J Supercomput (2012) 59:414–442
DOI 10.1007/s11227-010-0443-x

Performance-based dynamic loop scheduling in heterogeneous computing environments

Chao-Tung Yang · Wen-Chung Shih · Lung-Hsing Cheng

Published online: 26 May 2010

Abstract With the rapid advance of computing technologies, it becomes more and more common to construct high-performance computing environments with heterogeneous commodity computers. Previous loop scheduling schemes were not designed for this kind of environments. Therefore, better loop scheduling schemes are needed to further increase the performance of the emerging heterogeneous PC cluster environments. In this paper, we propose a new heuristic for the performance-based approach to partition loop iterations according to the performance weighting of cluster/grid nodes. In particular, a new parameter is proposed to consider HPCC benchmark results as part of performance estimation. A heterogeneous cluster and grid were built to verify the proposed approach, and three kinds of application program were implemented for execution on cluster testbed. Experimental results show that the proposed approach performs better than the previous schemes on heterogeneous computing environments.

Keywords Parallel loop · Self-scheduling · Performance · Heterogeneous · MPI

1 Introduction

Cluster computing systems usually connect several commodity computers in local-area networks to form a single, unified resource for parallel computing [1, 18, 20].

C.-T. Yang (✉) · L.-H. Cheng
High-Performance Computing Laboratory, Department of Computer Science, Tunghai University, Taichung 40704, Taiwan, R.O.C.
e-mail: ctyang@thu.edu.tw

W.-C. Shih
Department of Information Science and Applications, Asia University, Taichung 41354, Taiwan, R.O.C.
e-mail: wjshih@asia.edu.tw

As more and more inexpensive personal computers (PC) become available, clusters of PCs are becoming alternatives to the supercomputers many research projects cannot afford. As computer architectures become more and more diverse and heterogenic, and computer expiration rates are higher than before, we can put old and unused computers to efficient use in our research. Therefore, it is natural because of the fast development of information technology that clusters consist of computers with various processors, memories and hard disk drives. However, it is difficult to deal with such heterogeneity in a cluster [2, 7, 13, 14, 16, 19, 22–24].

Loop scheduling and load balancing on parallel and distributed systems are critical problems that are difficult to cope with, especially on the emerging PC-based clusters [3, 4]. In this aspect, an important issue is how to assign tasks to nodes so that the nodes' loads are well balanced. Conventional self-scheduling loop approaches [11] include static scheduling and dynamic scheduling. However, the former considers computing nodes as homogeneous resources, and thus not suitable for heterogeneous environments, and the latter, especially self-scheduling, can still be improved [6, 22, 24–26].

Previous researchers proposed some useful self-scheduling schemes applicable to PC-based clusters [6, 25, 26] and grid computing environments [22, 24]. These schemes consist of two phases. In the first phase, system configuration information is collected, and some portion of the workload is distributed among slave nodes according to their CPU clock speeds [22] or HINT measurements [6, 24, 25]. After that, the remaining workload is scheduled using some well-known self-scheduling scheme, such as GSS [11]. The performance of this approach depends on an appropriate choice of scheduling parameters since it estimates node performance using only CPU speed or HINT benchmark, which are factors affecting node performance. In [6], an enhanced scheme, which dynamically adjusts scheduling parameters according to system heterogeneity, is proposed.

Intuitively, we may want to partition loop iterations according to CPU clock speed. However, the CPU clock is not the only factor which affects the node performance. Many other factors also have dramatic influences, such as the amount of memory available, the cost of memory accesses, and the communication medium between nodes. Using the intuitive approach will result in degraded performance if the performance estimation is not accurate.

In this paper, we propose a general approach that utilizes performance functions to estimate performance weights for each node. To verify the proposed approach, a heterogeneous cluster was built, and three types of application program, matrix multiplication, circuit satisfiability, and Mandelbrot set computation [9], were implemented for execution on this testbed. Empirical results show that for heterogeneous cluster environments, the proposed approach obtained improved performance compared to the previous schemes.

Previous work in [6, 24–26] and this paper were all inspired by [22], the α self-scheduling scheme. However, this work has a different perspective and a unique contribution. First, while [6, 24] partition $\alpha\%$ of workloads according to CPU clock speed performance weighting in phase one, the proposed scheme partitions according to a general performance function (PF). In this paper, we do not define an explicit performance function. Instead, CPU clock and HPC Challenge (HPCC) [5] performance measurement are used to estimate the value of performance weighting (PW)

for all nodes. The PW obtained by HPCC measurement can be used to estimate cluster node performance rather accurately. PW calculation is presented below.

Second, the scheme in [22] utilizes a fixed α value, and [6, 24] adaptively adjust the α value according to the heterogeneity of the cluster. In other words, both schemes depend on a properly chosen α value for good performance. However, the proposed scheme focuses on accurate estimation of node performance, so the choice of α value is not very critical. Thus, we can roughly choose an α value from a larger range than the previous schemes can. Third, in our implementation, the master node participates in computation, whereas in previous schemes, only slave nodes do computation work.

The rest of this paper is organized as follows. In Sect. 2, we introduce several typical and well-known self-scheduling schemes, and a famous benchmark used to analyze computer system performance. In Sect. 3, we define our model and describe our approach. Next, our system configuration is specified and experimental results on three types of application program are presented in Sect. 4. Concluding remarks and future work are given in Sect. 5.

2 Background

In this section we first review previous loop scheduling schemes, then introduce the HPCC performance analyzer.

2.1 Review of loop self-scheduling schemes

Traditional static loop scheduling schemes make scheduling decisions at compiling time, and assign equal workloads to processors. This is applied when all iterations take roughly the same amount of time, and the compiler knows sufficient relevant information before compilation. Its advantage is in less scheduling overhead, while a possible disadvantage lies in load-imbalance. Well-known static scheduling schemes include Block Scheduling, Cyclic Scheduling, Block-D Scheduling, Cyclic-D Scheduling [10], etc. However, these schemes are not suitable for heterogeneous clusters.

In [22], a heuristic was proposed to distribute workloads according to CPU performance. In heterogeneous clusters, it is difficult to estimate node performance. In [12], it is indicated that many attributes influence the system performance, including CPU clock speed, available memory, communication cost, etc. In [24], the authors tried to evaluate computer performance using the HINT benchmark. However, HINT requires hours to execute, so it is not suitable for frequent use.

In contrast, dynamic scheduling is more suitable for load balancing. However, the runtime overhead must be taken into consideration. The schemes we focus on in this paper are self-scheduling, members of a large class of adaptive and dynamic centralized loop scheduling schemes. In a common self-scheduling scheme, p denotes the number of processors, N the total iterations and $\boldsymbol{f}()$ is a function for producing the chunk-size at each step. The design of function f depends on the scheduling strategy of the scheme, and its output is the chunk-size for the next iteration. For example, in GSS [10], $\boldsymbol{f}$ is defined as the number of iterations remaining in a parallel loop

Table 1 Partition size examples

Scheme	Partition size
PSS	1, 1, 1, 1, 1, 1, 1, 1, 1, ..., 1
CSS(128)	128, 128, 128, 128, 128, 128, 128, 128
GSS [7]	256, 192, 144, 108, 81, 61, 46, 34, 26, ...
FSS [10]	128, 128, 128, 128, 64, 64, 64, 64, 32, ...
TSS [17]	128, 120, 112, 104, 96, 88, 80, 72, 64, ...

divided by the number of available processors. At the ith scheduling step, the master computes the chunk-size C_i and the remaining number of tasks R_i:

$$R_O = N, \qquad C_i = f(i, p), \qquad R_i = R_{i-1} - C_i \tag{1}$$

where $f()$ may have more parameters than just i and p, such as R_{i-1}. The master assigns C_i tasks to an idle slave and the load-imbalance will depend on the execution time gap between the nodes [7, 10]. Different ways of computing C_i have given rise to various scheduling schemes. The most notable examples are Pure Self-Scheduling (PSS), Chunk Self-Scheduling (CSS), Factoring Self-Scheduling (FSS), Guided Self-Scheduling (GSS), and Trapezoid Self-Scheduling (TSS) [8, 11, 21]. Table 1 shows the various chunk sizes for a problem with iterations numbering $N = 1024$ and processors numbering $p = 4$.

Pure Self-Scheduling (PSS) was the first straightforward dynamic loop scheduling algorithm. In this paper, a processor is said to be idle if it has not been assigned a chunk of workload or it has finished its assigned workload. Whenever a processor falls idle, iterations are assigned to it. This algorithm achieves good load balancing but induces excessive overhead [10].

Chunk Self-Scheduling (CSS) assigns k iterations each time, where k, the chunk-size, is fixed and must be specified by either the programmer or the compiler. When k is 1, the scheme is purely self-scheduling, as discussed above. Large chunk sizes cause load-imbalance, while small chunk sizes are likely to produce excessive scheduling overhead [10].

Guided Self-Scheduling (GSS) can dynamically change the number of iterations assigned to idle processors [11]. More specifically, the next chunk-size is determined by dividing the number of iterations remaining in 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 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) assigns loop iterations to working processors in phases [8]. During each phase, only a subset of remaining loop iterations (usually half) is divided equally 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) tries to reduce the need for synchronization while still maintaining reasonable load balances [21]. 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 [21] proposed TSS($N/2p$, 1) as a general selection. In this case, the first chunk is of size $N/2p$, and successive 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.

In [22], the authors revise known loop self-scheduling schemes to fit all heterogeneous PC clusters environment when loops are regular. An approach is proposed for partitioning loop iterations in two phases and it achieves good performance in any heterogeneous environment: partition $\alpha\%$ of the workload according to CPU clock performance weighting in the first phase and the remainder $(100-\alpha)\%$ of the workload according to known self-scheduling in the second phase. The experimental results are from a cluster environment with six nodes in which the fastest computer was 6 times faster than the slowest ones in CPU-clock cycles. Various α values were applied to matrix multiplication with the best performance obtained at $\alpha = 75$.

2.2 Grid computing and its middleware

2.2.1 Globus toolkits

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 scheme 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 [2], organized by over 40 organizations. Participants include vendors, researchers, academics, software library developers and users. MPI offers portability, standardization, performance and functionality [2].

MPICH-G2 [1, 2] 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 [1].

2.3 HPC challenge benchmark

The HPC Challenge (HPCC) is a useful computer benchmarking tool [5]. It first examines the performance of HPC architectures using kernels with more challenging memory access patterns than High Performance Linpack (HPL). It also augments the TOP500 list. It provides benchmarks that bound the performance of many real applications as a function of memory access characteristics—e.g. spatial and temporal locality. Unlike conventional benchmarks, the HPCC benchmark consists of 7 basic tests, consisting of HPL, DGEMM, STREAM, PTRANS, Random Access, FFT, and Communication bandwidth and latency. In our work, we use the HPL measurement as a performance value and include it in our scheduling algorithm.

3 Proposed approach

Cluster computers have different performance scales in heterogeneous environments. In such situations, additional slave computers may not perform well because known self-scheduling schemes partition workloads according to formulas rather than computer performance. The aim of this work is to promote the performance of loop scheduling for the emerging heterogeneous computing environments. By considering a new parameter, β (proportion of HPCC benchmark results in performance estimation), performance of loop scheduling can benefit from more balanced load distribution. In FSS, for example, every slave gets a workload of size $N/2P$, where N is the total workload and P is the number of processors. If the performance difference between the fastest and the slowest computer is larger than $N/2P$, a load-imbalance occurs. In this section, we first introduce the system model, then describe the performance weighting parameters and static–workload ratio, and finally, present the skeleton algorithm for performance-based loop scheduling.

3.1 Performance function

In this context, we propose to partition $\alpha\%$ of the workload according to CPU clock speed performance weighting and HPCC [5] measurement of all nodes, and dispatch the remaining workload via some well-known self-scheduling scheme, such as GSS [11]. To use this approach, we need to know the real computer performance on the HPCC benchmark. We can then distribute appropriate workloads to each node and achieve load balancing. The more accurate the estimation is, the better the load balance will be.

To estimate node performance, we define a performance weighting (PW) for node j as

$$PW_f(V_1, V_2, \ldots, V_M) \tag{2}$$

where V_i, $1 < i < M$, is a variable of the performance weighting. In this paper, our PW for node j is defined as

$$PW_f = \beta \frac{CS_f}{\sum_{\forall node_i \in S} CS_i} + (1-\beta)\frac{HPL_f}{\Sigma_{\forall node_i \in S} HPL_i}, \quad 0 \le \beta \le 1 \tag{3}$$

where S is the set of all cluster nodes, CS_i is the CPU clock speed of node i, and is a constant attribute. HPL_i is the HPL measurement of HPCC, this value is analyzed above; β is the ratio between the two values.

Assume there are 24 loop iterations to be scheduled, and the value of parameter α is 50%, and the PW values of the three nodes are 3, 2, and 1. In other words, the scheduler will assign 6 iterations to the first node, 4 iterations to the second, and 2 to the third.

3.2 Algorithm

With this approach, the computing node with better performance gets more workload. Note that parameter α should not be too large or too small. If it is too large, the dominant computer will not finish its work, and if it is too small, the dynamic scheduling overhead will be significant. In any case, good performance cannot be attained without an appropriate α value.

We propose an algorithm for performance-based loop scheduling in heterogeneous cluster environments based on workload distribution and node performance information. This algorithm employs a message-passing paradigm, and consists of two modules: a master module and a slave module. The master module makes scheduling decisions and dispatches workloads to slaves, which then process the assigned work. This algorithm is just a skeleton, and detailed implementations, such as data preparation, parameter passing, etc., might differ according to the requirements of the various applications.

The algorithm consists of several steps. First, relevant information is acquired. Then, the Performance Weighting is calculated. Next, α percent of the total workload is statically scheduled according to the performance ratio among all slave nodes. Finally, the remainder of the workload is dynamically scheduled by some well-known self-scheduling scheme for load balancing. The algorithm is described below.

Algorithms MASTER and SLAVE in pseudo code:
Module MASTER /* scheduler */

```
/* perform task scheduling, load balancing and some computation */
[Initialization]
/* Stage 1: Gathering the information. */
Collect CPU clock speed and HPL measurements
/* Stage 2: Calculate the performance weighted */
Calculate PW_j by formula (3)
/* Stage 3: Static Scheduling */
r=0;
  for (i = 1; i < member_of_nodes; i+
  {
partition α% of loop iterations according to the performance
weighted;
send data to all nodes;
    r+
  }
Master does its own computation work
/* Stage 4: Dynamic Scheduling */
Partition (100−α) of loop iterations into the task queue using
some well-known self-scheduling scheme
/* Stage 5: Probe for returned results */
Do{
  Distinguish source and receive returned data
  If the task queue is not empty then
    Send another data to the idle slave
    r−
  else
    send TAG = to the idle slave
  }while (r >
[Finalization]
END MASTER
```

Module SLAVE /* worker */

```
[Initialization]
Probe if some data in
While (TAG >
  {
  Receive initial solution and size of subtask work and
  compute fine solution
  Send the result to the master
  Probe if some data in
  }
[Finalization]
END SLAVE
```

3.3 System model

The HPCC benchmark assisted us in comparatively analyzing all cluster nodes. We must have a proper response and an appropriate self-scheduling scheme for changeable system architectures and loop styles.

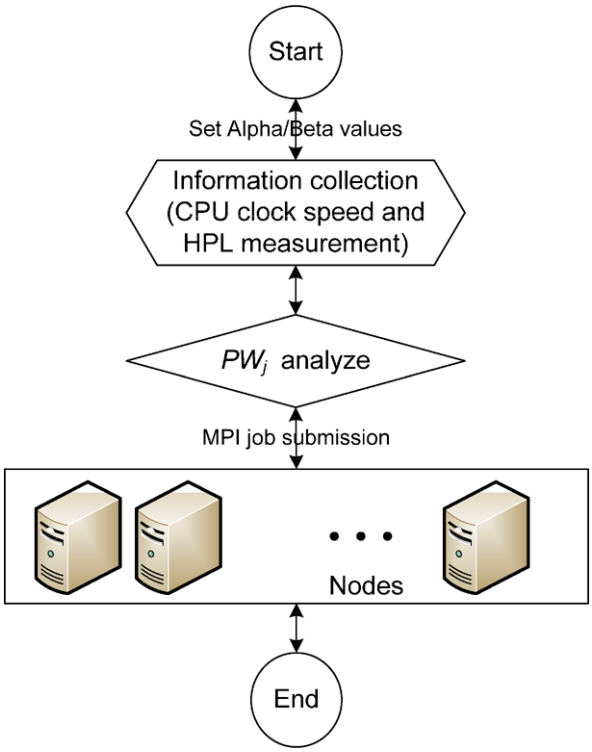


Fig. 1 The self-scheduling flow chart

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) and the Meta-System Approach. 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 regions where loop parts may be parallelized as much as possible. An example of how our new self-scheduling scheme works is illustrated in Fig. 1.

4 Experiments and results

A heterogeneous PC-based cluster was built to verify our approach, and three types of application program were implemented for execution on this testbed with MPI. We first illustrate our cluster environment and describe the terminology for our programs. The performance of our scheme is then compared with that of other static and dynamic schemes on the heterogeneous cluster, using matrix multiplication and Mandelbrot sets.

Table 2 Our implementation of all matrix multiplication programs

Master compute	Name	Description	Reference #
Y	matstat	Static scheduling	[10]
N	matgss	GSS	[11]
N	matngss0	Fixed α scheduling + GSS	[22]
Y	matngss1	Fixed α scheduling + GSS	[26]
N	matngss2	Adaptive α scheduling + GSS	[24, 25]
N	matngss3	Proposed scheduling + GSS	
Y	matngss4	Proposed scheduling + GSS	
N	matfss	FSS	[8]
N	matnfss0	Fixed α scheduling + FSS	[22]
Y	matnfss1	Fixed α scheduling + FSS	[26]
N	matnfss2	Adaptive α scheduling + FSS	[24, 25]
N	matnfss3	Proposed scheduling + FSS	
Y	matnfss4	Proposed scheduling + FSS	
N	mattss	TSS	[21]
N	matntss0	Fixed α scheduling + TSS	[22]
Y	matntss1	Fixed α scheduling + TSS	[26]
N	matntss2	Adaptive α scheduling + TSS	[24, 25]
N	matntss3	Proposed scheduling + TSS	
Y	matntss4	Proposed scheduling + TSS	

Conventional static scheduling schemes distribute the total workload equally to all workers at compiling time. However, such schemes are obviously not suitable for dynamic and heterogeneous environments. Therefore, a weighted static scheduling scheme is adopted in this experiment. The partitioning principle follows PW_s ratios. Faster nodes get proportionally more workload than the slower ones.

4.1 Experiments on three applications

We implemented three classes of application in C language, with message-passing interface (MPI) directives to parallelize code segments for execution on our testbed: Matrix Multiplication, Mandelbrot Set Computation, and Circuit Satisfiability. The first has regular workloads, the last irregular workloads. To enhance the readability of experimental results, brief descriptions of all implemented programs are given in Tables 2– 4.

4.2 Hardware configuration and terminology

We built a heterogeneous cluster consisting of eleven nodes. The hardware and software configurations are specified in Tables 5 and 6, respectively. Figures 2 and 3 show, respectively, the network route state and topology.

Table 3 Our implementation of all Mandelbrot set programs

Master compute	Name	Description	Reference #
Y	manstat	Static scheduling	[10]
N	mangss	GSS	[11]
N	manngss0	Fixed α scheduling + GSS	[22]
Y	manngss1	Fixed α scheduling + GSS	[26]
N	manngss2	Adaptive α scheduling + GSS	[24, 25]
N	manngss3	Proposed scheduling + GSS	
Y	manngss4	Proposed scheduling + GSS	
N	manfss	FSS	[8]
N	mannfss0	Fixed α scheduling + FSS	[22]
Y	mannfss1	Fixed α scheduling + FSS	[26]
N	mannfss2	Adaptive α scheduling + FSS	[24, 25]
N	mannfss3	Proposed scheduling + FSS	
Y	mannfss4	Proposed scheduling + FSS	
N	mantss	TSS	[21]
N	manntss0	Fixed α scheduling + TSS	[22]
Y	manntss1	Fixed α scheduling + TSS	[26]
N	manntss2	Adaptive α scheduling + TSS	[24, 25]
N	manntss3	Proposed scheduling + TSS	
Y	manntss4	Proposed scheduling + TSS	

Table 4 Our implementation of all circuit satisfiability programs

Master compute	Name	Description	Reference #
Y	satstat	Static scheduling	[10]
N	satgss	GSS	[11]
N	satngss0	Fixed α scheduling + GSS	[22]
Y	satngss1	Fixed α scheduling + GSS	[26]
N	satngss2	Adaptive α scheduling + GSS	[24, 25]
N	satngss3	Proposed scheduling + GSS	
Y	satngss4	Proposed scheduling + GSS	
N	satfss	FSS	[8]
N	satnfss0	Fixed α scheduling + FSS	[22]
Y	satnfss1	Fixed α scheduling + FSS	[26]
N	satnfss2	Adaptive α scheduling + FSS	[24, 25]
N	satnfss3	Proposed scheduling + FSS	
Y	satnfss4	Proposed scheduling + FSS	
N	sattss	TSS	[21]
N	satntss0	Fixed α scheduling + TSS	[22]
Y	satntss1	Fixed α scheduling + TSS	[26]
N	satntss2	Adaptive α scheduling + TSS	[24, 25]
N	satntss3	Proposed scheduling + TSS	
Y	satntss4	Proposed scheduling + TSS	

Table 5 Hardware configuration

Hostname	Processor model	Number of processors	Memory size	NIC	OS version
amd64-dual21	Dual-Core AMD Opteron(tm) Processor 2212	4	2 GB	1G	2.6.21-1.3194.fc7
amd64-dual11	AMD Opteron(tm) Processor 246	2	2 GB	1G	2.6.21-1.3194.fc7
amd64-dual31	AMD Opteron(tm) Processor 246	2	2 GB	1G	2.6.18-1.2798.fc6
xeon2	Intel(R) Xeon(TM)	2	1.5 GB	1G	2.6.15-1.2054_FC5
quad1	Intel(R) Core(TM)2 Quad Q6600	4	2 GB	1G	2.6.23.1-42.fc8
omega	Intel(R) Xeon(TM)	2	512 MB	1G	2.6.15-1.2054_FC5
circa	AMD Athlon(tm) MP 2000+	2	1 GB	100M	2.6.21-1.3194.fc7
amd-mpdual1	AMD Athlon(tm) MP 2000+	2	2 GB	100M	2.6.18-1.2798.fc6
condor1	Intel(R) Pentium(R) 4	2	512 MB	100M	2.6.18-1.2798.fc6
condor2	Intel(R) Pentium(R) 4	2	512 MB	100M	2.6.18-1.2798.fc6
s1	Intel(R) Xeon(R) CPU E5310	8	4 GB	1G	2.6.24.4-64.fc8

Table 6 CPU clock speeds and OS, compiler, and LAM/MPI versions

Hostname	Clock	BogoMIPS	HPCC	LAM/MPI ver.	GCC ver.
amd64-dual21	2000.080 MHz	4000.31	11.68 Gflops ($N = 15$k)	lam-7.1.2-10.fc7	gcc-4.1.2-27.fc7
amd64-dual11	1992.128 MHz	3983.54	6.376 Gflops ($N = 15$k)	lam-7.1.2-10.fc7	gcc-4.1.2-27.fc7
amd64-dual31	1991.652 MHz	3983.45	6.100 Gflops ($N = 15$k)	lam-7.1.2-8.fc6	gcc-4.1.1-30
xeon2	3056.757 MHz	6112.98	5.312 Gflops ($N = 13$k)	lam-7.1.2-1.fc5	gcc-4.1.0-3
quad1	2699.986 MHz	5399.77	24.61 Gflops ($N = 15$k)	lam-7.1.2-10.fc7	gcc-4.1.2-33
omega	3000.240 MHz	6000.62	5.372 Gflops ($N = 7$k)	lam-7.1.2-1.fc5	gcc-4.1.1-51.fc5
circa	1666.794 MHz	3333.05	3.732 Gflops ($N = 10$k)	lam-7.1.2-10.fc7	gcc-4.1.2-12
amd-mpdual1	1666.787 MHz	3333.77	3.837 Gflops ($N = 13$k)	lam-7.1.2-8.fc6	gcc-4.1.1-30
condor1	2806.465 MHz	5613.16	3.302 Gflops ($N = 7$k)	lam-7.1.2-8.fc6	gcc-4.1.1-30
condor2	2806.471 MHz	5613.14	3.317 Gflops ($N = 7$k)	lam-7.1.2-8.fc6	gcc-4.1.2-13.fc6
s1	1596.476 MHz	3192.85	14.39 Gflops ($N = 20$k)	lam-7.1.2-10.fc7	gcc-4.1.2-33

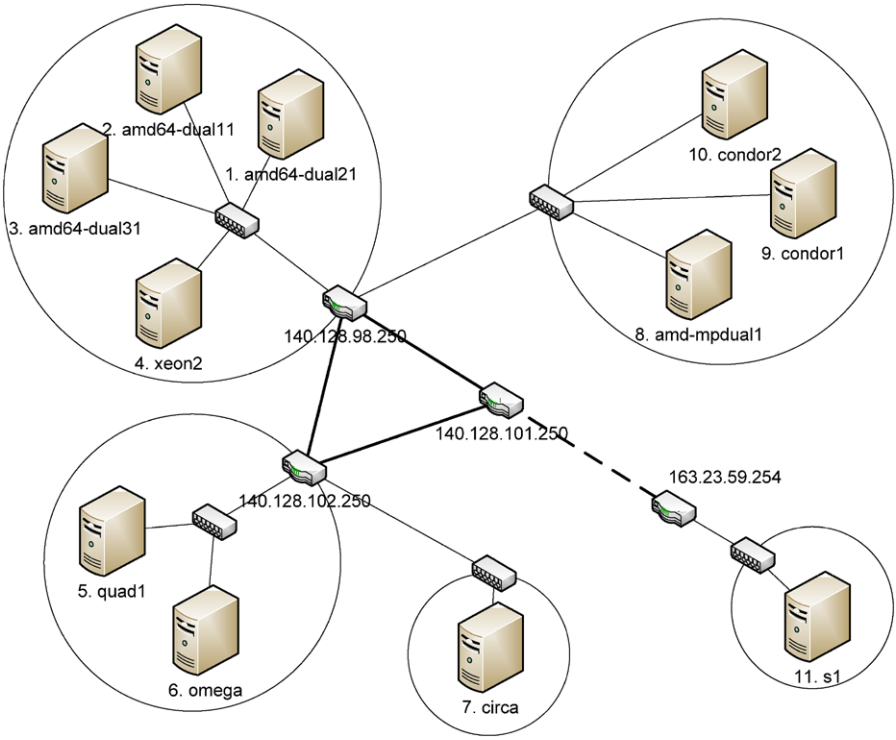
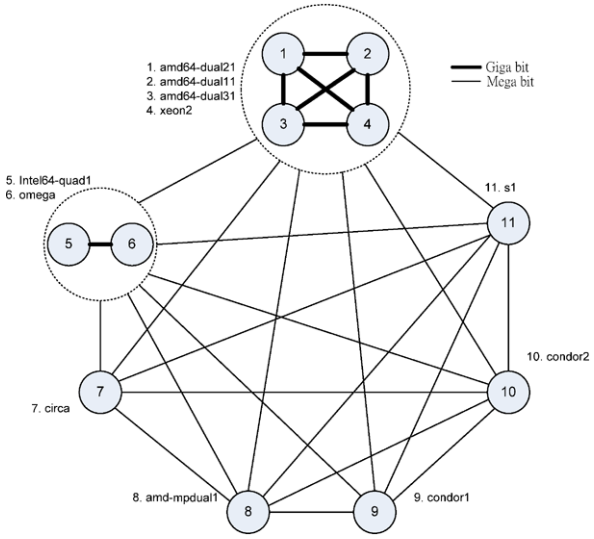


Fig. 2 Network route state

4.3 Experimental results

In our experiments, we first collected HPL measurements for all nodes, and then investigated the impact of parameters α, β, on performance. Parameters α and β are

Fig. 3 Network topology



set by the programmer and choosing appropriate values adaptable to dynamic environments is difficult. In this work, the master node also participated in computation.

4.3.1 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 developing efficient parallel matrix multiplication algorithms, and this will remain a task in the future as well. Many parallel algorithms for matrix multiplication have been designed, implemented, and tested on various parallel computers and clusters of workstations.

We implemented the proposed scheme for matrix multiplication. The master module is responsible for distributing workloads. When a slave node becomes idle, the master node sends two integers to it representing the beginning and end pointers to an assigned chunk. In other words, every node has a local copy of the input matrices, so data communication is not significant in this kind of implementation. This means the communication cost between the master and the slave is low, and the dominant cost is the matrix multiplication computation. The slave module C/MPI code fragment for matrix multiplication is listed below. As the source code shows, the column is the atomic unit of allocation.

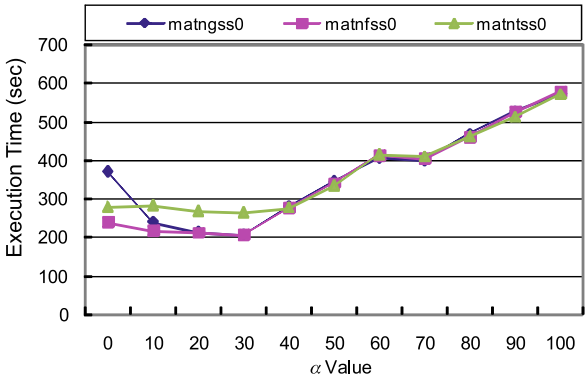


Fig. 4 Execution times for parameter α influence on matrix multiplication performance with matrix size 4096×4096

```
MPI_Recv(buf, count, MPI_FLOAT, source, tag. MPI_COMM_WORLD,
&status);
f=0;
while (status.MPI_TAG > 0)
{
  for (i=0; 1<(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 × size of(float));
c=(float*)malloc(count × size of(float));
MPI_Recv(buf, count, MPI_FLOAT, 0, MPI_ANY_TAG,
MPI_COMM_WORLD, &status);
}
```

Figures 4 and 5 show how parameters influence the performance. In the experiment, we found that the proposed schemes got better performance when α was 30 and β had various optimum values based on scheduling. After selecting α and β values, we carried out a set of experiments with them, and compared the results with the previous scheduling algorithms. Each experiment was run ten times and the average fetched in order to achieve a better accuracy.

We first investigated execution times on the heterogeneous cluster for the GSS group, then for the FSS group, and finally for the TSS group. In our experiments, the

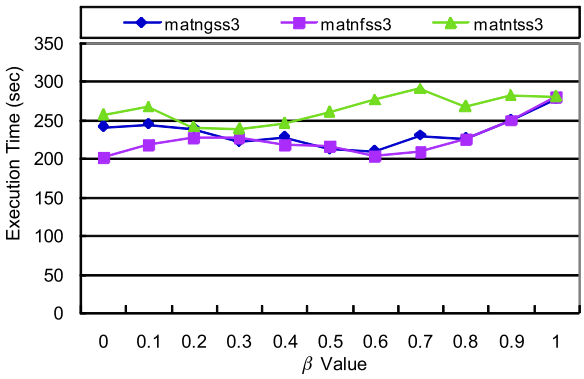


Fig. 5 Execution times for parameter β influence on matrix multiplication performance with matrix size 4096×4096

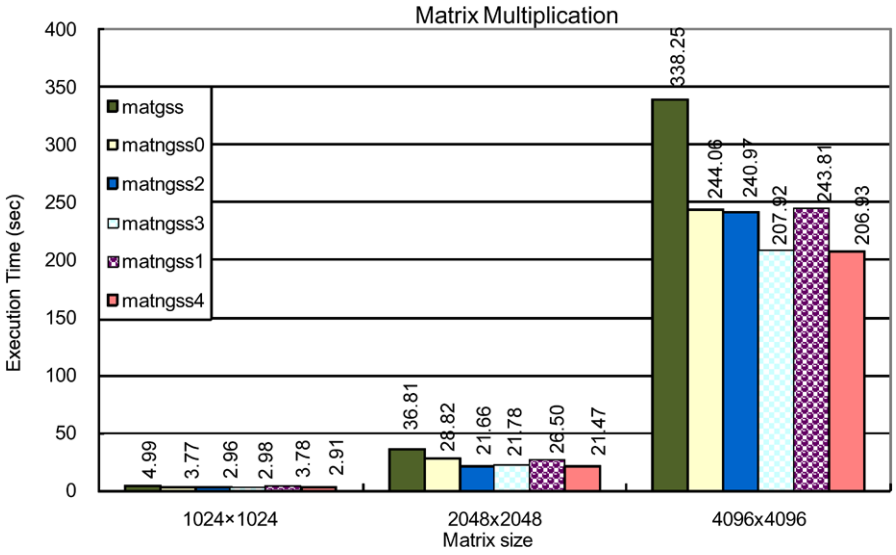


Fig. 6 Execution times for the proposed matrix multiplication scheduling compared with the previous GSS group schemes

execution times for static partitioning (matstat) were orders of magnitude worse than any of the dynamic approaches and made the results of the dynamic systems very difficult to distinguish visually from one another, so we state them as follows: matrix size 1024×1024 cost 57.83 seconds, matrix size 2048×2048 cost 251.7 seconds, and matrix size 4096×4096 cost 1853.27 seconds. Figures 6–8 show execution times for the conventional scheme (mat*ss), dynamic hybrid (matn*ss0-2), and the proposed scheme (matn*ss3-4), on the FSS, GSS, and TSS group approaches with input matrix sizes of 1024×1024, 2048×2048 and 4096×4096. Experimental results show that the proposed scheduling scheme got better performance than the static and previous schemes. Note that on the 4096×4096 matrix, our approach achieved speedups of 1.17, 1.27 and 1.07 over GSS, FSS and TSS in non-master participation, and speedups of 1.17, 1.22 and 1.05 in master participation.

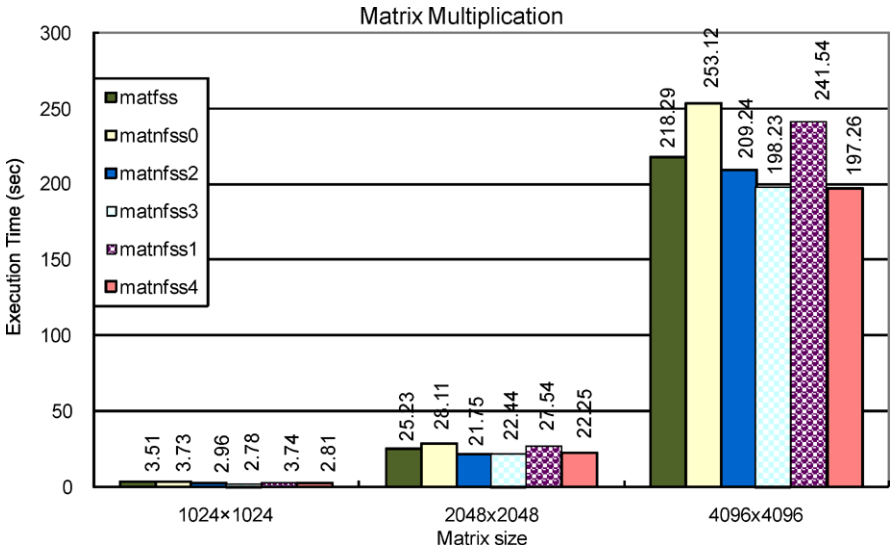


Fig. 7 Execution times for the proposed matrix multiplication scheduling compared with the previous FSS group schemes

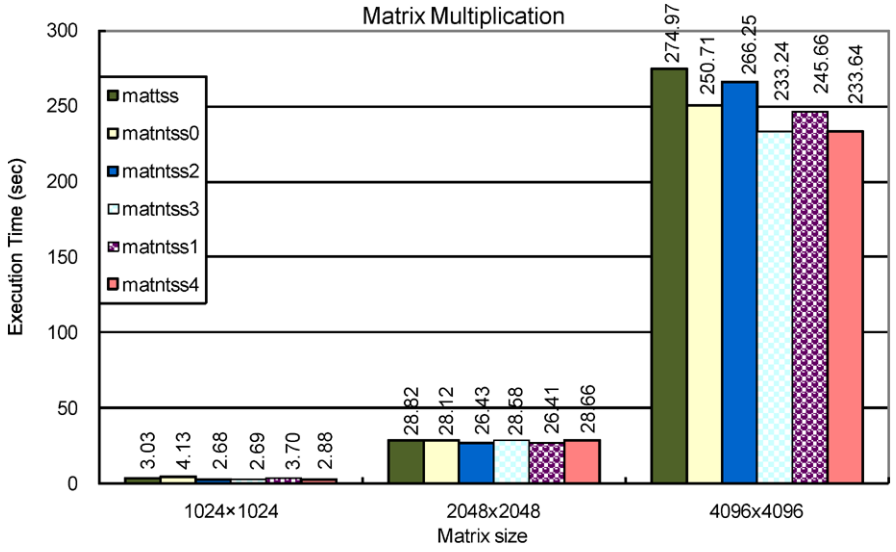


Fig. 8 Execution times for the proposed matrix multiplication scheduling compared with the previous TSS group schemes

4.3.2 Application 2: Mandelbrot set computation

A Mandelbrot set is a problem involving the same computation on different data points with different convergence rates [15]. Named after Benoit Mandelbrot, the Mandelbrot set is a fractal, a class of objects that display self-similarity at various

scales. Magnifying a fractal reveals small-scale details similar to its large-scale characteristics. Although the Mandelbrot set is self-similar at magnified scales, its 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 by n pixels. The resultant image is one of m by n pixels.

The proposed scheme was implemented for Mandelbrot set computation. The master module is responsible for workload distribution. When a slave node becomes idle, the master node sends two integers to it representing the beginning and end pointers to an assigned chunk. As in the matrix multiplication implementation, this keeps communication costs between the master and the slave low, and the dominant cost is the Mandelbrot set computation. The slave module C/MPI code fragment for Mandelbrot set computation is listed below. In this application, the workloads for outer loop iterations are irregular because the number of executions required for convergence is not fixed. Therefore, the workload distribution performance depends on the degree of variation between iterations.

```
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 */
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);
}
```

Figures 9 and 10 show how parameters influence the performance. In this experiment, we found that the proposed schemes got better performance when α was 40 and β was about 0.7. After selecting α and β values, we carried out a set of experiments

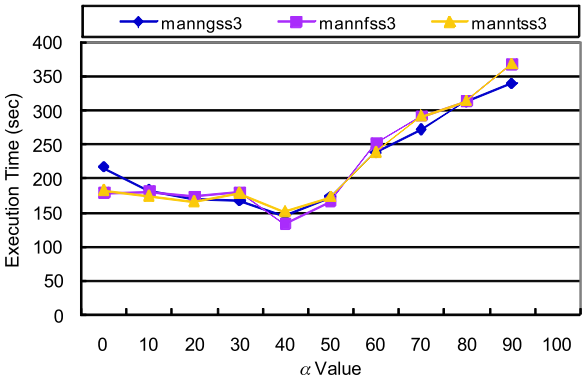


Fig. 9 Execution times for parameter α influence on Mandelbrot set computation performance at image size 2048×2048

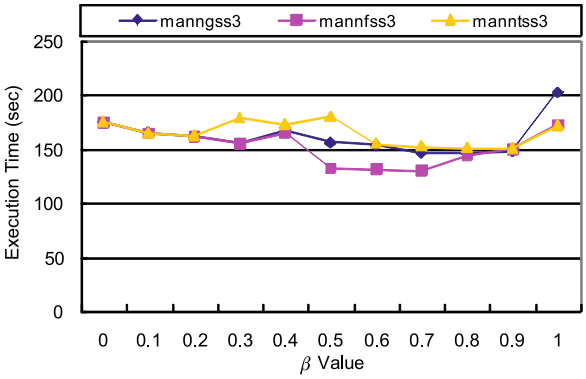


Fig. 10 Execution times for parameter β influence on Mandelbrot set computation performance at image size 2048×2048

with them, and compared the results with the previous scheduling algorithms. Each experiment was averaged over ten time runs in order to achieve a better accuracy.

We first investigated GSS group execution times on the heterogeneous cluster, then FSS group execution times, and finally TSS group execution times. In this experiment, static partitioning (manstat) execution times were: image size 512×512 cost 194.04 seconds, image size 1024×1024 cost 688.1 seconds, and image size 2048×2048 cost 3125.3 seconds.

Figures 11–13 show execution times for the static (manstat), conventional (man*ss), and dynamic hybrid (mann*ss0-2) schemes, and for the proposed scheme (mann*ss3-4), on the FSS, GSS, and TSS group approaches with input image sizes of 512×512, 1024×1024, and 2048×2048, respectively. Experimental results show the proposed scheduling scheme got better performance than the static and previous schemes. Master node participation in computation did not raise the performance in our experiments. Note that on image size 2048×2048, our approach achieved speedups of 1.17, 1.27 and 1.07 over GSS, FSS and TSS in non-master participation and of 1.17, 1.22 and 1.05 in master participation.

The largest size used in these experiments was 2048×2048, which is not very big but wastes a great deal of time when run with a node. It is hard to experiment with bigger sizes due to memory capacity and cache considerations, but we may find that the bigger the size the greater the efficiency. Note that in a heterogeneous

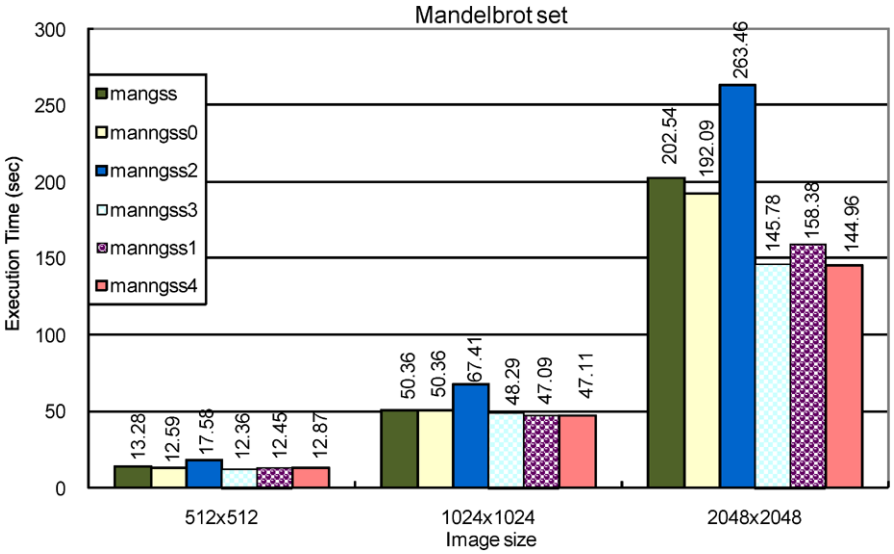


Fig. 11 Execution times for the proposed Mandelbrot set scheduling and the previous GSS group schemes

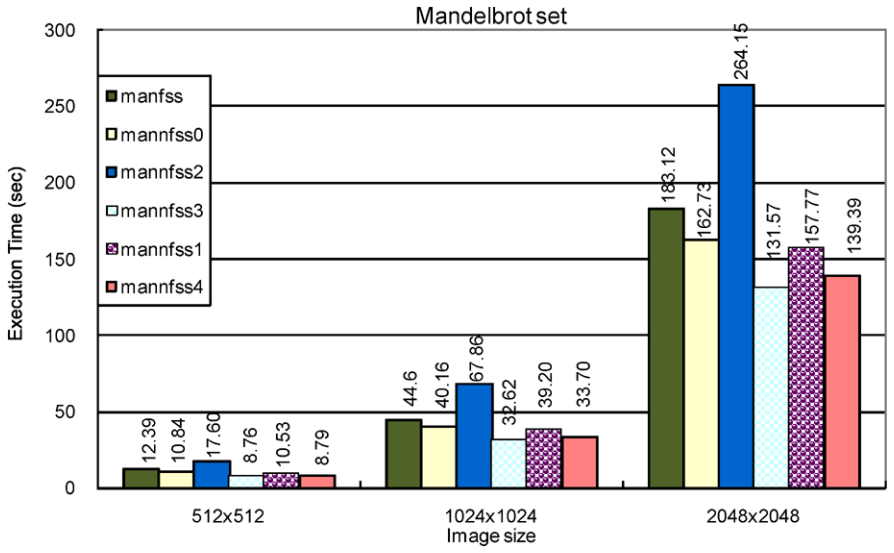


Fig. 12 Execution times for the proposed Mandelbrot set scheduling and the previous FSS group schemes

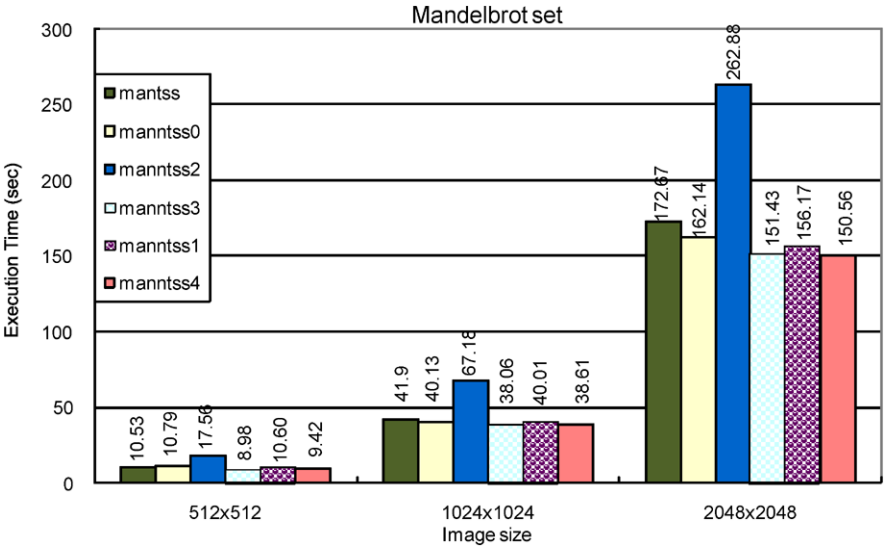


Fig. 13 Execution times for the proposed Mandelbrot set scheduling and the previous TSS group schemes

environment, manngss2 performed worse than any other schemes merely because the adaptive α value is not suitable to this environment.

4.3.3 Application 3: circuit satisfiability

The circuit satisfiability problem is one involving a combinational circuit composed of AND, OR, and NOT gates. Simply speaking, a circuit is satisfiable if there exists a set of Boolean input values that makes the output of the circuit be 1. The circuit satisfiability problem is NP-complete, and no known algorithms can solve it in polynomial time. In the experiment, we found the solutions through an exhaustive search. This operation gets a number as an input, the number of Boolean variables in the expression. After that, the algorithm exhaustively computes all combinations of these values. The circuit satisfiability problem is implemented in a similar way. The master module is responsible for workload distribution. When a slave node becomes idle, the master node sends two integers to it representing the beginning and end pointers to an assigned chunk. As in the matrix multiplication implementation, this keeps communication costs between the master and the slave low, and the dominant cost is the Mandelbrot set computation. The slave module C/MPI code fragment for Mandelbrot set computation is listed below. In this application, outer loop iteration workloads are irregular because the number of executions required to test for satisfiability is not fixed. Therefore, the workload distribution performance depends on the degree of variation between iterations.

```
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 */
  Count_true − 0;
  for (y = b[0]; y < b[1]; y++){
    count_true+ = check_circuit (rank_no, y);
  }//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);
}
```

Figures 14 and 15 show how parameters influence the performance. In this experiment, we found that the proposed schemes got better performance when α was 40 and β was about 0.6. After selecting α and β values, we carried out a set of experiments with them, and compared the results with the previous scheduling algorithms. Each experiment was averaged over ten time runs in order to achieve a better accuracy.

We first investigated GSS group execution times on the heterogeneous cluster, then FSS group execution times, and finally TSS group execution times. Static partitioning execution times (satstat) for this experiment were: 18 variable numbers cost 22.63 seconds, 19 variable numbers cost 96.12 seconds, and 20 variable numbers cost 412.45 seconds. Figures 16–18 show execution times for the static (satstat), conventional (sat*ss), and dynamic hybrid (satn*ss0-2) schemes, and the proposed scheme (satn*ss3-4), on the FSS, GSS, and TSS group approaches with 18, 19, and 20 variable numbers. Experimental results show the proposed scheduling scheme got better performance than the static and the previous ones. Master node participation in computation did not raise the performance in our experiments. In this case, our scheme achieved speedups of 1.12, 1.16, and 1.10 over GSS, FSS, and TSS for input size 20 in non-master participation and of 1.15, 1.20 and 1.10 in master participation.

We have built a Grid testbed consisting of eleven nodes. The hardware and software configurations are specified in Tables 3 and 4, respectively. Figures 1 and 2 show the network route state and topology.

4.4 Experimental result

In our experiments, first, the HPL measurements and CPU speed of all nodes were collected. Next, the impact of the parameters α, β, on performance was investigated.

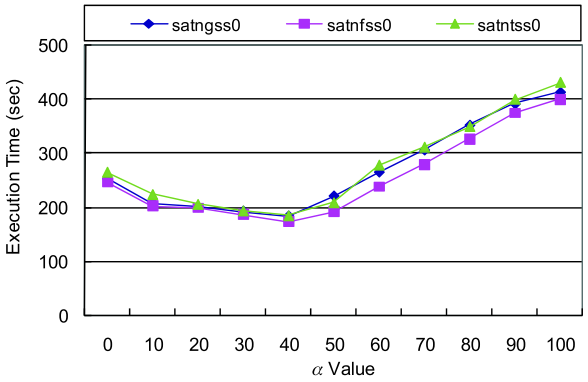


Fig. 14 Execution times for parameter α influence on circuit-satisfiability problem performance with 20 variable numbers

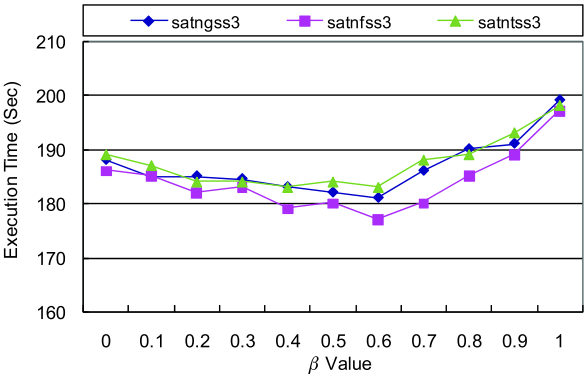


Fig. 15 Execution times for parameter β influence on circuit-satisfiability problem performance with 20 variable numbers

With this approach, a faster node will get more workload than a slower one proportionally. The principle of partitioning is according to the ratios of PW_s. The parameter α should not be too large or too small. In the former case, the dominant computer will not finish its work. In the latter case, the dynamic scheduling overhead is significant. In both cases, good performance cannot be attained. An appropriate parameter will lead to good performance.

In this work, the master node also participates in computation, and the scheduling parameter is set to be 60 for all α-scheduling schemes.

We have implemented the proposed scheme for matrix multiplication. The matrix multiplication is a fundamental operation in many numerical linear algebra applications [24]. This operation derives a resultant matrix by multiplying two input matrices, A and B, where A is a matrix of m rows by p columns and B is a matrix of p rows by n columns. The resultant matrix is one of m rows by n columns.

Figures 21 and 22 show how parameters influence the performance. In the experiment, we find that the proposed schemes get better performance when α is 50 and β is 0.3.

After selecting α and β values, we carry out the set of experiments with those two parameters, and compare with the previous scheduling. Each experiment is to run for ten times and fetch the average in order to reach a better accuracy.

In this work, the execution time of static scheme with input matrix of size 1024×1024 is about 51 seconds, 128 seconds is for size 2048×2048, and about

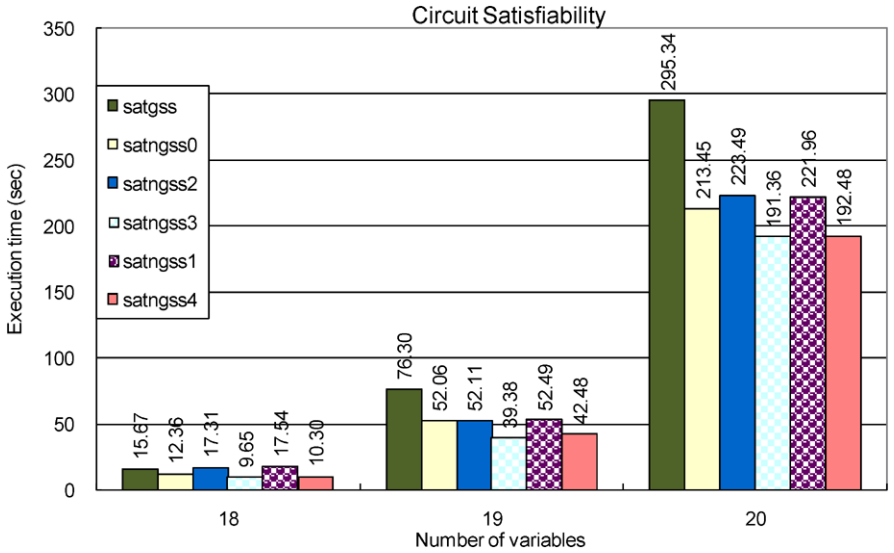


Fig. 16 Execution times for the proposed circuit satisfiability scheduling and the previous GSS group schemes

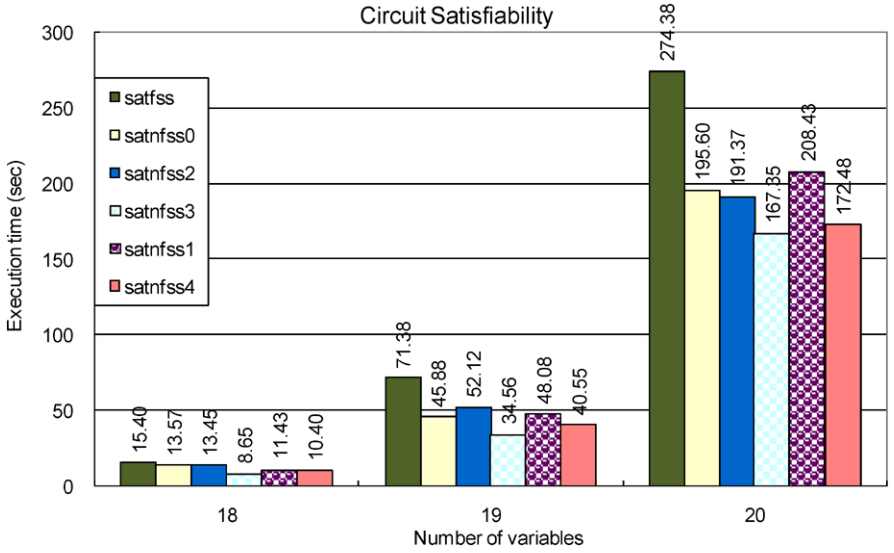


Fig. 17 Execution times for the proposed circuit satisfiability scheduling and the previous FSS group schemes

2738 seconds is for size 4096 × 4096. Figures 23–25 illustrate execution time of traditional scheme (mat*ss), dynamic hybrid (matn*ss0-2) and the proposed scheme (matn*ss3-4), with input matrix of size 1024 × 1024, 2048 × 2048 and 4096 × 4096, respectively. Experimental results show that the proposed scheduling scheme got better performance than the static and previous ones.

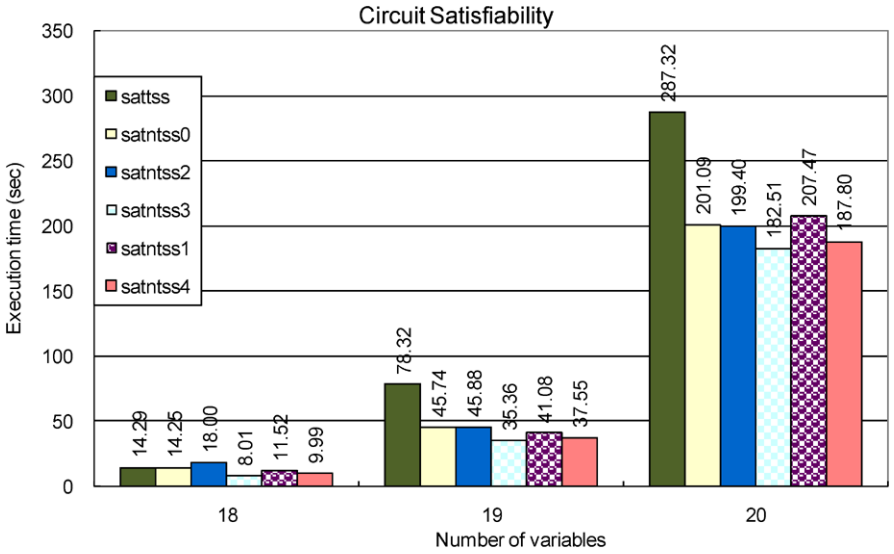


Fig. 18 Execution times for the proposed circuit satisfiability scheduling and the previous TSS group schemes

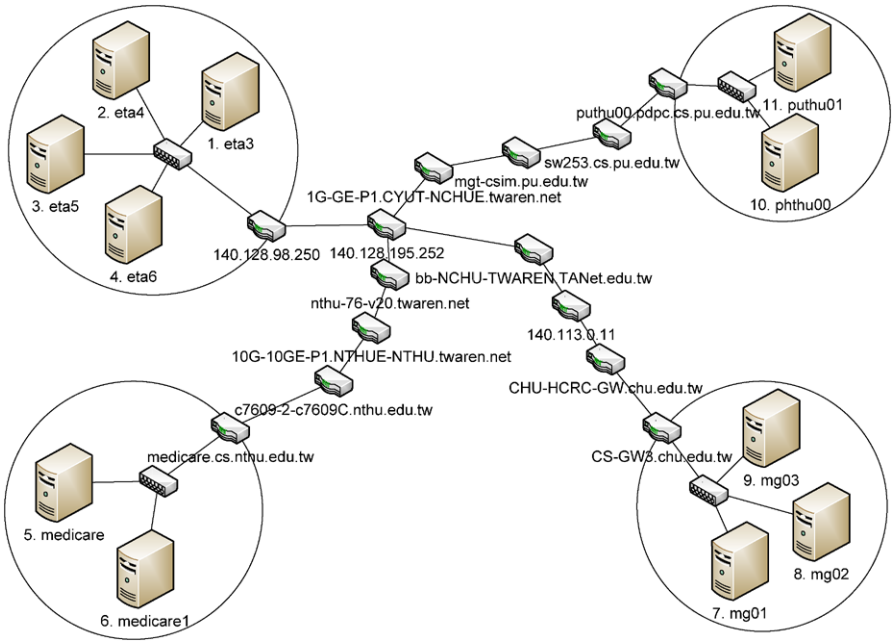


Fig. 19 Network route state

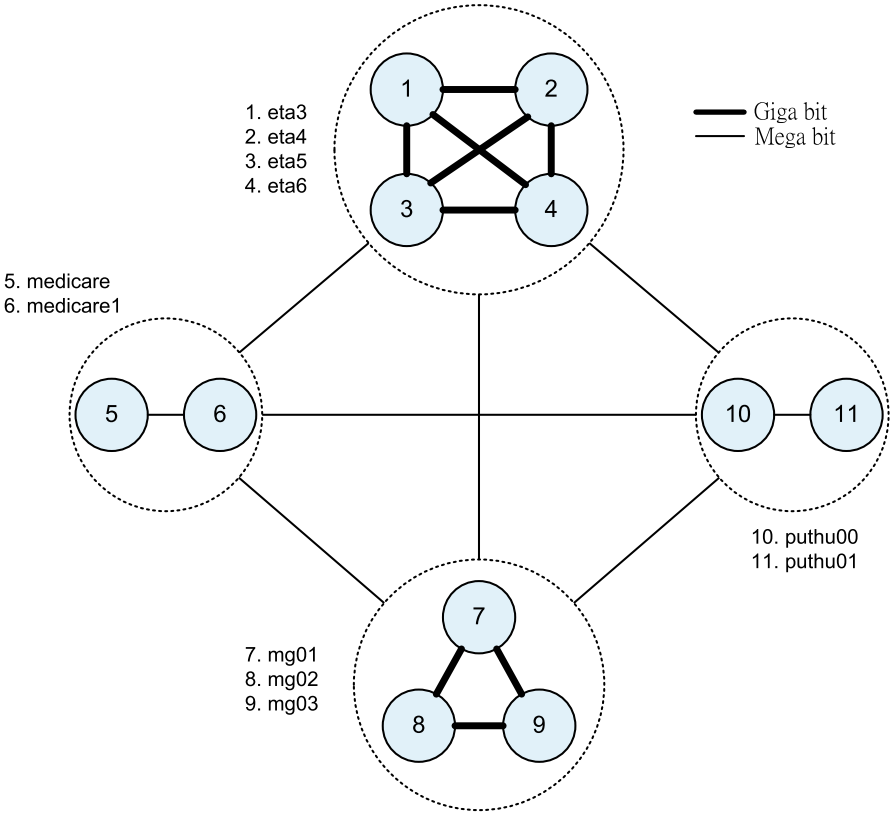


Fig. 20 Network topology

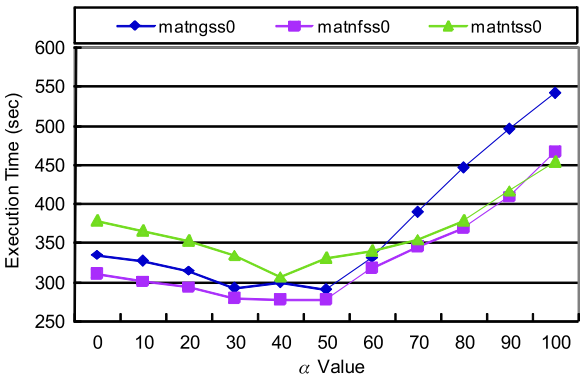


Fig. 21 Execution time of parameters α influence on matrix multiplication performance with matrix size 4096×4096

5 Conclusion

In this paper, we proposed a heuristic scheme that combines the advantages of static and dynamic loop scheduling schemes, and compared it with the previous algorithms

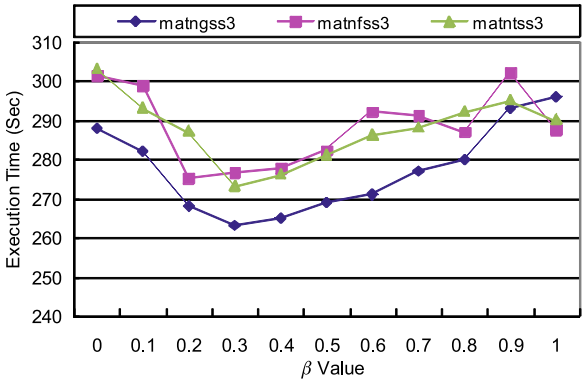


Fig. 22 Execution time of parameters β influence on matrix multiplication performance with matrix size 4096×4096

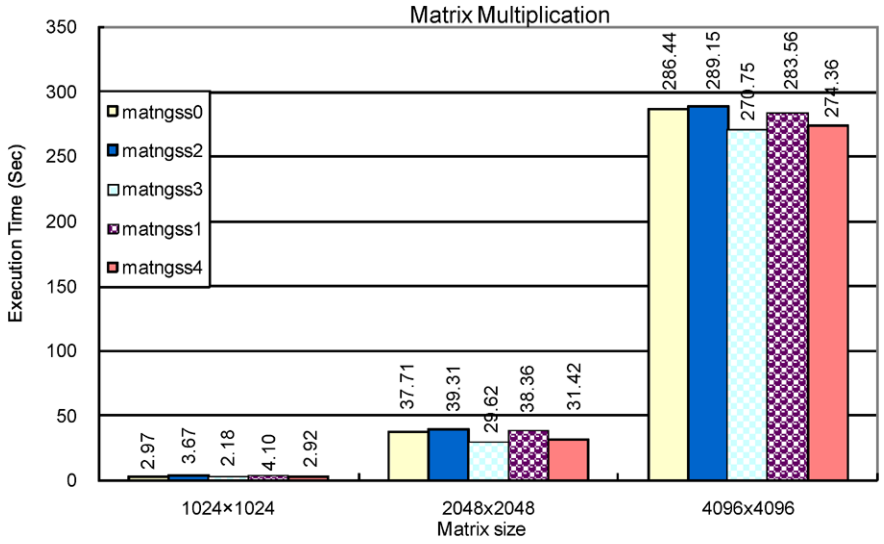


Fig. 23 Execution times for the proposed matrix multiplication scheduling compared with the previous GSS group schemes

in experiments on three types of application program in heterogeneous cluster environments. In each case, our approach obtained performance improvement over the previous schemes. Our approach is also less sensitive to α values than the previous schemes, i.e., more robust. In our future work, we will implement more application program types to verify our approach, and will carry out experiments with larger sizes. Also, we hope to find better ways of modeling performance weighting with other factors, such as amount of memory available, memory access costs, network information, and CPU loading. We will also address a theoretical analysis of the proposed method.

Acknowledgements This work is supported in part by the National Science Council, Taiwan, under grants no. NSC 98-2220-E-029-001-, NSC 98-2220-E-029-004- and NSC 98-2511-S-468-002-.

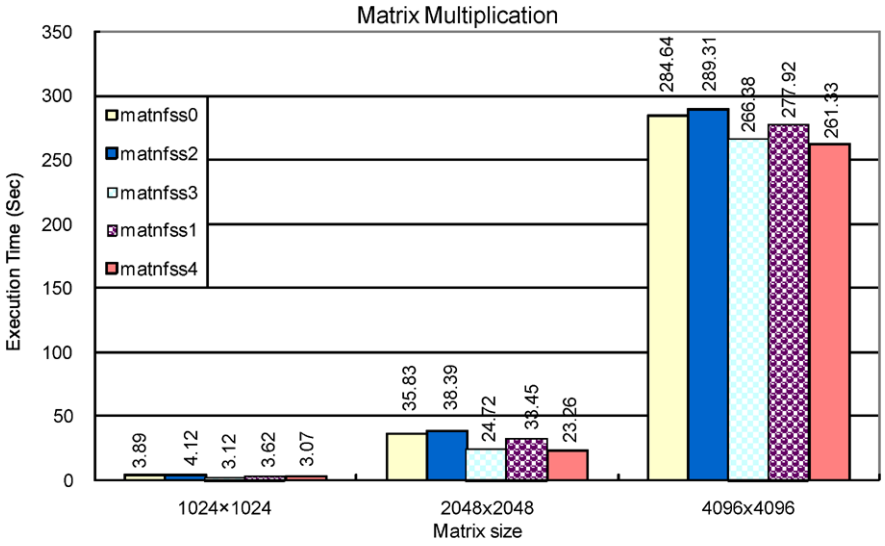


Fig. 24 Execution times for the proposed matrix multiplication scheduling compared with the previous FSS group schemes

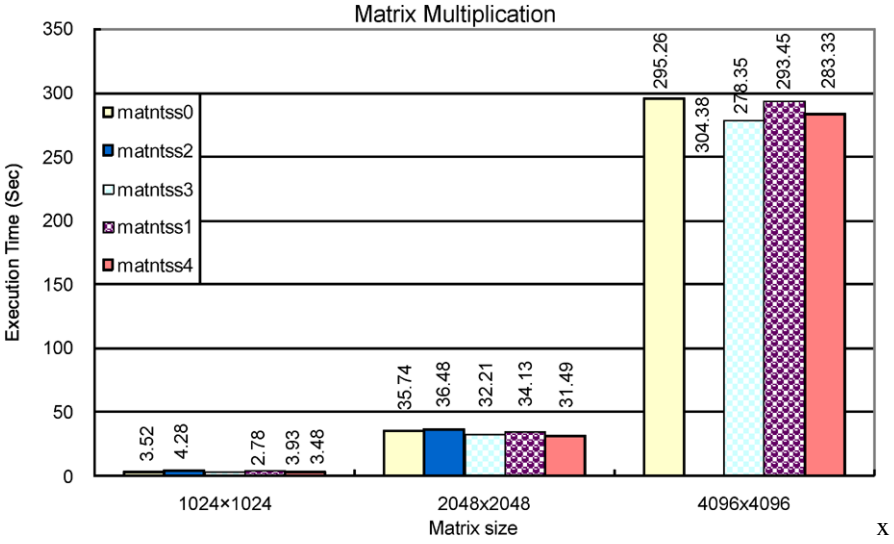


Fig. 25 Execution times for the proposed matrix multiplication scheduling compared with the previous TSS group schemes

References

1. Baker M, Buyya R (2002) Cluster computing: the commodity supercomputer. Softw Pract Exp 29(6):551–575, 1999
2. Beaumont O, Casanova H, Legrand A, Robert Y, Yang Y (2005) Scheduling divisible loads on star and tree networks: results and open problems. IEEE Trans Parall Distrib Syst 16:207–218

3. Bennett BH, Davis E, Kunau T, Wren W (2000) Beowulf parallel processing for dynamic load-balancing. Proc IEEE Aerosp Conf 4:389–395
4. Bohn CA, Lamont GB (2002) Load balancing for heterogeneous clusters of PCs. Future Gener Comput Syst 18:389–400
5. Challenge Benchmark HPC. http://icl.cs.utk.edu/hpcc/
6. Cheng K-W, Yang C-T, Lai C-L, Chang S-C (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
7. 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, pp 282–291
8. Hummel SF, Schonberg E, Flynn LE (1992) Factoring: a method scheme for scheduling parallel loops. Commun ACM 35:90–101
9. Introduction to the Mandelbrot Set, http://www.ddewey.net/mandelbrot/
10. Li H, Tandri S, Stumm M, Sevcik KC (1993) Locality and loop scheduling on NUMA multiprocessors. In: Proceedings of the 1993 international conference on parallel processing, vol II, pp 140–147
11. Polychronopoulos CD, Kuck D (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans Comput 36(12):1425–1439
12. Post E, Goosen HA (2001) Evaluation of 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)
13. Shih W-C, Yang C-T, Tseng S-S (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, Berlin, pp 48–55
14. Shih W-C, Yang C-T, Tseng S-S (2005) A hybrid parallel loop scheduling scheme on grid environments. In: Grid and cooperative computing: 4th international conference, GCC 2005. Lecture notes in computer science, vol 3795. Springer, Berlin, pp 370–381
15. Shih W-C, Yang C-T, Tseng S-S (2005) A hybrid parallel loop scheduling scheme on heterogeneous PC clusters. In: Proceedings of the 6th international conference on parallel and distributed computing, applications and technologies (PDCAT 2005), December 5–8, 2005, pp 56–58
16. Shih W-C, Yang C-T, Tseng S-S (2006) A performance-based approach to dynamic workload distribution for master–slave applications on grid environments. In: GPC 2006. Lecture notes in computer science, vol 3947. Springer, Berlin, pp 73–82
17. Shih W-C, Yang C-T, Tseng S-S (2007) A performance-based parallel loop scheduling on grid environments. J Supercomput 41(3):247–267
18. Sterling T, Bell G, Kowalik JS (2002) Beowulf cluster computing with Linux. MIT Press, Cambridge
19. Tang P, Yew PC (1986) Processor self-scheduling for multiple-nested parallel loops. In: Proceedings of the 1986 international conference on parallel processing, pp 528–535
20. The Scalable Computing Laboratory (SCL), http://www.scl.ameslab.gov/
21. Tzen TH, Ni LM (1993) Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. IEEE Trans Parallel Distrib Syst 4:87–98
22. Yang C-T, Chang S-C (2004) A parallel loop self-scheduling on extremely heterogeneous PC clusters. J Inf Sci Eng 20(2):263–273
23. Yang C-T, Cheng K-W, Li K-C (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, vol 3222. Springer, Heidelberg, pp 92–100
24. Yang C-T, Cheng K-W, Li K-C (2005) An enhanced parallel loop self-scheduling scheme for cluster environments. J Supercomput 34(3):315–335
25. Yang C-T, Cheng K-W, Shih W-C (2007) On development of an efficient parallel loop self-scheduling for grid computing environments. Parallel Comput 33(7–8):467–487
26. Yang C-T, Shih W-C, Tseng S-S (2007) Dynamic partitioning of loop iterations on heterogeneous PC clusters. J Supercomput 44(1):1–23