J Supercomput (2007) 41: 247–267
DOI 10.1007/s11227-007-0115-7

A performance-based parallel loop scheduling on grid environments

**Wen-Chung Shih · Chao-Tung Yang ·
Shian-Shyong Tseng**

Published online: 6 April 2007
© Springer Science+Business Media, LLC 2007

Abstract The effectiveness of loop self-scheduling schemes has been shown on traditional multiprocessors in the past and computing clusters in the recent years. However, parallel loop scheduling has not been widely applied to computing grids, which are characterized by heterogeneous resources and dynamic environments. In this paper, a performance-based approach, taking the two characteristics above into consideration, is proposed to schedule parallel loop iterations on grid environments. Furthermore, we use a parameter, *SWR*, to estimate the proportion of the workload which can be scheduled statically, thus alleviating the effect of irregular workloads. Experimental results on a grid testbed show that the proposed approach can reduce the completion time for applications with regular or irregular workloads. Consequently, we claim that parallel loop scheduling can benefit applications on grid environments.

Keywords Parallel loop scheduling · Performance · Self-scheduling ·
Grid computing · Globus Toolkit

W.-C. Shih · S.-S. Tseng
Department of Computer Science, National Chiao Tung University, Hsinchu, 30010, Taiwan

W.-C. Shih
e-mail: gis90805@cis.nctu.edu.tw

S.-S. Tseng
e-mail: sstseng@cis.nctu.edu.tw

C.-T. Yang (✉)
High-Performance Computing Laboratory, Department of Computer Science and Information Engineering, Tunghai University, Taichung, 40704, Taiwan
e-mail: ctyang@thu.edu.tw

S.-S. Tseng
Department of Information Science and Applications, Asia University, Taichung, 41354, Taiwan
e-mail: sstseng@asia.edu.tw

1 Introduction

As computers become more and more inexpensive and powerful, computational grids which consist of various computational and storage resources have become promising alternatives to traditional multiprocessors and computing clusters [1, 6–8]. Basically, grids are distributed systems which share resources through the internet. Users can access more computing resources through grid technologies. However, bad management of grid environments might result in using grid resources in an inefficient way. Moreover, the heterogeneity and dynamic changing of the grid environment make it different from conventional parallel and distributed computing systems, such as multiprocessors and computing clusters. Therefore, it becomes more difficult to utilize the grid efficiently.

Loop scheduling on parallel and distributed systems is an important problem, and has been thoroughly investigated on traditional parallel computers in the past [11, 12, 15, 20, 22]. Traditional loop scheduling approaches include static scheduling and dynamic scheduling. The former is not suitable in dynamic environments. The latter, especially self-scheduling, has to be adapted to be applied to heterogeneous platforms. Therefore, it is difficult to schedule parallel loops on the heterogeneous and dynamic grid environments. In recent years, several pieces of work has been devoted to parallel loop scheduling for cluster computing environments [2, 4, 5, 23, 25], addressing the heterogeneity of computing power. In [23], a useful self-scheduling scheme was proposed to be applicable to extremely heterogeneous PC clusters. This two-phased scheme collects system configuration information, and distributes α percentage of the total workload to slave nodes according to their CPU clock speed. After that, the remainder of the workload is scheduled by a conventional scheme, such as Guided Self-scheduling (GSS) [15]. Nevertheless, the performance of this approach depends on the appropriate choice of the parameter, α. In fact, it estimates node performance only by CPU speed, which is one of the factors affecting node performance. In [25], an enhanced scheme was proposed, which dynamically adjusted the parameter α according to system heterogeneity. However, the value of α should depend on dynamic environments and workloads, instead of the heterogeneity of systems.

Intuitively, we would partition the total workload according to CPU clock speed. However, the CPU speed is not the only factor which affects node performance. Many other factors also have dramatic influences in this respect, such as the amount of memory available, the cost of memory accesses, and the communication bandwidth between nodes, etc. Using this intuitive approach, the result will be degraded if the performance estimation is not accurate.

In this paper, we propose a general approach called PLS (Performance-based Loop Scheduling). This approach utilizes performance functions to estimate the performance of each node. To verify our approach, a grid testbed was built, and two types of applications, matrix multiplication and Mandelbrot set computation, were implemented and executed in this testbed. Experimental results showed that our approach could obtain performance improvement on grid environments.

Previous work in [16, 17] and this paper are all inspired by [23]. However, this work has different viewpoints and unique contributions. First, we show that parallel loop scheduling can still be applied to grid environments to reduce the completion

time of a program. Second, we put great emphasis on accurate estimation of node performance, rather than dynamically adjust scheduling parameters as in [25], to achieve efficient loop scheduling. In addition, both static configuration and dynamic environmental information are taken into consideration for estimation of node performance, which is not found in previous work. Finally, we have implemented two types of applications and investigate the setting of parameters on a grid testbed to verify our approach. Our previous work [16, 17] used a performance function to estimate the heterogeneous performance of nodes. However, the characteristics of dynamic grid environments and irregular workloads are not considered. In this paper, a more accurate performance function is proposed. In addition, dynamic information acquired from a monitoring tool is utilized to adapt to the dynamic environment. Furthermore, a sampling method is proposed to estimate the proportion of the workload to be assigned statically.

In [3, 24], the authors enhanced their self-scheduling schemes for clusters and applied this approach to grid platforms. However, the dynamically changing characteristic of computing grids has not been investigated. The reliability issue of the grid is addressed in [10], but this is out of the scope of this paper, which is focused on the performance issue. Besides, a theoretical view of dynamic loop scheduling is presented in [19], but its assumptions are not specific to grid environments.

The remainder of this paper is organized as follows. In Sect. 2, background on parallel loop scheduling and grid computing is reviewed. In Sect. 3, we describe our methodology to solve the parallel loop scheduling problem. Next, the configuration of our grid testbed is specified and experimental results on two types of applications are also presented in Sect. 4. Finally, the concluding remarks are given in the last section.

2 Background review

In this section, a prerequisite for our research is described. First, we review previous loop scheduling schemes. Then, the evolution of grid computing and its middleware are presented.

2.1 Loop scheduling schemes

Conventionally, loop scheduling schemes are classified according to the time when the scheduling decision is made. Static loop scheduling schemes make a scheduling decision at compile time, and equally assign the total iterations of a loop to processors. It is applied when each iteration of a loop takes roughly the same amount of time, and the compiler knows enough related information before compilation. Its advantage is less overhead at runtime, while the disadvantage is possible load imbalance. Well-known static scheduling schemes include Block Scheduling, Cyclic Scheduling, Block-D Scheduling, Cyclic-D Scheduling, etc. However, these schemes are not suitable for dynamic grid environments.

Dynamic loop scheduling schemes make a scheduling decision at runtime. Its disadvantage is more overhead at runtime, while the advantage is load balance. The schemes we focus in this paper are self-scheduling, which a large class of dynamic

loop scheduling schemes. Several self-scheduling schemes have been reviewed in [25], and they are restated here as follows.

- **Pure Self-scheduling (PSS)** This is a straightforward dynamic loop scheduling algorithm [12]. Whenever a processor becomes idle, a loop iteration is assigned to it. This algorithm achieves good load balance but also induces excessive overhead.
- **Chunk Self-scheduling (CSS)** Instead of assigning one iteration to an idle processor at one time, CSS assigns k iterations each time, where k, called the chunk size, is a constant. When the chunk size is one, this scheme is PSS, as discussed above. If the chunk size is set to the bound of the parallel loop equally divided by the number of processors, this scheme becomes static scheduling. A large chunk size will cause load imbalance while a small chunk size is likely to result in too much runtime overhead.
- **Guided Self-scheduling (GSS)** This scheme can dynamically change the number of iterations assigned to each processor [15]. More specifically, the next chunk size is determined by dividing the number of remaining iterations of a parallel loop by the number of available processors. The property of decreasing chunk size implies an effort is made to achieve load balance and to reduce the runtime overhead. By assigning large chunks at the beginning of a parallel loop, one can reduce the frequency of communication between the master and slaves.
- **Factoring Self-scheduling (FSS)** In some cases, GSS might assign too much work to the first few processors, so that the remaining iterations are not time-consuming enough to balance the workload. The Factoring algorithm addresses this problem [11]. The assignment of loop iterations to working processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Therefore, it balances loads better than GSS does when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of Factoring is not significantly larger than that of GSS.
- **Trapezoid Self-scheduling (TSS)** This approach tries to reduce the need for synchronization while still maintaining a reasonable load balance [22]. $TSS(N_s, N_f)$ assigns the first N_s iterations of a loop to the processor starting the loop and the last N_f iterations to the processor performing the last fetch, where N_s and N_f are both specified by the programmer or the system. This algorithm allocates large chunks of iterations to the first few processors and successively smaller chunks to the last few processors. Tzen and Ni proposed $TSS(N/2p, 1)$ as a general selection. In this case, the first chunk is of size $N/2p$, and consecutive chunks differ in size $N/8p^2$ iterations.

Table 1 shows the chunk sizes for the self-scheduling schemes above with respect to a loop with 1000 iterations. Besides, the number of available processors is 4.

In [23], the authors enhanced well-known loop self-scheduling schemes to fit an extremely heterogeneous PC cluster environment. A two-phased approach was proposed to partition loop iterations and it achieved good performance in heterogeneous testbeds. For example, GSS can be enhanced by partitioning α percentage of the total iterations according to their performance weighted by CPU clock in the first phase. Then, the remainder of the workload is still scheduled by GSS. In this paper, this enhanced scheme is called NGSS.

Table 1 Sample partition size

Scheme	Sample partition size
PSS	1, 1, 1, 1, 1, 1, 1, 1, 1, ...
CSS(125)	125, 125, 125, 125, 125, 125, 125, 125
GSS	250, 188, 141, 106, 79, 59, 45, 33, 25, ...
FSS	125, 125, 125, 125, 63, 63, 63, 63, 31, ...
TSS	125, 117, 109, 101, 93, 85, 77, 69, 61, ...

In [25], NGSS was further enhanced by dynamically adjusting the parameter α according to system heterogeneity. A performance benchmark was used to determine whether target systems are relatively homogeneous or relatively heterogeneous. In addition, the types of loop iterations were classified into four classes, and were analyzed respectively. The scheme enhanced from GSS is called ANGSS in this paper.

2.2 Grid computing and its middleware

The term “metacomputing” was first proposed in [18] and its goal was to provide the research community with a “Seamless Web” linking the user interface on the workstation and supercomputers. With the advent of networking technologies such as Ethernet and ATM, it has become possible to connect computers for the widespread, efficient sharing of data. As high performance networks have become less expensive, and as the price of commodity computers has dropped, it is now possible to connect a number of inexpensive computers through a high-speed network to conduct high performance computing.

Grid computing [6–8] can be thought of as distributed and large-scale cluster computing and as a form of networked parallel processing. It can be confined to the network of computer workstations within a corporation or it can be a public collaboration (in which case it is also sometimes known as a form of peer-to-peer computing). Grid computing appears to be a promising trend for three reasons. First, it promises to make more cost-effective use of a given amount of computer resources. Second, it has potential to solve problems that can not be approached without an enormous amount of computing power. Finally, it suggests that the resources of many computers can be cooperatively harnessed and managed as collaboration toward a common objective.

The Globus Project [9] provides software tools that make it easier to build computational grids and grid-based applications. These tools are collectively called the Globus Toolkit. The Globus Toolkit is used by many organizations to build computational grids that can support their applications. The composition of the Globus Toolkit can be pictured as three pillars: Resource Management, Information Services, and Data Management. Each pillar represents a primary component of the Globus Toolkit and makes use of a common foundation of security. GRAM implements a resource management protocol, MDS implements an information services protocol, and GridFTP implements a data transfer protocol. They all use the GSI security protocol at the connection layer.

The Monitoring and Discovery Service (MDS) is the information service component of the Globus Toolkit. It provides Grid information, such as resources that

are available and the state of the computational Grid. This information may include properties of the machines, computers, and networks in your Grid, such as the number of processors available, CPU load, network interfaces, file system information, bandwidth, storage devices, and memory.

2.2.1 Dynamic monitoring tools

Several tools have been developed to monitor a large number of machines in a standalone host as well as hosts in a cluster. For examples, Ganglia is an Open Source project and available on SourceForge at http://ganglia.sourceforge.net. It grew out from the University of California, Berkeley, Millennium Cluster Project in collaboration with the National Partnership for Advanced Computational Infrastructure (NPACI) Rocks Cluster Group. Ganglia provide a complete, real-time monitoring and execution environment based on a hierarchical design. It uses a multicast listen/announce protocol to monitor node status, and uses a tree of point-to-point connections to coordinate clusters of clusters and aggregate their state information. Ganglia uses the eXtensible Markup Language (XML) to represent data, eXternal Data Representation (XDR) for compact binary data transfers, and an open source package called RRDTool for data storage (in Round Robin databases) and for graphical visualization.

These tools can be useful because they monitor the availability of services on a host and detect if a host is overloaded, but they do not generally provide performance monitoring information at the level of detail needed to tune the performance of a Beowulf cluster. In this paper, Ganglia is utilized to acquire dynamic system information, such as CPU loading of available nodes.

2.2.2 MPI

MPI is a message-passing library standard that was published in May 1994. The "standard" of MPI is based on the consensus of the participants in the MPI Forums, organized by over 40 organizations. Participants include vendors, researchers, academics, software library developers and users. MPI offers portability, standardization, performance and functionality. MPICH-G2 [14] is a grid-enabled implementation of the MPI v1.1 standard. That is, using services from the Globus Toolkit, MPICH-G2 allows you to couple multiple machines, potentially of different architectures, to run MPI applications. MPICH-G2 automatically converts data in messages sent between machines of different architectures and supports multiprotocol communication by automatically selecting TCP for inter-machine messaging and (where available) vendor-supplied MPI for intra-machine messaging. Existing parallel programs written for MPI can be executed over the Globus infrastructure just after recompilation. In this paper, we used the C language associated with MPI to program the applications, such as Matrix Multiplication and Mandelbrot Set Computation [13].

3 Proposed approach

In this section, the system model is introduced first. Then, the parameters of performance ratio and static-workload ratio are described. Finally, we present the skeleton algorithm for the performance-based loop scheduling.

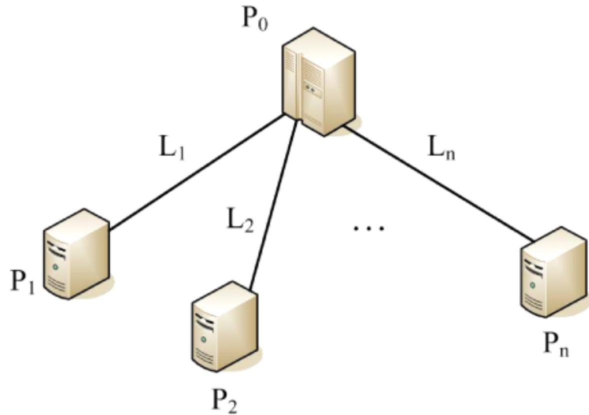


Fig. 1 The system model

3.1 The system model

The system in this work is modeled by a master-slave paradigm, which is represented by a star graph, $G = (N, E)$. In this graph, N means the set of all nodes on the grid, and E is the set of all edges between the master and the slaves. For example, as shown in Fig. 1, N is $\{P_0, P_1, \ldots, P_n\}$ and E is $\{L_1, L_2, \ldots, L_n\}$. In this example, P_0 is the master node and the other n nodes, $P_1, \ldots, P_n$, are slave nodes. Conceptually, there is a virtual link L_i connecting the master node and a slave node P_i. In reality, L_i may be composed of several networking segments connected by switches or/and routers.

In this model, there are two kinds of attributes associated with nodes, constants and variables. The values of the constant attributes do not vary during the lifetime of the node. For example, CPU clock speed, memory size, etc. are all constant attributes. On the other hand, the values of the variable attributes may fluctuate during the lifetime of the node. For example, CPU loading, available memory size, etc. are all constant attributes. In the following sections, the two kinds of attributes are utilized to model the heterogeneity of the dynamic grid.

Programming models are generally classified by the way memory is used. In the shared memory model each process accesses a shared address space, while in the message passing model processes communicate with other processes by sending and receiving messages. The message-passing paradigm is adopted in this paper. Basically, the programmer assumes the system consists of several processors, each with its own memory space, and writes a program to run on each processor. However, parallel programming generally requires communication between the processors to complete a task. The characteristic of the message-passing paradigm is that the processors communicate by sending messages instead of shared memory. Therefore, in the message-passing model, processors can not access each other's memory directly.

3.2 Performance ratio

The concept of performance ratio is previously defined in [18–20] in different forms and parameters, according to the requirements of applications. In this work, a different formulation is proposed to model the heterogeneity of the dynamic grid nodes. The purpose of calculating performance ratio is to estimate the current capability of

processing for each node. With this metric, we can distribute appropriate workloads to each node, and load balancing can be achieved. The more accurate the estimation is, the better the load balance is.

To estimate the performance of each slave node, we define a performance function (PF) for a slave node j as

$$PF_j(V_1, V_2, \dots, V_m) \tag{1}$$

where V_i, $1 < i < m$, is a variable of the performance function. In more detail, the variables could include CPU speed, networking bandwidth, memory size, etc. We propose to utilize a Grid Resource Monitoring Tool [21] to acquire the values of variable attributes for all slaves, and to acquire the values of constant attributes by MDS. In this paper, the *PF* for node j is defined as

$$PF_j = \frac{CS_j/CL_j}{\sum_{\forall \text{node}_i \in N} CS_i/CL_i} \tag{2}$$

where N is the set of all grid nodes; CS_i is the CPU clock speed of node i, and it is a constant attribute. The value of this parameter is acquired by the MDS service, as described in Sect. 2.2.1; CL_i is the CPU loading of node i, and it is a variable attribute. The value of this parameter is acquired by the Ganglia tool, as described in Sect. 2.2.2.

The performance ratio (*PR*) is defined to be the ratio of all performance functions. For instance, assume the values of PF_s of three nodes are 1/2, 1/3 and 1/4. Then, the *PR* is 1/2:1/3:1/4; i.e., the *PR* of the three nodes is 6:4:3. In other words, if there are 13 loop iterations, 6 iterations will be assigned to the first node, 4 iterations will be assigned to the second node, and 3 iterations will be assigned to the last one.

3.3 Static-workload ratio (*SWR*)

Another important factor to be estimated is the degree of variation for applications with irregular workload. For example, the Mandelbrot set computation is a problem involving the same computation on different data points which have different convergence rates [13]. This operation derives a resultant image by processing an input matrix which is an image of m pixels by n pixels. The resultant image is one of m pixels by n pixels. Figure 2 illustrates the workload distribution of a Mandelbrot set on $[-1.8, 0.5]$ to $[-1.2, 1.2]$ using a 800×800 pixel window. Obviously, a scheduling scheme which does not consider the effect of fluctuating workload would result in load imbalancing.

We propose to use a parameter, *SWR* (Static-Workload Ratio), to alleviate the effect of irregular workload. In order to take advantage of static scheduling, *SWR* percentage of the total workload is dispatched according to Performance Ratio. If the workload of the target application is regular, *SWR* can be set to be 100. However, if the application has irregular workload, such as Mandelbrot Set Computation, it is efficient to reserve some amount of workload for load balancing.

We propose to randomly take five sampling iterations, and compute their execution time. Then, the *SWR* of the target application i is determined by the following

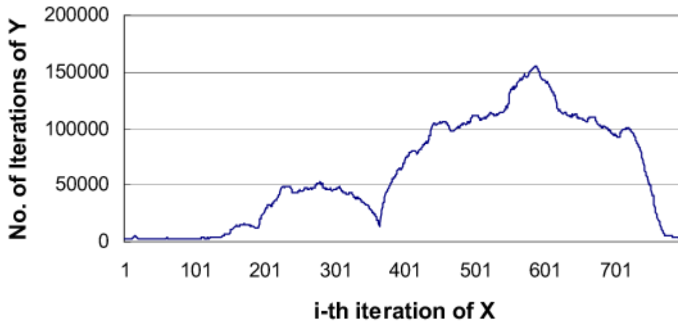


Fig. 2 The Mandelbrot set on $[-1.8, 0.5]$ to $[-1.2, 1.2]$ an 800×800 pixel window

formula.

$$SWR_i = \frac{\min_i}{\mathrm{MAX}_i} \tag{3}$$

where $\min_i$ is the minimum execution time of all sampled iterations for application i; MAX_i is the maximum execution time of all sampled iterations for application i.

For example, for a regular application with uniform workload distribution, the five sampled iterations are the same. Therefore, the *SWR* is 100%, and the whole workload can be dispatched according to Performance Ratio, with good load balance. However, for another application, the five sampling execution time might be 7, 7.5, 8, 8.5 and 10 seconds, respectively. Then the *SWR* is $7/10$, i.e. a percentage of 70. Therefore, 70 percentages of the iterations would be scheduled statically according to *PR*, while 30 percentages of the iterations would be scheduled dynamically by *GSS*.

3.4 The skeleton algorithm

Based on the information of workload distribution and node performance, we propose a skeleton algorithm for performance-based loop scheduling on grid environments. This algorithm is based on a message-passing paradigm, and consists of two modules: a master module and a slave module. The master module makes the scheduling decision and destinies workloads to slaves. On the other hand, the slave module processes the assigned work. This algorithm is just a skeleton, and the detailed implementation, such as data preparation, parameter passing, etc., might be different according to requirements of various applications.

Our algorithm is composed of four stages. In stage one, the related information are acquired. Then, stage two calculates the *SWR* and Performance Ratio. Next, *SWR* percentage of the total workload is statically scheduled according to the performance ratio among all slave nodes in stage three. Finally, the remainder of the workload is dynamically scheduled by Guided Self-Scheduling for load balancing. The algorithm of our approach is described as follows.

Module MASTER
```
Initialization
    /* Stage 1: Gathering the information */
```

```
collect the following information from
Grid_Monitoring_Tool and MDS:
    (1) CPU_Loading
    (2) CPU_Clock_Speed
collect the execution time of 5 sampled iterations

  /* Stage 2: Calculate two scheduling parameters */
calculate SWR of the workload
calculate Performance Ratio of all slave nodes

  /* Stage 3: Static Scheduling */
dispatch SWR percentage of workload according
to Performance Ratio
probe and receive for returned results

  /* Stage 4: dynamic Scheduling */
dispatch the (100-SWR)percentage of workload by GSS

Finalization
END MASTER
```

Module SLAVE

```
Initialization
While (a chunk of workload arrives) {
  receive the chunk of workload
    Compute on this chunk
  Send the result to the Master

}
Finalization
END SLAVE
```

4 Experimental results

To verify our approach, a grid testbed was built, and two types of application programs were implemented: Matrix Multiplication and Mandelbrot Set Computation. While the former has regular workload, the latter has irregular workload. First, the execution time of the PLS scheme was compared with those of previous schemes. Next, the impact of the parameter, *SWR*, on performance was investigated.

4.1 Grid testbed: TIGER Project [21]

A metropolitan-scale Grid computing platform named TIGER Grid [21] (standing for Taichung Integrating Grid Environment and Resource) has been built in a project leaded by Tunghai University. The TIGER grid interconnects computing resources of

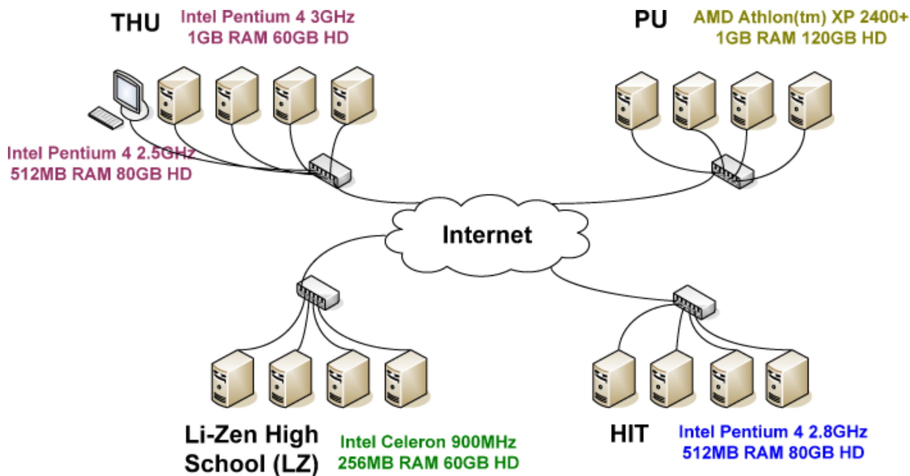


Fig. 3 The logical diagram of our grid test-bed

universities and high schools and shares available resources among them, for investigations in system technologies and high performance applications. This novel project shows the viability of implementation of such a project in a metropolitan city. The TIGER Grid computing platform consists of three universities and two high schools, all located in Taichung, Taiwan. The project of constructing such a grid infrastructure was to share computational resources of each institution.

The educational institutions participating in this project are Tunghai University (THU), Providence University (PU), HsiuPing Institute of Tech (HIT), National Dali High School (DL) and Li-Zen High School (LZ). They are interconnected by TANet (Taiwan Academic Network) with bandwidth of 1 Gbps. The TIGER Grid platform consists of 33 computing nodes, with 58 CPUs of different speed and total storage of more than 2TB. All these institutions are in Taiwan, and each is at least 10 kilometers away from THU geographically. All machines in this grid have Globus 3.2.1 or above installed.

In this paper, we have built a grid testbed based on part of the TIGER Grid and executed parallelized MPI programs on it. The master node is at Tunghai University (THU), and the slave nodes are located at Tunghai University (THU), Providence University (PU), Li-Zen High School (LZ), and Hsiuping Institute of Technology School (HIT). Figure 3 shows our grid testbed, and the specifications of the grid testbed are shown in Table 2. Figure 4 shows the real-time status of the grid testbed acquired by the monitoring tool, which is based on Ganglia [21].

4.2 Experiments on two applications

In this study, we have implemented two classes of applications in C language, with message passing interface (MPI) directives for parallelizing code segments to be processed by multiple CPUs. For readability of experimental results, the brief description of all implemented programs is listed in Table 3.

The conventional static scheduling scheme is to equally distribute the total workload to each worker at compile time. However, this scheme is obviously not suitable

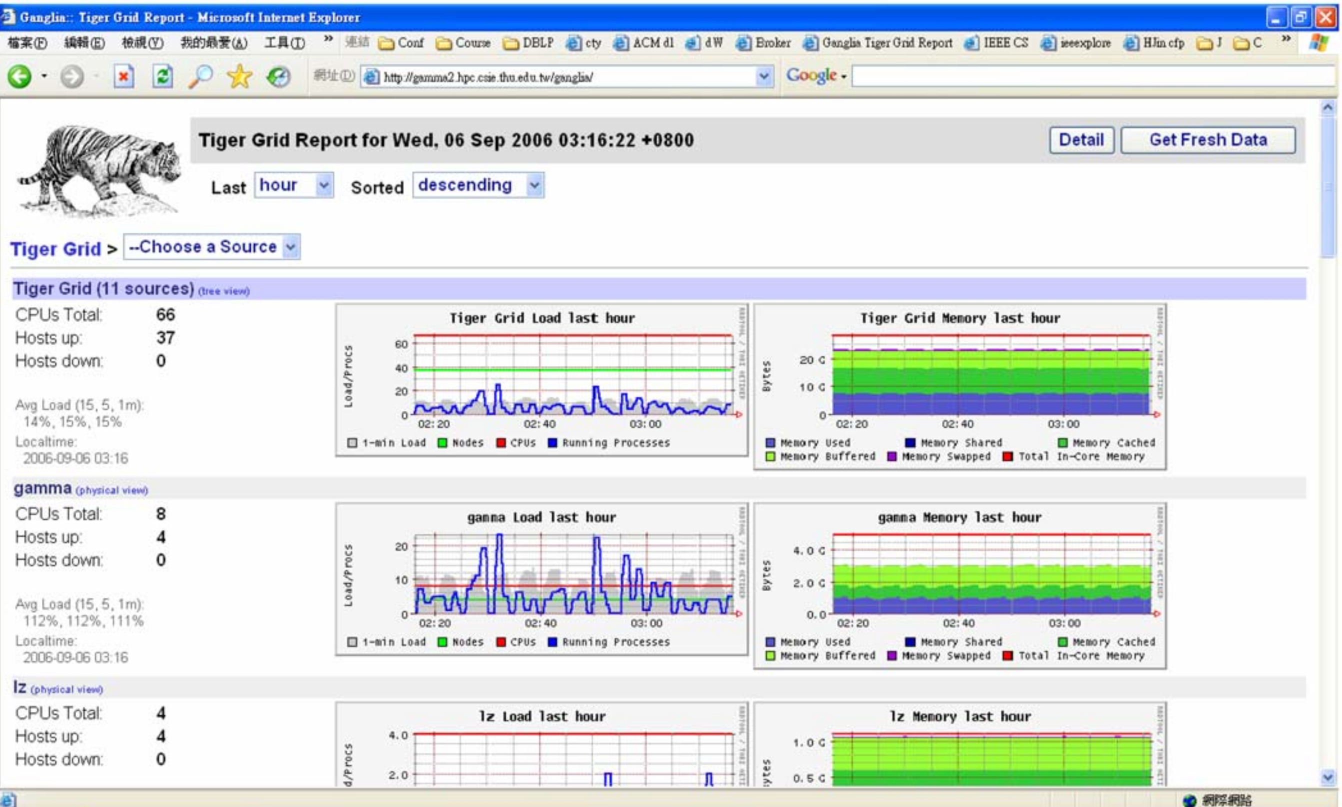


Fig. 4 The snapshot of the monitoring tool on the TIGER Grid

Table 2 Specifications of computing resources on the test-bed

Site	Host	CPU type	Clock (MHz)	RAM	NIC	Linux kernel	Globus version
THU	Delta1	Intel Pentium 4	3001	1 GB	1 G	2.6.12	4.0.1
	Delta2	Intel Pentium 4	3001	1 GB	1 G	2.6.12	4.0.1
	Delta3	Intel Pentium 4	3001	1 GB	1 G	2.6.12	4.0.1
	Delta4	Intel Pentium 4	3001	1 GB	1 G	2.6.12	4.0.1
LZ	lz01	Intel Celeron	898	256 MB	10/100	2.4.20	3.2.1
	lz02	Intel Celeron	898	256 MB	10/100	2.4.20	3.2.1
	lz03	Intel Celeron	898	384 MB	10/100	2.4.20	3.2.1
	lz04	Intel Celeron	898	256 MB	10/100	2.4.20	3.2.1
HIT	Gridhit0	Intel Pentium 4	2800	512 MB	10/100	2.6.12	3.2.1
	Gridhit1	Intel Pentium 4	2800	512 MB	10/100	2.6.12	3.2.1
	Gridhit2	Intel Pentium 4	2800	512 MB	10/100	2.6.12	3.2.1
	Gridhit3	Intel Pentium 4	2800	512 MB	10/100	2.6.12	3.2.1
PU	hpc09	AMD Athlon XP	1991	1 GB	1 G	2.4.22	3.2.1
	hpc10	AMD Athlon XP	1991	1 GB	1 G	2.4.22	3.2.1
	hpc11	AMD Athlon XP	1991	1 GB	1 G	2.4.22	3.2.1
	hpc12	AMD Athlon XP	1991	1 GB	1 G	2.4.22	3.2.1

Table 3 Description of all implemented programs

Scheduling scheme	Description	Reference
static	Weighted static scheduling	
gss	Dynamic scheduling (GSS)	[15]
fss	Dynamic scheduling (FSS)	[11]
tss	Dynamic scheduling (TSS)	[22]
ngss	Fixed α scheduling + GSS	[23]
angss	Adaptive α scheduling + GSS	[25]
pls	Proposed scheduling	

for dynamic and heterogeneous environments. Therefore, a weighted static scheduling scheme is adopted in this experiment. The principle of partitioning is according to the CPU clock speed of each processor. A faster node will get more workloads than a slower one proportionally.

4.2.1 Application 1: Matrix Multiplication

Matrix Multiplication is a fundamental operation in many numerical linear algebra applications. Its efficient implementation on parallel computers is an issue of prime importance when providing such systems with scientific software libraries. Consequently, considerable effort has been devoted in the past to the development of efficient parallel matrix multiplication algorithms, and this will remain a task in the fu-

ture as well. Many parallel algorithms have been designed, implemented, and tested on different parallel computers or cluster of workstations for matrix multiplication.

We have implemented the PLS scheme for Matrix Multiplication. The Master module is responsible for the distribution of workloads. When a slave node becomes idle, the master node sends two integers to the slave. The two numbers represent the beginning and ending pointers to the assigned chunk respectively. In other words, every node has a copy of the input matrices locally, so data communication is not significant in this kind of implementation. Therefore, communication cost between the master and the slave is low, and the dominant cost is the computation of matrix multiplication. The C/MPI code fragment of the Slave module for Matrix Multiplication is listed as follows. As the source code shows, a column is the atomic unit of allocation.

```
MPI_Recv(buf, count, MPI_FLOAT, source, tag,
MPI_COMM_WORLD, &status);
f=0;
while (status.MPI_TAG >0)
{
for (i=0; i<(count/SIZE); i++)
    for (j=0; j<SIZE; j++)
        c[i*SIZE+j]=0.0;

    /* computing */
    for (i=0; i<(count/SIZE); i++)
        for (j=0; j<SIZE; j++)
          for (k=0; k<SIZE; k++)
             c[i*SIZE+j] += buf[i*SIZE+k]*b[k*SIZE+j];

    /* sent result*/
    MPI_Send(c, count, MPI_FLOAT, 0, tag,
    MPI_COMM_WORLD);
    free(buf);
    free(c);

    /* get another size */
    MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD,
    &status);
    source = status.MPI_SOURCE;
    tag = status.MPI_TAG;
    MPI_Get_count(&status, MPI_FLOAT, &count);
    buf = (float*)malloc(count*sizeof(float));
    c = (float*)malloc(count*sizeof(float));
    MPI_Recv(buf, count, MPI_FLOAT, 0, MPI_ANY_TAG,
        MPI_COMM_WORLD, &status);
    }
}
```

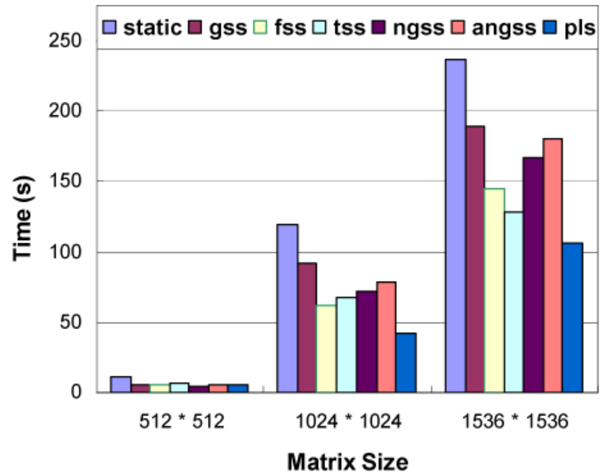


Fig. 5 Execution time for Matrix Multiplication with different input sizes

First, we want to compare the proposed PLS scheme with previous schemes with respect to the execution time. Figure 5 illustrates the execution time of weighted static scheduling, GSS, FSS, TSS, NGSS, ANGSS and our PLS scheme, with input matrix size 512×512, 1024×1024 and 1536×1536 respectively. The results are shown as follows.

- Among these schemes, PLS performs better than other schemes. The reason is that PLS accurately estimates the PR, and takes the advantage of static scheduling, thus reducing the runtime overhead.
- The static scheme obviously performs worse than other dynamic schemes. It is reasonable to say that the static scheme is not suitable for a dynamic environment, with respect to performance.
- It is interesting that traditional self-scheduling schemes (FSS and TSS) perform slightly better than NGSS and ANGSS. However, this result is inconsistent with that of previous research. The reason might be that the parameter α is set too high, 75. If the parameter α is set appropriately, it is possible for NGSS and ANGSS to perform better, as previous work has shown. This claim will be validated in the following experiment.

Next, it is interesting to compare the parameter α in NGSS and the *SWR* in PLS. Basically, the parameter α is set by the programmer, and it is difficult to choose an appropriate value adaptive to the dynamic environment. In contrast, *SWR* is automatically set by the algorithm. Figure 6 shows how scheduling parameters influence performance. As the figure shows, the behavior of PLS is more robust than the α self-scheduling schemes, NGSS. In this experiment, we find that NGSS gets near-optimal performance when α is 30. However, PLS shows very stable performance at every value of *SWR*. The reason is that PLS can accurately estimate the performance ratio and thus performs well when the workload is regular, even without the help of *SWR*.

4.2.2 Application 2: Mandelbrot set computation

The Mandelbrot set is a problem involving the same computation on different data points which have different convergence rates [13]. The Mandelbrot set, named after

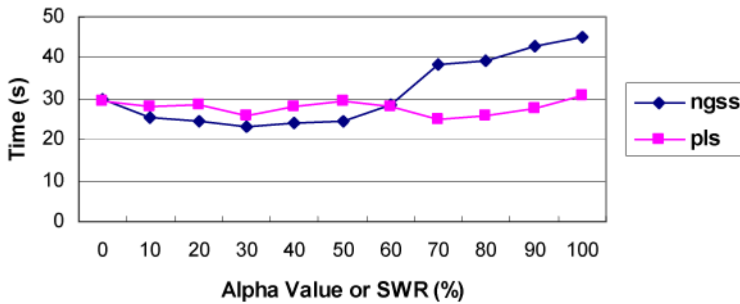


Fig. 6 Execution time for Matrix Multiplication with different values of parameters

Benoit Mandelbrot, is a fractal. Fractals are objects that display self-similarity at various scales. Magnifying a fractal reveals small-scale details similar to the large-scale characteristics. Although the Mandelbrot set is self-similar at magnified scales, the small scale details are not identical to the whole. In fact, the Mandelbrot set is infinitely complex. Yet the process of generating it is based on an extremely simple equation involving complex numbers. This operation derives a resultant image by processing an input matrix, A, where A is an image of m pixels by n pixels. The resultant image is one of m pixels by n pixels.

The PLS scheme has been implemented for Mandelbrot Set Computation. The Master module is responsible for the distribution of workload. When a slave node becomes idle, the master node sends two integers to the slave. As implemented in Matrix Multiplication, communication cost between the master and the slave is low, and the dominant cost is the computation of Mandelbrot Set. The C/MPI code fragment of the Slave module for Mandelbrot Set Computation is listed as follows.

```
MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
source = status.MPI_SOURCE;
tag = status.MPI_TAG;
MPI_Get_count(&status, MPI_INT, &count);
MPI_Recv(&b[0], count, MPI_INT, source, tag,
MPI_COMM_WORLD, &status);
while (status.MPI_TAG >0) {
/* Compute pixels in parallel */

   //t1 = MPI_Wtime();
   for (i = 0; i < Nx*Ny; i++)pix_tmp[i]=0.0;

   for (y = b[0]; y < b[1]; y++){
      for (x = 0; x < Nx; x++){
         c.real = Rx_min +
                  ((double) x * (Rx_max - Rx_min)/
                  (double) (Nx - 1));
```

```
        c.imag = Ry_min +
                 ((double) y * (Ry_max - Ry_min)/
                 (double) (Ny - 1));
        pix_tmp[y*Nx+x] = cal_pixel(c);
      }//for x
    }//for y
    /* sent result*/
    MPI_Send(&b[0], count, MPI_INT, 0, tag,
    MPI_COMM_WORLD);
    /* get another size */
    MPI_Probe(0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
    source = status.MPI_SOURCE;
    tag = status.MPI_TAG;
    MPI_Get_count(&status, MPI_INT, &count);
    MPI_Recv(&b[0], count, MPI_INT, source, tag,
    MPI_COMM_WORLD,&status);
    }
```

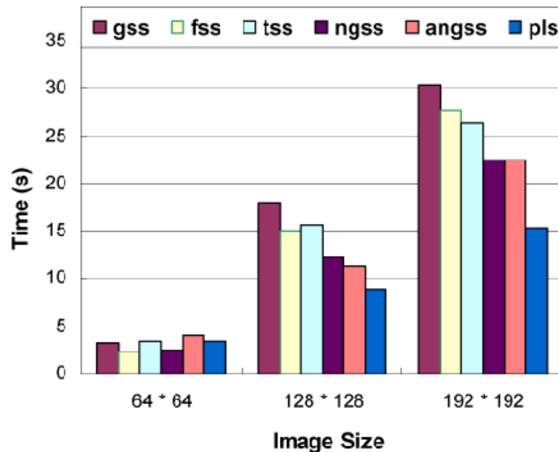


Fig. 7 Execution time for Mandelbrot set computation with different input sizes

In the following experiment, we want to compare the execution time of previous schemes with the proposed PLS. Figure 7 illustrates the execution time of GSS, FSS, TSS, NGSS, ANGSS and our PLS scheme, with input image size 64×64, 128×128 and 192×192 respectively. The execution time of weighted static scheduling is omitted due to its bad performance. According to the experience in Matrix Multiplication, the parameter α is set to 30. The results are shown as follows.

- Among these schemes, PLS still performs better than other schemes. The reason is also that PLS accurately estimates the PR, and takes the advantage of static scheduling, thus reducing the runtime overhead.
- Traditional self-scheduling schemes (GSS, FSS and TSS) perform worse than NGSS and ANGSS. The reason is that irregular workload is difficult to schedule. If the parameter α is set appropriately, it is certain for NGSS and ANGSS to perform better, as previous work has shown.

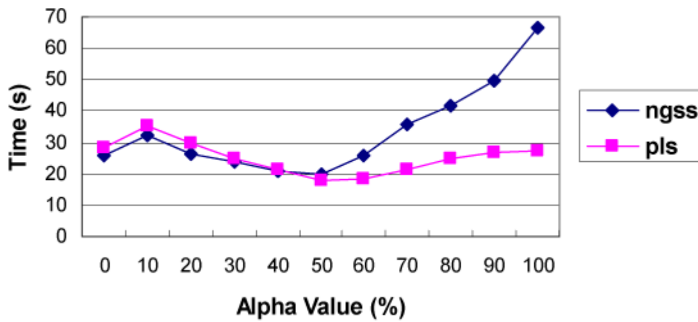


Fig. 8 Execution time for Mandelbrot set computation with different values of parameters

Next, the parameter α in NGSS and the *SWR* in PLS are compared for irregular workload. Figure 8 shows how scheduling parameters influence performance. In this experiment, we find that NGSS gets near-optimal performance when $\alpha = 50$. However, PLS shows stable performance at every value of *SWR*. In other words, PLS is more robust. However, the degree of fluctuation of PLS becomes larger than that for Matrix Multiplication. The reason is that, when the workload is irregular, the accuracy of estimation would be reduced slightly.

4.2.3 Characterization of irregular workload

In this section, we propose a parameter to characterize irregular workload. Based on this characterization, five types of irregular workloads are generated manually, and execution times of aforementioned schemes for these workloads are compared.

Figure 9 shows the proposed workload distribution model and two special cases. In this model, as shown in Fig. 9a, the x-axis means the workload size of one iteration, and the x-axis represents the frequency of some workload size. For example, the coordinate pair (9, 5) means that there are 5 iterations and each has a workload size of 9 (units). This model represents a workload distribution by two parameters: Width and Height. The distribution shown in Fig. 9b means all iterations have almost the same workload size. Therefore, this special case is an example of regular workload. For example, the workload distribution of Matrix Multiplication is like Fig. 9b. However, Fig. 9c illustrates a special case of irregular workload. In this figure, there are iterations with almost all sizes. In other words, the degree of workload variation is large, and it is a extreme irregular workload.

We define the Width-Height Ratio (*WHR*) as follows.

$$WHR = \frac{Width}{Height} \tag{4}$$

In (4), the parameters *Width* and *Height* are those shown in Fig. 9a.

To study the impact of irregular workload on execution time of loop scheduling schemes, we generate five workloads with different Width-Height Ratio (*WHR*), and execute aforementioned loop scheduling schemes on them. The results are shown in Fig. 10.

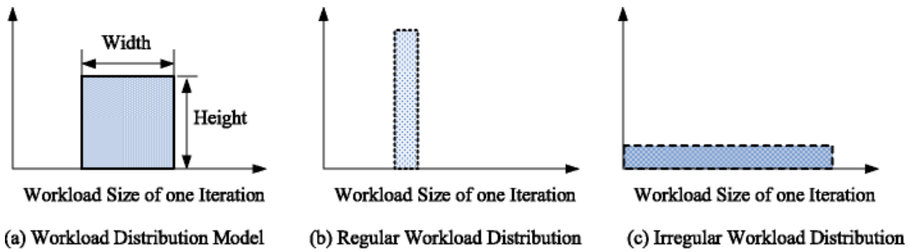
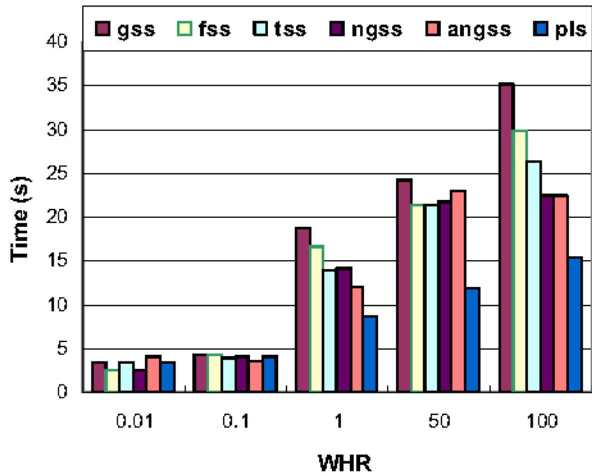


Fig. 9 Workload distribution model

Fig. 10 Execution time for five types of synthetic irregular workloads



4.2.4 *Overhead*

The overhead of the proposed approach results mainly from Stage 1 and Stage 2, in the Master module. The additional cost consists of information reading, sampled iteration execution, and parameter calculation.

The first step of Stage 1 is to collect information of CPU loading and CPU clock speed of each node. In our grid test-bed, the information is stored in the Master node and updated periodically by the Grid Monitoring Tool. Therefore, the cost of this step is $n \times T_{\text{read}}$, where n is the number of available nodes in the grid and T_{read} is the time to read a local variable.

The second step of Stage 1 is to gather the execution time of five sampled iterations, which are five quantile-numbers. For example, assume that there are 100 iterations. The five samples are: the first, 25th, 50th, 75th, and 100th iteration. The cost of this sampling process is a few simple arithmetic operations. Then, the five samples are executed. When the problem size is large enough, the execution time of the samples can be neglected. In Stage 2, *SWR* and *PR* are calculated. The cost is to evaluate formula (2) and (3).

Compared with the whole workload, these costs are insignificant. In fact, this claim is consistent with the experimental results in Sects. 4.2.1 and 4.2.2. In these experiments, the total execution time of the proposed approach includes these over-

head costs. When the problem size is small (64×64 in Mandelbrot set computation), the proposed approach did not outperform conventional schemes. Nevertheless, as the problem size increases, the proposed approach obviously outperforms conventional schemes. In other words, the benefit is larger than the overhead when the problem size is large enough.

5 Conclusions

In this paper, we have investigated the parallel loop scheduling problem on dynamic and heterogeneous grid environments. First, a performance-based approach was proposed to schedule parallel loops on grid environments. In this approach, the system heterogeneity is estimated by performance functions, and the dynamic environment is estimated by Static-Workload Ratio. On our grid platform, the proposed approach can obtain performance improvement on previous schemes. In our future work, we will implement more types of application programs to verify our approach. Furthermore, we will take network bandwidth into consideration when estimating the performance ratio for data-intensive applications.

Acknowledgements This paper is supported in part by National Science Council, Taiwan, ROC, under grants no. NSC94-2213-E-029-002, NSC95-2221-E-029-004 and NSC95-2622-E-029-003-CC3.

References

1. Baker MA, Fox GC (1999) Metacomputing: harnessing informal supercomputers. In: High performance cluster computing. Prentice-Hall
2. Banicescu I, Carino RL, Pabico JP, Balasubramaniam M (2005) Overhead analysis of a dynamic load balancing library for cluster computing. In: Proceedings of the 19th IEEE international parallel and distributed processing symposium, 2005
3. Cheng KW, Yang CT, Lai CL, Chang SC (2004) A parallel loop self-scheduling on grid computing environments. In: Proceedings of the 2004 IEEE international symposium on parallel architectures, algorithms and networks, KH, China, May 2004, pp 409–414
4. Chronopoulos AT, Penmatsa S, Xu J, Ali S (2006) Distributed loop-self-scheduling schemes for heterogeneous computer systems. Concurr Comput: Pract Exp 18:771–785
5. Chronopoulos AT, Andonie R, Benche M, Grosu D (2001) A class of loop self-scheduling for heterogeneous clusters. In: Proceedings of the 2001 IEEE international conference on cluster computing, 2001, pp 282–291
6. Foster I, Kesselman C (1997) Globus: a metacomputing infrastructure toolkit. Int J Supercomput Appl High Perform Comput 11(2):115–128
7. Foster I, Kesselman C, Tuecke S (2001) The anatomy of the grid: enabling scalable virtual organizations. Int J Supercomput Appl High Perform Comput 15(3):200–222
8. Foster I (2002) The Grid: a new infrastructure for 21st century science. Phys Today 55(2):42–47
9. The Globus Project. http://www.globus.org/
10. Herrera J, Huedo E, Montero RS, Llorente IM (2006) Loosely-coupled loop scheduling in computational grids. In: Proceedings of the 20th IEEE international parallel and distributed processing symposium, 2006
11. Hummel SF, Schonberg E, Flynn LE (1992) Factoring: a method scheme for scheduling parallel loops. Commun ACM 35:90–101
12. Kruskal C, Weiss A (1984) Allocating independent subtaskson parallel processors. IEEE Trans Softw Eng 11:1001–1016
13. Mandelbrot BB (1988) Fractal geometry of nature. Freeman, New York
14. MPICH-G2. http://www.hpclab.niu.edu/mpi/

15. Polychronopoulos CD, Kuck D (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans Comput 36(12):1425–1439
16. Shih WC, Yang CT, Tseng SS (2005) A hybrid parallel loop scheduling scheme on grid environments. In: Grid and cooperative computing—GCC 2005: fourth international conference, Lecture notes in computer science. Springer, November 2005
17. Shih WC, Yang CT, Tseng SS (2005) A performance-based parallel loop self-scheduling on grid environments. In: Network and parallel computing: IFIP international conference, NPC 2005, Lecture notes in computer science, vol 3779. Springer, November 2005, pp 48–55
18. Smarr L, Catlett C (1992) Metacomputing. Commun ACM 35(6):44–52
19. Tabirca S, Tabirca T, Yang LT (2006) A convergence study of the discrete FGDLS algorithm. IEICE Trans Inf Syst E89-D(2):673–678
20. Tang P, Yew PC (1986) Processor self-scheduling for multiple-nested parallel loops. In: Proceedings of the 1986 international conference on parallel processing, 1986, pp 528–535
21. TIGER Grid Report. http://gamma2.hpc.csie.thu.edu.tw/ganglia/
22. Tzen TH, Ni LM (1993) Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. IEEE Trans Parallel Distrib Syst 4:87–98
23. Yang CT, Chang SC (2004) A parallel loop self-scheduling on extremely heterogeneous PC clusters. J Inf Sci Eng 20(2):263–273
24. Yang CT, Cheng KW, Li KC (2004) An efficient parallel loop self-scheduling on grid environments. In: Jin H, Gao G, Xu Z (eds), NPC'2004 IFIP international conference on network and parallel computing, Lecture notes in computer science. Springer, Heidelberg
25. Yang CT, Cheng KW, Li KC (2005) An efficient parallel loop self-scheduling scheme for cluster environments. J Supercomput 34:315–335

Copyright of Journal of Supercomputing is the property of Springer Science & Business Media B.V. and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.