Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Parallel Computing 33 (2007) 467–487
PARALLEL COMPUTING
www.elsevier.com/locate/parco

On development of an efficient parallel loop self-scheduling for grid computing environments ☆

Chao-Tung Yang [a,*], Kuan-Wei Cheng [a], Wen-Chung Shih [b]

[a] *High Performance Computing Laboratory, Department of Computer Science and Information Engineering, Tunghai University, Taichung 40704, Taiwan, ROC*

[b] *Department of Computer and Information Science, National Chiao Tung University, Hsinchu 30010, Taiwan, ROC*

Received 14 March 2006; received in revised form 25 October 2006; accepted 6 January 2007
Available online 27 January 2007

Abstract

The approaches to deal with scheduling and load balancing on PC-based cluster systems are famous and well-known. Self-scheduling schemes, which are suitable for parallel loops with independent iterations on cluster computer system, they have been designed in the past. In this paper, we propose a new scheme that can adjust the scheduling parameter dynamically on an extremely heterogeneous PC-based cluster and Grid computing environments in order to improve system performance. A Grid computing environment consists of multiple PC-based clusters is constructed using Globus Toolkit and MPICH-G2 middleware. The experimental results show that our scheduling can result in higher performance than other similar schemes on Grid computing environments.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Parallel loop; Self-scheduling; Cluster computing; Grid computing; Globus toolkit; Load balancing

1. Introduction

Grid computing, most simply stated, is distributed computing taken to the next evolutionary level. The goal is to create the illusion of a simple yet large and powerful self-managing virtual computer out of a large collection of connected heterogeneous systems sharing various combinations of resources. The standardization of communications between heterogeneous systems created the Internet explosion. The emerging standardization for sharing resources, along with the availability of higher bandwidth, are driving a possibly equally large evolutionary step in Grid computing [1,3,7–11,13,14,17–19,21,22,25–27,30].

In exploiting the computing power of clustered computers, it is important to know how to assign parallel loops to computers such that computer loads are well balanced. That is, understanding how to map various

☆ This paper is supported in part by NSC (National Science Council) Taiwan R.O.C., under grants no. NSC94-2213-E-029-002, NSC95-2221-E-029-004, and NSC95-2622-E-029-003-CC3.

* Corresponding author. Tel.: +886 4 23590415; fax: +886 4 23591567.
E-mail address: ctyang@thu.edu.tw (C.-T. Yang).

0167-8191/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2007.01.003

parts of parallel coding to computing resources in order to minimize overall computing time and make efficient use of these resources. Loops often comprise a large portion of a program's parallelism. An efficient approach to minimizing application execution times is to concentrate on the parallelism available in the loops. A loop is called a DOALL loop if there is no cross-iteration dependence in the loop, i.e., all the loop iterations can be executed in parallel. If all iterations of a DOALL loop are distributed among system processors evenly, a high degree of parallelism can be exploited. Parallel loop scheduling is a method for scheduling DOALL loops evenly across multiprocessor systems [2,4–6,12,16,20,23,24,28,29,31,32].

If loop iterations are distributed to different processors as evenly as possible, the parallelism within loop iterations can be efficiently exploited. Loops can be roughly divided into four kinds, uniform workload, increasing workload, decreasing workload, and random workload loops, as shown in Fig. 1. These are the most commonly found in computer programs, and should cover most loop parallelism cases.

In relatively homogeneous cases, workloads can be partitioned proportionally to working computer nodes according to respective computing power, but this method will not work in relatively heterogeneous cases. A method for solving parallel loop scheduling problems in heterogeneous cluster environments using α-based self-scheduling schemes was introduced in [32,34,35]. The self-scheduling scheme works well in moderately heterogeneous and extremely heterogeneous environments. Quite large performance differences may also be noted between the fastest and slowest computers. After each round of executions is finished, run-time information report is provided to the algorithm in order to improve performance of the next task submission.

In this paper, we revise known loop self-scheduling schemes to fit Grid computing environments [33]. The HINT Performance Analyzer [15] is used to determine whether target systems are relatively homogeneous or relatively heterogeneous. We then partition loop iterations into four classes, based on typical cluster system cases to achieve good performance in any given computing environment.

Internet computing and Grid technologies promise to change the way we tackle complex problems. They will enable large-scale aggregation and sharing of computational, data and other resources across institutional boundaries. And harnessing these new technologies effectively will transform scientific disciplines ranging from high-energy physics to the life sciences. In this paper, a Grid computing environment is proposed and constructed on multiple PC clusters by using Globus Toolkit (GT) [30] and SUN Grid Engine (SGE) [26]. The experimental results are also conducted by using the matrix multiplication to demonstrate the performance. On the other hand, the approaches to deal with scheduling and load balancing on multiple heterogeneous PC clusters computer system are not mature. Self-scheduling schemes which are suitable for parallel loops with independent iterations on heterogeneous cluster computer system have been designed in the past. However, these schemes, such as FSS, GSS and TSS, can not achieve load balancing

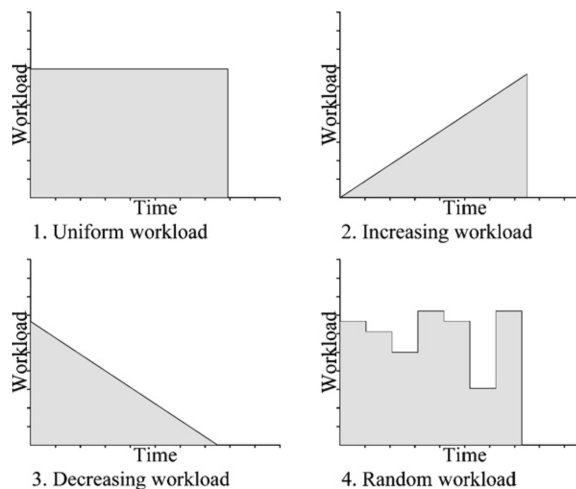

Fig. 1. Four types of loop.

in extremely heterogeneous environment. We propose a heuristic approach based upon α-based self-scheduling scheme to solve parallel regular loop scheduling problem on an extremely heterogeneous Grid computing environment.

The remainder of this paper is organized as follows. In Section 2, we introduce several typical and well-known self-scheduling schemes, and a famous benchmark used to analyze computer system performance. Our methodology is described in Section 3. In Section 4, experimental results are presented, and in Section 5 we give conclusions and future work.

2. Background review

2.1. Parallel loop self-scheduling

Parallel loop scheduling decisions in a parallel processing system can be made in two ways: statically at compile-time, and dynamically at run-time.

Static scheduling is usually applied when iterations are uniformly distributed among processors [4–6]. It has the drawback of creating load imbalances when:

- loop styles are not uniformly distributed,
- loop bounds cannot be known at compile-time,
- when the system is heterogeneous, and
- when locality management cannot be exercised.

Dynamic scheduling is more appropriate for load balancing; however, run-time overhead must be taken into consideration. Parallelizing compilers generally distribute loop iterations using only one scheduling algorithm type, static or dynamic.

Dynamic scheduling adjusts schedules during execution and is especially suitable when the number of iterations is uncertain and when each iteration takes a different amount of time. Although it is more suitable for load balancing between processors, its run-time overhead is costly.

Master/slave computation patterns are used to model problems, i.e., the master coordinates data distribution to slaves, which perform computations and transmit the results back to the master. The master is not responsible for workloads and idle-slave requests for new loop iterations, and no communication occurs between slaves. The number of iterations assigned to a slave is a critical issue. Improper assignment will cause bad system performance. A master/slave model is shown in Fig. 2.

Self-scheduling includes a large class of adaptive/dynamic centralized loop scheduling schemes. In a common self-scheduling scheme, p denotes the number of processors, N denotes total iterations, and f is a function for producing chunk-sizes at each step. At the ith scheduling step, the master computes the chunk-size C_i and the remaining number of tasks R_i, $R_0 = N$, $Ci = f(i,p)$, $R_i = R_i - 1 - C_i$, where f may have parameters other than just i and p, such as R_{i-1}. The master assigns C_i tasks to an idle-slave and load imbalances are dependent on the execution time gaps between t_j, for $j = 1,\ldots,p$, as shown in [24].

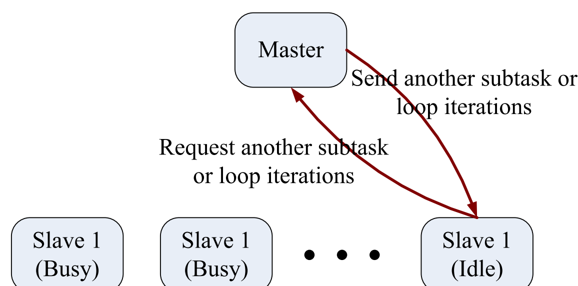

Fig. 2. A master/slave model.

Different ways of computing C_i have given rise to different scheduling schemes. The most notable ones are as follows:

Pure self-scheduling (PSS): the easiest and most straightforward dynamic loop scheduling algorithm [24]. Whenever a processor is idle, iterations are assigned to it. This algorithm achieves good load balancing but also induces excessive overhead [24].

Chunk self-scheduling (CSS): instead of assigning iterations to idle processors as in self-scheduling, CSS assigns k iterations each time, where k, called the chunk size, is fixed and must be specified by either the programmer or by the compiler [20]. When the chunk size is 1, the scheme is purely self-scheduling, as discussed above. When the chunk size is set to the boundary of the parallel loop equally divided by the number of processors, a static scheduling scheme results. Large chunk sizes cause load imbalances, while small chunk sizes are likely to produce excessive scheduling overhead. CSS(s) is used for various partitioning schemes. CSS(s) is a modified version of CSS; s refers to chunk size.

Guided self-scheduling (GSS): This algorithm can dynamically change the numbers of iterations assigned to idle processors [24]. 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 that an effort is made to achieve load balancing and to reduce the scheduling overhead. By assigning large chunks at the beginning of a parallel loop, one can reduce the frequency of communication between master and slaves. The small chunks at the end of a loop partition serve to balance the workload across all working processors.

Factoring self-scheduling (FSS): GSS may sometimes assign too much work to the first few processors such that the remaining iterations are not time-consuming enough to balance the workload [16]. This situation arises when initial iterations in a loop are much more time-consuming than the iterations that follow. The FSS algorithm addresses this problem [16]. The assignment of loop iterations to working processors proceeds in phases. During each phase, only a subset of remaining loops iterations (usually half) is equally divided among available processors. Because FSS assigns a subset of the remaining iterations in each phase, it balances workloads better than GSS when loop iteration computation times vary substantially. The synchronization overhead of FSS is not significantly greater than that of GSS.

Trapezoid self-scheduling (TSS): This approach tries to reduce the need for synchronization while still maintaining reasonable load balances [31]. 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 either the programmer or parallelizing compiler. 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 [31]. In this case, the first chunk is of size $N/2p$, and consecutive chunks differ in size $N/8p^2$ iterations. The size difference of successive chunks is always a constant in TSS, whereas it is a decreasing function in GSS and in FSS.

2.2. Grid computing platforms

This description of Grid architecture identifies requirements for general classes of component [1,3,7–11,13,14,17–19,21,22,25–27]. The result is an extensible, open architectural structure within which can be placed solutions to key user requirements. The architecture is organized into component layers, as shown below. Components within each layer share common characteristics but can build on capabilities and behaviors provided by any lower layer. The architectural description is high level and places few constraints on design and implementation. The layered Grid architecture and its relationship to the Internet protocol architecture is shown in Fig. 3.

The Grid fabric layer contains the resources that are to be shared. This could include computational power, data storage, sensors etc. This sharing is controlled by Grid protocols but the resource could include local networks. In this case the local protocols take over at this point. The Grid system is just concerned with access above this point.

The connectivity layer contains the communication and authentication protocols required for Grid-specific network transactions. Communication protocols enable the exchange of data between different fabric layer

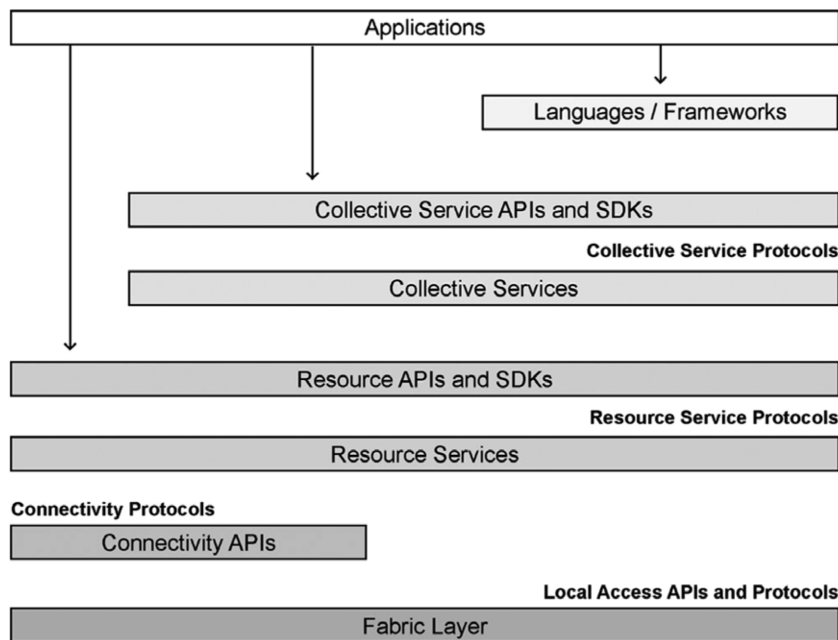

Fig. 3. The Grid architecture.

resources. Authentication protocols build on communication services to provide secure mechanisms for verifying the identity of users and resources.

The resource layer uses the communication and security protocols of the connectivity layer to control the secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. Resource layer protocols call fabric layer functions to access and control local resources. Resource layer protocols are concerned entirely with individual resources.

While the resource layer is focused on interactions with a single resource, the collective layer contains protocols and services that are global in nature and capture interactions across collections of resources. Collective components are so designed that they implement a wide variety of sharing behaviors without placing new requirements on the fabric resources being shared such as: A directory service may allow users to query for resources by name or by attributes such as type, availability, or load.

The final layer in the Grid architecture comprises the user applications. Applications are constructed in terms of, and by calling upon, services defined at each layer in the Grid structure. At each layer well-defined protocols provide access to some useful service: resource management, data access, resource discovery, and so forth. At each layer protocols and services are used to perform desired actions.

2.2.1. Globus Toolkit

The Globus Project provides software tools that make it easier to build computational Grids and Grid-based applications. These tools are collectively called The Globus Toolkit (http://www.globus.org/). We adopted it as infrastructure for our Grid testbed. The toolkit includes software for security, information infrastructure, resource management, data management, communication, fault detection, and portability.

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. The Globus Resource Allocation Manager (GRAM) implements a resource management protocol, the Metacomputing Directory Service (MDS) implements an information services protocol, and GridFTP implements a data transfer protocol. They all use the GSI security protocol at the connection layer.

GRAM provides an API for submitting and canceling job requests, as well as checking the statuses of submitted jobs. The specifications are written by the Resource Specification Language (RSL), and processed by GRAM as part of each job request.

MDS is the information services component of the Globus Toolkit and provides information about available resources on the Grid and their statuses. Via the default LDAP schema distributed with Globus, it gives current information about the Globus gatekeeper including CPU type and number, real memory, virtual memory, file systems and networks.

GridFTP is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. GridFTP protocol is based on FTP, the highly popular Internet file transfer protocol.

2.2.2. MPICH-G2

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 [22], organized by over 40 organizations. Participants include vendors, researchers, academics, software library developers and users. MPI offers portability, standardization, performance and functionality [22].

MPICH-G2 [21,22] is a Grid-enabled implementation of the MPI v1.1 standard. That is, using services from the Globus Toolkit® (e.g., job startup, security); 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 multi-protocol 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 [22].

3. Methodology

3.1. Loop styles analysis and our previous work

For the programs with regular loops, intuitively, we may want to partition problem size according to their CPU clock in heterogeneous environment. However, the CPU clock is not the only factor which affects computer performance. Many other factors also have dramatic influences in this aspect, such as the amount of memory available, the cost of memory accesses, and the communication medium between processors, etc. [5]. Using this intuitive approach, the result will be degraded if the performance prediction is inaccurate. A computer with largest inaccurate prediction will be the last one to finish the assigned job.

Loops can be roughly divided into four kinds, as shown in Fig. 1: *uniform workload*, *increasing workload*, *decreasing workload*, and *random workload* loops. They are the most common ones in programs, and should cover most cases. These four kinds can be classified two types: regular and irregular. The first kind is regular and the last three ones are irregular. Different loops may need to be handled in different ways in order to get the best performance. Since workload is predictable in regular loops, it is not necessary to process load balancing at beginning.

We propose to partition problem size in two stages. At first stage, partition $\alpha\%$ of total workload according to their performance weighted by CPU clock. In the way, the communication between master and slaves can be reduced efficiently. At second stage, partition following $(100 - \alpha)\%$ of total workload according to known self-scheduling scheme. In the way, load balancing can be archived. This approach can be suitable for all regular loops. An appropriate α value will lead to good performance.

In our previous work [32], we have proposed a α-based self-scheduling for parallel loop scheduling problem in cluster environments, The $\alpha\%$ portion of the workload was performed according to their performance weighed by CPU clock in the first phase, and the remaining $(100 - \alpha)\%$ of the workload according to known self-scheduling in the second phase. Experiments were performed on a PC-based cluster with six computing nodes; the fastest computer was 7.5 times faster than the slowest one in CPU clock rate. Various α values were applied to the matrix multiplication with the best performance coming when $\alpha = 75$. We also tested our schemes on two simulated increasing/decreasing workload loops and obtained

performance improvement. Our approach was thus found to be suitable for all applications with regular parallel loops. We then provided three improved self-scheduling schemes [32] using FSS, GSS, and TSS with the α-based Self-Scheduling Scheme: "N_FSS," "N_GSS," and "N_TSS" (originally "FSS", "GSS", and "TSS"); "N" signifying "new". Furthermore, dynamic load balancing approach should not be aware of the run-time behavior of the applications before execution. But in *GSS* and *TSS*, to achieve good performance, computer performance of each computer in the cluster has to be in order in extreme heterogeneous environment, which is not very applicable. With our schemes, this trouble will not exist. In the following example, the terminology "*FSS*-80" stand for "$\alpha = 80$, and remainder iterations use *FSS* to partition" and so on.

Example 1. Suppose that there is a cluster consisting of five slaves. Each of computing nodes has CPU clock of 200 MHz, 200 MHz, 233 MHz, 533 MHz, and 1.5 GHz, respectively. Table 1 shows the different chunk sizes for a problem with the number of iteration $I = 2048$ in this cluster. The number of scheduling steps is parenthesized.

3.2. New version of our method

The α-based self-scheduling scheme gave us the inspiration for this paper. It permits users to choose partition parameters α before execution of the initialization phase, but weaknesses remain. First, we noted that the fixed and monotonous parameter α is not changeable in run-time scheduling. Second, a parameter decision before run-time execution with machine information can affect scheduling results. Finally, if system architecture configurations change unexpectedly, our previous α-based Self-Scheduling Scheme [32] cannot cope with them. In this paper, we present some approaches to overcoming these weaknesses.

Hierarchical INTegration (HINT) [15] is a computer benchmarking tool developed at the Ames Laboratory Scalable Computing Laboratory (SCL), funded by the Office of Scientific Computing, U.S. Department of Energy (DOE). Unlike conventional benchmarks, HINT neither fixes the problem size nor the calculation time. Instead, it uses a measure called QUIPS (QUality Improvement Per Second). Significant HINT characteristics are listed as below:

- Problem size in HINT is scalable; not specific as in LINPACK or NPB2. Hence, it can analyze performance issues on computing systems with different subsystem main memory sizes. It can also predict performance for various problem sizes.
- HINT uses several data types (integer, floating point, short integer, etc.). Thus, it helps us to understand variations in computer processing of different data types.
- It reveals many aspects of computer performance: operating speed, precision, memory bandwidth, and usable memory size. It is completely unlike conventional computer benchmarks. HINT measures "QUIPS" (QUality Improvement Per Second) as a function of time. "Net QUIPS" is a single number that summarizes QUIPS over time. The faster the computer, the greater the need for high precision, fast access and large usable storage at all levels. Unlike conventional benchmarks, shortcomings in any of these are reflected in the figure of merit (Net QUIPS).

As mentioned above, more and more computing systems today are composed of homogeneous and heterogeneous workstation clusters in which the workstations may have similar or different CPU performance capabilities, amounts of memory, cache sizes, I/O, etc. Before we submit tasks to computational resources, we have to determine the differences between these systems in order to get better performance. Therefore, we use HINT to collect system information that helps us develop decision policies for differentiating between typical cluster system cases.

The HINT Performance Analyzer Tool assisted us in comparatively analyzing two cluster systems. We must have a proper response and an appropriate self-scheduling scheme process for when system architectures and loop styles are changeable. Parallel loop analysis is essential since parallel loops can be roughly divided into the four types shown in Fig. 1. These are the most common parallel coding, and cover most parallel loop cases.

Table 1
Sample partition size of Example 1 by using our previous work

GSS	410, 328, 262, 210, 168, 134, 108, 86, 69, 55, 44, 35, 28, 23, 18, 14, 12, 9, 7, 6, 5, 4, 3, 2, 2, 2, 1, 1, 1, 1 ($N = 30$)
GSS-80	923, 328, 144, 123, 121, 82, 66, 53, 42, 34, 27, 21, 17, 14, 11, 9, 7, 6, 4, 4, 3, 2, 2, 1, 1, 1, 1, 1 ($N = 28$)
FSS	205, 205, 205, 205, 205, 103, 103, 103, 103, 103, 51, 51, 51, 51, 51, 26, 26, 26, 26, 26, 13, 13, 13, 13, 13, 6, 6, 6, 6, 6, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1, 1 ($N = 43$)
FSS-80	923, 328, 144, 123, 121, 41, 41, 41, 41, 41, 21, 21, 21, 21, 21, 10, 10, 10, 10, 10, 5, 5, 5, 5, 5, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1 ($N = 39$)
TSS	204, 194, 184, 174, 164, 154, 144, 134, 124, 114, 104, 94, 84, 74, 64, 38 ($N = 16$)
TSS-80	923, 328, 144, 123, 121, 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 1 ($N = 23$)

We developed a new method incorporating dynamic parameterization and systematic adjustment for the new improved algorithm version. For convenience, the new improved algorithms were named A_N_FSS, A_N_GSS, and, A_N_TSS, where A means "Adjustment". System definition is the first step in our approach. The HINT Performance Analyzer [2] is given for helping us to distinguish whether the target system is relatively homogeneous or relatively heterogeneous. We gather CPU performance capabilities, amounts of memory, cache sizes, and basic system performance by HINT. An optimal "alpha" value with running the HINT benchmark can be specified when system architecture has changed at each time. Users first know which resources are available just before their application run by using "Monitoring and Information Services" of our Grid resource broker in Grid environment, which is described at the end in Section 4.2.1. Therefore, they are easy to specify the alpha value every execution time. This cost may not affect on their total execution time and the experimental results in the paper do not count the cost. An updatable library, called System Information Array (SIA), is build to record the collection of the information. Define the two Cluster System Typical Cases as follows:

Gather CPU information, P_1, P_2, ..., and P_n,

Assume P_1 is the node that has the worst performance (working ability) of all.

Say, $P_n = r^n P_1$.

Partition α% of workload according to their performance weighted by CPU clock and the rest $(100 - \alpha)$% of workload according to known self-scheduling scheme, such as FSS, GSS, or TSS.

(1) Define *Heterogeneous ratio* (HR), $\mathrm{HR} = \frac{P_1}{P_n} \approx \frac{\mathrm{MinQUIPS}}{\mathrm{MaxQUIPS}} \approx \frac{1}{r^n} < \alpha'/100$, where α' is the temporary value of α.
(2) Case 1: If $\alpha' < \mathrm{HR}$, then we say the target system is relatively heterogeneous case.
Case 2: If $\alpha' > \mathrm{HR}$, then we say the target system is relatively homogeneous case.
(3) If the target system is relatively heterogeneous system, we start the α-based self-scheduling scheme with $\alpha = \alpha'\%$.

If the target system is relatively homogeneous, then we run the HINT benchmark to build (and update) the SIA, and start the α-based self-scheduling scheme with $\alpha = 100\%$.

The alpha value is pre-computed before each execution. However, it is not necessary to re-compute the alpha value every time. Only when the system configuration is modified, the re-computation of the alpha value is needed. Also, the alpha value depends on system configuration, and in this paper, the same alpha value is used for different applications. In this work, the execution time does not include the execution time of HINT.

There is still a point for attention: not always update the SIA before each time of job submission, only when the system has one or more new nodes added, SIA-update will be needed and α will be properly adjusted. Partial program source code is shown below:

```
void master(int num_procs)
{
float *a, *buf;
int i, j, rowc, r, source, tag, count, r1, r2, resource;
MPI_Status status;
min_CPU = quips[0];
```

```
max_CPU = quips[0];
for (i = 0 ; i < num_procs − 1 ; i++)
{
  if(quips[i] > max_CPU)
    max_CPU = quips[i];
  if(quips[i] < min_CPU)
    min_CPU = quips[i];
}
prs = min_CPU / max_CPU;
/* get every trunk size */
ptrfirst = (struct ss *)NULL;
r1 = (SIZE*prs)/100;
r2 = SIZE − r1;
```

In the above code, "ptrfrist = (strcuct ss *)NULL" is for linked list initialization. The variable r1 stores the workload size to be dispatched according to performance ratio of nodes. The variable r2 stores the value of remaining workload. We implemented dynamic adjustment of scheduling parameters to fit multiform system architectures, and message-passing interface (MPI) directives for parallelizing code segments to be executed by multiple CPUs. We chose MPI for our programming environment since MPI is more suited to cluster computing than other programming environments such as the Parallel Virtual Machine (PVM) [12] and the Meta-System Approach [13]. In the MPI programming environment, we write just one program for both master and slave nodes. Thus, MPI code is easy to use and maintain. The Parallel Virtual Machine (PVM) requires the development of two individual program modules, one for the master node and the other for the slave nodes, which makes it inefficient and hard to maintain. Our scheduling codes must be easy to insert into the target source code in the region where the loop parts may be parallelized as much as possible.

4. Experimental environments and results

4.1. Previous experimental results

We use matrix multiplication as example. The matrix operation derives a resultant matrix by multiplying two input matrices, X and Y, where matrix X is a matrix of n rows by p columns and matrix Y is of p rows by m columns. The resultant matrix Z is of n rows by m columns. Serial execution of this operation is quite straightforward, as shown below.

```
for (k=0;k < M;k++)
  for (i=0;i < N;i++) {
    Z[i][k] = 0.0;
    for (j = 0;j < P;j++)
      Z[i][k]+ = X[i][j]*Y[j][k];}
```

This algorithm requires n^3 multiplications and n^3 additions, leading to a sequential time complexity of $O(n^3)$.

The first test environment, described in Table 2, we build two clusters to form a multiple cluster environment. Each cluster has two slave nodes and one master node. Each nodes are interconnected through 3COM 3C9051 10/100 Fast Ethernet Card to Accton CheetahSwitch AC-EX3016B Switch HUB; each master node is running SGE QMaster daemon and SGE execute daemon to running, manage and monitor incoming job and Globus Toolkit v3.0.2. Each slave node is running SGE execute daemon to execute income job only.

The experiment consists of three scenarios: single Personal Computer, Cluster environment and Grid environment. At first step, we run a MPI program on a PC or SMP system to evaluate the system performance. Second step, we connect three Personal Computers together to form a cluster computing environment (our

Table 2
Hardware configuration

Cluster 1			
Hostname	Grid	Grid1*	Grid2
FQDN	grid.hpc.csie.thu.edu.tw	grid1.hpc.csie.thu.edu.tw	grid2.hpc.csie.thu.edu.tw
IP	140.128.101.172	140.128.101.173	140.128.101.174
CPU	Intel Pentium 3 –1 GHz × 2	Intel Celeron 1.7 GHz	Intel Celeron 300 MHz
RAM	512 MB SDRAM	512 MB DDR RAM	256 MB SDRAM
Cluster 2			
Hostname	Grid3*	Grid4	Grid5
FQDN	grid3.hpc.csie.thu.edu.tw	grid4.hpc.csie.thu.edu.tw	grid5.hpc.csie.thu.edu.tw
IP	140.128.102.187	140.128.102.188	140.128.102.189
CPU	Intel Celeron 1.7 GHz	Intel Pentium 3 – 866 MHz × 2	Intel Pentium 3 – 366 MHz
RAM	256 MB DDR RAM	512 MB DDR RAM	256 MB SDRAM

* Stand for Master node of the cluster, the others is slave node.

testbed consists of Cluster 1 and Cluster 2). Then, we rerun the same MPI program to evaluate the system performance. Third step, through the WAN connection, we connect the Cluster 1 and Cluster 2 together to form a Grid computing environment. Finally, we rerun the same MPI program to evaluate the system performance.

For a start, we propose an experiment to compare the performance of cluster computing and Grid computing (see Fig. 4 and Table 3). And then we give the second experimental result on Grid computing environment about the three self-scheduling schemes based on our α-based self-scheduling approach above (see Fig. 5 and Table 4). In this figure, "*N*"FSS means the FSS self-scheduling scheme based on our α-based self-scheduling approach.

4.2. Experimental results for regular and simulated irregular workload

4.2.1. System hardware and software of Grid

We build three clusters to form a multiple cluster environment. Each master node is running SGE Master Daemon and SGE execute daemon to running, manage and monitor incoming job and Globus Toolkit v3.0.2. Each slave node is running SGE execute daemon to execute income job only. Our Grid architecture is implemented on top of Globus Toolkit, name Grid-cluster. The operating system is RedHat Linux release 9. Parallel application we use MPICH-G2 v1.2.5 for message passing. It is built three PC clusters to form a computational Grid environment (Figs. 6–8). Software specification and hardware specification of our testbed environment are described in Tables 5 and 6, respectively.

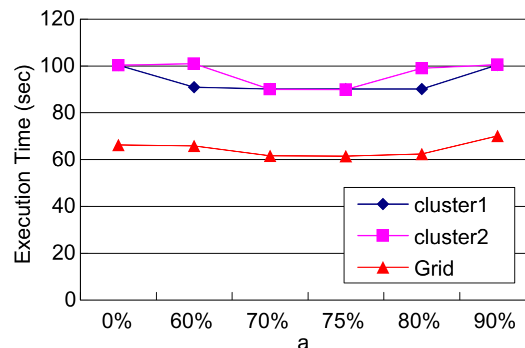

Fig. 4. Execution time of 1024 × 1024 matrix multiplication on different platform.

Table 3
Execution time (sec) of matrix multiplication with matrix size 1024×1024

α-based scheme	0%	60%	70%	75%	80%	90%
Cluster 1	100.312	90.887	90.214	90.213	90.164	100.419
Cluster 2	100.357	100.954	90.092	89.922	99.040	100.630
Grid	66.177	65.912	61.616	61.454	62.433	70.142

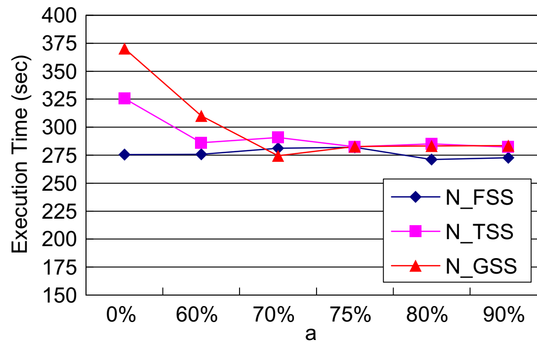

Fig. 5. Execution time of 2048×2048 matrix multiplication on Grid computing environment.

Table 4
Execution time (s) of matrix multiplication with size 2048×2048 on Grid environment

α-based scheme	0%	60%	70%	75%	80%	90%
FSS	275.622	275.898	281.222	282.091	271.182	272.567
TSS	325.498	286.093	290.645	282.401	285.064	282.134
GSS	370.199	310.193	274.505	282.875	283.216	283.598

- Alpha site: Four PCs and each with dual Athlon MP 2000 MHz processor, 512 MB DDRAM and Intel PRO100 VE interface.
- Beta site: Four PCs and each with single Celeron 1700 processor, 256 MB DDRAM, and 3Com 3c9051 interface.
- Gamma site: Four PCs with and each with dual Pentium III 866 MHz processors, 512 MB SDRAM and 3Com 3c9051 interface.

Sites 1–3 are located at different department and lab in Tunghai University, Taiwan. We use general applications to benchmark network traffic from Sites 1–3. Among the group of sites (1, 2, and 3), the average network latency is 3 ms and the maximum transfer speed is 7600 Kbytes.

In the previous work [37], we implemented a computational Grid resource broker which is used to discover and evaluate Grid resources, and make informed job submission decisions by matching requirements of a job with an appropriate Grid resource to meet user and deadline requirements. Users could easily make use of our resource broker through a common Grid portal [37]. The primary task of Resource Broker is to compare requests of users and resource information provided by Information Service. The primary purpose of Information Service is to collect related resource information (processors, memory, disk, and network bandwidth) of all machines in the Grid and provide the analyzed information. The goal of Monitoring Service is to acquire the information maintained by Information Service, and present it in graphical form. When users want an optimal "alpha" value with running the HINT benchmark, they first know which resources are available just before their application run by using this monitoring and information services of Grid environment. There-

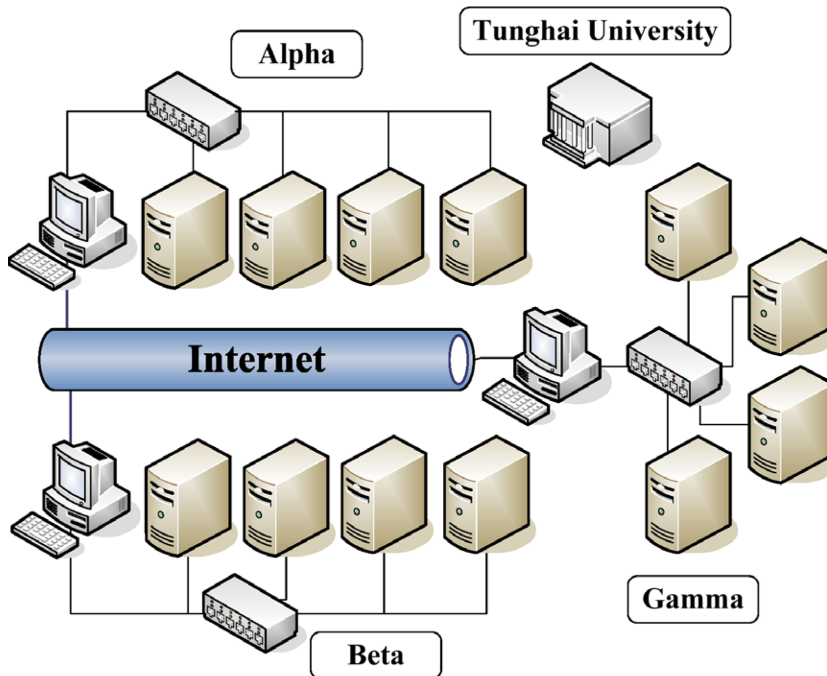

Fig. 6. Our THU Grid testbed consists of three PC clusters.

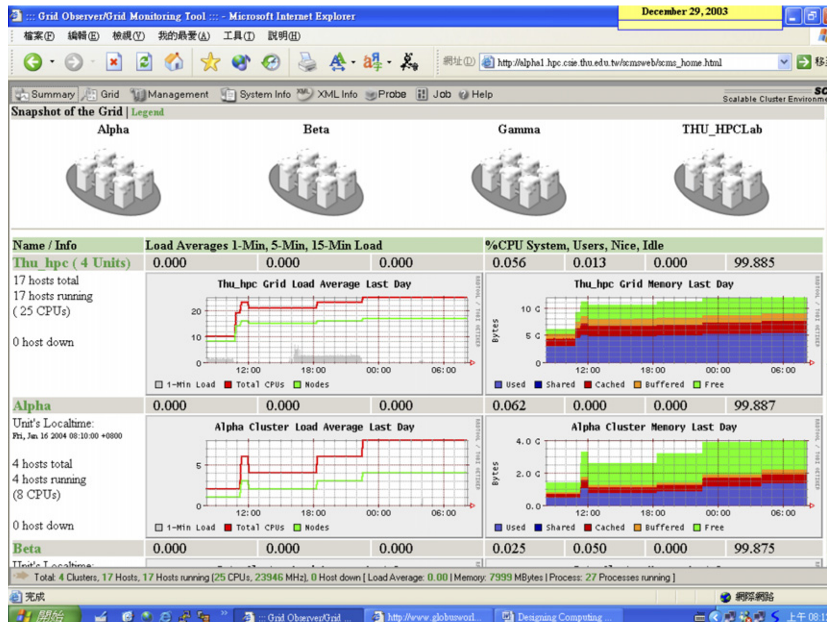

Fig. 7. THU Grid static summary monitoring.

fore, users are easy to specify the alpha value every execution time. This cost may not affect on their total execution time and the experimental results in the paper do not count the cost. The tasks and relations of these components are described as follows. Fig. 9 shows the system stack of Monitoring and Information Services.

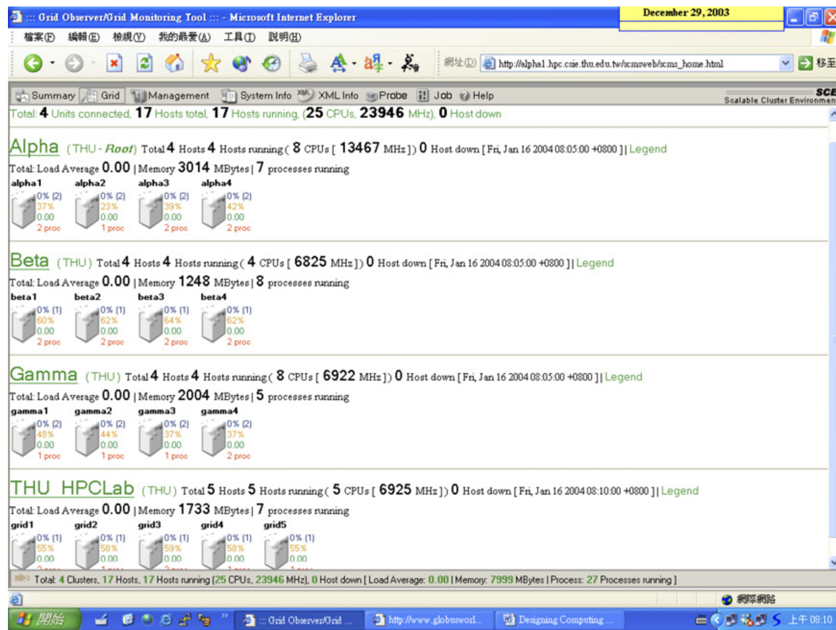

Fig. 8. THU Grid summary view.

Table 5
Software configuration

OS and software configuration
Linux Redhat 9.0
Globus Toolkit 3.0.2
Sun Grid Engine 5.3
MPICH 1.2.5 and MPICH-G2

Table 6
Hardware configuration

Alpha			
Alpha1[a]	Alpha2	Alpha3	Alpha4
alpha1.hpc.csie.thu.edu.tw AMD MP-2000 × 2 512 MB DDR RAM	alpha2.hpc.csie.thu.edu.tw AMD MP-2000 × 2 512 MB DDR RAM	alpha3.hpc.csie.thu.edu.tw AMD MP-2000 × 2 512 MB DDR RAM	alpha4.hpc.csie.thu.edu.tw AMD MP-2000 × 2 512 MB DDR RAM
Gamma			
Gamma1[a]	Gamma2	Gamma3	Gamma4
gamma1.hpc.csie.thu.edu.tw Intel Pentium 3 – 866 MHz × 2 512 MB SDRAM	gamma2.hpc.csie.thu.edu.tw Intel Pentium 3 – 866 MHz × 2 512 MB SDRAM	gamma3.hpc.csie.thu.edu.tw Intel Pentium 3 – 866 MHz × 2 512 MB SDRAM	gamma4.hpc.csie.thu.edu.tw Intel Pentium 3 – 866 MHz × 2 512 MB SDRAM
Beta			
Beta1[a]	Beta2	Beta3	Beta4
beta1.hpc.csie.thu.edu.tw Intel Celeron 1.7 GHz 256 MB DDR RAM	beta2.hpc.csie.thu.edu.tw Intel Celeron 1.7 GHz 256 MB DDR RAM	beta3.hpc.csie.thu.edu.tw Intel Celeron 1.7 GHz 256 MB DDR RAM	beta4.hpc.csie.thu.edu.tw Intel Celeron 1.7 GHz 256 MB DDR RAM

[a] Stand for Master node of the cluster.

We integrate four important components into this system: (1) JRobin [38], (2) Tomcat server [39], (3) Ganglia [40], and (4) Network Weather Service [41]. Also, we build friendly GUI interface for inexperienced users, no matter if they know what computational Grid is.

4.2.2. Regular workload/uniform workload

The experiment in this section consists of three different scenarios: (1) Differences performance presentation of scheduling schemes in uniform workload. (2) Different gird environment and (3) matrix multiplication with different matrix sizes. At first step, we run a MPI program on different Grid system to evaluate the system performance. Second step, we connect these Grid systems together to form a Grid environment (in our testbed is Grid Alpha, Beta and Gamma). Then, running the same MPI program to evaluate the system performance. Third step, through the different system topologies, we connect the system characteristics together for a performance analysis. Finally, we run the same MPI program to evaluate the system performance of different system architectures.

Figs. 6–8 note that our approach connects these Grid systems together to form a Grid environment (In our testbed is Grid Alpha, Beta and Gamma) Then, running the same MPI program to evaluate the system performance and implements FSS, GSS, and TSS group approach. Our new scheme can guarantee whether what kind of parallel loop scheduling situation happen, they can be properly well-arranged in our approach and achieved better performance than other scheme developed before, all of the performance analysis are presented in Figs. 10–12, respectively. In previous methods, N_FSS, N_TSS, and N_GSS get worse performance than new scheme with dynamic parameterization and systematic adjustment automatically.

4.2.3. Irregular workload

The experiment in this subsection consists of three scenarios: differences performance presentation of scheduling schemes in (1) Increasing workload. (2) Decreasing workload and (3) Random workload. Figs. 13–15 show the performance in different combinations of Grid system, called α, β, and γ, implementing the irregular workload situations with FSS, GSS, and TSS group approach comparison. Finally, we examine the performance of different Grid system (in our testbed is Grid Alpha, Beta and Gamma), the result is shown as Fig. 16. The results of these figures show that, by gradually increasing processors, the time consumed drops significantly.

Therefore, using our new approach in 1024×1024 matrix multiplication of simulated increasing workload loop will reduce 11.3%, 19.9% and 30.9% execution time less than N_FSS, N_GSS and N_TSS, respectively; and reduce 29.2%, and 18.6% execution time less than N_FSS and N_TSS respectively in 1024×1024 matrix multiplication of simulated decreasing workload loop. In 1024×1024 matrix multiplication of simulated random workload loop, our approach will reduce 18.8% and 16.6% execution time less than N_FSS and N_TSS, respectively.

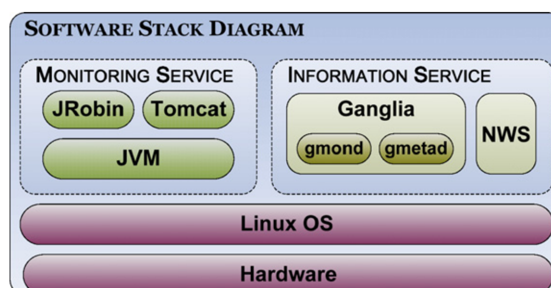

Fig. 9. Software stack of Monitoring and Information Services.

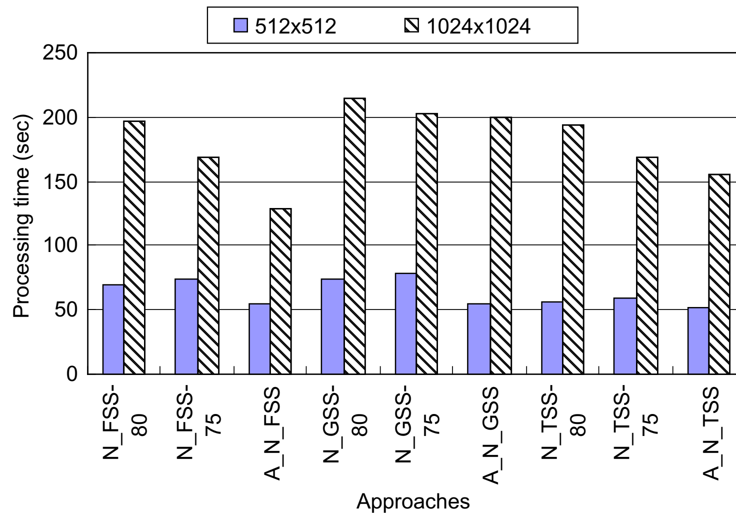

Fig. 10. Execution time chart of different sizes of matrix multiplication using Grid $\alpha + \beta + \gamma$.

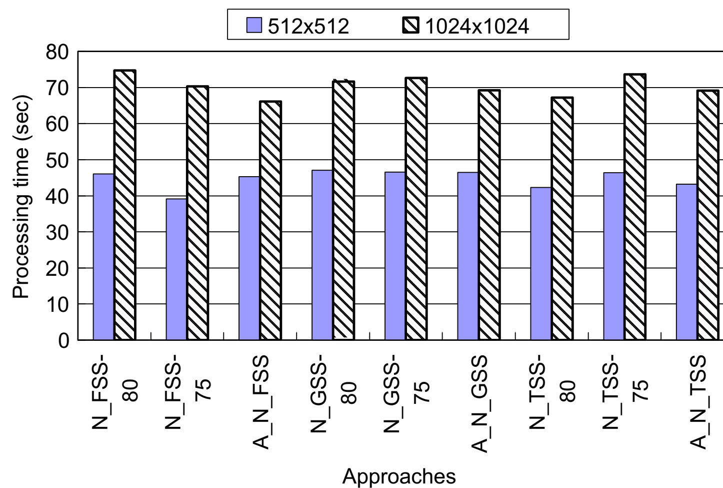

Fig. 11. Execution time chart of different sizes of matrix multiplication using Grid $\beta + \gamma$.

4.3. Experimental results for real irregular workload: the Mandelbrot set

The Mandelbrot set is a problem involving the same computation on different data points which have different convergence rates. The Mandelbrot set, named after 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. Therefore, it is an appropriate example for applications with irregular workload.

We have configured two scenarios from this testbed to show the performance of our approach. The experimental results indicate that our approach can automatically adjust the alpha parameter according to the heterogeneity of the Grid configuration. In this experiment, the Mandelbrot set is computed on $[0.5, -1.8]$ to $[1.2, -1.2]$ using a 800×800 pixel window size.

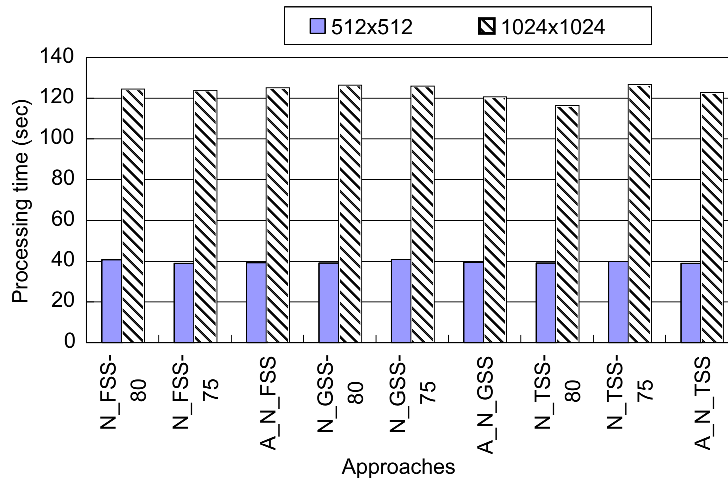

Fig. 12. Execution time chart of different sizes of matrix multiplication using cluster γ.

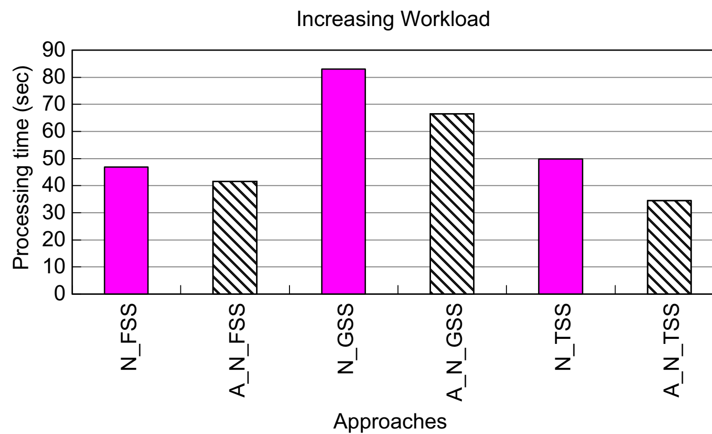

Fig. 13. Execution time chart simulated increasing workload loop by various self-scheduling approaches Grid $\alpha + \beta + \gamma$.

4.3.1. System configuration of Grid

To conduct the experiments in this section, we have integrated six cluster sites to form a Grid testbed. Our Grid architecture is implemented on top of Globus Toolkit, name Grid-cluster. The operating system is Red-Hat Linux release 9. In parallel applications, we use MPICH-G2 v1.2.5 for message passing.

- Alpha site: Four PCs and each with dual Athlon MP 2000 MHz processor, 512 MB DDRAM and Intel PRO100 VE interface.
- Gamma site: Four PCs with and each with dual Pentium III 866 MHz processors, 256 MB SDRAM and 3Com 3c9051 interface.
- PU site: Four PCs and each with single Athlon XP 2400+ processor, 1 GB DDRAM and Intel PRO100 VE interface.
- HIT site: Four PCs and each with single Intel Pentium 4 2.80 GHz processor with HT (Hyper Threading), 512 MB DDRAM and Intel PRO100 VE interface.
- LZ site: Four PCs and each with single Intel Pentium III 866 MHz processor, 256 MB SDRAM and Intel PRO100 VE interface.

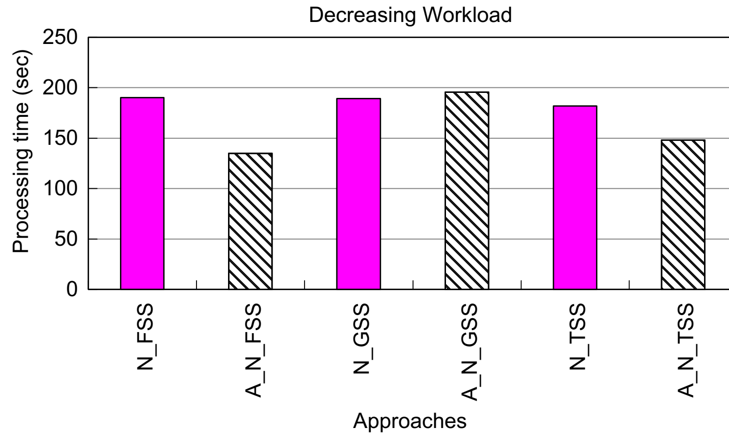

Fig. 14. Execution time chart simulated decreasing workload loop by various self-scheduling approaches Grid $\alpha + \beta + \gamma$.

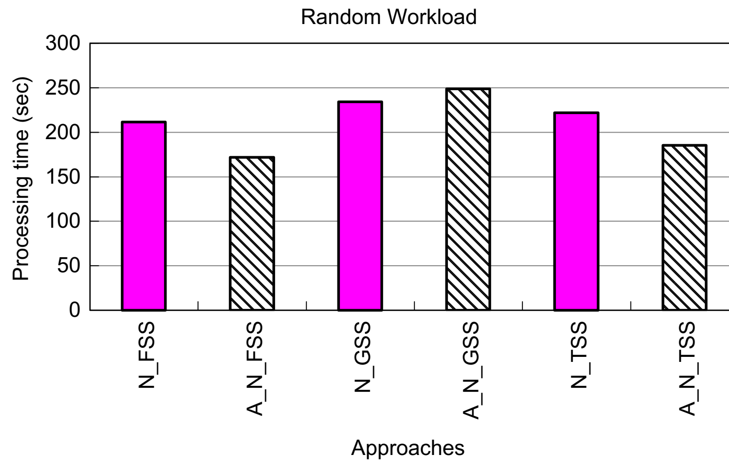

Fig. 15. Execution time chart simulated random workload loop by various self-scheduling approaches Grid $\alpha + \beta + \gamma$.

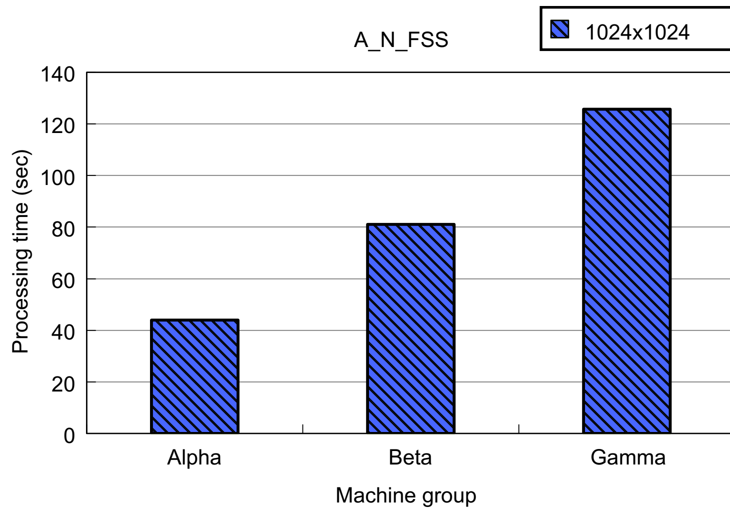

Fig. 16. Execution time chart of A_N_FSS self-scheduling approach with matrix size 1024×1024 in Grid α, β, and γ, respectively.

- TC site: Two PCs and each with single Intel Pentium 4 1.80 GHz processor, 128 MB DDRAM and Intel PRO100 VE interface.

The Alpha site and the Gamma site are located at different departments in Tunghai University, Taiwan. The other sites are located in Providence University, Hsiuping Institute of Technology, Lizen High School and National Dali Senior High School, respectively.

4.3.2. Scenario one: a configuration consisting of six sites

In this scenario, we use 21 nodes of the six sites to run the Mandelbrot set computation. First, we execute previous self-scheduling schemes with different alpha parameters, from 0 to 100, to find the possibly optimal alpha value for previous self-scheduling schemes. As Fig. 17 shows, the three schemes get better performance when the alpha parameter is set to the range of [40, 50].

In our scheme, the alpha value is computed automatically according to the heterogeneity of computing environment. In this scenario, our scheme automatically sets the alpha parameter to 31, and this setting produces rather good performance compared with previous schemes. As shown in Fig. 18, when inappropriate alpha values are adopted by previous schemes, the performance might degrade dramatically. The top value of the white part means the max processing time with an inappropriate alpha setting, and the top value of the dark part means the max processing time of the proposed approach.

4.3.3. Scenario two: a configuration consisting of three sites

In this scenario, we use 12 nodes of the first three sites to run the Mandelbrot set computation. Also, we execute previous self-scheduling schemes with different alpha parameters, from 0 to 100, to find the possibly optimal alpha value for previous schemes. As Fig. 19 shows, the three schemes get better performance when the alpha parameter is less than 60.

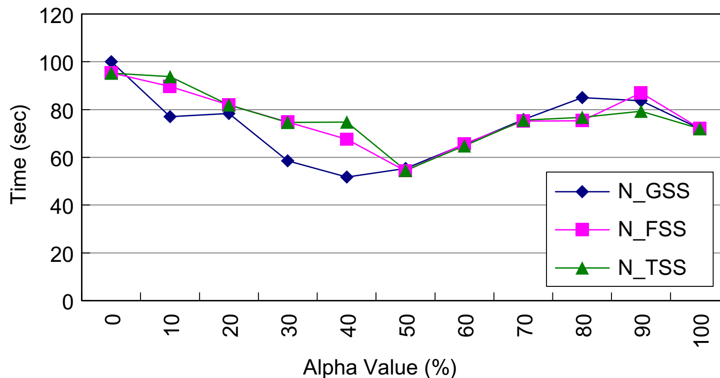

Fig. 17. Execution time chart of previous self-scheduling approach for scenario one.

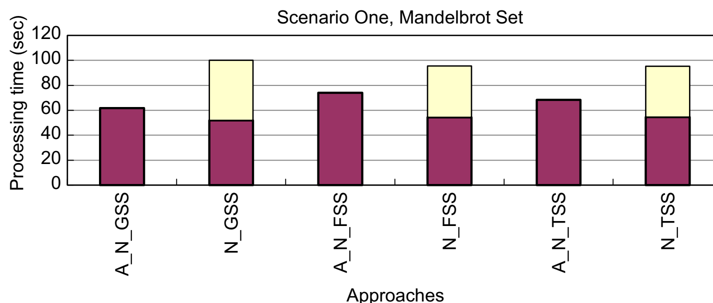

Fig. 18. Execution time chart of our and previous self-scheduling approaches for scenario one.

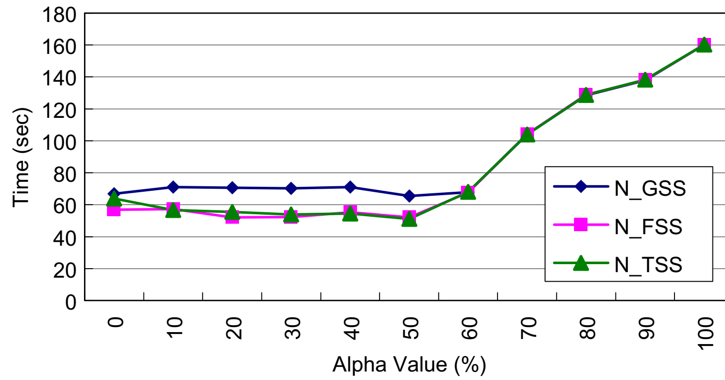

Fig. 19. Execution time chart of previous self-scheduling approach for scenario two.

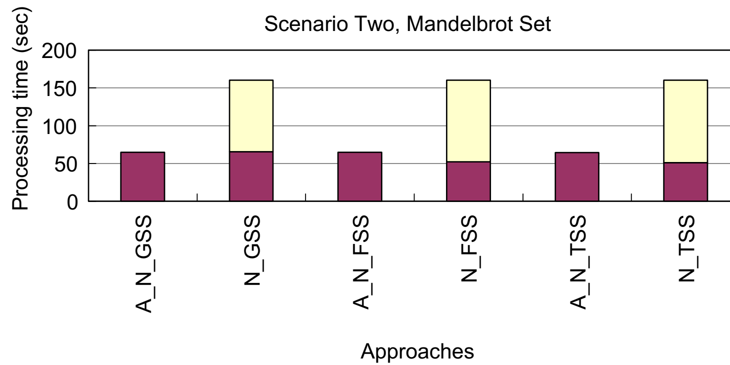

Fig. 20. Execution time chart of our and previous self-scheduling approaches for scenario two.

In our scheme, the alpha value is computed automatically according to the heterogeneity of computing environment. In this scenario, our scheme automatically sets the alpha parameter to 59, and this setting produces rather good performance compared with previous schemes. As shown in Fig. 20, when inappropriate alpha values are adopted by previous schemes, the performance might degrade seriously. The experimental results indicate that the proposed approach can automatically adjust the alpha parameter according to the heterogeneity of the Grid configuration. However, the performance of previous schemes depends on a good choice of the alpha parameter. In our other paper [36], experimental results show that the proposed approach is more efficient than previous schemes. In addition, as the number of nodes increases, the proposed approach is still effective.

5. Conclusions

In this paper, we presented the design and implementation of a complete parallel loop self-scheduling software system. The system, which covers the four types of parallel loop shown as Fig. 1, can appropriately arrange scheduling and achieve better performance than previously developed schemes. We revised known loop self-scheduling schemes to fit in both PC-based cluster environments and Grid computing environments whether loop types are regular or irregular.

After enough feedback information has been discovered, collected, and analyzed, performance will improve each time feedback information is collected. To date, we have combined the HINT Performance Analyzer Tool, our new α self-scheduling scheme, and dynamic adjustment of scheduling parameters in a complete newly developed approach. The goal of achieving good performance on parallel loop self-scheduling using

our approach is convenience and practicality. Our future work will focus on finding an appropriate method for investigating performance trends after new computing nodes have been added, and an improved way to adjust the value of α.

References

[1] B. Allcock, S. Tuecke, I. Foster, A. Chervenak, C. Kesselman, Protocols and services for distributed data-intensive science. In: ACAT2000 Proceedings, 2000, pp. 161–163.
[2] R. Buyya, High Performance Cluster Computing:System and Architectures, vol. 1, Prentice Hall PTR, New Jersey, 1999.
[3] Mark A. Baker, Geoffery C. Fox, Metacomputing: Harnessing Informal Supercomputers, High Performance Cluster Computing, Prentice-Hall, 1999, ISBN 0-13-013784-7.
[4] Christopher A. Bohn, Gary B. Lamont, Load balancing for heterogeneous clusters of PCs, Future Generation Computer Systems 18 (2002) 389–400.
[5] A.T. Chronopoulos, R. Andonie, M. Benche, D.Grosu, A class of loop self-scheduling for heterogeneous clusters, in: Proceedings of the 2001 IEEE International Conference on Cluster Computing, pp. 282–291.
[6] A.T. Chronopoulos, R. Andonie, M. Benche, and D. Grosu, A class of loop self-scheduling for heterogeneous clusters, in: Proceedings of 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001), 2001, pp. 282–291.
[7] I. Foster, N. Karonis, A grid-enabled MPI: message passing in heterogeneous distributed computing systems, in: Proceedings of the 1998 SC Conference, November, 1998.
[8] I. Foster, C. Kesselman, S. Tuecke, The anatomy of the grid: enabling scalable virtual organizations, International Journal of Supercomputer Applications 15 (3) (2001).
[9] I. Foster, C. Kesselman, Globus: a metacomputing infrastructure toolkit, International Journal of Supercomputer Applications 11 (2) (1997) 115–128.
[10] I. Foster, The grid: a new infrastructure for 21st century science, Physics Today 55 (2) (2002) 42–47.
[11] I. Foster, C. Kesselman (Eds.), The Grid: Blueprint for a New Computing Infrastructure, first ed., Morgan Kaufmann, 1999, January.
[12] Y.W. Fann, C.T. Yang, S.S. Tseng, C.J. Tsai, An intelligent parallel loop scheduling for multiprocessor systems, Journal of Information Science and Engineering 16 (2000) 169–200, Special Issue on Parallel and Distributed Computing.
[13] A.S. Grimshaw, Meta-systems: an approach combining parallel processing and heterogeneous distributed computing systems, in: Workshop on Heterogeneous Processing, International Parallel Processing Symposium, 1992, pp. 54–59.
[14] Global Grid Forum. <http://www.ggf.org/>.
[15] HINT performance analyzer. <http://hint.byu.edu/>.
[16] S.F. Hummel, E. Schonberg, L.E. Flynn, Factoring: a method scheme for scheduling parallel loops, Communications of the ACM 35 (1992) 90–101.
[17] Introduction to Grid Computing with Globus, 2002. <http://www.ibm.com/redbooks>.
[18] KISTI Grid Testbed. <http://Gridtest.hpcnet.ne.kr/>.
[19] LHC – The Large Hadron Collider Home Page. <http://lhc-new-homepage.web.cern.ch/>.
[20] H. Li, S. Tandri, M. Stumm, K.C. Sevcik, Locality and loop scheduling on NUMA multiprocessors, in: Proceedings of the 1993 International Conference on Parallel Processing, vol. II, 1993, pp. 140–147.
[21] MPICH-G2. <http://www.hpclab.niu.edu/mpi/>.
[22] MPICH. <http://www-unix.mcs.anl.gov/mpi/mpich/>.
[23] E. Post, H.A. Goosen, Evaluating the parallel performance of a heterogeneous system, in: Proceedings of 5th International Conference and Exhibition on High-Performance Computing in the Asia-Pacific Region (HPC Asia 2001). <http://parallel.hpc.unsw.edu.au/HPCAsia/papers/12.pdf>.
[24] C.D. Polychronopoulos, D. Kuck, Guided self-scheduling: a practical scheduling scheme for parallel supercomputers, IEEE Transactions on Computers 36 (1987) 1425–1439.
[25] V.S. Sunderam, PVM: a framework for parallel distributed computing, Concurrency: Practice and Experience 2 (4) (1990) 315–339, December.
[26] Sun ONE Grid Engine. <http://wwws.sun.com/software/Gridware/>.
[27] TeraGrid. <http://www.teraGrid.org/>.
[28] T.H. Tzen, L.M. Ni, Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers, IEEE Transactions on Parallel and Distributed Systems 4 (1) (1993) 87–98, January.
[29] P. Tang, P.C. Yew, Processor self-scheduling for multiple-nested parallel loops, in: Proceedings of the 1986 International Conference on Parallel Processing, 1986, pp. 528–535.
[30] The Globus Project. <http://www.globus.org/>.
[31] T.H. Tzen, L.M. Ni, Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers, IEEE Transactions on Parallel and Distributed Systems 4 (1993) 87–98.
[32] Chao-Tung Yang, Shun-Chyi Chang, A parallel loop self-scheduling on extremely heterogeneous PC clusters, Journal of Information Science and Engineering 20 (2) (2004) 263–273, Institute of Information Science, Academia Sinica, Nankang, Taipei, 115 Taiwan.

[33] Chao-Tung Yang, Kuan-Wei Cheng, Kuan-Ching Li, An efficient parallel loop self-scheduling on grid environments, in: Hai Jin, Guangrong Gao, Zhiwei Xu (Eds.), Network and Parallel Computing: IFIP International Conference, NPC 2004, Lecture Notes in Computer Science, vol. 3222, Springer-Verlag, 2004, pp. 92–100, October.

[34] Chao-Tung Yang, Kuan-Wei Cheng, Kuan-Ching Li, An enhanced parallel loop self-scheduling scheme for cluster environments, The Journal of Supercomputing 34 (1) (2005) 1–21, September.

[35] Chao-Tung Yang, Kuan-Wei Cheng, Kuan-Ching Li, An enhanced parallel loop self-scheduling scheme for heterogeneous cluster environments, in: Proceedings of the International Conference on Advanced Information Networking and Applications (AINA 2005) INA'2005 The First International Workshop on Information Networking and Application, vol. 2, Tamkang University, Taipei, Taiwan, March 28–30, 2005, pp. 207–210.

[36] W.C. Shih, C.T. Yang, S.S. Tseng, 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-Verlag, 2005, pp. 48–55, November.

[37] Chao-Tung Yang, Cheng-Fang Lin, Sung-Yi Chen, A workflow-based computational resource broker with information monitoring in grids, in: Proceedings of the 5th International Conference on Grid and Cooperative Computing (GCC 2006), IEEE CS Press, China, 2006, October.

[38] JRobin. <http://www.jrobin.org/>.

[39] Tomcat. <http://tomcat.apache.org/>.

[40] Ganglia. <http://ganglia.sourceforge.net/>.

[41] Network Weather Service. <http://nws.cs.ucsb.edu/ewiki/>.