Futur. Gener. Comp. Syst., 25, 617-626 (2009)

Derivation of Self-Scheduling Algorithms for heterogeneous distributed computer systems: Application to Internet-based Grids of computers

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

Computational Chemistry and High Performance Computing group, Escuela Superior de Informática, Universidad de Castilla-La Mancha, Paseo de la Universidad 4, 13071, Ciudad Real, Spain.

Abstract

Self-scheduling algorithms are useful tools for achieving load balance in heterogeneous computational systems. Therefore, they can be applied 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 deriving an explicit form for the rate of variation of the chunks distribution function. From the first family, we propose a Quadratic Self-Scheduling (QSS) algorithm. From the second, two new algorithms, Exponential Self-Scheduling (ESS) and Root Self-Scheduling (RSS) are introduced. QSS, ESS and RSS are tested 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. However, RSS shows a poor performance, a fact traced back to the curvature of the chunks distribution function.

Key words: Self-Scheduling Algorithms, Chunk size, Load balancing, Computational Grid

* Corresponding author.
Email addresses: javier.diaz@uclm.es (Javier Díaz), sebastian.reyes@uclm.es (Sebastián Reyes), alfonso.nino@uclm.es (Alfonso Niño), camelia.munoz@uclm.es (Camelia Muñoz-Caro).

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 computational resources. This is specially important when the resources are not only distributed, but also heterogeneous, as in a computational Grid [15]. 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. 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 [5]). In particular, dynamic self-scheduling algorithms are extensively used in practical applications. These algorithms represent adaptive schemes where tasks are allocated in run-time. Self-scheduling algorithms were initially developed to solve parallel loop scheduling problems in homogeneous memory-shared systems, see for instance [25]. 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 [3, 4, 7–9, 11, 17, 21, 23, 30, 39]. In addition, some works about their performance on Grid connected clusters of computers have also been reported [12, 13, 18, 29, 36].

In the general case, a computational Grid can be thought of as a dynamic heterogeneous system. Here, by dynamic, as opposed to static, we mean that the characteristics defining the system are not fixed. In particular, the availability and performance of computer nodes (CPUs) and network links can change over the time. From this standpoint, a suitable strategy to the development of efficient Grid scheduling strategies is to start by deriving efficient scheduling schemes for static heterogeneous distributed computer systems. Later, the dynamic behaviour of the system can be incorporated into the parameters defining the schemes. Therefore, as core components of more efficient Grid scheduling models, in this work we present new self-scheduling schemes. Thus, two general families of self-scheduling algorithms are developed. From these families, new flexible self-scheduling algorithms, oriented toward heterogeneous distributed systems, are derived. These algorithms are compared with prior self-scheduling strategies in a dedicated, static, heterogeneous Grid of computers.

The rest of this paper is organized as follows. Section II presents an overview of the field of self-scheduling algorithms. In section III, we introduce two families

of self-scheduling algorithms and new self-scheduling schemes are presented. Section IV details the methodology used for comparing the new approaches with well-established self-scheduling schemes. Section V presents and interprets the results found in the comparative study. Finally, in section VI we present the main conclusions of this paper and in section VII the perspectives for future works.

2 Background

When trying to allocate a set of tasks among several processors we have two complementary effects: load balance and communication overhead. The theoretical work of Kruskal and Weiss [24] showed 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 achieving good load balance but causing too much communication and scheduling overhead. On the other hand, we have Chunk Self-Scheduling (CSS) [37], 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 for an equilibrium between load balance and overhead. In this context, Kruskal and Weiss [24] 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.

The simplest of these algorithms is Guided Self-Scheduling (GSS), which dynamically changes the number of tasks assigned to each processor [31]. 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, in the early chunks GSS may assign too many tasks to the processors, and the remaining tasks may not be sufficient to guarantee a good load balance.

On the other hand, we have Factoring Self-Scheduling (FSS) that 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 [20]. The chunk size is determined by dividing the number of remaining tasks by the product of the number of processors and by 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 [38]. 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 sizes balance 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 [38] proposed as suboptimal values $F = I/2P$ 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 such as computational Grids. However, they could be not enough flexible (they have not enough degrees of freedom) to adapt efficiently to this heterogeneous and dynamic environment. Different algorithms have been proposed for this case, such as Affinity Scheduling (AS) [26]. AS combines static and dynamic scheduling strategies. Other approaches introduce additional degrees of freedom in the model. Thus, we have proposed previously a new self-scheduling algorithm called Quadratic Self-Scheduling (QSS) [11], [12]. This algorithm is based on a quadratic form for the chunks distribution function. Therefore, we have three degrees of freedom, which provide higher adaptability to distributed heterogeneous systems. A different approach is to take into account the environment characteristics. Here, most of the algorithms are based in a master-slave architecture, where the master obtains the information necessary to allocate the different tasks among the slaves. Thus, Weighted Factoring (WF) uses a variant of Factoring Self-Scheduling, which incorporates weight factors for each processor (slave) [19]. These factors are experimentally determined from the relative computing power of the processors. In WF each processor weight remains constant throughout the parallel execution. Banicescu and Liu [2] have proposed a method called Adaptive Factoring (AF). AF adjusts the processors weights according to timing information reflecting variations in the available computing power of the slaves. Chronopoulos et al. [7] extended the TSS algorithm, proposing the Distributed TSS (DTSS). In this algorithm, chunks sizes are weighted by the relative power of the server processors and by the number of processes in their run-queue at the moment of requesting work from the master. Ciorba at el. extended in [6] the DTSS algorithm, proposing DTSS+SP to handle loops with dependencies on heterogeneous clusters via synchronization points (SPs). In [8], Chronopoulos et al. proposed a Hierarchical DTSS (HDTSS). This is a hierarchical method, which tries to avoid the bottleneck problems of the centralized schemes like DTSS. Also, there are self-scheduling algorithms based on the history of task timing results, like SAS [34]. This self-scheduling algorithm attempts to balance dynami-

cally 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) [22]. 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, there is another kind of self-scheduling algorithms that can redistribute the workload when a load imbalance is detected, for example RAS [21]. This 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. Other approach is TREES [10] [23]. This is a distributed load balancing scheme that statically arranges the processors in a logical communication topology based on the computing powers of the processors involved. When a processor becomes idle, it asks for work from a single, predefined partner, which migrates half its work to the idle processor.

As the above examples show, self-scheduling algorithms are at the core of more refined techniques for task scheduling in Grid environments. Therefore, new, more efficient algorithms are of interest for defining new Grid scheduling approaches.

3 New Self-Scheduling Schemes

This section presents two different families of Self-Scheduling methods. As usual [38], both of them are derived using a continuous chunks distribution function, $C(t)$. This function 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

A general treatment of the scheduling problem can be devised in the following form. Consider, using the nomenclature of Tzen and Ni [38], an arbitrary parallel system composed by P processors with a total of I tasks to schedule. As in reference [38], we define a 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 chunk, with t defined in the interval $[0, N]$. The main strategy is to assign each chunk 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,

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

This condition means that $C(t)$ is a monotonic decreasing function and, therefore,

$$C(t_b) \leq C(t_a) \qquad \forall\ t_b > t_a \quad \in [0, N] \tag{2}$$

The second condition is that the number of tasks I is fixed. This translates to,

$$I = \int_0^N C(t)dt \tag{3}$$

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 two cases we have Guided Self-Scheduling [31] and Factoring Self-Scheduling [20], respectively.

In this work, we introduce another approach to determine $C(t)$. $C(t)$ can be expressed by a Taylor expansion. In this way, assuming that $C(t)$ is n-derivable in a point $t_a \in [0, N]$, we can build a Taylor polynomial $P(t)$ of n order [1],

$$P(t) = \sum_{k=0}^{n} \frac{C^k(t_a)}{k!}(t - t_a)^k \tag{4}$$

where,

$$C^k(t_a) = \frac{d^k C(t_a)}{dt^n} \tag{5}$$

and

$$C^0(t_a) = t_a \tag{6}$$

$P(t)$ is an approximation to $C(t)$ and, therefore:

$$C(t) = P(t) + E_n(t) = \sum_{k=0}^{n} \frac{C^k(t_a)}{k!}(t - t_a)^k + E_n(t) \tag{7}$$

where $E_n(t) = C(t) - P(t)$ is the approximation error. The equation (7) defines the Taylor formula with remainder [1]. It can be shown [1] that

$$E_n(t) = \frac{1}{n!}\int_{t_a}^{t}(t - s)^n C^{(n+1)}(s)ds \tag{8}$$

from which it is deduced [1] that given two values m and M such as,

$$m \leq C^{(n+1)}(t) \leq M \qquad \forall\ t \quad \in [0, N] \tag{9}$$

then

$$m\frac{(t - t_a)^{n+1}}{(n+1)!} \leq E_n(t) \leq M\frac{(t - t_a)^{n+1}}{(n+1)!} \quad if \quad t > t_a \tag{10}$$

and

$$m\frac{(t_a - t)^{n+1}}{(n+1)!} \leq (-1)^{n+1}E_n(t) \leq M\frac{(t - t_a)^{n+1}}{(n+1)!} \quad if \quad t_a > t \tag{11}$$

Equations (10) and (11) represent a bracketing of the error. They show that the error is reduced (it depends inversely on $(n + 1)!$) when the terms of the polynomial $P(t)$ are increased. This conclusion will be applied to our treatment. Approximating $C(t)$ by a Taylor polynomial we have

$$C(t) \simeq C(t_0) + \left[\frac{dC(t)}{dt}\right]_{t_0}(t - t_0) + \frac{1}{2}\left[\frac{d^2C(t)}{dt^2}\right]_{t_0}(t - t_0)^2 + \ldots \tag{12}$$

or in a more compact form,

$$C(t) \simeq a + bt + ct^2 + \ldots \tag{13}$$

Limiting the expansion of equation (13) 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. Here, we can obtain the $C(t)$ function with a couple of $(C(t), t)$ points. The first $(C_0, 0)$

and last (C_N, N) points used in the TSS algorithm are $(I/2P, 0)$ and $(1, N)$, respectively. These points were chosen as conservative values to keep between bounds the load balance [38]. Using these points, we have that equation (13) reduces for a linear expansion to.

$$C(t) = C_0 + \frac{(C_N - C_0)}{N} t \tag{14}$$

where $C_N = 1$ and $C_0 = I/2P$. This result is equivalent to that derived in reference [38].

Different algorithms can be obtained by retaining additional terms of the Taylor expansion. For the quadratic case we have Quadratic Self-Scheduling [11], [12], [13].

3.1.1 Quadratic Self-Scheduling (QSS)

If we retain up to the quadratic term of the Taylor expansion (13), we obtain, in the sense of equations (10) and (11), a more flexible and less approximate model than the TSS algorithm. In this case,

$$C(t) = a + bt + ct^2 \tag{15}$$

where t represents the t-th chunk assigned to a processor. To apply the QSS algorithm we need the a, b and c coefficients of equation (15). The most direct solution, albeit approximate, is to select three reference points $(C(t), t)$ and using equation (13) solve for the resulting system of equations. In previous studies [11], [12], [13], the two first points were selected as in the TSS linear case, that is $(I/2P, 0)$ and $(1, N)$. 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 the system of equations for a, b and c, we obtain,

$$\begin{aligned} a &= C_0 \\ b &= (4C_{N/2} - C_N - 3C_0)/N \\ c &= (2C_0 + 2C_N - 4C_{N/2})/N^2 \end{aligned} \tag{16}$$

where N is obtained by applying equation (3), which yields,

$$N = 6I/(4C_{N/2} + C_N + C_0) \tag{17}$$

On the other hand, the $C_{N/2}$ value is given by,

$$C_{N/2} = \frac{C_N + C_0}{\delta} \tag{18}$$

and since C_0 and C_N are fixed, the $C_{N/2}$ value determines the slope of equation (15) at a given point. Therefore, depending on δ, the slope of the quadratic function for a fixed t is higher or smaller than that of the linear case (TSS algorithm), which corresponds to $\delta = 2$. So, we have case a), where the slope is higher than in the linear model, and case b), where the slope is smaller than in the linear case. These two cases are defined as

$$\begin{aligned} Case\ a \Rightarrow C_{N/2} > \frac{C_0 + C_N}{2} \ \Rightarrow \frac{d^2C(t)}{dt^2} < 0 \\ Case\ b \Rightarrow C_{N/2} < \frac{C_0 + C_N}{2} \ \Rightarrow \frac{d^2C(t)}{dt^2} > 0 \end{aligned} \tag{19}$$

The two cases are compared with the linear model in Figure 1. As we can see, case a) is a concave down function. Here $N_q < N_l$, where the q and l subscripts are used to distinguish the quadratic from the linear model. The smaller number of chunks in the QSS algorithm translates into 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), where the function is concave up. Here, chunk sizes are smaller than in the linear case (almost all the way) to the expense of a higher number of chunks, $N'_q > N_l$. 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 of load balance to overhead by selecting an appropriate value of the $C_{N/2}$ coefficient.

3.2 Methods based in the slope of the chunks distribution function

Here, the starting point is the slope of the chunks distribution function, $C(t)$. In this case, we model 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 will have the general expression,

$$\frac{dC(t)}{dt} = f(t) \tag{20}$$

Equation (20) defines a differential equation. After integration we will have an

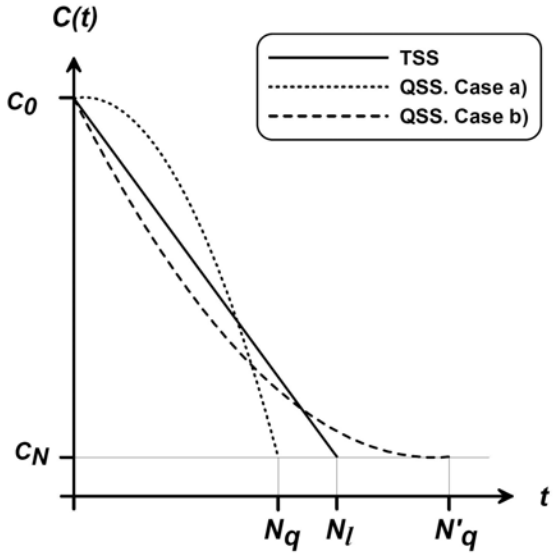

Fig. 1. Evolution of the $C(t)$ function for the TSS and QSS algorithms as a function of the number of chunks, t.

explicit functional form for $C(t)$. Different expressions for $f(t)$ yield different algorithms. Two examples are presented here.

3.2.1 Exponencial Self-Scheduling

A first approach is to consider that the slope (negative) is proportional to the chunk size:

$$\frac{dC(t)}{dt} = -kC(t) \tag{21}$$

Here, k is a parameter and t represents the t-th chunk assigned to a processor. Equation (21) can be integrated by separation of variables yielding,

$$C(t) = \left(\frac{I}{2P}\right) e^{-kt} \;\Rightarrow \frac{d^2C(t)}{dt} = k^2 \left(\frac{I}{2P}\right) e^{-kt} > 0 \tag{22}$$

where we have used $C(0) = I/2P$, as proposed by Tzen and Ni [38]. Equation (21) defines a new self-scheduling method that we call Exponential Self-Scheduling (ESS). In this method, k is a parameter related to the working environment. The second derivative in equation (22) shows that the curve is concave up. In addition, from equation (3) we get,

$$N = -ln[1 - 2Pk]/k \tag{23}$$

Equation (23) shows that the number of chunks, N, is independent of the number of tasks, I.

3.2.2 Root Self-Scheduling

A second approach is to consider that the slope (negative) is inversely proportional to C(t)

$$\frac{dC(t)}{dt} = -k/C(t) \tag{24}$$

Here, k is a parameter and t represents the t-th chunk assigned to a processor. Equation (24) can be integrated by separation of variables yielding,

$$C(t) = \sqrt{\left(\frac{I}{2P}\right)^2 - 2kt} \Rightarrow \frac{d^2C(t)}{dt} = \frac{-k^2}{\sqrt{\left(\left(\frac{I}{2P}\right)^2 - 2kt\right)^3}} < 0 \tag{25}$$

where we have used $C(0) = I/2P$, as proposed by Tzen and Ni [38]. Equation (24) defines a new self-scheduling method that we call Root Self-Scheduling (RSS). As in ESS, k is a parameter related to the working environment. The second derivative in equation (25) shows that the curve is concave down.

The parameters defining the previous algorithms, QSS, ESS and RSS, are considered constant with time. This corresponds formally to a parallel system with fixed characteristics. Therefore, to determine the usefulness of the new algorithms, we must test their efficiency in a controlled heterogeneous parallel system. Thus, QSS, ESS and RSS are compared against previous self-scheduling schemes found in the literature in a dedicated Grid of computers.

4 Methodology

Our aim is to compare, at the application level, the performance of QSS, ESS and RSS against previous self-scheduling algorithms in an Internet-based Grid of computers. Thus, apart from QSS, ESS and RSS we select Chunk Self-Scheduling, Guided Self-Scheduling, Factoring Self-Scheduling and Trapezoid Self-Scheduling.

The tests are 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 4.0.3 flavour [16]. To allocate the tasks on the Grid we use the 5.0 version of the GridWay metascheduler [27] through its DRMAA API [14] [28].

The different parameters of the self-scheduling algorithms have been configured as follows. In Chunk Self-Scheduling, the chunk size is by definition given as 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 used a value of 2 for the α parameter, the optimal value obtained in the original statistical analysis of the FSS method [20]. In Trapezoid Self-Scheduling we have used the optimal values found by Tzen and Ni [38] for the initial and final chunk sizes. In QSS we have selected C_0 as the initial optimal 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 and RSS we have selected C_0 as the optimal starting chunk size of TSS. The parameter k will be experimentally determined for both algorithms.

The use of adjustable parameters permits to adapt our algorithms to the environment. In this study, we will obtain these parameters from actual, experimental measures. This approach is useful for assessing the performance of the algorithms. However, it can be inefficient for the regular use. A more practical approach could be based on heuristics. This point will be discussed in the Future Works section.

In the tests, we consider sets of tasks without any predefined relationship among their durations. This is a simple way to represent real situations such as the computation of the Mandelbrot set, computer graphics rendering, BLAST [32] searches in bioinformatics, or parameter sweep computations such as the calculation of potential energy hypersurfaces in molecules [33] or the determination of group difference pseudopotentials for hybrid quantum mechanics-molecular mechanics calculations [35]. Thus, we have built an application with a stochastically determined computing time. The application performs several times a product of square matrices of floating point numbers. The size of the matrices is randomly determined in run time between 300 and 1000. Sets of several thousand products (tasks) are used for allocation in the Grid using the different self-scheduling algorithms. Each chunk size corresponds to a number of matrix products (tasks). Thus, each chunk defines a 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 different tests, we have used two system configurations. The first uses 20 processors of our Grid distributed 8 from Hermes, 8 from Tales and 4 from Popo. The second, uses 26 processors distributed 9 from Hermes, 11 from Tales and 6 from Popo. The starting order of assignment of chunks to the Grid nodes (in the present case the three clusters used) is chosen randomly but kept in all the tests. This order is: Hermes, Tales and Popocatepetl. No additional characteristics of the Grid are taken into account in the scheduling process. The objective is to make the performance of each self-scheduling algorithm a function of only its intrinsic ability to balance workload and network overhead in the Grid environment. However, as commented in the Future works section of this paper, this initial approach can be improved by considering system characteristics such as the relative computational power of the Grid nodes or the network performance.

To carry out the performance study, we consider a total of three test cases involving several thousand tasks. The cases correspond to different 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 three test cases are: 2804/20, 5608/20 and 5608/26. The calculations needed for each test have been performed three times, to obtain average results. As performance index we use the speedup relative to the worst case.

5 Results and Discussion

We perform a two-dimensional, 2D, study to obtain the optimal values of the $C_{N/2}$ and C_N parameters for the QSS algorithm. Thus, a grid of points for different values of $C_{N/2}$ and C_N is considered. This study has been performed for each test case using averages of three measures for each point. The $C_{N/2}$ values are obtained by modifying the δ parameter, see equation (18). The δ values range from three to seven in increments of one. We showed previously [12] that values smaller or higher than those affect negatively the efficiency of the algorithm. In particular, for $\delta < 2$, case a) of Figure 1, the curve is concave down. On the other hand, the C_N parameter will range from one to eight in increments of one.

Figure 2 shows the results for QSS in the three test cases. The first case, bottom of the figure, corresponds to the 2804/20 test. Here, we find a valley along δ for $C_N = 1 - 3$. The global minimum is found for $\delta = 3$ and $C_N = 2$, with a local minimum appearing for $\delta = 3$, $C_N = 6$. It is interesting to indicate that even for the worst combination of δ and C_N, QSS is more efficient that the rest of algorithms, except ESS.

On the other hand, the 5608/20 case is shown in the middle of Figure 2. Here,

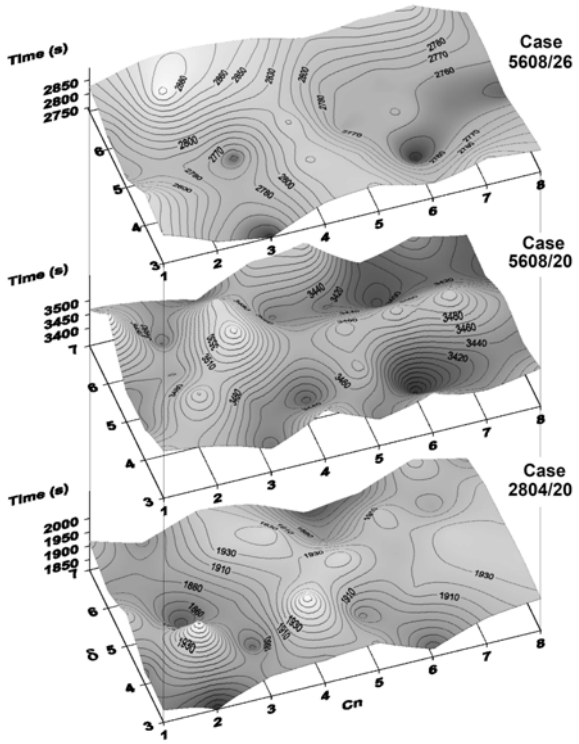

Fig. 2. Performance of the QSS algorithm as a function of δ and C_N parameters. Temporal data in seconds. Interval between isocontour lines 10 seconds. Darker zones correspond to lower value zones.

the number of tasks has been increased. Therefore, the behaviour of QSS exhibits some changes. We observe that the algorithm has the best behaviour for δ values between 4 to 6 and C_N values between 5 to 7. The global minimum is found for $\delta = 4$ and $C_N = 6$. Now, a local minimum appears for $\delta = 3$ and $C_N = 3$.

Finally, the results for 5608/26 case are shown at the top of Figure 2. In this case, the number of processors has been increased with respect to the previous test. However, the zone where the algorithm has the best behaviour is similar, although it is larger than in the previous case. The global minimum is found in the same point, *i.e.*, $\delta = 4$ and $C_N = 6$. Again, a local minimum appears for $\delta = 3$ and $C_N = 3$.

The present results show that the best values of δ and C_N can be different depending on the number of tasks and the number of processors. In this study, we have selected the best values found for each case.

Respect to ESS, the k parameter has been also optimized for each test case. We have selected k values ranging from 0.010 to 0.024, in increments of 0.001. Smaller or larger k values do not provide any significant information, 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,

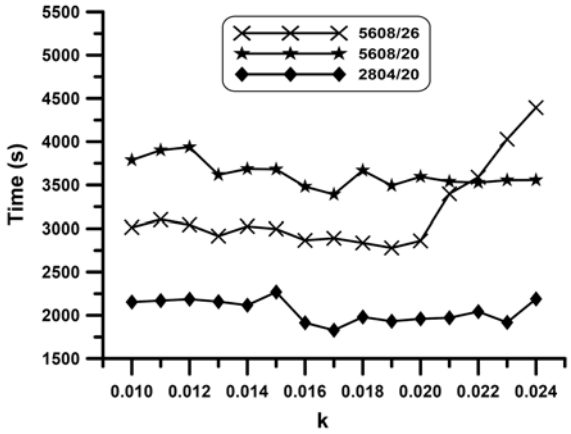

Fig. 3. Performance of the ESS algorithm as a function of the k parameter. Temporal data in seconds.

which leads to very small chunk sizes and, therefore, to a maximization of communication overhead.

Figure 3 represents the performance of the ESS algorithm as a function of the k parameter for the different test cases. Average of three measures are used for each point. The cases 2804/20 and 5608/20 exhibit a similar behaviour. They show a generally negative slope until $k = 0.017$, where the minimum is found. From here a generally increasing trend is manifested. The minimum found corresponds to the point where communication overhead compensates load balance in our system. On the other hand, the case 5608/26 has a similar behaviour, with a decreasing trend in the range $k = 0.017 - 0.019$. Now the minimum is reached for $k = 0.019$. From now on we will use for each test case the corresponding optimal k value.

The last algorithm is RSS that depends on a single, k, parameter. As in ESS, the parameter has been optimized for each test case. Thus, we have selected k values ranging from 1 to 41, in increments of 2. We consider this interval to be representative of the RSS behaviour, since smaller or larger k values yield few and long chunks. This fact minimizes the communication overhead, but increases load imbalance.

Figure 4 represents the working of the RSS algorithm as a function of the k parameter, in the three test cases. Again, averages of three measures are used for each point. Here, all cases have a similar behaviour with a generally increasing trend as k decreases. The minimum is found for $k = 35$.

After calibration of QSS, ESS and RSS, we compare their performance against the rest of self-scheduling algorithms. Each algorithm uses the optimal values of its parameters. The same three tests are used. The number of chunks allocated in the Grid for the different cases are collected in Table 1. The number of chunks follows the order: CSS<RSS<ESS<TSS<QSS<GSS<FSS for

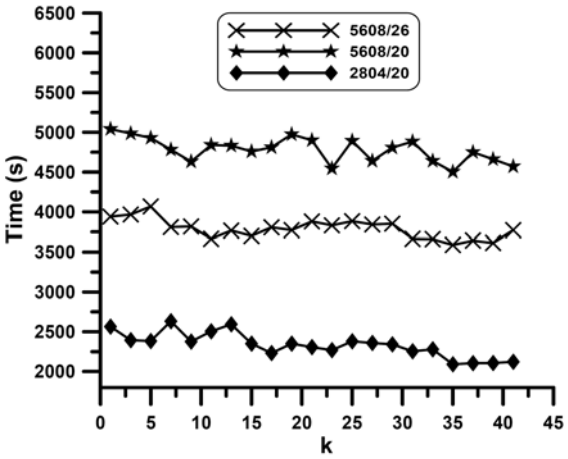

Fig. 4. Performance of the RSS algorithm as a function of the k parameter. Temporal data in seconds.

Table 1
Number of chunks allocated by the self-scheduling algorithms in the different test cases.

Test case	CSS	GSS	TSS	FSS	QSS	ESS	RSS
2804/20	20	108	72	144	84	65	50
5608/20	20	122	72	168	98	66	42
5608/26	26	152	95	200	129	175	57

20 processors and CSS<RSS<TSS<QSS<GSS<ESS<FSS for 26 processors. The difference arises because of the large increase of chunks in ESS when the number of processors goes from 20 to 26, see equation (23). Clearly, in this order the communication overhead increases.

Table 2 collects the time used in the tests (average values) for the different algorithms and the corresponding standard deviation, σ. The changing values of the standard deviation, in a range of about 10 to 170 seconds, reflect the dynamic nature of an actual Grid as the one used here. Anyway, in the worst case (TSS in the 2804/20 test) the σ represents just a 6.4 % of the average time. Table 2 shows that CSS gives the poorest performance. This is what we can expect due to its static nature (and therefore the lack of adaptability), with a small number of large chunks. Thus, using CSS as reference, we compare the relative performance using the speedup, here defined as the quotient between the time used by CSS and the time used by each algorithm. Figure 5 collects the speedups so obtained.

In all cases, QSS outperforms CSS, GSS, TSS, FSS, RSS 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 tests. It is remarkable than ESS, with only two parameters has a similar performance to QSS. Therefore, this new algorithm is not only simple, but seems to be almost as efficient as QSS. On the other

Table 2
Average total time used by the different self-scheduling algorithms in the tests performed. Standard deviation is included. Temporal data in seconds.

Test Case	CSS	GSS	TSS	FSS	QSS	ESS	RSS
2804/20	2910	2065	2068	2121	1811	1828	2091
σ	57.79	118.93	132.88	106.07	48.52	39.00	124.28
5608/20	5051	3501	3577	3417	3331	3391	4503
σ	171.13	161.28	128.92	41.84	134.89	104.97	148.09
5608/26	4509	3052	2980	2877	2734	2778	3590
σ	150.46	15.01	57.18	22.89	51.11	10.40	129.73

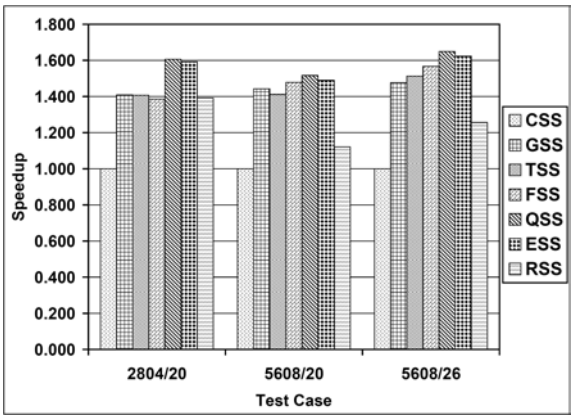

Fig. 5. Speedup (S), respect to CSS, for the considered self-scheduling algorithms in the tests performed.

hand, RSS, shows a poor perfomance and it only improves with respect to CSS. Both, RSS and CSS have a similar behaviour, generating a small number of chunks, see Table 1. Despite this, RSS uses a larger number of chunks than CSS, see Table 1, allowing for a better load balancing. It is interesting to note that RSS corresponds to a concave down curve, see equation (25), as the case a) of QSS, see equation (19). Concave down functions for $C(t)$ are translated in a small number of large chunks and therefore they exhibit, as shown here for RSS and in [12] for QSS, a bad behaviour due to inefficient load balancing. Therefore, RSS and the case a) of QSS show that concave down functions do not provide good scheduling schemes.

Comparing the data in Table 1 with the speedups of Figure 5 we observe that the performance does not degrade when the number of chunks, and therefore the communication overhead, increases. The present results show that the main factor affecting negatively the performance in a Grid system is load imbalance.

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. We focus in the development of new self-scheduling algorithms as core components of more refined scheduling methods in computational Grid environments. Thus, we introduce two families of self-scheduling algorithms. The first one is defined by methods looking for an explicit form for the chunks distribution function. The second corresponds to methods focusing in an explicit form for the rate of variation of the chunks distribution function. From the first family, we selected a Quadratic Self-Scheduling (QSS) algorithm. From the second, two new self-scheduling algorithms, Exponential Self-Scheduling (ESS) and Root Self-Scheduling (RSS), are introduced. The parameters of QSS, ESS and RSS models are obtained in an Internet-based Grid of Computers.

We perform tests to determine the relative performance of the new algorithms against previous self-scheduling schemes, namely: Chunk Self-Scheduling (CSS), Guided Self-Scheduling (GSS), Trapezoid Self-Scheduling (TSS), and Factoring Self-Scheduling (FSS). As heterogeneous system we use an Internet-based Grid of computers that involves a transatlantic connection and processors with different architectures. Here, we allocate sets of several thousand tasks of random duration. Three different tests are performed involving different combinations of number of tasks/number of processors. All measures have been performed three times, to obtain an average. In particular, the following combinations were used: 2804/20, 5608/20, and 5608/26.

The tests show that the QSS algorithm outperforms all the other self-scheduling schemes and slightly ESS, because of its higher adaptability to a heterogeneous environment. CSS and RSS present the worst behaviour. This is due to the use of very large chunks. This fact produces a large load imbalance. The FSS, GSS and TSS algorithms have similar behaviour in the different tests.

The ordering of the considered algorithms with respect to the number of chunks is maintained in the different test cases, except for ESS. Here, the number of chunks increases greatly when the number of processors grows. Despite this, ESS remains as one of the most efficient approaches. In general, we observe that a small number of chunks is associated to the poorest performing algorithms. However, the opposite is not true. A small number of chunks implies a larger load imbalance, whereas a larger number of chunks implies a larger communications overhead. Therefore, the present tests show that load imbalance is the main factor determining the performance of an Internet-based Grid of computers. The tests have also shown that concave down chunks distribution functions are inefficient, because they generate fewer number of chunks

than concave up functions. This causes important load imbalances.

7 Future Works

In the present study, the heterogeneity of the systems is incorporated in the new self-scheduling strategies (QSS and ESS) through the calibration of the $C_{N/2}$, C_N and k parameters. Previous works [4, 7–9, 17, 19] 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. Here, QSS and ESS use constant parameters in its definition. This formally corresponds to a parallel system where its defining characteristics are constant with time. Therefore, since an actual Grid system is dynamic in nature, we will have a different value of the parameters each time its characteristics change. In other words, the most direct solution to incorporate the dynamic behaviour of an actual Grid system in our self-scheduling algorithms is to make their parameters a function of time. One way to achieve this would be to determine a new set of parameters each time one or several chunks have been allocated. To such a goal, a direct calibration in the actual system is clearly impractical. An approach based in a simulation of the actual system seems much more appropriate.

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. They 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] T.M. Apostol, Calculus, second ed., Reverté, Barcelona, Spain, 1998.

[2] I. Banicescu, and Z. Liu: Adaptive Factoring: A Dynamic Scheduling Method Tuned to the Rate of Weight Changes, in: Proc. of the High Performance Computing Symp., 2000, pp. 122-129.

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

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao: Application-Level Scheduling on Distributed Heterogeneous Networks, in: Proc. of Supercomputing '96, 1996, p. 39.

[5] T.L. Casavant, J.G. and Kuhl: A Taxonomy of Scheduling in General-Purpose Distributed Computing. IEEE Trans. on Soft. Eng. 14 (1988) 141-154.

[6] F.M. Ciorba, T. Andronikos, I. Riakiotakis, A.T. Chronopoulos, and G. Papakonstantinou: Dynamic Multi Phase Scheduling for Heterogeneous Clusters, in: Proc. of the 20th IEEE Int. Parallel & Distributed Processing Symp. (IPDPS 2006), 2006, p. 10.

[7] A.T. Chronopoulos, M. Benche, D. Grosu, and R. Andonie: A Class of Loop Self-Scheduling for Heterogeneous Clusters, in: Proc. of the 2001 IEEE Int. Conf. on Cluster Computing, 2001, pp. 282-294.

[8] A.T. Chronopoulos, S. Penmatsa, N. Yu: Scalable Loop Self-Scheduling Schemes for Heterogeneous Clusters, in: Proc. 2002 IEEE Int. Conf. on Cluster Computing, 2002, pp. 353-359.

[9] A.T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali, Distributed Loop Scheduling Schemes for Heterogeneous Computer Systems, Concurr.: Pract. Exp. 18 (2006) 771-785.

[10] S.P. Dandamudi, and T.K. Thyagaraj: A hierarchical processor scheduling policy for distributed-memory multicompute system, in: Proc. 4th Int. Conf. on High Performance Computing, 1997, pp. 218-223.

[11] J. Díaz, S. Reyes, A. Niño, and C. Muñoz-Caro: Un Algoritmo Autoplanificador Cuadrático para Clusters Heterogéneos de Computadores, in: XVII Jornadas de Paralelismo, 2006, pp. 379-382.

[12] J. Díaz, S. Reyes, A. Niño, and C. Muñoz-Caro: A Quadratic Self-Scheduling Algorithm for Heterogeneous Distributed Computing Systems, in: Proc. of the 5th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar '06), 2006, pp. 1-8.

[13] J. Díaz, S. Reyes, A. Niño, C. and Muñoz-Caro: New Self-Scheduling Schemes for Internet-Based Grids of Computers, in: 1st Iberian Grid Infrastructure Conference (IBERGRID), 2007, pp. 184-195.

[14] Distributed Resource Management Application API Working Group - The Global Grid Forum. http://www.drmaa.org. Last access November 2007.

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

[16] I. Foster: Globus Toolkit Version 4: Software for Service-Oriented Systems, in: IFIP Int. Conf. on Network and Parallel Computing, in: Springer-Verlag LNCS 3779, 2005, pp. 2-13.

[17] B. Hamidzadeh, D.J. Lilja and Y. Atif, Dynamic Scheduling Techniques for Heterogeneous Computing Systems, Concurr.: Pract. Exp. 7 (1995) 633-652.

[18] J. Herrera, E. Huedo, R.S. Montero, and I.M. Llorente: Ejecución Distribuida de Bucles en Grids Computacionales, in: 3^a Reunión para la Red Temática en Grid Middleware, 2005.

[19] S.F. Hummel, J. Schmidt, R.N. Uma, and J. Wein: Load-Sharing in Heterogeneous Systems via Weighted Factoring, in: Proc. of the 8th Annu. ACM Symp. on Parallel Algorithms and Architectures, 1996, pp. 318-328.

[20] S.F. Hummel, E. Schonberg, and L.E. Flynn, Factoring: A Method for Scheduling Parallel Loops, Comm. of the ACM 35 (1992) 90-101.

[21] Y. Kee, and S. Ha: A Robust Dynamic Load-Balancing Scheme for Data Parallel Application on Message Passing Architecture, in: Int. Conf. on Parallel and Distributed Processing Techniques and Applications, 1998, pp. 974-980.

[22] A. Kejariwal and A. Nicolau and C.D. Polychronopoulos: An Efficient Approach for Self-Scheduling Parallel Loops on Multiprogrammed Parallel Computers, in: The 18th Int. Workshop on Languages and Compilers for Parallel Computing (LCPC), 2005, pp. 441-449.

[23] T.H. Kim and J.M. Purtilo: Load Balancing for Parallel Loops in Workstation Clusters, in: Proc. of Int. Conf. on Parallel Processing, 1996, pp. 182-189.

[24] C.P. Kruskal and A. Weiss, Allocating Independent Subtasks on Parallel Processors, IEEE Trans. Soft. Eng. 11 (1985) 1001-1016.

[25] D.J. Lilja, Exploiting the Parallelism Available in Loops, IEEE Computer 27 (1994) 13-26.

[26] E.P. Markatos and T.J. LeBlanc, Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors, IEEE Trans. Parallel Distrib. Syst. 5 (1994) 379-400.

[27] R.S. Montero, E. Huedo and I.M. Llorente: The GridWay Approach for Job Submission and Management on Grids, in: iAstro Workshop on Distributed Processing, Transfer, Retrieval, Fusion and Display of Images and Signals: High Resolution and Low Resolution in Data and Information Grids, 2003.

[28] Open Grid Forum. http://www.ogf.org, Last access May 2008.

[29] S. Penmatsa, A.T. Chronopoulos, N.T. Karonis and B. Toonen: Implementation of Distributed Loop Scheduling Schemes on the TeraGrid, in: Proc. of the 21st IEEE Int. Parallel and Distributed Processing Symp. (IPDPS 2007), 4th High Performance Grid Computing Workshop, 2007, pp. 1-8.

[30] T. Philip, and C.R. Das: Evaluation of Loop Scheduling Algorithms on Distributed Memory Systems, in: Proc. of Int. Conf. on Parallel and Distributed Computing Systems, 1997.

[31] C.D. Polychronopoulos, and D. Kuck, Guided Self-Scheduling: a Practical Scheduling Scheme for Parallel Supercomputers, IEEE Trans. on Computers 36 (1987) 1425-1439.

[32] S.F. Altschul, W. Gish, W. Miller, E.W. Myers and D.J. Lipman, Basic local alignment search tool, J. Molecular Biology 215 (1990) 403-410.

[33] S. Reyes, C. Muñoz-Caro, A. Niño, R.M. Badia and J.M. Cela, Performance of computationally intensive parameter sweep applications on Internet-based Grids of computers: the mapping of molecular potential energy hypersurfaces, Concurr.: Pract. Exp. 19 (2007) 463-481.

[34] I. Riakiotakis, F.M. Ciorba, T. Andronikos and G. Papakonstantinou: Self-Adapting Scheduling for Tasks with Dependencies in Stochastic Environments, in: Proc. of the 5th Int. Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar '06), 2006, pp. 1-8.

[35] W. Sudholt, K.K. Balgridge, Parameter Scan of an Effective Group Difference Pseudopotencial Using Grid Computing, Grid System for Life Sciences 22 (2003) 137-146.

[36] 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, New York, 2006.

[37] P. Tang, and P.C. Yew: Processor Self-Scheduling for Multiple Nested Parallel Loops, in: Proc. of the 1986 Int. Conf. on Parallel Processing, 1986, pp. 528-535.

[38] T.H. Tzen and L.M. Ni, Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers, IEEE Trans. Parallel Distrib. Syst. 4 (1993) 87-98.

[39] C. Yang and S. Chang, A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters, J. Information Science and Engineering 20 (2004) 263-273.

Javier Díaz received the MSc degree in Computer Science from the Castilla-La Mancha University, Spain, in 2005. Actually, he is a PhD candidate in the Castilla-La Mancha University. He is member of the UCLM Computational Chemistry and High Performance Computing (QCyCAR) research group. His research interests are in the areas of computational grids and scheduling algorithms.

Sebastián Reyes is Computer Science Engineer from the Granada University, Spain. He is currently a member of the Department of Information Technologies and Systems at the Castilla-La Mancha University (UCLM), Spain, and also member of the UCLM Computational Chemistry and High Performance Computing (QCyCAR) research group. He is a lecturer in Computer Networks. His present research interests are in clustering and Grid computing.

Alfonso Niño received his PhD degree from the Complutense University in Madrid, Spain. His multidisciplinary work involves Computer Science and Molecular Physics. Formerly, he worked at the National Research Council of Spain (CSIC) and is currently a member of the Department of Information Technologies and Systems at the Castilla-La Mancha University (UCLM), Spain. He is also member of the UCLM Computational Chemistry and High Performance Computing (QCyCAR) research group. He has been visiting professor at the Brock University (Ontario, Canada) and currently participates in the Theoretical Chemistry graduate program of the Puebla University (Mexico). He has been a lecturer in Theoretical Chemistry, Software Engineering and Object Oriented Programming. His research interests are in the fields of clustering and Grid computing applied to Molecular Modelling.

Camelia Muñoz-Caro received her PhD degree from the Complutense University in Madrid, Spain. Her work covers Computer Science and Molecular Physics. Formerly, she worked at the National Research Council of Spain (CSIC), being currently a member of the Department of Information Technologies and Systems at the Castilla-La Mancha University (UCLM), Spain. At present, she leads the Computational Chemistry and High Performance Computing (QCyCAR) research group of the UCLM. She has been visiting professor at the Brock University (Ontario, Canada) and currently participates in the Theoretical Chemistry graduate program of the Puebla University (Mexico). She has been a lecturer in Theoretical Chemistry, Management Information Systems and Object Oriented Programming. Her research interests are in the fields of clustering and Grid computing applied to Molecular Modelling.