J Supercomput (2012) 59:42–60
DOI 10.1007/s11227-010-0418-y

Designing parallel loop self-scheduling schemes using the hybrid MPI and OpenMP programming model for multi-core grid systems

Chao-Chin Wu · Chao-Tung Yang · Kuan-Chou Lai · Po-Hsun Chiu

Published online: 12 March 2010

Abstract Loop scheduling on parallel and distributed systems has been thoroughly investigated in the past. However, none of these studies considered the multi-core architecture feature for emerging grid systems. Although there have been many studies proposed to employ the hybrid MPI and OpenMP programming model to exploit different levels of parallelism for a distributed system with multi-core computers, none of them were aimed at parallel loop self-scheduling. Therefore, this paper investigates how to employ the hybrid MPI and OpenMP model to design a parallel loop self-scheduling scheme adapted to the multi-core architecture for emerging grid systems. Three different featured applications are implemented and evaluated to demonstrate the effectiveness of the proposed scheduling approach. The experimental results show that the proposed approach outperforms the previous work for the three applications and the speedups range from 1.13 to 1.75.

Keywords Grid computing · Loop scheduling · Multi-core computer · MPI · OpenMP

C.-C. Wu (✉) · P.-H. Chiu
Department of Computer Science and Information Engineering, National Changhua University of Education, Changhua City, 500, Taiwan
e-mail: ccwu@cc.ncue.edu.tw

C.-T. Yang
High-Performance Computing Laboratory, Department of Computer Science and Information Engineering, Tunghai University, Taichung, 40704, Taiwan

K.-C. Lai
Department of Computer and Information Science, National Taichung University, Taichung City, 403, Taiwan

1 Introduction

As computers become more and more inexpensive and powerful, computational grids which consist of various computational and storage resources have become promising alternatives to traditional multiprocessors and computing clusters [6, 7]. Basically, grids are distributed systems which share resources through the Internet. Users can access more computing resources through grid technologies. However, bad management of grid environments might result in using grid resources in an inefficient way. Moreover, the heterogeneity and dynamic changing of the grid environment makes it different from conventional parallel and distributed computing systems, such as multiprocessors and computing clusters. Therefore, it becomes more difficult to utilize the grid efficiently.

Loop scheduling on parallel and distributed systems is an important problem, and has been thoroughly investigated on traditional parallel computers in the past [11, 12, 17, 22]. Traditional loop scheduling approaches include static scheduling and dynamic scheduling. The former is not suitable in dynamic environments. The latter, especially self-scheduling, has to be adapted to be applied to heterogeneous platforms. Therefore, it is difficult to schedule parallel loops on heterogeneous and dynamic grid environments. In recent years, several pieces of work have been devoted to parallel loop scheduling for cluster computing environments [1, 3, 4, 23–26, 28, 29], addressing the heterogeneity of computing power.

To adapt to grid systems, we have an approach to enhance some well-known loop self-scheduling schemes [27]. The HINT Performance Analyzer [10] is used to determine whether target systems are relatively homogeneous or relatively heterogeneous. We then partition loop iterations into four classes to achieve good performance in any given computing environment. Finally, a heuristic approach based upon α-based self-scheduling scheme is used to solve parallel regular loop scheduling problem on an extremely heterogeneous Grid computing environment. Furthermore, we have proposed another improved loop self-scheduling approach called PLS (Performance-based Loop Scheduling) for grid systems [18]. In this work, dynamic information acquired from a monitoring tool is utilized to adapt to the dynamic environment. Furthermore, a sampling method is proposed to estimate the proportion of the workload to be assigned statically.

Although our previous approaches improved system performance, they did not investigate multi-core architecture [18, 27] or they did not consider the feature of grid systems [23, 24]. Recently, more grid systems are including multi-core computers because nearly all commodity personal computers have the multi-core architecture. The primary feature of multi-core architecture is multiple processors on the same chip that communicate with each other by directly accessing the data in shared memory. Unlike multi-core computers, each computer in a distributed system has its own memory system and thus relies on a message-passing mechanism to communicate with other computers. The MPI library is usually used for parallel programming in the grid system because it is a message-passing programming model [14]. However, MPI is not the best programming model for multi-core computers. OpenMP is suitable for multi-core computers because it is a shared-memory programming model. Therefore, we propose using hybrid MPI and OpenMP programming mode to design

the loop self-scheduling scheme for a grid system with multi-core computers. Three different featured applications, Matrix Multiplication, sparse Matrix Multiplication and Mandelbrot set calculation, are implemented and evaluated to demonstrate the effectiveness of the proposed scheduling approach. These applications are featured according to whether the workload distribution is regular and whether data communication is needed at each scheduling step. The experimental results show that the proposed approach outperforms the PLS approach with the speedups ranging from 1.13 to 1.75.

The rest of this paper is organized as follows. In Sect. 2, we introduce several typical and well-known self-scheduling schemes. In Sect. 3, we describe how to employ the hybrid MPI and OpenMP programming model to develop the two-level self-scheduling schemes. 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 and future work are given in the last section.

2 Background review

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

2.1 Loop scheduling schemes

Various self-scheduling schemes have been proposed to achieve better load balance with less scheduling overhead, including Pure Self-Scheduling (PSS) [12], Chunk Self-Scheduling (CSS) [12], Guided Self-Scheduling (GSS) [17], Factoring Self-Scheduling (FSS) [11] and Trapezoid Self-Scheduling (TSS) [22]. The rules of calculating the next chunk size for the five self-scheduling schemes are shown in Table 1. Furthermore, Table 2 shows the chunk sizes for the five self-scheduling schemes with respect to a loop with 1000 iterations. The number of available processors is 4.

Table 1 Chunk sizes assigned at each time for five self-scheduling schemes

Scheme	Chunk size assigned at each time
PSS	1
CSS(k)	k, users have to specify the value of k
FSS	Allocation in phases. During each phase of FSS, only a subset of remaining loop iterations (usually half) is equally distributed to available processors
GSS	Dividing the number of the remaining iterations by the number of available processors
TSS(N_s, N_f, δ)	The chunk sizes decrease linearly. Users have to specify the first chunk size, N_s, and the last chunk size, N_f, and consecutive chunks difference, δ

Table 2 Partition sizes

Scheme	Partition size
PSS	1, 1, 1, 1, 1, 1, 1, ...
CSS(125)	125, 125, 125, 125, 125, 125, 125, 125
FSS	125, 125, 125, 125, 63, 63, 63, 63, 31, ...
GSS	250, 188, 141, 106, 79, 59, 45, 33, 25, ...
TSS(125, 1, 8)	125, 117, 109, 101, 93, 85, 77, 69, 61, ...

2.2 Grid computing and programming models

Grid computing [5, 6, 8] can be thought of as distributed and large-scale cluster computing and as a form of networked parallel processing. It can be confined to the network of computer workstations within a corporation or it can be a public collaboration. In this paper, Ganglia is utilized to acquire dynamic system information, such as CPU loading of available nodes.

MPI is a message-passing library standard that was published in May 1994. MPICH-G2 [15] is a grid-enabled implementation of the MPI v1.1 standard. In contrast, Open Multi-Processing (OpenMP), a kind of shared-memory architecture API [16], provides a multi-threaded capacity. A loop can be parallelized easily by invoking subroutine calls from OpenMP thread libraries and inserting the OpenMP compiler directives. In this way, the threads can obtain new tasks, the un-processed loop iterations, directly from local shared memory.

2.3 Related work

The EasyGrid middleware is a hierarchically distributed Application Management System (AMS) which is embedded automatically into a user's parallel MPI application without modifications to the original code by a scheduling portal [2]. Each EasyGrid AMS is a three-level hierarchical management system for application-specific self-scheduling and distinct scheduling policies can be used at each level, even within the same level. A low intrusion implementation of a hybrid scheduling strategy has been proposed to cope with the dynamic behavior of grid environments.

TITAN is a multi-tiered scheduling architecture for grid workload management, which employs a performance prediction system (PACE) and task distribution brokers to meet user-defined deadlines and improve resource usage efficiency [19]. The PACE system is developed to facilitate model generation for applications. The scheduler uses the evaluation engine to identify expected execution run-time from the application models and from the resource models that represent the cluster or multiprocessor system of homogeneous processing nodes.

Herrera et al. proposed a new loop distribution scheme to overcome the following three limitations when the MPI programming model is adopted in computational Grids [9]. (1) All required resources must be simultaneously allocated to begin execution of the application. (2) The whole application must be restarted when a resource fails. (3) Newly added resources cannot be allocated to a currently running application. Their approach is implemented using the Distributed Resource Management Application API (DRMAA) standard and the GridWay meta-scheduling framework.

Chronopoulos et al. proposed a distributed self-scheduling algorithm which takes CPU computation powers as weights that scale the size of iterations each computer is assigned to compute [2, 3]. Their method can be used to improve the performance of various self-scheduling algorithms.

This paper is a continuation of our earlier work. We proposed several loop self-scheduling schemes for heterogeneous cluster systems [23–26, 28, 29]. In our first work [25], a heuristic was proposed for distributing workloads according to CPU performance when loops are regular. It partitions loop iterations in two phases. In the first phase, $\alpha\%$ of the workload is partitioned according to performance weighted by CPU clock. In the second phase, the remaining $(100 - \alpha)\%$ of the workload is distributed according to a conventional self-scheduling scheme. However, many factors influence system performance, including CPU clock speed, available memory, communication cost, and so forth. Therefore, in our second work, we tried to evaluate computer performance using the HINT benchmark [26]. We also adjusted the value of α adaptively according to the heterogeneity of the cluster. In the most recent scheme [28], we use application execution times to estimate performance function values for all nodes. The performance function values are then used for iteration partitioning. We also proposed a loop self-scheduling approach based on the hybrid MPI and OpenMP programming model for cluster systems [23, 24]. For each scheduler of the proposed two-level software architecture, each application adopts one kind of well-known self-scheduling schemes to allocate loop iterations in one phase. In [29], we proposed an approach that combines both the advantages of static and dynamic schemes and uses the HPC Challenge Benchmarks to define each node's performance weighting. Because grid systems have long communication latency and their computing resources are highly heterogeneous, this paper proposes a new algorithm especially for grid systems.

3 Hierarchical loop scheduling

3.1 The main idea

A grid system is comprised of multiple computational nodes connected by the Internet. If multi-core computational nodes are included, this kind of system can be regarded as a two-level hierarchical structure. The first level consists of computational nodes and the second level consists of processor cores. Because each computational node has its own memory system and address space, it must communicate with others by explicitly sending messages through the Internet. In contrast, because the processor cores on a multi-core computational node all share the same memory, they can communicate with each other by accessing the data in shared memory. Accordingly, the communications can be divided into inter-node and intra-node communications. Because the inter-node communications have longer latencies than the intra-node communications, the former should be minimized for optimized communications. However, the previously proposed self-scheduling schemes for grid systems totally ignored such a communication issue [18, 27]. Intra-node communications are all based on message-passing paradigms, rather than share-memory paradigms.

Each computational node runs an MPI process regardless how many processor cores it has. However, OpenMP is used for intra-node communications. Each MPI process will fork OpenMP threads depending on the number of processor cores in its underlying computational node. Every processor core runs one OpenMP thread. OpenMP is a shared-memory multi-threaded programming model, which matches the multi-core computational node feature.

The scheduling scheme we proposed consists of one global scheduler and multiple local schedulers. Each worker computational node has one local scheduler. One processor core from each worker computational node is responsible for local scheduler execution. The processor core running the local scheduler is called the master core and the others are called worker cores. The global scheduler and the local schedulers are all MPI processes.

In the first-level scheduling, the global scheduler is responsible for deciding how many iterations will be assigned whenever a local scheduler issues a request. The number of processor cores in the computational node, from which the request comes, should be taken into consideration when the decision is made. In the second-level scheduling, because all of the processor cores are homogeneous, the local scheduler dispatches the iterations assigned by the global scheduler to all processor cores primarily based on whether the iteration workload is regular or not. The iteration workload of a loop is *regular* if the difference between the execution times for any two of the iterations is very small. Otherwise, the iteration workload is *irregular*. Basically, static scheduling is preferred in the second level. However, dynamic scheduling is adopted if the iteration workload distribution is irregular. Nevertheless, the default OpenMP built-in self-scheduling scheme, i.e., static scheduling, is adopted for the second level in this work.

3.2 Proposed approach

In the first-level scheduling, we propose a two-phase scheduling approach as follows. In the first phase, the *SWR* (Static-Workload Ratio) and Performance Ratio are calculated and then *SWR* percentage of the total workload is statically scheduled according to the performance ratio among all worker nodes [18]. In the second phase, the remainder of the workload is dynamically scheduled using any well-known self-scheduling scheme such as CSS, GSS, FSS or TSS. When an MPI process requests new iterations at each scheduling step, we must take the number of processor cores into consideration when the master core determines the number of iterations to be allocated for the worker core because the assigned iterations will be processed by parallel OpenMP threads. If there are p_i processor cores in the computational node i, the master will use the applied self-scheduling scheme, such as GSS, to calculate the total number of iterations by adding up the next p_i allocations. For instance, if there are 4 processor cores in a computational node and the CSS scheme with a chunk size of 128 iterations adopted, the master will assign 512 iterations whenever the MPI process running on the computational node asks for new iterations.

In the second-level scheduling, the local scheduler dispatches the iterations assigned by the global scheduler to the parallel OpenMP threads by invoking the OpenMP built-in scheduling routine. The scheduling scheme can be any one of

the following OpenMP built-in schemes: static, GSS and CSS schemes. Note that there is implicit barrier synchronization at the end of every parallel OpenMP section, which will cause additional run-time overhead. Whenever all assigned iterations are processed by OpenMP threads, the local scheduler issues another request to the global scheduler for the next chunk of iterations.

We describe the performance function used in this paper as follows. Let M denote the number of computational nodes, P denote the total number of processor cores. Computational node i is represented by m_i, and the total number of processor cores in computational node m_i is represented by p_i, where $1 \leq i \leq M$. In consequence, $P = \sum_{i=1}^{M} p_i$. The jth processor core in computational node i is represented by c_{ij}, where $1 \leq i \leq M$ and $1 \leq j \leq p_i$. N denotes the total number of iterations in some application program and $f()$ is an allocation function to produce the chunk size at each step. The output of f is the chunk size for the next iteration. At the sth scheduling step, the global scheduler computes the chunk size C_s for the computational node i and the remaining number of tasks R_s,

$$R_0 = N, \qquad C_s = f(s, i), \qquad R_s = R_s - 1 - C_s, \tag{1}$$

where $f()$ possibly has more parameters than just s and i, such as R_{i-1}. The concept of performance ratio is previously defined in [20, 21] in different forms and parameters, according to the requirements of applications. In this work, a different formulation is proposed to model the heterogeneity of the dynamic grid nodes.

The purpose of calculating performance ratio is to estimate the current capability of processing for each node. With this metric, we can distribute appropriate workloads to each node, and load balancing can be achieved. The more accurate the estimation is, the better the load balance is.

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

$$\mathrm{PF}_i(V_1, V_2, \ldots, V_X), \tag{2}$$

where V_r, $1 \leq r \leq X$, is a variable of the performance function. In this paper, our PF for a computational node i is defined as

$$\mathrm{PF}_i = \frac{\sum_{k=1}^{p_i} \frac{\mathrm{CS}_{ik}}{\mathrm{CL}_{ik}}}{\sum_{q=1}^{M} \sum_{k=1}^{p_i} \frac{\mathrm{CS}_{qk}}{\mathrm{CL}_{qk}}}, \tag{3}$$

where CS_{ij} is the CPU clock speed of processor core j in computational node i, and it is a constant attribute. The value of this parameter is acquired by the MDS service; CL_{ij} is the CPU loading of processor core j in computational node i, and it is a variable attribute. The value of this parameter is acquired by the Ganglia tool.

The performance ratio (PR) is defined to be the integer-ratio of all performance functions. For instance, assume the values of PFs of three nodes are 1/2, 1/3 and 1/4. Then, 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.

We also propose to use a parameter, SWR, to alleviate the effect of irregular workload. In order to take advantage of static scheduling, SWR percentage of the total workload is dispatched according to Performance Ratio. We propose to randomly take five sampling iterations, and compute their execution time. Then, the SWR of the target application i is determined by the following formula.

$$\mathrm{SWR} = \frac{\min_i}{\mathrm{MAX}_i} \tag{4}$$

where $\min_i$ is the minimum execution time of all sampled iterations for application i; MAX_i is the maximum execution time of all sampled iterations for application i. If the workload of the target application is regular, SWR can be set to be 100. However, if the application has an irregular workload, it is efficient to reserve some amount of workload for load balancing.

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

4 Performance evaluations

To verify our approach, we constructed a Grid system consisting of four sites, 19 computational nodes and 49 processor cores. The configurations of the constituent computational nodes are shown in Tables 3 and 4 in Appendix. The four sites are built at four educational institutions, including National Changhua University of Education (NCUE), Tatung University (TTU), National Taichung University (NTCU) and A-Lien Elementary School (ALES). The locations, machine types, node counts and core counts per node of the four Grid sites are shown in Table 3 in Appendix. These four Grid sites are distributed over Taiwan as shown in Fig. 8 in Appendix.

Three types of application programs are implemented to verify our approach in this testbed: Matrix Multiplication, sparse matrix multiplication and Mandelbrot set computation. The Performance-based Loop Scheduling (PLS) proposed by Yang et al. [18] is compared with our approach, Hierarchical Loop Scheduling (HLS). The PLS approach adopts the single-level, two-phase scheduling method. The pure MPI programming paradigm is used for parallel programming in PLS. The two-phase scheduling adopted by PLS is similar to that in HLS except the following two designs. (1) Each processor core, instead of a computational node, has its own performance ratio. Therefore, the performance functions adopted by PLS and HLS are different. (2) GSS is the only scheme adopted in the second phase. In the following subsections, the experimental results from the three applications are presented.

4.1 Application 1: Matrix Multiplication

We implemented the proposed scheme for Matrix Multiplication. The input matrix A will be partitioned into a set of rows and kept in the global scheduler. At run-time, after the global scheduler decides which rows will be assigned at each scheduling step, the corresponding row data will be sent to the requesting worker process. On the other hand, every local scheduler has a copy of the input matrix B because it is needed to calculate every row of matrix C. Since the result of one row of matrix C can be derived from multiplying one row of matrix A by the whole matrix B, Matrix Multiplication can be parallelized in this way to become an embarrassingly parallel computation.

The global scheduler, viz. the Master module, is responsible for the workload distribution to the local schedulers, viz. worker nodes. When a local scheduler becomes idle, the global scheduler sends the local scheduler one integer indicating how many rows will be assigned. The global scheduler then sends the corresponding data to the local scheduler. Finally, the OpenMP threads will follow the specified scheduling scheme such as guided self-scheduling to calculate the assigned rows. The Matrix Multiplication application has a regular workload distribution and requires data communication at each scheduling step. The C/MPI+OpenMP code fragment of the Worker module for Matrix Multiplication is listed as shown in Fig. 1. As the source code shows, a row is the atomic unit of allocation.

We compare Yang's proposed PLS approach and our proposed HLS approach using execution time, as shown in Fig. 2. The label PLS(X) in the legend denotes Yang's proposed PLS approach [27] where the X scheme is adopted in the second phase. The label HLS(X) in the legend represents our proposed HLS approach where the X scheme is adopted by the global scheduler in the second phase. In this experiment, HLS outperforms PLS if the same self-scheduling scheme is adopted. Therefore, the local HLS scheduler plays a key role in performance. In HLS, only the local scheduler, rather than every processor core, will request tasks from the global scheduler, resulting in reduced message traffic between master and worker cores at a computational node. Furthermore, the local scheduler will adopt the OpenMP built-in self-scheduling scheme to assign the received tasks to all the processor cores in the same node. The local assignment is through low-latency shared memory.

Observe the case that the matrix size is 4096×4096. The FSS scheme provides the best performance for PLS while the TSS scheme has the shortest execution time for HLS. Theoretically, FSS and TSS are both designed to address the problem that GSS might assign too much work to the first few processors. The FSS feature assigns iterations in phases. During each FSS phase, only a subset of the remaining loop iterations (usually half) is equally distributed to the available processors. In contrast, the chunk sizes decrease linearly in TSS. For PLS, each processor core requests tasks from the global scheduler. If FSS is adopted in PLS, each processor core in the same computational node acquires the same number of iterations during the same phase. However, if TSS is adopted for PLS, processor cores in the same computational node acquire different numbers of iterations even though they are assigned consecutively. Because the workload distribution of Matrix Multiplication is regular, PLS favors FSS. On the other hand, for HLS, only the local scheduler, rather than

```
WORKER Module /*worker*/
int **partial_C,**partial_A;
omp_set_num_threads(num_core); /* create the threads only once */
/* Do the following instructions in the first phase*/
Receive the job_size and initialize the matrix_size;
Receive the source data and compute by OpenMP parallel thread methods;
Return the resultants to the master;
do{
    MPI_Send(&req_tag,1,MPI_INT,source,tag,MPI_COMM_WORLD);
    MPI_Recv(&termination,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    MPI_Recv(&job_size,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    partial_A = (int **)calloc(job_size,sizeof(int *));
    partial_C = (int **)calloc(job_size,sizeof(int *));
    for (i=0;i<job_size;i++) {
       partial_A[i] = (int *)calloc(Row_SIZE,sizeof(int ));
       partial_C[i] = (int *)calloc(Row_SIZE,sizeof(int ));
    }
    for(i=0;i<job_size;i++){
       MPI_Recv(&partial_A[i], Row_SIZE, MPI_INT, source,tag,
                  MPI_COMM_WORLD, &status);
    }
        /* computing*/
    #pragma omp parallel private(j,k){
        #pragma omp for schedule(static)  /* the method can be changed*/
                for (i=0;i<job_size;i++)
                    for (j=0;j< Row_SIZE;j++)
                        for (k=0;k< Row_SIZE;k++)
                            *(partial_C[i]+j) += (*(partial_A[i]+k))*(*(B[k]+j));
    }
        /*send results to the master*/
     MPI_Send(&job_size,1, MPI_INT, 0, myid, MPI_COMM_WORLD);
     for (i=0;i<job_size;i++) {
        MPI_Send(partial_C[i], Row_SIZE,MPI_INT,source,tag,MPI_COMM_WORLD);
     }
} while(!termination);
```

Fig. 1 The local scheduler algorithm of Matrix Multiplication

every processor core, requests tasks from the global scheduler. After receiving tasks form the global scheduler, the local scheduler assigns the tasks to all processor cores in the same computational node. Therefore, no matter the global scheduler adopts

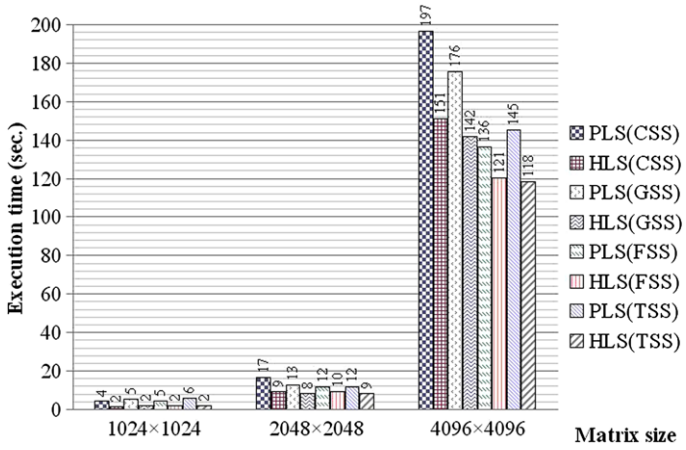

Fig. 2 Execution time comparisons for Matrix Multiplication. The chunk size of CSS is 64

Fig. 3 The algorithm of sparse Matrix Multiplication

```
for (i=0;i<job_size;i++)
    for (j=0;j< Row_SIZE;j++)
        for (k=0;k< Row_SIZE;k++)
            if((*(partial_A[i]+k)) != 0)
                *(partial_C[i]+j) += (*(partial_A[i]+k))*(*(B[k]+j));
```

FSS or TSS in HLS, the local scheduler can balance workload distribution among the processor cores in the same computational node. In fact, HLS(FSS) and HLS(TSS) have similar execution times while PLS(FSS) has much shorter execution time than that of PLS(TSS).

4.2 Application 2: Sparse Matrix Multiplication

Sparse Matrix Multiplication is the same as Matrix Multiplication, as described in Sect. 4.1, except that the input matrix A is a sparse matrix. Assume that 50% of elements in matrix A are zero and all the zeros are in the lower rectangular. If an element in matrix A is zero, the corresponding calculation is omitted as shown in Fig. 3. Therefore, Sparse Matrix Multiplication has irregular workload distribution and requires data communication at each scheduling step.

We compare Yang's proposed PLS approach and our proposed HLS approach by execution time, as shown in Fig. 4. If the same self-scheduling scheme is adopted by the global scheduler, HLS always outperforms PLS. Although FSS and TSS are both proposed to improve GSS performance, FSS cannot outperform GSS for PLS when the matrix size is 1024×1024 or 2048×2048. This is because the workload distribution of Sparse Matrix Multiplication is irregular but FSS assigns a subset of remaining loop iterations equally to available processors during each phase. Only when the matrix size is large enough can FSS outperform GSS. On the other hand, FSS and TSS both outperform GSS for HLS. For any matrix size, TSS always provides the best performance for both PLS and HLS.

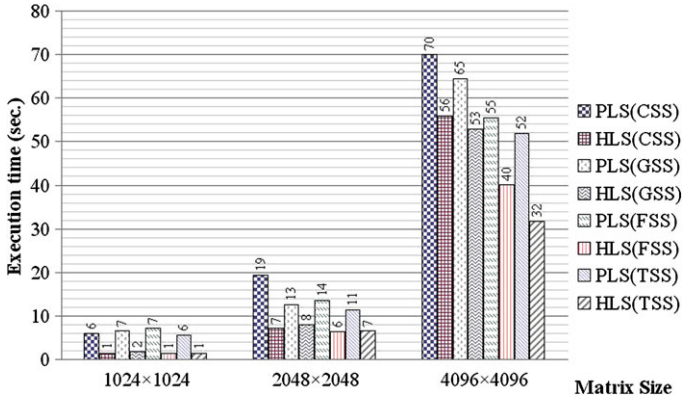

Fig. 4 Execution time comparisons for Sparse Matrix Multiplication. The chunk size of CSS is 64

4.3 Application 3: Mandelbrot set computation

This operation derives a resultant image by processing an input matrix, A, where A is an image of m pixels by n pixels [13]. The resultant image is one of m pixels by n pixels.

The proposed scheme was implemented for Mandelbrot set computation. The global scheduler is responsible for the workload distribution. When a local scheduler becomes idle, the global scheduler sends two integers to the local scheduler. The two numbers represent the beginning index and the size of the assigned chunk, respectively. The tasks assigned to the local scheduler are then dispatched to OpenMP threads based on a specified self-scheduling scheme. Unlike Matrix Multiplication, the communication cost between the global scheduler and local scheduler is low, and the dominant cost is the Mandelbrot set computation. The C/MPI+OpenMP code fragment for the local scheduler for Mandelbrot set computation is listed as shown in Fig. 5. In this application, the workload for each outer loop iteration is irregular because the number of executions for convergence is not a fixed number. Moreover, it requires no data communication at each scheduling step. Therefore, the performance for workload distribution depends on the degree of variation for each iteration.

We compare Yang's proposed PLS approach and our proposed HLS approach using execution time, as shown in Fig. 6. If the same self-scheduling scheme is adopted by the global scheduler, HLS will outperform PLS. Surprisingly, CSS has the best performance for PLS. The reason is described as follows. The long communication latency in Grid increases the scheduling overhead. When the execution time for running one chunk of iterations is too short, the scheduling overhead for such a small chunk has a significantly negative effect on the overall performance. Unlike GSS, FSS and TSS, the chunk size for CSS is fixed to 64 iterations at each scheduling step, avoiding assigning too small chunks in the end of scheduling.

On the other hand, TSS rather than CSS provides the best performance for HLS. It is because HLS uses another way to avoid assigning too small chunks to one computational node in the end of scheduling. In HLS, all the processor cores in the same computational node rely on the local scheduler to issue requests for them. Conse-

```
WORKER Module /* worker */
double scale_real,scale_imag,max_real=2,max_imag=2,min_real=-2,min_imag=-2;
int **resultant;
struct complex c;
scale_real=(double) (max_real - min_real) /(double) (IMAGE_SIZE);
scale_imag=(double) (max_imag - min_imag) /(double) (IMAGE_SIZE);
omp_set_num_threads(num_core);
/* Do the following instructions in the first phase*/
Receive the job_size and head;
Compute by OpenMP parallel thread methods;
Return the resultants to the master;
do{
    MPI_Send(&req_tag,1,MPI_INT,source,tag,MPI_COMM_WORLD);
    MPI_Recv(&termination,1,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    MPI_Recv(&arr[0],2,MPI_INT,source,tag,MPI_COMM_WORLD,&status);
    head = arr[0];
    job_size = arr[1];
    resultant = (int **)calloc(job_size,sizeof(int *));
    for (i=0;i<job_size;i++)
       resultant[i] = (int *)calloc(IMAGE_SIZE,sizeof(int ));

    #pragma omp parallel private(c,x){
        #pragma omp for schedule(guided,1)        /* the method can be changed */
            for (y=0; y<job_szie; y++)
                for (x= 0; x<IMAGE_SIZE; x++){
                    c.real = min_real + ((double) x * scale_real);
                    c.imag = min_imag + ((double) (y+head) * scale_imag);
                    *(resultant[y]+x) = cal_pixel(c);
                }
     }
    /*send results to the master*/
   MPI_Send(&job_size,1,MPI_INT,0,myid,MPI_COMM_WORLD);
   for (i=0;i<job_size;i++) {
    MPI_Send(resultant[i],IMAGE_SIZE,MPI_INT,source,tag,MPI_COMM_WORLD);
} while(!termination);
```

Fig. 5 The local scheduler algorithm of Mandelbrot set computation

quently, the amount of message traffic is significantly reduced. Furthermore, at each scheduling step, the size of the chunk assigned by the global scheduler to a multi-core computational node is proportional to the number of processor cores in the node. In

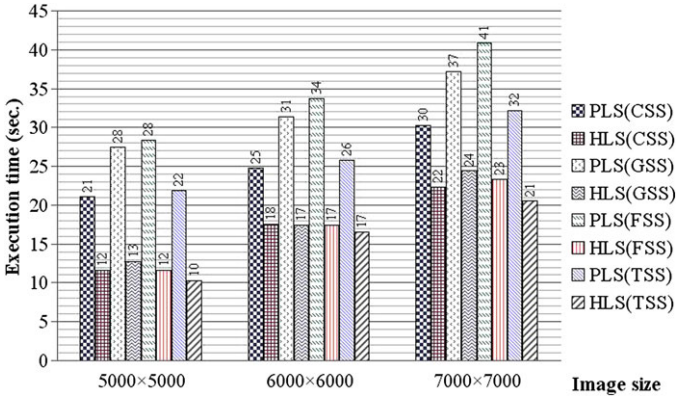

Fig. 6 Execution time comparisons for Mandelbrot set computation. The chunk size of CSS is 64

effect, HLS uses a single message to assign a larger task to all processor cores in the same node. Consequently, HLS can reduce the overhead of each scheduling step by increasing the chunk size near the end of scheduling.

4.4 Summary

According to the results shown in the above three subsection, HLS outperforms PLS for three different kinds of applications no matter which self-scheduling scheme is adopted. Furthermore, TSS is the best choice for the global scheduler in HLS because it requires the shortest execution time for any one of the three applications.

We summarize the performance improvements obtained by our proposed HLS for three applications in Fig. 7. The speedup is derived from dividing the execution time for PLS by the execution time for HLS, where the same self-scheduling scheme is adopted. In the figure, MM represents Matrix Multiplication of size 4096×4096, SMM represents Sparse Matrix Multiplication of size 4096×4096, and MS represents Mandelbrot set computation of size 7000×7000.

For the Matrix Multiplication, it has regular workload distribution and requires data communications at each scheduling step. The speedups range from 1.13 to 1.30. Because FSS is the best choice for PLS, the speedup of HLS over PLS is the smallest if FSS is adopted. On the other hand, if CSS is adopted, HLS has the best speedup with a factor of 1.30. For the Sparse Matrix Multiplication, it has irregular workload distribution and requires data communications at each scheduling step. Compared with Matrix Multiplication, HLS can provide better speedups for Sparse Matrix Multiplication. In other words, HLS is more effective than PLS for irregularly workload distributed applications requiring data communication at each scheduling step. The speedups range from 1.22 to 1.63. HLS can provide the best speedup when TSS is adopted. For the Mandelbrot set computation, it has irregular workload distribution and requires little data communications at each scheduling step. The speedups range from 1.36 to 1.75. The best speedup is provided when FSS is adopted. It is because FSS is not suitable for this kind of applications when PLS is applied because too much

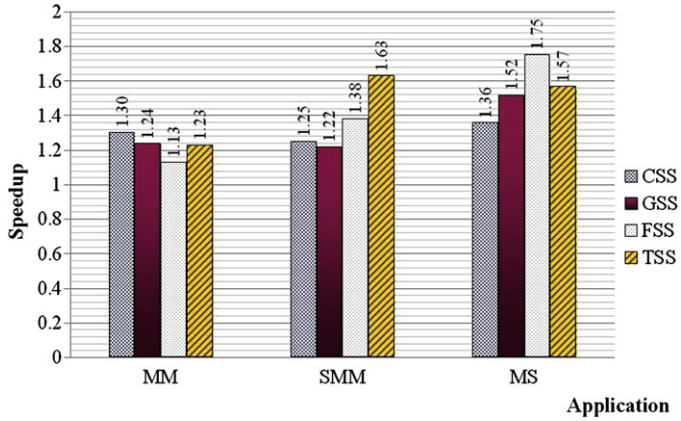

Fig. 7 Speedup comparisons for three applications. MM represents the Matrix Multiplication application. The matrix size is 4096 × 4096. SMM represents the Sparse Matrix Multiplication of 4096 × 4096 matrix size. MS represents the Mandelbrot set computation of 7000 × 7000 image size

scheduling overhead near the end of scheduling. Compared with the other two applications, HLS can provide the best performance improvements for all self-scheduling schemes except TSS. If TSS is adopted, HLS can improve the performance of Sparse Matrix Multiplication most.

5 Conclusions and future work

This paper uniquely investigated how to employ the hybrid MPI and OpenMP programming mode to design parallel loop self-scheduling schemes for emerging grid systems with multi-core computational nodes. The proposed scheduling approach, called HLS, is based on our previous work adopting the pure MPI model. In the proposed approach, only one MPI process will be created in each computational node no matter how many processor cores it has. The MPI process will request new loop iterations from the master MPI process. After receiving the assigned iterations at each scheduling step, the MPI process will fork OpenMP threads for parallel processing on the iterations. One OpenMP thread is created for each processor core. The MPI process will return the results to the master MPI process whenever the assigned iterations are finished. Because the iterations assigned to one MPI process will be processed in parallel by the processors cores in the same computational node, the number of loop iterations to be allocated to one computational node at each scheduling step also depends on the number of processor cores in that node.

Three applications, Matrix Multiplication, Sparse Matrix Multiplication, and Mandelbrot set computation, have been implemented to verify the effectiveness of the proposed approach. These applications were executed and evaluated in a Taiwan-wide grid system. The experimental results showed that the proposed approach outperforms the previous work for all three applications regardless which self-scheduling scheme was adopted by the global scheduler. Which extant self-scheduling scheme

is the best choice depends on the characteristics of the respective application. The speedups range from 1.13 to 1.75.

In our future work, we will implement more application program types to verify our approach. Moreover, although the experiments are carried out on a Taiwan-wide Grid system, the average parallelism per node is only 2.6, i.e., 49/19. We will study the performance impact when the computational nodes have more cores. According to our experiences, allocating too much workload at a time may degrade the performance for GSS. Therefore, it is possible that we have to design a new approach when the computational nodes have more cores. Furthermore, we hope to find better ways of modeling performance weighting using factors, such as amount of memory available, memory access costs, network information, and CPU loading. We will also address theoretical analysis of the proposed method.

Acknowledgements The authors would like to thank the National Science Council, Taiwan, for financially supporting this research under Contract No. NSC98-2221-E-018-008-MY2 and NSC98-2220-E-029-004.

Appendix

Table 3 The configuration of our Grid system—part 1

	E2160 TS Series	*Q6600 TS Series*	*E5405 RS Series*	*E5320 RS Series*
Processor	Intel Core 2 Duo E2160 *1	Intel Core 2 Quad Q6600 *1	Intel Xeon E5405 *2	Intel Xeon E5320 *2
Clock Rate	1809 Mhz	2394 Mhz	1995 Mhz	1862 Mhz
Core Count per Node	2 cores per processor	4 cores per processor	4 cores per processor	4 cores per processor
Memory	DDR2-667 512 MB *1	DDR2-533 2048 MB Register ECC *1	DDR2-667 1024 MB Register ECC *3	DDR2-1066 1024 MB Register ECC *4
Hard Disc	SATA 80 GB	SATA 160 GB	SATA-II 320 GB	SATA-II 320 GB
Swap Space	1024 MB	1024 MB	1024 MB	1024 MB
OS	Slackware Linux 12.2	Slackware Linux 12.2	Slackware 12.2	Slackware Linux 12.2
Kernel	2.6.27.7-smp	2.6.27.7-smp	2.6.27.7-smp	2.6.27.7-smp
Compiler	GNU Compiler Collection 4.2.4	GNU Compiler Collection 4.2.4	GNU Compiler Collection 4.2.4	GNU Compiler Collection 4.2.4
Grid Middleware	Globus Toolkit 4.0.5	Globus Toolkit 4.0.5	Globus Toolkit 4.0.5	Globus Toolkit 4.0.5
MPI Library	MPICH-G2 1.2.7p1	MPICH-G2 1.2.7p1	MPICH-G2 1.2.7p1	MPICH-G2 1.2.7p1
Node Count	6	4	2	1

Table 4 The configuration of our Grid system—part 2

	AO270 RS Series	O2214 RS Series	P4200 TS Series	P3800 TS Series
Processor	AMD Opteron 270 Dual Core *2	AMD Opteron 2214 Dual Core *2	Intel Pentium 4 2.0 Ghz *1	Intel Pentium III 800 Mhz *1
Clock Rate	1990 Mhz	2194 Mhz	2018 Mhz	799 Mhz
Core Count Per Node	2 cores per processor	2 cores per processor	1 core per processor	1 core per processor
Memory	DDR-400 1024 MB Register ECC *4	DDR2-667 1024 MB Register ECC *3	DDR-333 256 MB *1	PC-133 256 MB *1
Hard Disc	SATA-II 320 GB	SATA-II 320 GB	ATA-133 80 GB	ATA-66 20 GB
Swap Space	1024 MB	1024 MB	1024 MB	1024 MB
OS	Slackware Linux 12.2	Slackware Linux 12.2	Slackware Linux 12.2	Slackware Linux 12.2
Kernel	2.6.27.7-smp	2.6.27.7-smp	2.6.27.7-smp	2.6.27.7-smp
Compiler	GNU Compiler Collection 4.2.4	GNU Compiler Collection 4.2.4	GNU Compiler Collection 4.2.4	GNU Compiler Collection 4.2.4
Grid Middleware	Globus Toolkit 4.0.5	Globus Toolkit 4.0.5	Globus Toolkit 4.0.5	Globus Toolkit 4.0.5
MPI Library	MPICH-G2 1.2.7p1	MPICH-G2 1.2.7p1	MPICH-G2 1.2.7p1	MPICH-G2 1.2.7p1
Node Count	2	1	2	1

Table 5 The locations, bandwidths, machine types, node counts and core counts per node of the four Grid sites

Site	Location	Bandwidth	Machine Type	Node Count	Core Count per Node
NCUE	Changhua City	1 Giga bps	E2160 TS Series	6	2
			Q6600 TS Series	2	4
			E5405 RS Series	2	4
			E5320 RS Series	1	4
			AO270 RS Series	2	2
			O2214 RS Series	1	2
TTU	Taipei City	1 Giga bps	P3800 TS Series	1	1
NTCU	Taichung City	400 Mbps	P4200 TS Series	2	1
			Q6600 TS Series	1	4
ALES	Kaohsiung County	100 Mbps	Q6600 TS Series	1	4

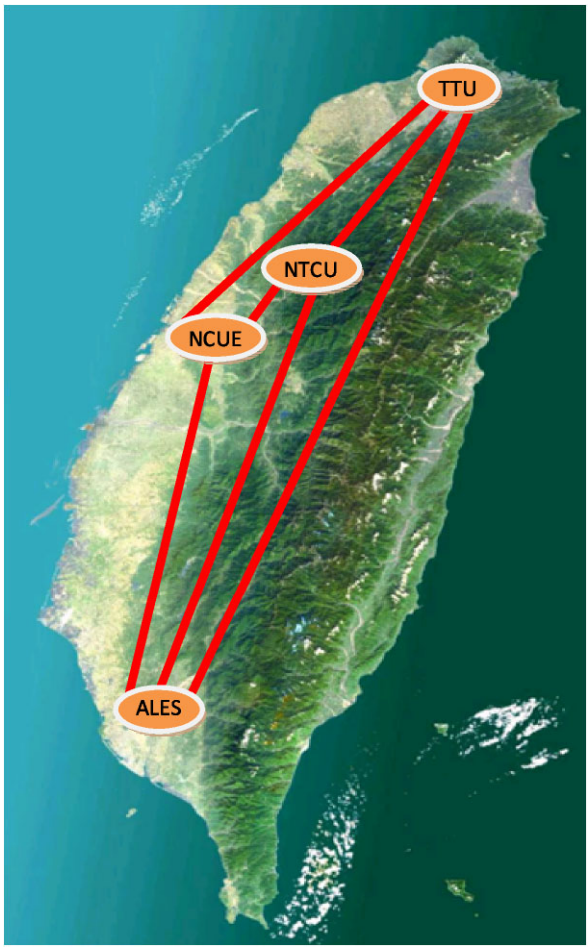

Fig. 8 The distribution of the four Grid sites over Taiwan

References

1. Banicescu I, Carino RL, Pabico JP, Balasubramaniam M (2005) Overhead analysis of a dynamic load balancing library for cluster computing In: Proceedings of the 19th IEEE international parallel and distributed processing symposium, pp 122.2
2. Boeres C, Nascimento AP, Rebello VEF Sena AC (2005) Efficient hierarchical self-scheduling for MPI applications executing in computational Grids. In: Proceedings of the 3rd international workshop on middleware for grid computing, pp 1–6
3. Chronopoulos AT, Penmatsa S, Yu N (2002) Scalable loop self-scheduling schemes for heterogeneous clusters. In: Proceedings of the 2002 IEEE international conference on cluster computing, pp 353–359
4. Chronopoulos AT, Penmatsa S, Xu J, Ali S (2006) Distributed loop-self-scheduling schemes for heterogeneous computer systems. Concurr Comput Pract Experience 18(7):771–785
5. Foster I (2002) The Grid: a new infrastructure for 21st century science. Phys Today 55(2):42–47
6. Foster I, Kesselman C (1997) Globus: a metacomputing infrastructure toolkit. Int J Supercomput Appl High Perform Comput 11(2):115–128
7. Foster I, Kesselman C (2003) The Grid 2: blueprint for a new computing infrastructure. Morgan Kaufmann, San Mateo

8. Foster I, Kesselman C, Tuecke S (2001) The anatomy of the grid: enabling scalable virtual organizations. Int J Supercomput Appl High Perform Comput 15(3):200–222
9. Herrera J, Huedo E, Montero RS, Llorente IM (2006) Loosely-coupled loop scheduling in computational grids. In: Proceedings of the 20th IEEE international parallel and distributed processing symposium, 6 pp
10. HINT performance analyzer. http://hint.byu.edu/
11. Hummel SF, Schonberg E, Flynn LE (1992) Factoring: a method scheme for scheduling parallel loops. Commun ACM 35(8):90–101
12. 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
13. Mandelbrot BB (1988) Fractal geometry of nature. Freeman, New York
14. MPI. http://www.mcs.anl.gov/research/projects/mpi/
15. MPICH-G2. http://www.hpclab.niu.edu/mpi/
16. OpenMP. http://en.wikipedia.org/wiki/OpenMP/
17. Polychronopoulos CD, Kuck D (1987) Guided self-scheduling: a practical scheduling scheme for parallel supercomputers. IEEE Trans Comput 36(12):1425–1439
18. 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
19. Spooner DP, Jarvis SA, Cao J, Saini S, Nudd GR (2003) Local grid scheduling techniques using performance prediction. IEE Proc-Comput Digit Tech 150(2):87–96
20. Tabirca S, Tabirca T, Yang LT (2006) A convergence study of the discrete FGDLS algorithm. IEICE Trans Inf Syst E89-D 2:673–678
21. Tang P, Yew PC (1986) Processor self-scheduling for multiple-nested parallel loops. In: Proceedings of the 1986 international conference on parallel processing, 1986, pp 528–535
22. Tzen TH, Ni LM (1993) Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. IEEE Trans Parallel Distrib Syst 4:87–98
23. Wu C-C, Lai L-F, Chiu P-H (2008) Parallel loop self-scheduling for heterogeneous cluster systems with multi-core computers. In: Proceedings of Asia-pacific services computing conference, vol 1, pp 251–256
24. Wu C-C, Lai L-F, Yang C-T, Chiu P-H (2009) Using hybrid MPI and OpenMP programming to optimize communications in parallel loop self-scheduling schemes for multicore PC clusters. J Supercomput. doi:10.1007/s11227-009-0271-z
25. 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
26. 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
27. 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
28. Yang C-T, Shih W-C, Tseng S-S (2008) Dynamic partitioning of loop iterations on heterogeneous PC clusters. J Supercomput 44(1):1–23
29. Yang C-T, Chang J-H, Wu C-C (2009) Performance-based parallel loop self-scheduling on heterogeneous multi-core PC clusters. In: Proceedings of the second international conference on high performance computing and applications