J Supercomput (2008) 44: 1–23
DOI 10.1007/s11227-007-0146-0

Dynamic partitioning of loop iterations on heterogeneous PC clusters

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

Published online: 9 September 2007

Abstract Loop partitioning on parallel and distributed systems has been a critical problem. Furthermore, it becomes more difficult to deal with on the emerging heterogeneous PC cluster environments. In the past, some loop self-scheduling schemes have been proposed to be applicable to heterogeneous cluster environments. In this paper, we propose a performance-based approach, which partitions loop iterations according to the performance ratio of cluster nodes. To verify the proposed approach, a heterogeneous cluster is built, and three types of application programs are implemented to be executed in this testbed. Experimental results show that the proposed approach performs better than traditional schemes.

Keywords Parallel loops · Self-scheduling · Cluster computing · MPI programming · Heterogeneous · PC clusters

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

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

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

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

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

1 Introduction

As more and more inexpensive personal computers (PC) are available, clusters of PCs have become alternatives of supercomputers which many research projects cannot afford. Usually, a cluster connects several commodity computers by a local area network. Therefore, it is natural that a cluster consists of computers with various processors, memories and hard disk drives because of the fast development of information technology. However, it is difficult to deal with the heterogeneity in a cluster [1–4, 6, 13–15, 17, 18].

Loop scheduling and load balancing on parallel and distributed systems are critical problems, but it is difficult to cope with these problems, especially on the emerging PC-based clusters. Traditional loop self-scheduling approaches include static scheduling and dynamic scheduling. However, the former considers computing nodes as homogeneous resources, thus not suitable for heterogeneous environments. Besides, the latter, especially self-scheduling, still can be improved.

Previous researchers propose some useful self-scheduling schemes, which are applicable to PC-based cluster [17, 18] and grid computing environments [5, 12, 19]. These schemes are composed 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 speed [17] or HINT measurements [5, 18, 19]. After that, the remaining work load is scheduled by some well-known self-scheduling scheme, such as GSS [10]. Nevertheless, the performance of this approach depends on the appropriate choice of scheduling parameters. Besides, it estimates node performance only by CPU speed or HINT benchmark, which is one of the factors affecting node performance. In [5], 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 node performance. Many other factors also have dramatic influences in this respect, such as the amount of memory available, the cost of memory accesses, and the communication medium between nodes, and so forth. Using this intuitive approach, the result will be degraded if the performance estimation is not accurate.

In this paper, we propose a general approach which utilizes performance functions to estimate the performance ratio of each node. To verify the proposed approach, a heterogeneous cluster is built, and three types of application programs, matrix multiplication, Mandelbrot and circuit satisfiability, are implemented to be executed in this testbed. Empirical results show that the proposed approach can obtain performance improvement on previous schemes, for heterogeneous cluster environments.

Previous work in [5, 12, 18, 19] and this paper are all inspired by [17], the α self-scheduling scheme. However, this work has different viewpoints and unique contribution. First, while [5, 18] partition $\alpha\%$ of workload according to performance weighted by CPU clock speed in phase one, the proposed scheme conducts the partition according to a general performance function (PF). In this paper, we do not define performance function explicitly. Instead, application execution time is used to estimate the value of PF for all nodes. The PF obtained by simulation execution can estimate performance of cluster nodes rather accurately. The calculation of PR is presented later.

Second, the scheme in [17] utilizes a fixed α value, and [5, 18] adaptively adjust the α value according to the heterogeneity of the cluster. In a word, both schemes depend on a properly chosen α value to get good performance. Nevertheless, the proposed scheme focuses on accurate estimation of node performance, so the choice of α value is not very critical. In other words, we can roughly choose the α value from a larger range than previous schemes can. Third, in our implementation, the master node also participates in computation. However, in previous schemes, only slave nodes do computation work.

The rest of this paper is organized as follows. In Sect. 2, the background about parallel loop scheduling and cluster computing is reviewed. 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 programs are also presented in Sect. 4. Finally, the conclusion remarks are given in the last section.

2 Review of loop self-scheduling schemes

Traditional static loop scheduling schemes make scheduling decisions at compiling time, and equally assign workload to processors. It is applied when each iteration takes roughly the same amount of time, and the compiler knows enough relative information before compilation. Its advantage is less scheduling overhead, while the disadvantage is possible load-imbalance. Well-known static scheduling schemes include block Scheduling, cyclic Scheduling, block-D scheduling, cyclic-D scheduling [9], and so forth. However, these schemes are not suitable for heterogeneous clusters.

In [17], a heuristic was proposed to distribute workload according to CPU performance. In heterogeneous clusters, it is difficult to estimate node performance. In [11], it is indicated that many attributes influence system performance, include CPU clock speed, available memory, communication cost, and so forth. Yang et al. [18] tried to evaluate computer performance by HINT benchmark. Nevertheless, HINT requires hours to execute, so it is not suitable for frequent execution.

In contrast, dynamic scheduling is more suitable for load balancing. Nevertheless, the runtime overhead must be taken into consideration. The schemes we focus in this paper are self-scheduling, which is a large class of adaptive and dynamic centralized loop scheduling schemes. In a common self-scheduling scheme, p denotes the number of processors, N denotes the total iterations and $f()$ is a function to produce the chunk-size at each step. The output of f is the chunk-size for the next iteration. The design of the function f depends on the scheduling strategy of the scheme. For example, in GSS [9], f is defined as the number of remaining iterations of a parallel loop 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,

$$R_0 = N, \qquad C_i = f(i, p), \qquad R_i = R_{i-1} - C_i, \tag{1}$$

where $f()$ possibly has 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 [6, 9]. Different ways to compute C_i have given rise to different scheduling schemes. The most notable examples are Pure Self-Scheduling

Table 1 Example partition size

Scheme	Example partition size
PSS	1, 1, 1, 1, 1, 1, 1, 1, 1, ...
CSS(128)	128, 128, 128, 128, 128, 128, 128, 128
GSS [10]	256, 192, 144, 108, 81, 61, 46, 34, 26, ...
FSS [7]	128, 128, 128, 128, 64, 64, 64, 64, 32, ...
TSS [16]	128, 120, 112, 104, 96, 88, 80, 72, 64, ...

(PSS), Chunk Self-Scheduling (CSS), Factoring Self-Scheduling (FSS), Guided Self-Scheduling (GSS), and Trapezoid Self-Scheduling (TSS) [7, 10, 16]. Table 1 shows the different chunk sizes for a problem with the number of iteration $N = 1024$ and the number of processor $p = 4$.

Pure Self-Scheduling (PSS) is 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 the assigned workload. That is, an idle node does not have a chunk of workload to execute. Whenever a processor gets idle, iterations are assigned to it. This algorithm achieves good load balancing but induces excessive overhead [9].

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 by the compiler. When k is 1, the scheme is purely self-scheduling, as discussed above. Large chunk sizes cause load imbalances, while small chunk sizes are likely to produce excessive scheduling overhead [9].

Guided Self-Scheduling (GSS) can dynamically change the numbers of iterations assigned to idle processors [10]. 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) assigns loop iterations to working processors in phases [7]. 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) tries to reduce the need for synchronization while still maintaining reasonable load balances [16]. 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 [16] proposed TSS($N/2p$, 1) as a general selection.

In this case, the first chunk is of size $N/2p$, and consecutive chunks differ in size $N/8p^2$ iterations. The size difference of successive chunks is always a constant in TSS, whereas it is a decreasing function in GSS and in FSS.

In [17], the authors revise known loop self-scheduling schemes to fit all heterogeneous PC clusters environment when loop is regular. An approach is proposed to partition loop iterations by two phases and it achieves good performance in any heterogeneous environment: partition $\alpha\%$ of workload according to their performance weighted by CPU clock in the first phase and the rest $(100-\alpha)\%$ of workload according to known self-scheduling in the second phase. The experimental results are conducted on a cluster environment with six nodes and the fastest computer is 6 times faster than the slowest ones in CPU-clock cycle. Many various α values are applied to the matrix multiplication and a best performance is obtained with $\alpha = 75$.

3 The proposed approach: HPLS (Heuristic Parallel Loop Scheduling)

In this section, the performance function is defined first, and then the proposed algorithm is described.

3.1 Performance function

We propose to partition $\alpha\%$ of workload according to the performance ratio of all nodes, and the remaining workload is dispatched by some well-known self-scheduling scheme, such as GSS [10]. Using this approach, we do not need to know the real computer performance. However, a good performance ratio is desired to estimate performance of nodes accurately.

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

$$\mathrm{PF}_j(V_1, V_2, \ldots, V_M), \tag{2}$$

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

$$\mathrm{PF}_j = w \times \frac{1/t_j}{\sum_{\forall \mathrm{node}_i \in S} 1/t_i}, \tag{3}$$

where S is the set of all cluster nodes, t_i is the execution time (sec.) of node i for some application program, such as matrix multiplication, and w is the weight of this term.

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

In this paper, we do not define performance function explicitly. Instead, application execution time is used to estimate the value of PF for all nodes. The PF obtained by simulation execution can estimate performance of computing nodes accurately.

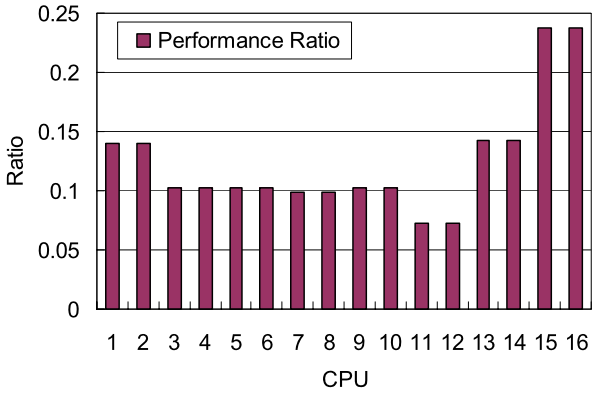


Fig. 1 Performance ratio of 16 processors for our heterogeneous cluster

The calculation of PR is presented as follows, and the performance ratio is illustrated in Fig. 1.

First, the PFs of all nodes are estimated by experimental simulation. Execution time of the target program on all computing nodes is recorded, and their reciprocals are taken to form the performance function values. A good performance ratio is desired to predict performance of nodes accurately. The proposed approach is based on the simulated performance of computing nodes to distributed work load, so it is important to know the simulated performance of all nodes. For example, NGSS [17] is executed on each node, and the size of input matrix is 512×512.

To normalize the values of performance functions, we compute performance ratios (PR) as follows. First, the reciprocal of performance function of each node is taken. Then, the ratios of these reciprocals are defined as PR of cluster nodes. For instance, assume the PF of two nodes are $1/2$ and $1/3$. Then, the PR is $1/2 : 1/3$. Consequently, the PR of the two nodes is 3 and 2, respectively. In other words, if there are 5 loop iterations, 3 will be assigned to the former and 2 will be assigned to the latter.

3.2 Algorithm

With this approach, the computing node with better performance will get more data to process. The parameter α should not be too large or too small. In 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 can not be attained. An appropriate α value will lead to good performance.

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

The algorithm is composed of several steps. First, the related information are acquired. Then, the performance ratio is calculated. Next, α percent of the total work-

load is statically scheduled according to the performance ratio among all slave nodes. Finally, the remainder of the workload is dynamically scheduled by guided self-scheduling for load balancing. The algorithm is described as follows.

Algorithms MASTER and SLAVE in pseudo code:
Module MASTER

```
/* perform task scheduling, load balancing and some computation
*/
Initialization
/*  Performance Ratio is defined before execution. */
r = 0;
for (i = 1; i < number_of_nodes; i++) {
   partition α% of loop iterations according to the performance
functions;
   send data to all nodes;
   r++;
}
Master does its own computation work
Partition (100-α)% of loop iterations into the task queue using
some known self-scheduling scheme
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 = 0 to the idle slave
} while (r > 0)
Finalization
END MASTER
```

Module SLAVE /* worker */

```
Initialization
Probe if some data in
While (TAG > 0) {
       Receive initial solution and size of subtask work and
compute to fine solution
     Send the result to the master
     Probe if some data in
}
Finalization
END SLAVE
```

4 Experimental results

To verify our approach, a heterogeneous PC-based cluster is built, and three types of application programs are implemented with MPI to be executed on this testbed. To begin with, our cluster environment is illustrated, and terminologies for our programs are described. Next, performance of our scheme is compared with that of other static and dynamic schemes on the heterogeneous cluster, with respect to matrix multiplication, Mandelbrot and circuit satisfiability. Finally, performance comparison on a homogeneous cluster is discussed.

Table 2 Hardware configuration

Host	CPU type	CPU speed	Number of CPU	RAM
Cluster 1: Heterogeneous cluster				
hpc	Intel Xeon^{TM}	2.4 GHz	2	1 GB
amd1	AMD Athlon^{TM} MP	1.8 GHz	2	2 GB
amd1-dual1	AMD Athlon^{TM} MP	2.2 GHz	2	512 MB
amd1-dual01	AMD Athlon^{TM} MP	2.0 GHz	2	2 GB
dna2	AMD Athlon^{TM} MP	2.0 GHz	2	2 GB
piii-dual1	Intel Pentium III	866 MHz	2	1 GB
xeon2	Intel Xeon^{TM}	3.0 GHz	2	512 MB
hpc2	Intel Xeon^{TM}	3.0 GHz	2	1 GB
Cluster 2: Homogeneous cluster				
amd1	AMD Athlon^{TM} MP	1.8 GHz	2	2 GB
amd2	AMD Athlon^{TM} MP	1.8 GHz	2	2 GB
amd3	AMD Athlon^{TM} MP	1.8 GHz	2	2 GB
amd4	AMD Athlon^{TM} MP	1.8 GHz	2	2 GB

4.1 Grid hardware configuration and terminology

We have built a heterogeneous cluster and a homogeneous one. The former consists of 8 PCs, and each has 3COMTM 3C9051 10/100 Fast Ethernet NIC interconnected via an Accton CheetahSwitch AC-EX3016B Switch HUB. Similarly, the latter is composed of 4 PCs, and each has 3COMTM 3C9051 10/100 Fast Ethernet NIC interconnected via an Accton CheetahSwitch AC-EX3016B Switch HUB. The hardware configuration of the two clusters is specified in Table 2.

We have implemented three categories of application programs in C language, with message passing interface (MPI) directives for parallelizing code segments to be processed by multiple CPUs. For readability of experimental results, the description of our implementation for all programs is listed in Table 3. In this paper, the scheduling parameter α is set to be 50 for all hybrid schemes, except for the schemes by [17], of which α is dynamically adjustable according to cluster heterogeneity.

4.2 Application 1: matrix multiplication

The matrix multiplication is a fundamental operation in many numerical linear algebra applications. Many parallel algorithms have been designed, implemented, and tested on different parallel computers or cluster of workstations for matrix multiplication. 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 matrix B is one of p rows by n columns. The resultant matrix is one of m rows by n columns.

We have implemented the proposed algorithm for matrix multiplication. The master module is responsible for the distribution of workloads. When a slave node becomes idle, the master node sends two integers to the slave. The two numbers represent the beginning and ending pointers to the assigned chunk, respectively. In other

Table 3 Description of our implementation for all programs

AP	Name	Description	Reference
Matrix multiplication	matstat	Static scheduling	[9]
	matgss	Dynamic scheduling (GSS)	[10]
	matngss0	Fixed α scheduling + GSS	[17]
	matngss2	Adaptive α scheduling + GSS	[5]
	mathgss	Our hybrid scheduling + GSS	
	matfss	Dynamic scheduling (FSS)	[7]
	matnfss0	Fixed α scheduling + FSS	[17]
	matnfss2	Adaptive α scheduling + FSS	[5]
	mathfss	Our hybrid scheduling + FSS	
	mattss	Dynamic scheduling (TSS)	[16]
	matntss0	Fixed α scheduling + TSS	[17]
	matntss2	Adaptive α scheduling + TSS	[5]
	mathtss	Our hybrid scheduling + TSS	
Mandelbroot set	manstat	Static scheduling	[9]
	mangss	Dynamic scheduling (GSS)	[10]
	manngss0	Fixed α scheduling + GSS	[17]
	manngss2	Adaptive α scheduling + GSS	[5]
	manhgss	Our hybrid scheduling + GSS	
	manfss	Dynamic scheduling (FSS)	[7]
	mannfss0	Fixed α scheduling + FSS	[17]
	mannfss2	Adaptive α scheduling + FSS	[5]
	manhfss	Our hybrid scheduling + FSS	
	mantss	Dynamic scheduling (TSS)	[16]
	manntss0	Fixed α scheduling + TSS	[17]
	manntss2	Adaptive α scheduling + TSS	[5]
	manhtss	Our hybrid scheduling + TSS	
Circuit satisfiability	satstat	Static scheduling	[9]
	satgss	Dynamic scheduling (GSS)	[10]
	satngss0	Fixed α scheduling + GSS	[17]
	satngss2	Adaptive α scheduling + GSS	[5]
	sathgss	Our hybrid scheduling + GSS	
	satfss	Dynamic scheduling (FSS)	[7]
	satnfss0	Fixed α scheduling + FSS	[17]
	satnfss2	Adaptive α scheduling + FSS	[5]
	sathfss	Our hybrid scheduling + FSS	
	sattss	Dynamic scheduling (TSS)	[16]
	satntss0	Fixed α scheduling + TSS	[17]
	satntss2	Adaptive α scheduling + TSS	[5]
	sathtss	Our hybrid scheduling + TSS	

words, every node has a copy of the input matrices locally, so data communication is not significant in this kind of implementation. Therefore, communication cost between the master and the slave is low, and the dominant cost is the computation of matrix multiplication. The C/MPI code fragment of the slave module for matrix multiplication is listed as follows. As the source code shows, a column is the atomic unit of allocation. The workload for each column can be considered equal, which is composed of a fixed number of computation. Therefore, the distribution of workload can attain good performance.

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

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

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

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

First, execution time on the heterogeneous cluster for GSS group is investigated. Figure 2(a) illustrates execution time of static (matstat), dynamic (matgss) and our hybrid scheme (mathgss), with input matrix size 512×512, 1024×1024 and 1536×1536, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 1536×1536 got 37% and 26% performance improvement over the static one and the dynamic one, respectively.

Figure 2(b) illustrates execution time of previous hybrid schemes (matngss0 and matngss2) and our hybrid scheme (mathgss), with input matrix size 512×512, 1024×1024 and 1536×1536 respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 1536×1536 got 4% and 14% performance improvement over the static one and the dynamic one, respectively.

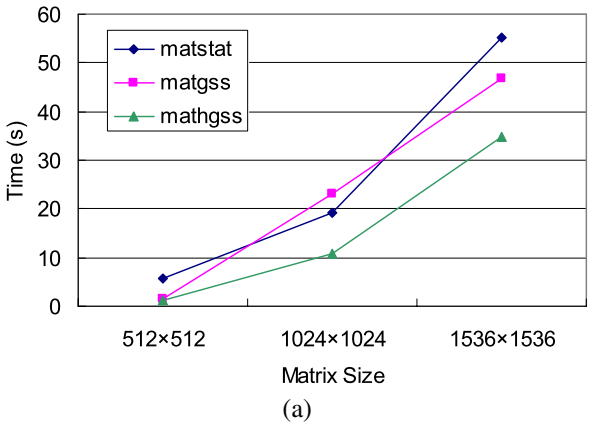


Fig. 2 Matrix multiplication execution time on the heterogeneous cluster for GSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matngss0, matngss2 and our mathgss

Next, execution time on the heterogeneous cluster for FSS group is investigated. Figure 3(a) illustrates execution time of static (matstat), dynamic (matfss) and our hybrid scheme (mathfss), with input matrix size 512 × 512, 1024 × 1024 and 1536 × 1536, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 1536 × 1536 got 38% and 1% performance improvement over the static one and the dynamic one, respectively.

Figure 3(b) illustrates execution time of previous hybrid schemes (matnfss0 and matnfss2) and our hybrid scheme (mathfss), with input matrix size 512 × 512, 1024 × 1024 and 1536 × 1536, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 1536 × 1536 got 5% and 15% performance improvement over the static one and the dynamic one, respectively.

Finally, execution time on the heterogeneous cluster for TSS group is investigated. Figure 4(a) illustrates execution time of static (matstat), dynamic (mattss) and our hybrid scheme (mathtss), with input matrix size 512 × 512, 1024 × 1024 and 1536 × 1536, respectively. Experimental results show that our hybrid scheduling scheme got

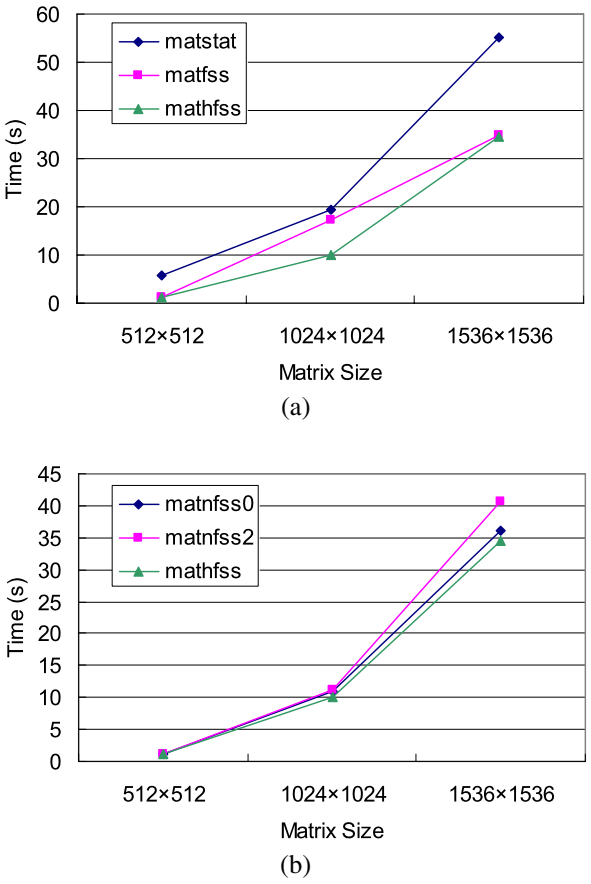


Fig. 3 Matrix multiplication execution time on the heterogeneous cluster for FSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matnfss0, matnfss2 and our mathfss

better performance than static and dynamic ones. In this case, our scheme for input size 1536 × 1536 got 23% and 13% performance improvement over the static one and the dynamic one, respectively.

Figure 4(b) illustrates execution time of previous hybrid schemes (matntss0 and matntss2) and our hybrid scheme (mathtss), with input matrix size 512 × 512, 1024 × 1024 and 1536 × 1536, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 1536 × 1536 got 6% performance improvement over the static one.

4.3 Application 2: Mandelbrot set computation

The Mandelbrot set is a problem involving the same computation on different data points which have different convergence rates [8]. 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,

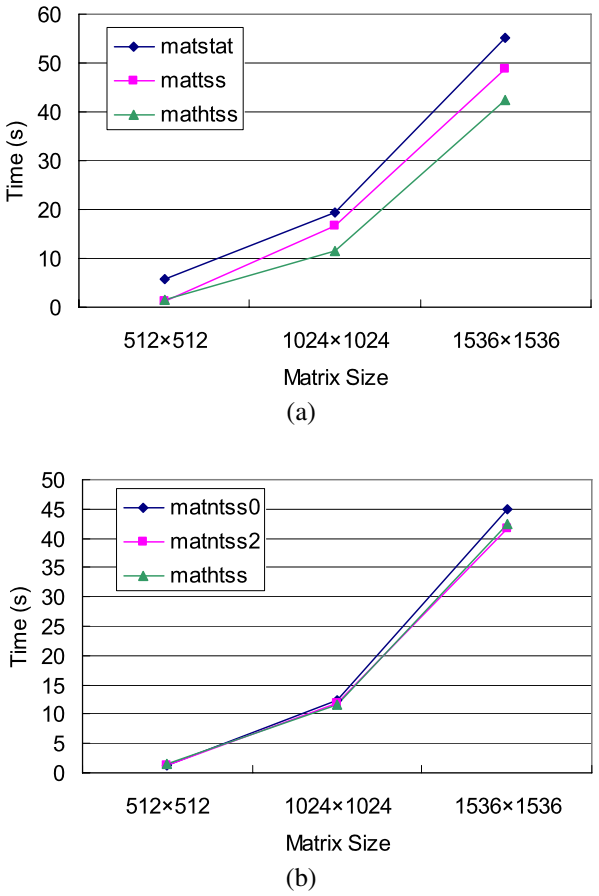


Fig. 4 Matrix multiplication execution time on the heterogeneous cluster for TSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matntss0, matntss2 and our mathtss

the small scale details are not identical to the whole. In fact, the Mandelbrot set is infinitely complex. Yet the process of generating it is based on an extremely simple equation involving complex numbers. This operation derives a resultant image by processing an input matrix, A, where A is an image of m pixels by n pixels. The resultant image is one of m pixels by n pixels.

The PLS scheme has been implemented for Mandelbrot set computation. The master module is responsible for the distribution of workload. When a slave node becomes idle, the master node sends two integers to the slave. As implemented in matrix multiplication, communication cost between the master and the slave is low, and the dominant cost is the computation of Mandelbrot set. The C/MPI code fragment of the slave module for Mandelbrot set computation is listed as follows. In this application, the workload for each iteration of the outer loop is irregular because the number of execution for convergence is not a fixed number. Therefore, the performance for workload distribution depends on the degree of variation for each iteration.

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

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

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

First, execution time on the heterogeneous cluster for GSS group is investigated. Figure 5(a) illustrates execution time of static (manstat), dynamic (mangss) and our hybrid scheme (manhgss), with input image size 64 × 64, 128 × 128 and 192 × 192, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 192 × 192 got 95% and 29% performance improvement over the static one and the dynamic one, respectively.

Figure 5(b) illustrates execution time of previous hybrid schemes (manngss0 and manngss2) and our hybrid scheme (manhgss), with input image size 64 × 64, 128 × 128 and 192 × 192, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 192 × 192 got no performance improvement over the static one and the dynamic one, respectively.

Next, execution time on the heterogeneous cluster for FSS group is investigated. Figure 6(a) illustrates execution time of static (manstat), dynamic (manfss) and our hybrid scheme (manhfss), with input image size 64 × 64, 128 × 128 and 192 × 192, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 192 × 192 got 95% and 27% performance improvement over the static one and the dynamic one, respectively.

Figure 6(b) illustrates execution time of previous hybrid schemes (mannfss0 and mannfss2) and our hybrid scheme (manhfss), with input image size 64 × 64, 128 × 128 and 192 × 192, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case,

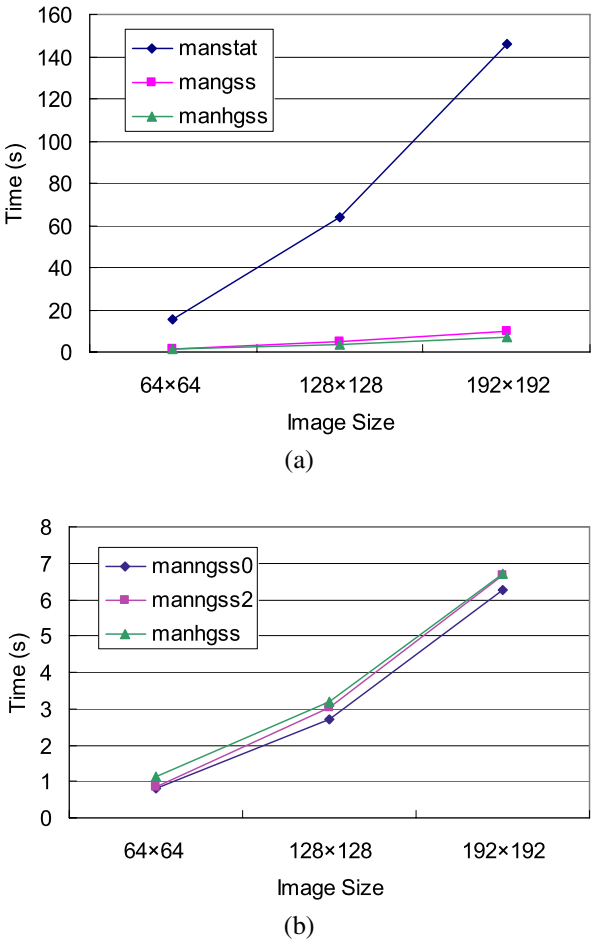


Fig. 5 Mandelbrot execution time on the heterogeneous cluster for GSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matngss0, matngss2 and our mathgss

our scheme for input size 192 × 192 got 4% and 15% performance improvement over the static one and the dynamic one, respectively.

Finally, execution time on the heterogeneous cluster for TSS group is investigated. Figure 7(a) illustrates execution time of static (manstat), dynamic (mantss) and our hybrid scheme (manhtss), with input image size 64 × 64, 128 × 128 and 192 × 192, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 192 × 192 got 95% performance improvement over the static one.

Figure 7(b) illustrates execution time of previous hybrid schemes (manntss0 and manntss2) and our hybrid scheme (manhtss), with input image size 64 × 64, 128 × 128 and 192 × 192, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 192 × 192 got no performance improvement over the static one and the dynamic one, respectively.

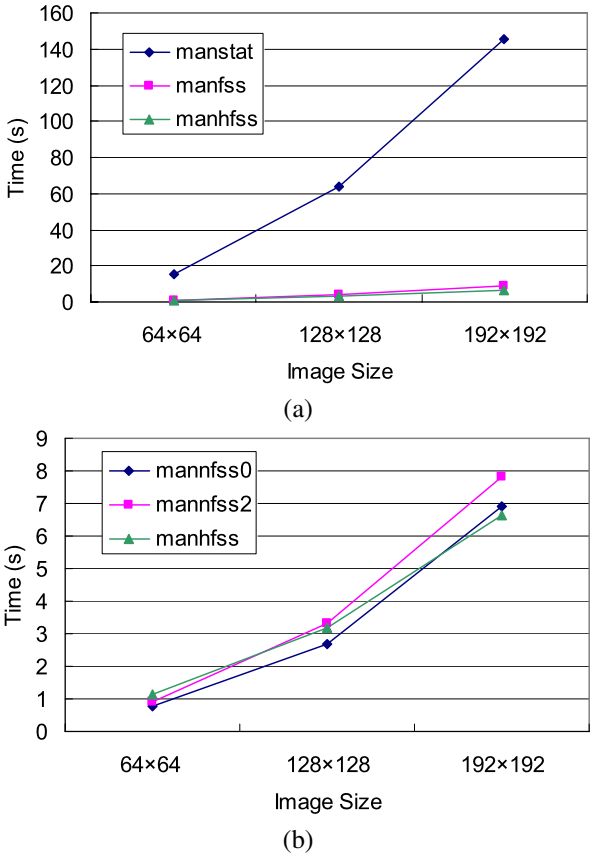


Fig. 6 Mandelbrot execution time on the heterogeneous cluster for FSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matnfss0, matnfss2 and our mathfss

4.4 Application 3: circuit satisfiability

The circuit satisfiability problem is one involving a combinational circuit composed of AND, OR, and NOT gates. Simply speaking, the question can be described as follows: is there an assignment of Boolean values to the inputs that makes the output to be 1? A circuit is satisfiable if there exists a set of Boolean input values that makes the output of the circuit to be 1. The circuit satisfiability problem is NP-complete, and no known algorithms can solve it in polynomial time. In the experiment, we find the solutions through an exhaustive search. This operation gets a number as input, which is the number of Boolean variables in the expression. After that, the algorithm exhaustively computes all combinations of these Boolean values.

The circuit satisfiability problem is also implemented in a similar way. The master module is responsible for the distribution of workload. When a slave node becomes idle, the master node sends two integers to the slave. As implemented in matrix multiplication, communication cost between the master and the slave is low, and the dominant cost is the computation of Mandelbrot set. The C/MPI code fragment of the slave module for Mandelbrot set computation is listed as follows. In this application, the workload for each iteration of the outer loop is irregular because the number of

execution for testing satisfiability is not a fixed number. Therefore, the performance for workload distribution depends on the degree of variation for each iteration.

```
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);
    }

    /* 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);
    }
```

First, execution time on the heterogeneous cluster for GSS group is investigated. Figure 8(a) illustrates execution time of static (satstat), dynamic (satgss) and our hybrid scheme (sathgss), with input variable numbers 14, 15 and 16, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 16 got 47% performance improvement over the static one.

Figure 8(b) illustrates execution time of previous hybrid schemes (satngss0 and satngss2) and our hybrid scheme (sathgss), with input size 14, 15 and 16, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 16 got no performance improvement over the static one and the dynamic one, respectively.

Next, execution time on the heterogeneous cluster for FSS group is investigated. Figure 9(a) illustrates execution time of static (satstat), dynamic (satfss) and our hybrid scheme (sathfss), with input variable numbers 14, 15 and 16, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 16 got 50% performance improvement over the static one.

Figure 9(b) illustrates execution time of previous hybrid schemes (satnfss0 and satnfss2) and our hybrid scheme (sathfss), with input size 14, 15 and 16, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 16 got no performance improvement over the static one and the dynamic one, respectively.

Finally, execution time on the heterogeneous cluster for TSS group is investigated. Figure 10(a) illustrates execution time of static (satstat), dynamic (sattss) and our

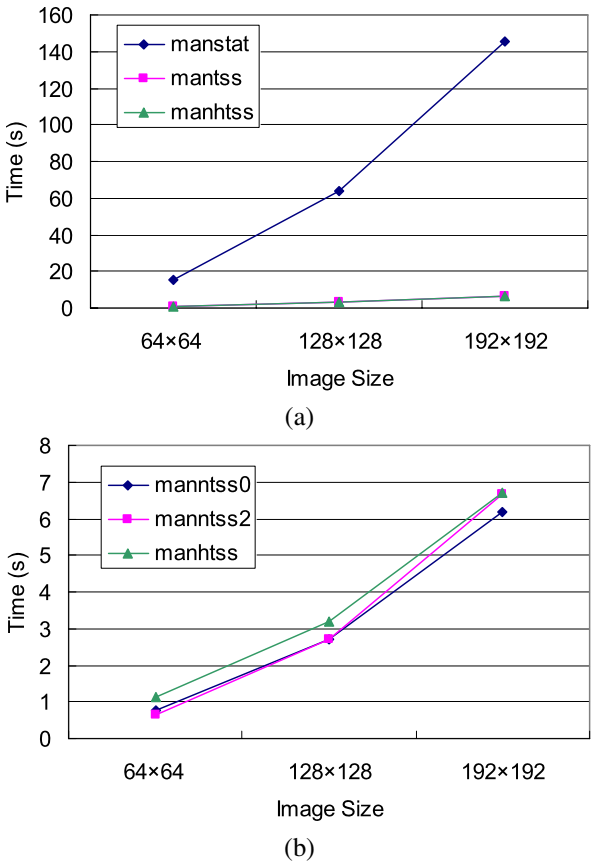


Fig. 7 Mandelbrot execution time on the heterogeneous cluster for TSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matntss0, matntss2 and our mathtss

hybrid scheme (sathtss), with input variable numbers 14, 15 and 16, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 16 got 49% performance improvement over the static one.

Figure 10(b) illustrates execution time of previous hybrid schemes (satntss0 and satntss2) and our hybrid scheme (sathtss), with input size 14, 15 and 16, respectively. Experimental results show that our hybrid scheduling scheme got better performance than static and dynamic ones. In this case, our scheme for input size 16 got no performance improvement over the static one and the dynamic one, respectively.

4.5 Performance on the homogeneous cluster

For comparison, execution time on the homogeneous cluster for GSS group is investigated. Figure 11 illustrates execution time of static (matstat), α scheduling (matngss0) and our hybrid scheme (mathgss), with input matrix size 512×512, 1024×1024, 1536×1536 and 2048×2048, respectively. Experimental results show that our hybrid scheduling scheme got slightly worse performance than static one.

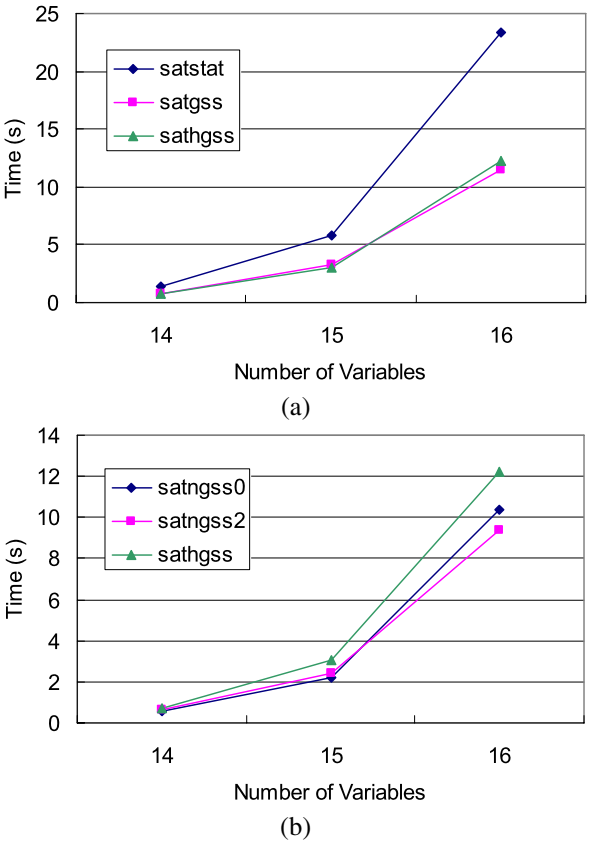
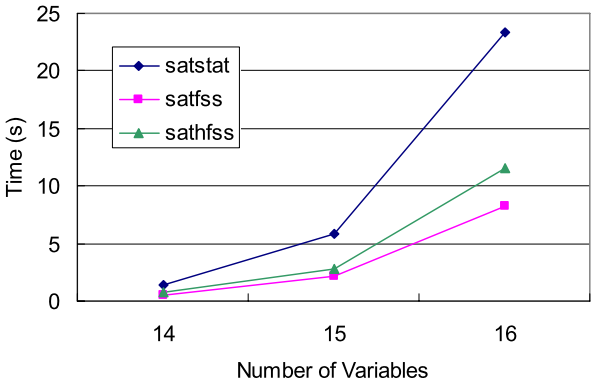


Fig. 8 Circuit satisfiability execution time on the heterogeneous cluster for GSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matngss0, matngss2 and our mathgss

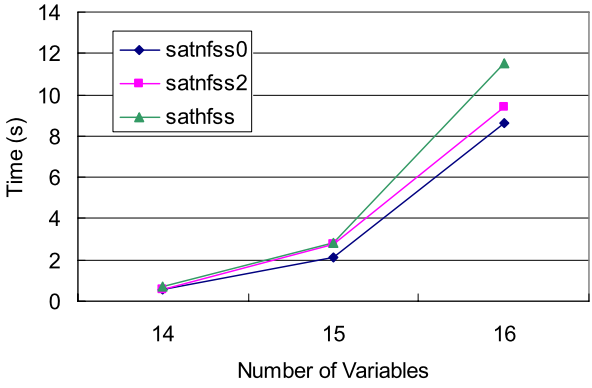
4.6 Summary and discussion

We have conducted experiments for three types of applications. In this section, experimental results are summarized, and descriptions for analysis and justification are given.

- Our HPLS got performance improvement on static scheduling schemes for heterogeneous clusters. Among these schemes, HPLS performs better than other schemes. The reason is that HPLS accurately estimates the PR, and takes the advantage of static scheduling, thus reducing the runtime overhead. The static scheme obviously performs worse than other dynamic schemes. It is reasonable to say that the static scheme is not suitable for a dynamic and heterogeneous environment, with respect to performance.
- Our scheme also got performance improvement on self-scheduling schemes for heterogeneous cluster environments. Traditional self-scheduling schemes (GSS, FSS and TSS) perform worse than HPLS. The reason is that HPLS takes heterogeneity of computing nodes into account. Traditional schemes might assign a large chunk of workload to a slower CPU, thus resulting in a bottleneck.



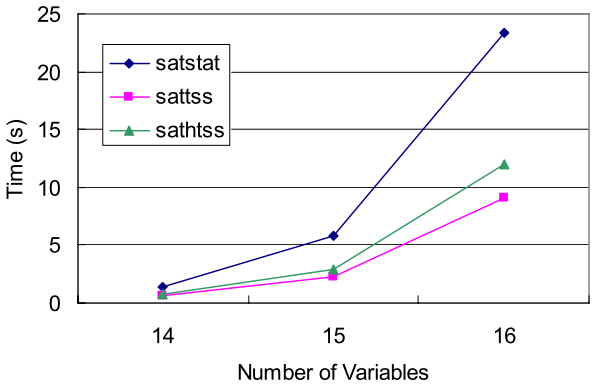
(a)



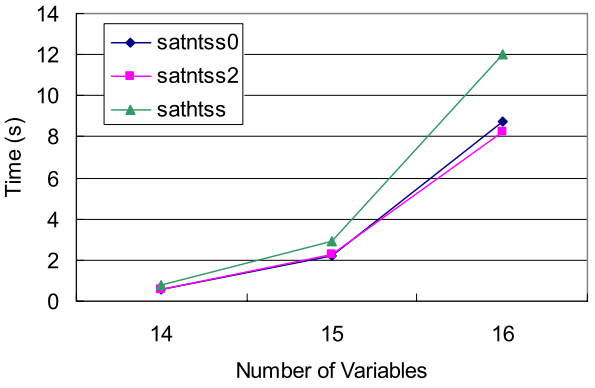
(b)

Fig. 9 Circuit satisfiability execution time on the heterogeneous cluster for FSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matnfss0, matnfss2 and our mathfss

- The workload distribution of matrix multiplication is more regular than the other two applications. When the workload of each iteration is roughly the same as other ones, it is easier for scheduling schemes to make scheduling decisions, attaining load balance. Although HPLS does not consider the effect of irregular workload, its two-phased dispatching strategy can alleviate the performance degradation probably brought by the irregularity.
- In homogeneous cluster environments, our scheme got slightly worse performance than static scheme did. The main reason is the overhead of HPLS for collecting and calculating performance ratio. The static scheme does not consider the heterogeneity of computing nodes, and always equally dispatches workload to each node. When the environment is homogeneous and the application has regular workload distribution, the static scheme can attain good performance. However, real world environments are usually not like this case. A small variation in the computing platform or workload distribution can result in disaster for the static approach.



(a)



(b)

Fig. 10 Circuit satisfiability execution time on the heterogeneous cluster for TSS group schemes. **a** Static, dynamic and our hybrid scheme; **b** hybrid schemes: matntss0, matntss2 and our mathtss

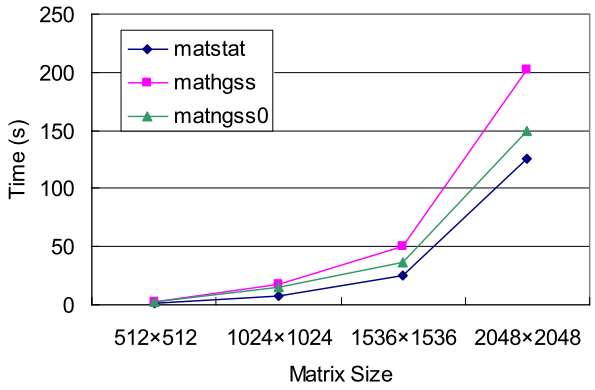


Fig. 11 Matrix multiplication execution time on the homogeneous cluster for GSS group schemes: static, matngss0 and our hybrid scheme

5 Conclusions and future work

In this paper, we propose a heuristic scheme, which combines the advantages of static and dynamic loop scheduling schemes, and compare it with previous algorithms by experiments on three types of application programs in heterogeneous cluster environment. In each case, our approach can obtain performance improvement on previous schemes. Besides, our approach is less sensitive to α values than previous schemes; in other words, it is more robust. In our future work, we will implement more types of application programs to verify our approach. Furthermore, we hope to find better ways of modeling the performance function, incorporating network information. Also, a theoretical analysis of the proposed method will be addressed.

References

1. Baker M, Buyya R (1999) Cluster computing: the commodity supercomputer. Int J Softw Pract Exp 29(6):551–575
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 Parallel Distrib Syst 16:207–218
3. Bennett BH, Davis E, Kunau T, Wren W (2000) Beowulf parallel processing for dynamic load-balancing. In: Proceedings of IEEE aerospace conference, vol 4, pp 389–395
4. Bohn CA, Lamont GB (2002) Load balancing for heterogeneous clusters of PCs. Future Gener Comput Syst 18:389–400
5. 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
6. 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
7. Hummel SF, Schonberg E, Flynn LE (1992) Factoring: a method scheme for scheduling parallel loops. Commun ACM 35:90–101
8. Introduction to the Mandelbrot set, http://www.ddewey.net/mandelbrot/
9. 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
10. Polychronopoulos CD, Kuck D (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans Comput 36(12):1425–1439
11. Post E, Goosen HA (2001) Evaluation 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)
12. 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
13. Sterling T, Bell G, Kowalik JS (2002) Beowulf cluster computing with Linux. MIT Press, Cambridge
14. 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
15. The Scalable Computing Laboratory (SCL), http://www.scl.ameslab.gov/
16. Tzen TH, Ni LM (1993) Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. IEEE Trans Parallel Distrib Syst 4:87–98
17. 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
18. 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
19. 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

Chao-Tung Yang received a B.S. degree in computer science and information engineering from Tunghai University, Taichung, Taiwan in 1990, and the M.S. degree in computer and information science from National Chiao Tung University, Hsinchu, Taiwan in 1992. He received the Ph.D. degree in computer and information science from National Chiao Tung University in July 1996. He won the 1996 Acer Dragon Award for outstanding Ph.D. Dissertation. He has worked as an associate researcher for ground operations in the ROCSAT Ground System Section (RGS) of the National Space Program Office (NSPO) in Hsinchu Science-based Industrial Park since 1996. In August 2001, he joined the faculty of the Department of Computer Science and Information Engineering at Tunghai University, where he is currently an associate professor. His researches have been sponsored by Taiwan agencies National Science Council (NSC), National Center for High Performance Computing (NCHC), and Ministry of Education. His present research interests are in grid and cluster computing, parallel and high-performance computing, and internet-based applications. He is both member of the IEEE Computer Society and ACM.

Wen-Chung Shih received a B.S. degree in Computer and Information Science from National Chiao Tung University in 1992 and an M.S. degree in Computer and Information Science from National Chiao Tung University in 1994. He is a Ph.D. candidate in the Computer Science Department at National Chiao Tung University, Taiwan. He passed the second class of the National Higher Examination in Information Processing field in 1994 and in Library Information Management field in 2004, respectively. Since 2004, he has worked as an executive officer in National Chi Nan University library, Taiwan. His research interests include parallel loop scheduling, grid computing, data mining, expert systems and e-learning.

Shian-Shyong Tseng received the Ph.D. degree in computer engineering from the National Chiao Tung University in 1984. Since August 1983, he has been on the faculty of the Department of Computer and Information Science at National Chiao Tung University, and is currently a Professor there. From 1988 to 1991, he was the Director of the Computer Center at National Chiao Tung University. From 1991 to 1992 and 1996 to 1998, he acted as the Chairman of Department of Computer and Information Science. From 1992 to 1996, he was the Director of the Computer Center at Ministry of Education and the Chairman of Taiwan Academic Network (TANet) management committee. From 1999 to 2003, he has participated in the National Telecommunication Project and acted as the Chairman of the Network Planning Committee, National Broadband Experimental Network (NBEN). From 2003 to 2006, he has acted as the principal investigator of the Taiwan SIP/ENUM trial project and the Chairman of the SIP/ENUM Forum Taiwan. In December 1999, he founded Taiwan Network Information Center (TWNIC) and was the Chairman of the board of directors of TWNIC from 1999 to 2005. Since August 2005, he is the Dean of the College of Computer Science, Asia University. He is also the Director of the e-learning and application research center at National Chiao Tung University. His current research interests include expert systems, data mining, computer-assisted learning, and internet-based applications. He has published more than 100 journal papers.