Simulation for bulk synchronous parallel superstep task assignment in desktop grids characterised by gaussian parameter distributions

Edscott Wilson Garcia and Guillermo Morales-Luna

*Instituto Mexicano del Petróleo, 07700 México D.E. México
bCentro de Investigación y Estudios Avanzados del IPN, 07360 México, D.F., México

Abstract. Modern load distribution schemes for parallel task assignment often refer to parallel computers with shared memory or dedicated clusters with local address spaces. For parallel computing performed on dedicated clusters, the communication topology between processors can be considered static. On dealing with the execution of parallel programs on desktop grids, the previous assumption will lead to an inexact parameter determination for network cost due to the actual network topology being determined not only by the hardware characteristics but also by the instantaneous bandwidth availability for the processes to communicate. The dynamic bandwidth depends not only on location, time and date, but also on the habits and behaviour of the individual users of the computers that conform the grid environment and will lead to an inconstant network topology. This paper presents a novel simulator design which takes these factors into consideration for the BSP parallel computer model. The simulator is used as a performance evaluation tool for several case studies to point out variations in performance obtained from different task assignment strategies as the size of the desktop grid increases and the number of parallel threads changes.

Keywords: Grid simulation, desktop grids, parallel computing, BSP

1. Introduction

Constructing a simulator to reproduce experimental results is not a difficult task. The intention presented hence is to construct a simulator based upon a sound mathematical principle (The Law of Large Numbers) in order to illuminate pathways to trip the potential of the computational resources which will be available in the near future with Grids composed of a large number of ordinary desktop computers.

A few simulators are currently available to test application scheduling in Grid environments. These include MicroGrid [17], SimGrid [5], GridSim [4], and most recently GangSim [6].

The MicroGrid simulator [17] developed at the University of California is a toolkit undertaken for the specific architecture of the Globus [7] controlled environment. Thus the toolkit is focused towards applications constructed with Globus. An important feature of MicroGrid is the ability to verify simulation results under the Globus environment. The simulator does not address the issue of examining
the performance of a parallel application where barrier synchronisation is an important issue, nor does it make any mathematical considerations dealing with parameter determination on large grid sizes.

The SimGrid toolkit, also developed at the UCSD, is a powerful system that allows the creation of tasks in terms of their execution times and resources with respect to a standard machine capability. Tasks can be assigned according to the scheduling policy being tested. Nonetheless, SimGrid is limited to a single scheduling entity and it is difficult to simulate multiple competing users in a market-like Grid computing environment. No mathematical considerations for parameter determination on large size desktop grids is currently available to simulate the performance of a parallel application.

GridSim [4] is a Java toolkit which fills in many of the limitations presented by SimGrid. GridSim provides a comprehensive facility for simulation of different classes of heterogeneous resources, users, applications and schedulers. The focus, in this case as with the previous simulators, is towards distributed computing and not parallel applications. With GridSim, all users need to submit their jobs to a central scheduler, which can be targeted to perform global optimisation or overall user satisfaction depending on resource allocation policy or optimise for high priority users.

Several shortcomings of GridSim can be pointed out. Task assignment can only be performed from the central scheduler and no considerations are made to simulate an environment where assignments are made from multiple entry points to a Grid with a morphing network topology – which is one of the main characteristics of a desktop grid. Furthermore, while Java is a powerful tool for the construction of high level applications, a lower level approach with pointer arithmetic is much easier to code in a language such as C and provides for faster execution. This last detail is fundamental for scaling the simulator to large grid sizes.

In one of the latest additions to the Grid simulator collection, GangSim [6] focuses on job scheduling with resource allocation policies adopted by sites and virtual organisations. This simulator models not only the sites but also virtual organisation users and planners and is able to model usage policies. The scenario where virtual organisations appear and disappear, and where resource allocation policies are never constant is not covered by the current status of development in GangSim.

The Desktop Grid Simulator (DGS) strives to cover terrain not included by the excellent tools which have been developed so far. For this reason a comparison of the results produced by DGS with the previously mentioned is not included in this paper: parameter considerations are fundamentally different.

In addition to this last point, previous simulators are presented as toolkits, while the DGS is set forth as a set of rules with which to generate desktop grid parameters and proceed with task assignment policies: the scope of the DGS at this point is for parallel programs under the BSP model. These rules were programmed in C because of the convenience presented by this language, but they can used to extend GridSim, for example. The novelty of the approach is the use of the mathematical tool comprised by the Law of Large Numbers to define the Grid parameters. A drawback of this approach is that it is only applicable to large Grids. A Grid composed of a few dozen scientific clusters with 10K processors each would certainly not be modelled by these assumptions. On the other hand, the growing number of desktop computers which are being integrated into the Internet with each day is a different target focus: this is the motivation for the DGS simulator.

The DGS also takes into consideration the fragmentation of the Grid. The focal point of the DGS is not distributed computing but rather the scheduling of a parallel application under the BSP model: with a sufficiently large Grid size this becomes feasible. While it is possible to incorporate these new conditions upon existing simulators, the verification and testing of such rules on a standalone environment is necessary before any such undertaking. This is the void filled by DGS.
As a tool to evaluate the performance of task assignment strategies for desktop computer grids, the DGS can be used to obtain performance evaluations for different grid sizes and number of parallel threads.

The rules defined by the DGS to generate Grid parameters could be incorporated into any of the previous simulators. Nevertheless, the purpose of this paper is not to extend existing simulators and complicate them further, but rather to develop the procedures for dealing with large desktop Grids whose size will tend to increase with time.

Current literature on the subject of load balancing parallel applications to increase performance deals with both task assignment and task migration. The former has the trial grounds of systems such as Condor [14,19] while the latter has received much attention from practical applications [11,15]. A substantial quantity of literature exists on the subject of load balancing. Among these, a practical approach can be found in [20] or a history driven method in [3]. Systems such as Condor address issues such as job scheduling, access control and user/resource management, and mechanisms are provided for sharing resources on desktop machines to provide high computational throughput with off-the-shelf components.

When a grid is build by the linking of common desktop computers used for ordinary day-to-day tasks, such parameter estimation is not so evident. The technical difficulty for building a simulator for the desktop grid lies in the large amount of individual computers which compose a desktop grid and for which resource parameters must be generated at each time interval to characterise the state of the system. Even though the Law of Large Numbers indicates that the method to estimate the parameters is quite straightforward, doing so for very large grid sizes is difficult since the simulator must keep the state of the grid in memory. This involves a data structure for every node in the desktop grid. Furthermore, in order to obtain results which are mathematically coherent with the Law of Large Numbers assumption, the simulations must be executed repeatedly to form a large population of results from which to draw statistical conclusions. To illustrate the difficulty involved, the paper includes results for the simulation of a 50,000 node desktop grid.

One of the most prominent characteristics of desktop grids is their tendency to grow not only in size, but also in performance and memory capacity. A successful task assignment algorithm should take this fact into consideration to extend the usefulness of the algorithms for a reasonable amount of time.

Random task-to-processor assignments usually have good results. In a desktop grid, however, the resources available at each processor are variable. This includes all different parameters which characterise the remote processor. A random assignment which does not consider this variable would incur the risk of assigning a task to a processor which does not have sufficient resources. The task would therefore be rejected with a consequent loss in time. Furthermore, another attempt to assign the task to a different processor has only a slightly smaller probability of error. Contemporary scientific applications are of such complexity that there is a continuously increasing demand for computing power and access to larger datasets [8]. In this case, keeping tabs on the resources that will lead to a more effective task assignment strategy.

The need for the development of the rules by which the simulation must establish the parameters follows from the development of the force field task assignment method [9]. A parallel program using this assignment method must have access to the entire grid, but will use only a small portion at any one time due to Amdal's Law limitations. The desktop grid as a whole must be separated from the effect individual users have upon the desktop grid parameters: these individual users also include those who are executing a BSP parallel program with a small number of threads per superstep relative to the total grid size. This is modelled best by the Law of Large Numbers.
At the San Diego Supercomputing Center, empirical studies of desktop grids have provided statistical basis for the construction of desktop grid simulators [13]. These results indicate the tendency of Grid parameters towards a Gaussian distribution and as grid size grows, this tendency is likely to increase, just as mathematical theory predicts. The development of these tools can be used to evaluate new schemes for task assignment and compare them with those already known and commonly used.

The main contributions of the work presented in this paper are as follows:

- The algorithm behind the simulator which considers the tabulation of Grid parameters upon the basis of the Law of Large Numbers for the execution of parallel programs is described.
- Simulations results for several task assignment policies which may or may not use network cost or cycle availability considerations are compared.
- The strategies which have better performance as the size of the desktop grid increases and the number of parallel threads is varied are shown.

The rest of the paper is organised as follows. Section 2 gives an overview of the BSP model and task assignment methods. Section 3 presents the simulator design for desktop grids. Section 4 presents case studies with comparative evaluations for several task assignment schemes. Finally, Section 5 provides concluding remarks and further work to be done.

2. Background

In many cases, the computational and communicational overhead implied by the collection of data to solve the task-processor assignment algorithm, may exceed the benefit derived from the improved assignment strategy. While this may be true for grids built from the grouping of shared memory super computers and dedicated clusters of workstations—this premise is not necessarily applicable for desktop grids.

In the construction of the DGS, the virtual parallel computer is viewed under the BSP model with cost considerations for both latency and bandwidth. In this model, program execution is divided into a sequential series of supersteps. Parallel threads are contained within these supersteps. To increase the performance of a computer which behaves according to this model specification, the completion time for the superstep should be reduced.

Task failures are not currently incorporated into the simulator because the purpose of the tool is to produce comparisons between different task assignment methods and task failures will affect the different task assignment methods under comparison equally and thus have no effect on the relative results. None of the compared methods uses task duplication. The use of task duplication may be used as a complementary tool by any assignment method and may be used to extend the simulator in the future.

2.1. BSP parallel computer model

The BSP model is quite adequate—from the C programmer’s point of view—when applied to the desktop grid conditions: the high possibility of individual nodes of the desktop grid entering and exiting the grid without any previous warning determines that a synchronous model with known system states will provide the basis for more robust application source code: this is because error recovery is not only feasible but inherently natural when the system reaches a known state. The unpredictable environment of the desktop grid can be dealt with at the low level of the model, thus leaving the programmer free from the complications which the desktop grid implies. This may seem rather academic and of little practical
relevance but the proliferation of the ubiquitous interconnected computer reveals that the vast amount of computing power represented by desktop computers will tend to increase and will require new methods to harness this power without distracting the programmer.

The BSP model divides parallel applications into a series of supersteps, each of which consists of parallel running threads with any number of operations. The most important aspect of the BSP model is the barrier synchronisation which takes place at the end of each superstep. The individual parallel threads perform only local communication until the barrier. At this point all global communication takes place. Since local communication is allowed for each thread, it is possible for a thread to fork into several parallel sub-threads. This would only make sense — of course — if the thread that will fork is executing in a multiprocessor remote computer: in such a case the splitting of the thread into more than one sub-thread can take advantage of hardware enabled parallel processing at the remote node.

In general terms, the BSP model consists of a set of nodes — each with an associated memory area — a global communications network and a mechanism for efficient barrier synchronisation [16].

Figure 1 represents the basic building block for a BSP machine with \( n \) nodes executing a parallel application. Upon initial barrier synchronisation, the parallel application forks into \( m \) (\( m < n \)) parallel tasks. These tasks are then assigned to individual nodes.

Once the tasks are assigned, each node is on its own. These nodes may send messages to other nodes at any moment, but the communication requests shall not be completed until barrier synchronisation.

Once all running threads have reached the barrier synchronisation and the program is ready to proceed by assigning the tasks of the next superstep to remote processors, the assignment strategy should consider the current status of communications network and computational resources at remote nodes.

Strategies used include the minimisation of communication costs [1,10] and also the achievement of a reasonable computational load balance [12] between the participating nodes. While these strategies will produce good results with small grids, methods such as the Force-Field approach [9] which consider both limiting factors of a desktop grid produce reduced parallel program execution times as the grid size grows. This is further analysed in Section 4.

In order to combine communication cost with the data from idle cycles at remote nodes, task assignment strategies which rely on such measures must express both variables in the same units: this may be done by
expressing time units as a certain number of computation cycles on a reference processor. Communication cost is measured by the information packets sent over the network. As processor speeds increase at a faster rate than that of communication hardware, the ratio between the two also increases. Parallel applications with a heavy demand for communications, such as data distribution, may achieve increased performance from improved task assignment methods. Since all tasks must synchronize at the barrier, a particular detail must not be overlooked in order to combine the communication cost variable with the data from idle cycles at remote nodes. The network status must be measured relative to the cost of computation. As processor speeds increase at a faster rate than that of communication hardware, the ratio between the two also increases. Parallel applications with a heavy demand for communications, such as data distribution [18] can readily profit from optimized task assignment methods.

The model based on the Law of Large Numbers is a sound mathematical principle which is used in many models with a great deal of success. The applicability of this model to the desktop grid environment depends on the following: a large number of desktop nodes and many simulation experiments are necessary. Problems such as network congestion are also taken into consideration by the Law of Large Numbers due to the fact that with a large enough population, such network congestion will also be normally distributed. Thus the communication cost the simulator utilizes takes network congestion into consideration (reflected by the effective available bandwidth). Furthermore, instances of varying network congestion can be simulated by a linear variation of the network communication costs. Issues such as a generalized network congestion (due to a very successful computer virus for example) would constitute a "denial-of-service" attack on the desktop grid and is not considered by the simulator since the conditions that define the desktop grid environment would cease to exist.

2.2. Force field assignment

The rationale which exists in representing the assignment problem by using a force field arises primarily from a philosophical argument. If motion of particles in a physical universe is approximated well by force field, then motion of tasks in the virtual universe defined by a desktop grid may be well ruled by a force field. As any scientific hypothesis, if proved true, this does not mean the force field approach is the best conceivable method: "The sweetest song remains yet to be sung". The concept of force field has proved very successful in physics to approximate what is observed in reality. From Leibnitz's philosophical point of view, the universe is created to function in the best possible way. Mathematically speaking, looking at the way the universe moves, and taking closest mathematical approximation science has developed to explain motion, the force field is reached, Thus the force field model hypotheses that this analogy for motion of tasks on the desktop grid is likely to be successful. The analogies for the force-field are quite intuitive: computers which are far away—high communication cost—should have very little attraction for tasks unless they are massive—i.e., have extraordinary idle-cycle availability—. Recent theoretical proposals in physics postulate the existence of gravitons which travel between bodies to transmit the force of gravity. These can be viewed as the information packets which need to travel between computers with idle-cycle and communication cost information. As Zhuge pointed out [21], in the future interconnection environment resources must flow from high to low energy nodes. Energy leads to force fields. The simulator is aimed at testing how effective the force field consideration actually is, as well as any other method which may be proposed for the conditions of the desktop grid.

Walt Whitman, Leaves of Grass.
The Force-Field [9] task assignment method takes both communication cost and idle cycles into consideration and combines them into a scalar force field. The *net attraction* of each remote node for a particular task will thus be the strength or affinity for receiving a particular task. This attraction in the force field will determine which remote computers are assigned the tasks that compose the superstep. Remote computers which do not have enough resources will produce a repulsive force.

Let the term *computational ease* refer to the reciprocal of the cost in cycles required to complete a task. What distinguishes the Force-Field task assignment method is, in general terms, the fact that force is in inverse ratio to the square of the cost of communications and direct ratio to the computational resources at the remote node multiplied with the computational ease of the task to be assigned.

The equation which determines the force field is obtained through an analogy to Coulomb's law:

\[
F = \frac{q_j M_{ij}}{r_{ij}^2}.
\]

**Remote computer charge.** Represented by \(q_j\), this variable is the computational potential of node \(j\), and is equivalent to the available computational cycles per unit time. This number is always positive or equal to zero. If \(q_j = 0\), then neither attraction nor repulsion exists. Whether or not the node actually receives a task depends on the force field values determined for other nodes on the grid.

**Task charge.** Represented by \(M_{ij}\), this variable is the computational ease associated to the memory requirements of task \(i\) with regard to node \(j\). An important consideration which is often overlooked is the fact that the current bottleneck in the execution of most tasks is not in processor speed but rather the moving of data between memory storage and processor registers. This variable will be less than zero, unless the memory resources at node \(j\) are insufficient to satisfy the requirements of task \(i\). In this second case, \(M_{ij}\) will correspond to a value greater than zero, ensuring a net repulsive force.

**Processordistance.** Represented by \(r_{ij}\), this is the cost in communications from the remote node \(i\) to the local node \(j\) from where the task assignment will take place.

On applying Eq. (1), \(F < 0\) implies an attraction or tendency to move a task towards the remote node, while \(F > 0\) implies repulsion. With the Force-Field task assignment, the nodes with the most negative forces will be assigned the tasks: the assignment of tasks to nodes is done simultaneously in such a manner that the node with the greatest attractive force will receive the task associated with this force.

### 2.3. Other assignment methods and minimum

The random task assignment method is the most common scheme to assign a task to a remote processor. The method consists in randomly assigning the task to any available processor. An advantage of this task assignment method is that no information is required on the status of the remote processors nor network cost. This method is best suited to grids where the configuration of the network and of the remote processor's resources is quite stable.

This method may prove counterproductive with a desktop grid for two reasons. The first is that the remote processor which receives the task may not have enough memory resources and thus must reject the task. Thus the assignment fails and must be attempted again. The second reason is that both network communications and remote resources are variable in a desktop grid, and therefore it is improbable that the assignment will optimise the resources available on the grid.

Figure 2 shows the mean value of task assignment rejections produced for an increasing number of parallel task to be executed within a 1000 node desktop grid. This figure has been produced with the utilisation of the desktop grid simulator described in this paper. The conditions which produced
the results are those described in the section relative to the description of the parameters used in the experiments and are obtained by reference to the Law of Large Numbers. Each missed assignment will set the superstep back a fixed interval of time. This delay will depend on the communication costs. Furthermore, the results shown in Section 4 show that this is not an adequate method for desktop grids.

For the greedy idle-cycle task assignment [2], the main factor to be considered is the number of idle computer cycles per unit time available at the remote node. For the distribution of $n$ parallel tasks, the remote computers with highest number of idle cycles per unit time will be assigned the tasks in order, provided the memory resources are greater than or equal to those demanded by these tasks. When dealing with tasks where communication requirements are small or insignificant, this should be the method of choice. Nonetheless, applications where there is little need for communication are fewer and farther apart each day.

The greedy communications cost task assignment method [2] considers the communication costs among processors. The remote computers with the least communications cost—and with enough memory resources—will be assigned the tasks which compose the superstep. For data intensive applications where computation cycles are not in high demand, this would be the method of choice.

Another policy that utilises both information of computation cost and communication cost is likely to be available. In fact, policies that use both variables to solve the assignment problem are not only infinite in number, but form a space of functions whose members are uncountable. The simulator presented is a means to test these methods against each other when confronted to the conditions of the desktop grid.

Considering that the simulator is designed to evaluate a parallel computer acting under the BSP model, the minimum is the combination of tasks to processors in which the maximum time for all the task resolutions is the smallest. In other words:

Let $P = \{p_1, p_2, \ldots, p_n\}$ be the set of processors in the grid and $T = \{t_1, t_2, \ldots, t_r\}$ the set of tasks to be solved in parallel within the superstep. Suppose that $m \leq n$. Then, for all $t \in T$ and all $p \in P$, there exists an execution time $\tau(t, p)$. $\tau$ is thus a mapping $\tau : T \times P \rightarrow N$. Define a task assignment $\phi$...
as an injective mapping $\phi : T \rightarrow P$. Then there exist $C^m_n = n!/m!$ combinations from where to pick $\phi$. Let $\Phi$ be the set of all assignments and for each $\phi \in \Phi$ define: $\omega(\phi) = \max_{t \in T} (\tau(t, \phi(t)))$. Thus the optimisation problem consists of finding an assignment $\phi_0 \in \Phi$ such that $\omega(\phi_0) = \min_{\phi \in \Phi} \omega(\phi)$. Note that there may be multiple optimum assignments $\phi_0, \phi_1, \ldots, \phi_i$.

With the knowledge of idle cycles for all tasks to be solved during the superstep (piece of data which is not available to the task assignment strategies under test), the minimum as described above is approximated and shown in Figs 14–16 and 18–20. The computation of the exact minimum is not computationally feasible, being an NP-hard problem.

3. Design

3.1. Parameter determinations

The general tendencies which characterise a desktop grid composed by a large number of nodes can be correctly inferred with the Law of Large Numbers developed by the great mathematician, Carl Friedrich Gauss.

The Law of Large Numbers indicates that regardless of the underlying distribution of the variables which characterise the desktop grid, the average of the observed values will converge to a standard normal distribution. In other words, given a large number of data points, the probability function can be simulated by the Gaussian distribution in Eq. (2):

$$Y(x) = Y(x; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\frac{(x-\mu)^2}{\sigma^2}\right).$$

The DGS generates values for the variables which characterise the desktop grid and the tasks to be assigned at any given point in time. A fact to be duly noted is that this assumption is valid for a large collection of data points. Therefore the performance evaluation tool shall be best applicable to desktop grids of large size.

The first variable to be assigned is the number of cycles available at each computer within the desktop grid. Considering that most desktop users will be using a word processor, surfing the web, or processing e-mail—if not distracted from the screen—the number of idle cycles shall be high. Users who are actually executing programs with a large demand for computational cycles which do not require keyboard I/O—and which consume a high percentage of the available idle cycles—will be distributed along the edges of the Gaussian distribution.

The next variable to be assigned pertains to the available memory resources at each individual computer. This variable—which is independent from the idle cycle availability—is basically the amount of virtual memory the remote computer can assign to a remote task without incurring into a swap-idle swap-out problem. Since each task to be assigned has a predetermined amount of memory resources necessary for execution, this variable will be used to determine whether a particular task may be assigned to a specific remote processor or not. Most current applications for the desktop user—such as e-mail, work processing, web browsers—tend to be memory hogs, especially with each new release. This determines that the number of machines with high memory availability for remote processes will tend to be along the edges of the Gaussian distribution.

Thus the general profile of the typical desktop grid machine will be that of high idle cycle and low free memory. To generate these values, a Gaussian distribution for each variable is split into a number of points which matches the number of computers in the grid. Then each point of the Gaussian distributions
is randomly assigned to a specific remote computer. The assignment for each distribution is independent from each other, as noted above.

The third variable that characterises the desktop grid is the communication cost between the remote processor and the processor where the tasks are generated. As most users will not be downloading or uploading files most of the time, computers with high cost will be along the edges of the Gaussian distribution. The cost assignment is done in the same manner as the first two variables.

Finally, the design deals with the variables which characterise the remote tasks to be assigned. Each one of these tasks will have certain memory requirements and computational cost. Both these variables are obtained from a Gaussian distribution in the manner detailed above. The task memory requirements must be known by any task assignment scheme. The computational cost, on the other hand, is generally unknown. Thus the simulator does not make this variable known to any of the schemes where performance is being evaluated. Nonetheless, this is used to tabulate all possible task/processor combination to obtain an absolute minimum. This evaluation is the most difficult step with respect to computation for the simulator, as described below.

Considering that the simulator is designed to evaluate a parallel computer working upon the BSP model, the minimum is attained by the tasks to processors in which the maximum time for all task resolutions is the smallest. In other words:

Let \( P \) be the set of processors which are in the grid, \( P = \{p_1, p_2, \ldots, p_n\} \), and \( T \) the set of tasks to be solved in parallel within the superstep, \( T = \{t_1, t_2, \ldots, t_m\} \). Note that \( n \leq m \). Then, for all \( t \in T \) and all \( p \in P \), there is an execution time \( \tau(t, p) \). \( \tau \) is thus a map \( \tau : T \times P \rightarrow N \). Define \( \phi \) as an injective task assignment mapping \( \phi : T \rightarrow P \). Thus there are \( C_n^m = m! / n! \) arrangements to choose \( \phi \) from. Let the set of all arrangements be \( \Phi \) and for each \( \phi \in \Phi \) define: \( \omega(\phi) = \max_{t \in T} \left[ \tau(t, \phi(t)) \right] \). Thus, the optimization problem consists of finding an assignment \( \phi_0 \in \Phi \) such that \( \omega(\phi_0) = \min_{\phi \in \Phi} \omega(\phi) \). Note that there may be multiple optimum assignments \( \phi_0, \phi_1, \ldots, \phi_k \).

For the tabulation of characteristics of individual nodes which compose the desktop grid, instantaneous values for network cost and idle cycle availability at any one time are not very useful. The characterising value would certainly be better represented by an average value during the particular time interval where the instantaneous value is located. Thus, if the instantaneous value is \( \beta(t) \) (which could be the network cost or the remote computer available idle cycles), the moving average \( \hat{\beta}(t) \) would be given by Eq. 3:

\[
\hat{\beta}(t) = \frac{1}{\Delta t} \int_{t-\Delta t}^{t} \beta(\xi) d\xi
\]

From a practical point of view, in order to make the force field method applicable, a well known method such as the trapezoid or Simpson's rule should be applied.

Summing up, the parameters which the DGS will determine by means of a Gaussian distribution are the following:

- Remote processor available idle cycles per unit time
- Remote processor available memory
- Communication cost to remote processor
- Task memory requirements
- Task computation cost in cycles

This provides for different grid conditions on each simulation run, and requires the simulator to be executed for a large number of times to obtain representative statistical results. Further analysis is detailed in Section 4.
3.2. Parameters used in the experimentations

To obtain the parameters used in the experiments, in accordance to the Law of Large Numbers, a normal distribution for communication cost, available idle cycles and memory resources was generated. This normal distribution for communication cost is shown in Fig. 3 with individual nodes of the desktop grid ordered according to the normal distribution. For each individual simulation run (thousands of runs need to be performed under different conditions) the grid node ID is randomly assigned to the grid number in the normal distribution. In a single simulation run—for example—the resulting grid communication cost parameters used in the experimentation is shown in Fig. 4. In the same manner, the parameters for available idle cycles and memory are generated and shown in Figs 5–6.

As can be observed, the mean value of these normal distributions are all the same. This allows for varying the system parameters by linear variation of the Gaussian distribution to account for network congestion or dwindling overall available idle cycles.

As for the generation of the parameters which characterise the tasks to be assigned, two approaches were used. The first was to assume a normal distribution such as that shown in Fig. 7. With such a
distribution, the required computational cycles and memory resources are randomly assigned to the task ID, as shown in Figs 8–9. The second approach, detailed in Section 4 consists of analysing a parallel problem in particular and assigning the parameters to characterise the tasks.

3.3. Simulation procedure

The input parameters for each run include the size of the Grid (N), the communication to computation ratio for the application (R), and the number of parallel threads to be executed in the superstep (M).

1. Assign an identification number to each node within the Grid and to each task to be executed in the superstep.
3. For each segment midpoint obtain the value of the Gaussian distribution curve (array $G_1[N]$).
4. Make random assignments of each node id to the elements of array $G_1[N]$. This will be the available computation cycles for the remote nodes ($A[N]$). The units of measurement are transformed so that they will coincide with the simplest mathematical expression for the Gaussian distribution.
5. Make a different random assignment of each node id to the elements of array $G_1[N]$ multiplied by the communication to computation ratio $R$. This will be the communication cost for the remote nodes, $(B[N])$.

6. Make a third different random assignment of each node id to the elements of array $G_1[N]$. This will be the available memory resources for the remote nodes $(C[N])$. The units are again transformed so that they will coincide with the simplest mathematical expression for the Gaussian distribution.

7. If the task memory requirements are known (as in the borehole problem), assign these values to the array $T_m[M]$ and proceed to step 11 otherwise continue at 8.


9. For each segment midpoint obtain the value of the Gaussian distribution curve (array $G_2[M]$).

10. Make random assignments of each task id to the elements of may $G_2[M]$. This will be the required memory for the tasks $(T_m[N])$. The units of measurement are transformed so that they will coincide with the simplest mathematical expression for the Gaussian distribution. This coincides with the units in 6.
11. If the task computational cycles requirements are known (as in the borehole problem), assign these values to the array \( T_m[M] \) and proceed to step 15 otherwise continue at 12.

12. Divide the interval \([-3, 3]\) into \( M \) segments.

13. For each segment midpoint obtain the value of the Gaussian distribution curve (array \( G_2[N] \)).

14. Make random assignments of each task id to the elements of array \( G_2[M] \). This will be the required computation cycles for the tasks \( (T_c[N]) \). The units of measurement are transformed so that they will coincide with the simplest mathematical expression for the Gaussian distribution. This coincides with the units in 4.

15. With the entire grid parameters in arrays \( A, B \) and \( C \), perform a random task assignment to the remote nodes.

16. If any assigned task has more memory requirements than those available at the assigned node, log the failure and add the time associated to the network cost for the remote node to the superstep time for the random assignment method.

17. Repeat 16 until all tasks are successfully assigned or all nodes are exhausted. If all nodes are exhausted, log a "Grid congestion" message and repeat procedure at 4. The final superstep time is tabulated as the first element of the array \( D[M] \). The superstep time will be the maximum execution and communication time of all the tasks, plus the time lost due to task rejections.

18. At this point the simulator may be configured to use all the data in arrays \( A, B \) and \( C \) (continue at 20) or else to simulate a real life situation where nodes which are too far away are ignored.

19. Arrays \( A, B \) and \( C \) are coded as elements of a structured array \( S[A, B, C] \), and thus performing a quick – sort on \( S \) will arrange all three with the same criterion. \( S \) is sorted according to \( B \) and reduced by eliminating the trailing elements. The amount of elements to consider in the task assignment schemes is user configurable and is set as a multiple of the number of tasks to be assigned.

20. The superstep time for each assignment method is obtained from \( S \). The method chosen to program the above is a memory intensive and avoids moving \textit{values} between memory locations: \( S \) is rearranged by pointer relocations for each different task assignment methods.

21. A vector containing \textit{superstep times} for all considered methods is appended to a plain text file which can be fed to statistical software or streamed to the second stage of the simulator: a statistical analysis to produce output fit for plotting. This second stage includes obtaining mean, average, maximum and minimum values for all \textit{analysed} assignment methods.
22. If the number of simulation runs is complete, stop. If not, repeat procedure from 4.

This is procedure is illustrated in Fig. 10. As can be observed, the simulation rules are simple and straightforward. Nonetheless, the procedure illustrated in Fig. 10 must be repeated a number of times. The results for every task assignment method should be saved in a separate array.

In the procedure described above, the implementation was performed in C with the GNU compiler. All the arrays are optimised by the utilisation of arrays of pointers. Thus only the pointers are moved and referenced more than once by different arrays.

The implementation of the distributed version on the 200 node cluster was performed by means of shell scripts generated with Perl.

3.4. Effect of task assignment itself

A question which may arise is whether the actual task assignment schemes will have an effect on the parameters to be measured for the task assignment itself. While this is certainly true, the effect is uniformly distributed through out the entire desktop grid, producing a systematic error. Systematic errors are errors that produce a result that differs from the true value by a fixed amount. These errors result from biases introduced by instrumental method, or human factors. In this case, the effect of the task assignment itself on the grid conditions.

Another case to consider is what will happen to the desktop grid when an increased number of parallel applications vie for grid resources. Under the assumption that the applications for a particular desktop grid will tend to be similar in nature—banking, CFD, chemistry — the duration of the super step can be assumed similar. Thus, instead of dealing with the complexities of a simulation involving ten applications...
The combination is kept in memory to approximate the minimum infectious value.

4. Case Studies

With 20 nodes, the results will not be identical, but general tendencies will surface.

Each with 5 parallel threads per super step, for example, consider the assymmetries of a single super step.

![Graph 1](image1.png)

![Graph 2](image2.png)

![Graph 3](image3.png)
reassigned resources and different grid conditions 1000 times (not to be confused with the grid size which remained constant at 1000 elements) in order to process the results statistically and obtain the mean value. Other values obtained by the simulator include the average, the maximum, the minimum, and the sequential: the sequential is the time the superstep would take to complete if all threads were executed sequentially on the reference processor. Thus for each parameter weight conditions (low, normal and high communication cost) 45 processes are generated, amounting to a 67.5 percent use of the available processors. Further below, when the borehole simulation is performed, the number of threads in the superstep remains constant and different processes are generated for each grid size with a step of 100. In this case the cluster use was complete while many tasks remained in queue.

4.1. Preliminary results

Figures 11–13 reflect point data for simulations of an application program with 20 parallel threads which is mainly restrained by the computation cycles, but where communication requirements are not insignificant. In these figures the horizontal axis represents individual runs of the simulator. For each run the resources are reassigned and represent different grid conditions. All simulation points in the graph correspond to a 1000 node grid. The vertical axis is expressed in relative time units — i.e., equivalent time for a fixed amount of computation cycles on a base reference processor. In these figures, the parameters are weighted to reflect normal communication cost conditions.

Figure 11 shows the results for the average superstep time when the task assignment is performed by the greedy idle-cycle scheme. Figures 12 and 13 show the same simulations but use the greedy communications cost and Force-Field methods respectively. Note that the scale in Fig. 12 is different, and reflect a poor performance of the greedy communications cost method when computation cycle requirements are the dominant factor. From these graphs — on first analysis — the Force-Field method is superior to the other two methods compared, yet further analysis is necessary to determine the conditions when this assertion will remain true. In other words, the data from these graphs is indicative of tendencies, and further analysis is necessary. For this effect, by obtaining the average of each graph, and repeating the simulation for a different number of parallel tasks to be assigned simultaneously, emerging patterns may be observed in the results that follow.
4.2. Random large number task cost

Simulations were performed for the greedy idle cycle, the greedy communications cost and the Force-Field algorithms. The duration of the superstep in time units for a 1000 node desktop grid can be observed in Figs 11 through 13. In these initial simulations, the computation to communication ratio was set to a low value. In other words, the most important factor characterizing the tasks was the computational requirements, not the need for interprocess communication.

Figure 11 shows the scatter point data when task assignment was performed by means of the greedy idle-cycle method. The maximum time for the superstep was less than 1.4 time units and greater than 1.0. With respect to the greedy communications cost shown in Fig. 12, the wall time for the super step is within the interval [0,100]. The values which appear most often are in the interval [0,10].

Finally, in Fig. 13, the Force-Field algorithm is evaluated under the same conditions. In this case the maximum interval [1,1.9] is greater than that returned by the greedy idle cycle method, but most values are concentrated in the interval [1,1.1]. At the first impression this seems a better overall performance. Nonetheless, statistical analysis is necessary to prove this intuitive claim. This is done with the results detailed below.

The first statistical analysis involves the use of the mean of the average task durations within the superstep. This is shown in Figs 14-16. In Fig. 14 (low communication cost conditions) shows that the greedy cycle-availability is better than the Force-Field algorithm only after 30 simultaneous tasks on a 1000 node desktop grid.

In order to view the effect of the different task assignment methods for typical computations where computational requirements are still the most important but where communication costs are no longer

---

Footnote: Time units are expressed relative to the computational cycles of a reference processor; these abstract time units are used instead of wall clock units because the focus of the simulator is to produce comparative results for different task assignment schemes under the same network end cycle availability conditions rather than to predict actual performance of a parallel BSP program on a desktop grid.
Insignificant, refer to Fig. 15. In this figure the greedy communication cost method exceeds a random assignment only when there are less than 10 parallel tasks in the superstep. On the other hand, the Force-Field method which takes communication cost into account, leads to a better performance than any other method and deviates very little from the absolute minimum value. Furthermore, considering tasks where communication is the most important requirement, Fig. 16 shows that the Force-Field method is practically equivalent to the minimum time obtainable.

These results indicate the direction task assignment algorithms should take with parallel applications on desktop grids of relatively small size. The effect as the grid size grows is shown in the Section 4.3.
Table I

<table>
<thead>
<tr>
<th>Group</th>
<th>Number of Elements</th>
<th>Relative Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>1.0000</td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>0.9659</td>
</tr>
<tr>
<td>3</td>
<td>12</td>
<td>0.8560</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>0.7071</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>0.5000</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
<td>0.2588</td>
</tr>
</tbody>
</table>

and confirms the tendency of the Force-Field algorithm to outperform other methods.

4.3. Application to Borehole Tomography

The previous results were generated considering that the computational task requirements were randomly located along a Gaussian distribution. In this section the simulator is applied to a problem in geophysics.

In the oil industry, as the non-renewable resource is depleted, drilling much deeper for reservoirs has proved necessary. Since the hydrocarbon is trapped in thin slices of the Earth’s crust, economics dictate the need to drill wells that enter the oil impregnated rock horizontally and traverse the reservoir along the longest path. This will ensure that the surface contact area will be as great as possible and thus production will be maximized during the well’s lifetime.

In order to determine the exact geometric configuration of the oil reservoir and determine the horizon entry point, several exploratory wells are drilled around the potential reservoir. Sonic seismic waves are then generated and received in these wells. By measuring the wave speed retardation, differences in rock density can be detected and a reconstruction of the reservoir’s geometry can be obtained by inversion methods. A different approach is to use the direct method of solving the wave equation. The direct method is not generally used due to the complexity of the problem. But as computer resources continue to grow and are grouped into desktop grids, the possibility looms on the horizon. The following material shows what can be expected from different task assignment methods for this problem in particular.

For a symmetric 12-borehole tomography, there are 66 wave paths to be solved by means of the wave equation. Each wave path —illustrated in Fig. 17— represents a task to be solved during the superstep. In this figure the exploratory wells are located around a potential reservoir and the wave paths to solved for each iteration are illustrated by the connecting lines. The partial differential equation to solved is hyperbolic and can be solved by well known finite difference methods such as Lax–Wendroff or MacCormack’s. More complicated methods such as variations of Godunov’s are not required since strong shock waves and phase discontinuities are not involved. Thus the computational requirements for solving each wave problem is directly proportional to the distance between the wells. The computational requirements for all wave paths is not equal, but a proportionality relationship exists. There are four groups. The first has full diameter length and six elements. All other groups have 12 elements. The diameter of the circle represent the computational cost of the longest path. If the longest path has computational cost equal to one, then the relationship between the groups is shown in Table 1.

If a single instance of the wave problem requires 10% of a researcher’s desktop computer (this percentage will depend on the relative complexity of the problem and can be varied at will) the DGS can generate the parallel tasks with the appropriate complexity with the data from Table 1.
Figures 18–20 shows the result of the simulations as the size of the desktop grid grows. In Fig. 18 the assumption is that the desktop grid is interconnected with state-of-the-art communications equipment. In this figure, whilst the Force-Field method starts out with greater times than the greedy idle-cycle availability, this disappears after the desktop grid size reaches 2000 processors and continues to drop as the grid size grows.

In Fig. 19 the communication network of the grid restrains the problem resolution and in this case the situation shows that the Force-Field method exceeds other schemes with regard to the superstep performance. Note that in Fig. 20 the force attraction and minimum time curves are practically the same. When communication costs are critical, the results are unequivocal as to the merits of the Force-Field task distribution method, as confirmed in Fig. 20.

Note that in Figs 18–20 the curve which shows the results obtained from the random task assignment is not smooth. This is a consequence of the method making poor assignments in a random fashion. This non-smooth behaviour is also observed in Figs 18–19 for the communication greedy scheme since in both these situations the parameter which determines the superstep time is the available computational
cycles at the remote node.

4.4. Fragmenting the grid

The focus of this paper is on the features which the simulator provides which are not covered by other simulator approaches, rather than on simulation performance in relation to other simulators. This is illustrated in this section: a 5000 node grid is simulated in which the task assignment is performed on dynamically changing subset of the Grid. The simulation shows how the information collection problem in large grids may be solved, using the force field model.
Figures 21(a) and 21(b) reveal the behaviour of communication cost for information collection on a fragmented grid. In Fig. 21(a) message cutoff occurs after receiving three times the tasks to be assigned, and it is clear that the force field method has less communications cost associated time. In Fig. 21(b) the difference in information cost is less, but even so the force field assignment is superior.

Since the super-step execution time remains relatively constant when the cutoff value is set to three times the amount of parallel tasks, it is logical that the application of the force field method with this cutoff value is sufficient. For all the messages which take longer to arrive, the distance in the force field is thus assumed infinite and thus the attraction force is null. Those nodes are outside the fragmented Grid from the vantage point where the task assignment for the parallel program is taking place.

5. Conclusions

The work presented in this paper are the design considerations for a Desktop Grid Simulator (DGS). Critical evaluation. In contrast with other grid simulators such as MicroGrid [17], SimGrid [5], GridSim [4], and most recently GangSim [6], the DGS simulator focuses only on a narrow segment of multiprocessor computation, namely, parallel processing under the Bulk Synchronous Parallel model. Furthermore, only one type of grid is considered: that which is characterised by Gaussian parameter distributions. This limits the applicability of the simulator to very large grids where processors are relatively independent from each other. If the above conditions are not met, then one of the excellent simulators mentioned previously should be preferred. Application to distributed computation is not considered.

The DGS is used to evaluate methods for task assignment. The rules which are defined by the DGS allow testing the task assignment methods on large desktop grid conditions. This leads the simulator to be an important tool to show that the Force-Field method outperforms other schemes such as the greedy idle cycle, the greedy communications cost, or the random assignment. The use of the DGS for the evaluation of task assignment strategies with different grid configurations and varying parallel
applications is one of the main benefits provided for new algorithm development. The DGS represents an important set of rules which can be included in other simulators, as long as code is optimised to permit multiple executions to generate a large quantity of information from which to gather statistical conclusions.

The DGS simulator is an important tool to show that task assignment strategies which take both communication cost and idle cycle availability into consideration, such as the Force-Field algorithm, should be preferred when there is a reason to assume that performances shall be better than the respective greedy strategies. These results produced by the simulator have the solid foundation of the Law of Large Numbers and has shown utility in solving the wave equation by finite difference methods for a 12
References

The approximation of the minimization of the minimum line approximation is the most memory
and computational costly part of the DCG.

The approximation of the minimization of the minimum line approximation is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.

The minimization is meant to demonstrate the scalability of the minimization methods. As the
similarity factor is increased, the minimization of the minimization is the most memory
and computational costly part of the DCG.
E.W. Garcia and G. Morales-Luna / Simulation for bulk synchronous parallel superstep task assignment


Authors' Bios

Edscott Wilson Garcia holds a Master's Degree in Mathematics from the Universidad Autonoma Metropolitana and since 1989 has been working in the research area of the Mexican Petroleum Institute. Currently he is finishing his Ph.D. dissertation at the Centro de Investigacion y Estudios Avanzados del Instituto Politecnico Nacional.

Guillermo Morales Luna holds a Ph.D. in Mathematics from the Institute of Mathematics of the Polish Academy of Sciences and since 1985 is full time professor and researcher of the Computer Science Department of the Centro de Investigacion y Estudios Avanzados del Instituto Politecnico Nacional. His areas of interest are Theory of Complexity and Mathematical Logic.