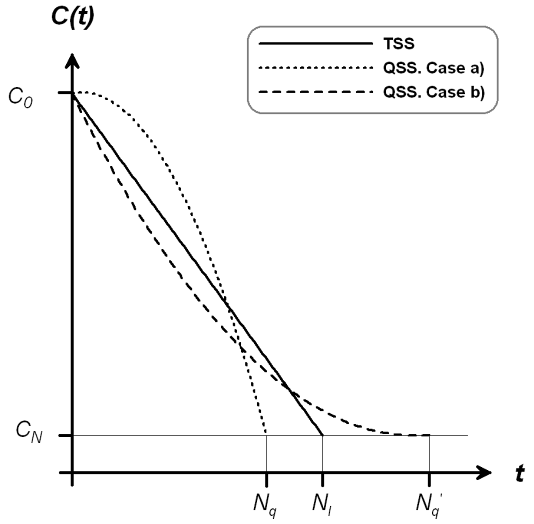

Figure 1. Evolution of the C(t) function for the TSS and QSS algorithms as a function of the number of chores, t.

4. Methodology

Our aim is to compare, at the application level, the working of QSS against other well-established self-scheduling algorithms in an Internet-based Grid of computers. Thus, apart from QSS we select the chunk self-scheduling, guided self-scheduling, factoring self-scheduling and trapezoid self-scheduling algorithms. Comparison with weighted self-scheduling schemes is left out of this initial study.

The tests have been carried out in an Internet-based Grid of computers formed by three clusters. Two of them (Tales and Hermes) are 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 (Popo) is at 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 2.4 flavour [7].

The different parameters of the self-scheduling algorithms have been configured as follows. In Chunk Self-Scheduling, the chunk size is giving 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 [12]. In Trapezoid Self-Scheduling we used the values proposed by Tzen and Ni [23] for the initial and final chunk sizes. As described in the previous section of this paper, for QSS we have selected C_0 and C_N as the initial and last chunk size of TSS. The $C_{N/2}$ parameter is defined as,

$$C_{N/2} = \frac{C_0 + C_N}{\delta} \qquad (13)$$

δ will be experimentally determined for the two cases shown in Equation (12).

In our tests, we will consider sets of tasks without any predefined relationship among their durations. Thus, we build an application able to use an amount of computing time stochastically determined. The application performs a given number of times a product of square matrices. In each product, the size of the matrices is randomly determined in run time between 900 and 1500. Sets of several thousand 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. The chunks are allocated directly using globus calls. 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 different tests, we have used two system configurations. The first one uses 20 processors of our Grid distributed 8 from Hermes, 8 from Tales and 4 from Popo. The second, uses 25 processors distributed 10 from Hermes, 10 from Tales and 5 from Popo.

To carry out our relative performance study, we consider a total of four test cases involving several thousand tasks. The cases correspond to combinations of number of allocatable tasks and number of processors. Each case is labeled as: number of tasks/number of processors. With this convention the four cases are: 2000/20, 2000/25, 5000/20, 5000/25. In this order we increase the number of different processors and the number of transatlantic connections between Spain and Mexico. Thus, we have defined our four tests in an increasing order of heterogeneity.

Initial tests have shown variation of the time used for each algorithm to complete a set of jobs. This fact is due to the variation of network load in the Spain-Mexico Internet connection. The maximum difference was found between week and weekend days. This difference amounted to 36 minutes, over a total time of about three and a half hours, for 5000 jobs allocated in 20 processors with FSS. Within weekdays, the typical difference found was 3-5 minutes. To minimize the influence of the Internet traffic, the four tests for the performance analysis are done in similar conditions. Thus, all calculations have been carried out in weekdays. In addition, each test case was completed in a single day applying consecutively the scheduling algorithms. The four tests were carried out using the same timetable.

5. Results and Discussion

The first step is to determine the $C_{N/2}$ parameter for the QSS model. We have considered the two cases collected in Equation (12). Thus, values for δ, see Equation (13), greater and smaller than 2 have been taken into account. The tests for the different values of δ are carried out on 20 processors in the Grid, using a total of 2000 independent tasks.

The results for cases a) and b) are collected in Tables 1 and 2, respectively.

Table 1. Total time evolution of the QSS algorithm as a function of the δ value in the case a) of $C_{N/2}$. 2000 tasks and 20 processors were used. Temporal data are reported as hh:mm:ss.

δ	1/3	1/2	1	2
Time:	2:11:00	1:45:41	1:58:03	1:24:13

Table 2. Total time evolution of the QSS algorithm as a function of the δ value in the case b) of $C_{N/2}$. 2000 tasks and 20 processors were used. Temporal data are reported as hh:mm:ss.

δ	2	3	4	5
Time:	1:24:13	1:07:15	1:11:20	1:21:17
δ	6	7	8	9
Time:	1:25:41	1:15:25	1:27:22	1:29:44

The results are graphically represented in Figure 2. This figure shows that case a) reaches a minimum for δ=1/2. Case b), on the other hand, exhibits a negative slope until δ=3, where a generally increasing trend is manifested. Considering that for δ=2 QSS reduces to TSS, we observe that in our heterogeneous Grid, QSS in case a) performs worse than TSS. Figure 1 shows that in case a) QSS uses fewer chunks that TSS. In these conditions, communication overhead is reduced. Therefore, load imbalance must be responsible for the increase in computing time.

For case b), Equation (9) shows that increasing δ we increase the number of chores, N. Thus, the number of small chunks augments due to the asymptotic trend of the C(t) tail, see Figure 1. This will correct for the load imbalance observed in case a), but overhead will increase. Thus, the minimum found in Figure 2 corresponds to the point where overhead balances load balance in our system. We will select this point, δ=3, for defining the QSS $C_{N/2}$ value.

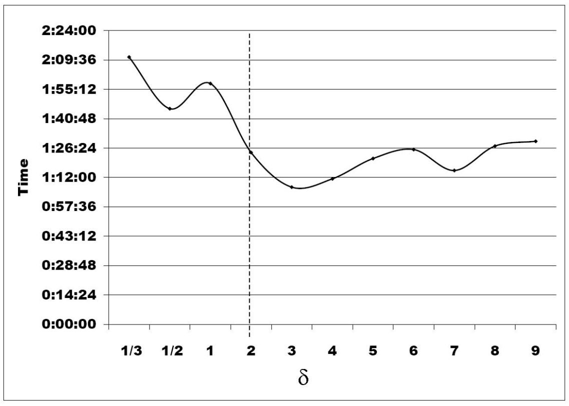

Figure 2. Performance of the QSS algorithm as a function of the δ parameter. The 2000/20 test case is used. The dotted line represents the border between a and b cases. Temporal data are reported as hh:mm:ss.

After selecting the δ value we carry out the set of tests. The number of chunks (chores) allocated in the Grid in the different cases are collected in Table 3. The number of chores follows the order: CSS<<TSS<QSS<GSS<FSS. In this same order the communication overhead increases. Thus, QSS incurs in an overhead between the extremal cases of CSS and FSS.

Table 3. Number of chores allocated by the self-scheduling algorithms in the different test cases.

Test case	CSS	GSS	TSS	FSS	QSS
2000/20	20	102	71	140	81
2000/25	25	122	91	175	100
5000/20	20	119	72	160	84
5000/25	25	144	91	200	106

The temporal results of the tests in our heterogeneous Grid are collected in Table 4. CSS clearly exhibits always the worst behaviour. This is what we can expect due to the static nature (and therefore the lack of adaptability) of CSS. Using CSS as reference, we can compare the relative performance using speedup. Here, the speedup (S) is defined, within each case, as the quotient between the time used by

CSS and the time used by each algorithm. Figure 3 collects the speedups for the different tests.

Table 4. Total time used by the different self-scheduling algorithms to complete each test case. Temporal data are reported as hh:mm:ss.

Test case	CSS	GSS	TSS	FSS	QSS
2000/20	2:20:38	1:25:05	1:25:17	1:45:42	1:07:15
2000/25	1:42:38	1:18:13	1:11:00	1:27:42	1:09:11
5000/20	5:11:34	3:23:56	3:32:21	3:52:29	3:16:04
5000/25	4:24:49	3:11:17	3:15:59	3:29:53	2:59:24

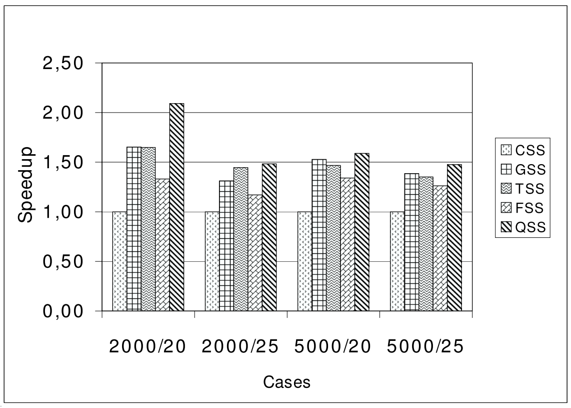

Figure 3. Speedup (S), respect to CSS, for the considered self-scheduling algorithms in the tests performed.

Figure 3 shows that QSS outperforms CSS, GSS, TSS, and FSS, in all the tests. We must consider that the results can be affected by the variability of the transmission rate between the nodes of the Grid placed in Spain and that in Mexico. However, the consistency of the results in the different tests, shows that QSS exhibits the highest performance among the tested self-scheduling algorithms. We observe that, when the number of processors and tasks increases, the QSS speedup reaches about 1.5. The better performance of QSS can be attributed to the higher adaptability (due to its three degrees of freedom) of the model to the heterogeneous environment and conditions used in the tests.

It is remarkable than GSS performs better than FSS in all cases. The same behaviour is found respect to TSS except in the 2000/25 case. This seems to contrast with the results of Tzen and Ni [23], and Hummel and Flynn [12], which show that GSS exhibits worse performance than TSS and FSS. However, these works were carried out on homogeneous systems where TSS and FSS are optimized solutions, whereas GSS is essentially a rule of thumb. In our conditions, one of the weakness of GSS (too small chunk sizes in the last chores) becomes an advantage, since the small chunks permit to compensate better the load imbalance.

Comparison of TSS and FSS shows that TSS performs better than FSS in all cases. This seems a logical result in a heterogeneous environment, since FSS is developed assuming all the computing times are independent, but equal, random variables. Therefore, FSS is a good approach only in a homogeneous environment. Since all processors are assumed equivalent, FSS works in stages. At each stage, chunks of equal size are assigned to all the processors. This strategy must incur in a load imbalance that increases as the heterogeneity of the system increases. TSS in these conditions is more adaptive, since the chunks (decreasing in size) are assigned to each free processor, obtaining a better load balance in a heterogeneous environment.

6. Conclusions

We consider in this work the problem of job scheduling at the application level in heterogeneous computing systems. In particular, the work is directed toward Internet-based Grids of computers. Focusing in adaptive, self-scheduling algorithms, we present a general formulation of the self-scheduling problem. From this formulation, we derive a new, quadratic, self-scheduling algorithm. The algorithm has more degrees of freedom than the previous ones. Thus, it is more adaptable for dealing with heterogeneity.

We perform initial tests to determine the relative performance of the new Quadratic Self-Scheduling (QSS) algorithm against previous self-scheduling schemes, namely: Chunk Self-Scheduling (CSS), Guided Self-Scheduling (GSS), Trapezoid Self-Scheduling (TSS), and Factoring Self-Scheduling (FSS). To compare in similar conditions, no specific account of the individual performance of each machine is introduced. As heterogeneous system we use an Internet-based Grid of computers that involves a transatlantic connection. Here, we allocate sets of several thousand tasks of random duration. Four different tests are performed involving different combinations of number of tasks/number of processors. In particular, the following combinations were used: 2000/20, 2000/25, 5000/20, and 5000/25.

Our results show that the communication overhead follows the order CSS<<TSS<QSS<GSS<FSS. However, the performance variation is QSS>GSS$\geq$TSS>FSS>CSS. The QSS algorithm outperforms all the other self-scheduling schemes, as a consequence of its higher adaptability to a heterogeneous environment. As expected, CSS presents the worst behaviour because the use of very large chunks produces a larger load imbalance than the other algorithms. 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. TSS gives better results because each chunk is assigned to the first free processor. Since the chunks decrease in size, TSS achieves a better load balance than FSS. On the other hand, in GSS the last chunks tend to be smaller than in FSS and TSS. Thus, in a heterogeneous environment, GSS seems to give better load balance than FSS and TSS. The present results suggest that, at the application level and working in an Internet-based Grid of computers, the load balance is the main factor defining the performance of the system.

In the present initial study, the heterogeneity of the systems is reflected in the new self-scheduling strategy (QSS) through the calibration of the $C_{N/2}$ value, see Equation (13). Previous works [9, 2, 4, 6, 5, 11] have shown that customization of self-scheduling algorithms in a heterogeneous environment, using the particular characteristics of the processors, greatly increases the performance. Mathematically, this is a consequence of the inclusion of additional degrees of freedom in the model, which makes it more adaptable. A clear example is the weighted-factoring strategy applied by Flynn Hummel et al. [11]. In the present paper, we assess the higher performance of QSS over previous approaches in a heterogeneous environment. Therefore, to increase its efficiency, the next step is to incorporate in the QSS model the specific characteristics of the machines conforming the system.

7. 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 Popo cluster.

References

[1] F. Berman. *High-performance schedulers*. in: I. Foster, C. Kesselman (Eds.). The Grid: Blueprint for a New Computing Infrastructure, Morgan-Kaufmann, pp. 279-309, 1999.

[2] 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.

[3] T. L. Casavant and J. G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing. *IEEE Transactions on Software Engineering*, 14(2):141–154, 1988.

[4] A. T. Chronopoulos, M. Benche, D. Grosu, and R. Andonie. A class of loop self-scheduling for heterogeneous clusters. *Proceeding of the 2001 IEEE International Conference on Cluster Computing*, pages 282–294, Newport Beach, USA, October 2001.

[5] A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali. Distributed loop scheduling schemes for heterogeneous computer systems. *Concurrency and Computation: Practice and Experience*, 18, Issue 7:771–785, 2006.

[6] A. T. Chronopoulos, S. Penmatsa, and N. Yu. Scalable loop self-scheduling schemes for heterogeneous clusters. *Proceeding of the 2002 IEEE International Conference on Cluster Computing*, Chicago, USA, September 2002.

[7] I. Foster. Globus toolkit version 4: Software for service-oriented systems. In *IFIP International Conference on Network and Parallel Computing*, pages 2–13, Springer-Verlag LNCS 3779, 2005.

[8] I. Foster and C. Kesselman. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kauffman Publishers, San Francisco, 1999.

[9] B. Hamidzadeh, D. J. Lilja, and Y. Atif. Dynamic scheduling techniques for heterogeneous computing systems. *Concurrency: Practice and Experience*, 7:633–652, 1995.

[10] J. Herrera, E. Huedo, R. S. Montero, and I. M. Llorente. Ejecución distribuida de bucles en grids computacionales. 3^a *Reunión para la Red Temática en Grid Middleware*, Logroño 2005.

[11] S. F. 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*, pages 318–328, Padua, Italy, June 1996.

[12] S. F. Hummel, E. Schonberg, and L. E. Flynn. Factoring: A method for scheduling parallel loops. *Comm. ACM*, 35(8):90–101, August 1992.

[13] 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*, pages 974–980, Las Vegas, July 1998.

[14] A. Kejariwal, 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.

[15] T. H. Kim and J. M. Purtilo. Load balancing for parallel loops in workstation clusters. *Proc. of Intl. Conference on Parallel Processing*, 3:182–189, 1996.

[16] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Trans. Software Engineering*, SE-11(10):1001–1016, October 1985.

[17] D. J. Lilja. Exploiting the parallelism available in loops. *IEEE Computer*, 27(2):13–26, 1994.

[18] E. P. Markatos and T. J. LeBlanc. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 05(4):379–400, April 1994.

[19] 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.

[20] C. D. Polychronopoulos and D. Kuck. Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. *IEEE Trans. on Computers*, 36:1425–1439, December 1987.

[21] P. J. Sokolowski, 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, to be appear in June 2006.

[22] P. Tang and P. C. Yew. Processor self-scheduling for multiple nested parallel loops. *In Proceedings of the 1986 International Conference on Parallel Processing*, pages 528–535, 1986.

[23] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, January 1993.

[24] C.-T. Yang and S.-C. 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.