Designing Parallel Loop Self-Scheduling Schemes by the Hybrid MPI and OpenMP Model for Grid Systems with Multi-Core Computational Nodes

Chao-Chin Wu, Chao-Tung Yang*, Kuan-Chou Lai**, Syun-Sheng Jhan***, Po-Hsun Chiu
Department of Computer Science and Information Engineering
National Changhua University of Education, Taiwan
**Department of Computer Science and Information, Engineering, Tunghai University, Taiwan*
***Department of Computer and Information Science, National Taichung University, Taiwan*
****Department of Information Management, Ling Tung University, Taiwan*
*ccwu@cc.ncue.edu.tw, *ctyang@thu.edu.tw, **kclai@mail.ntcu.eud.tw*
***janss@mail.ltu.edu.tw, *s94610032@mail.ncue.edu.tw*

Abstract

Loop scheduling on parallel and distributed systems has been thoroughly investigated in the past. However, none of them considers the feature of multicore architecture dominating the current markets of desktop computers, laptop computers, servers, etc. On the other hand, although there have been many studies proposed to employ the hybrid MPI and OpenMP programming model to exploit different levels of parallelisms for the distributed system with multicore computers, none of them aimed at the design of parallel loop self-scheduling. Therefore, this paper investigates how to employ the hybrid MPI and OpenMP model to design parallel loop self-scheduling scheme to adapt to the feature of multicore architecture for emerging grid systems. The proposed scheduling approach is based on our previous work adopting the pure MPI model. Preliminary experimental results show that the proposed approach outperforms the previous work with the average speedup of 3.39.

Keywords: Grid computing, Self-scheduling, Loop scheduling, MPI, OpenMP.

1. Introduction

As computers become more and more inexpensive and powerful, computational grids which consist of various computational and storage resources have become promising alternatives to traditional multiprocessors and computing clusters [1, 2]. Basically, grids are distributed systems which share resources through the Internet. Users can access more computing resources through grid technologies. However, bad management of grid environments might result in using grid resources in an inefficient way. Moreover, the heterogeneity and dynamic changing of the grid environment make it different from conventional parallel and distributed computing systems, such as multiprocessors and computing clusters. Therefore, it becomes more difficult to utilize the grid efficiently.

Loop scheduling on parallel and distributed systems is an important problem, and has been thoroughly investigated on traditional parallel computers in the past [3-6]. Traditional loop scheduling approaches include static scheduling and dynamic scheduling. The former is not suitable in dynamic environments. The latter, especially self-scheduling, has to be adapted to be applied to heterogeneous platforms. Therefore, it is difficult to schedule parallel loops on the heterogeneous and dynamic grid environments. In recent years, several pieces of work has been devoted to parallel loop scheduling for cluster computing environments [7-11], addressing the heterogeneity of computing power.

For grid systems, we have revised known loop self-scheduling schemes to fit Grid computing environments [12]. The HINT Performance Analyzer [13] is used to determine whether target systems are relatively homogeneous or relatively heterogeneous. We then partition loop iterations into four classes, based on typical cluster system cases to achieve good performance in any given computing environment. Finally, a heuristic approach based upon α-based self-scheduling scheme to solve parallel regular loop

scheduling problem on an extremely heterogeneous Grid computing environment.

Intuitively, we would partition the total workload according to CPU clock speed. However, the CPU speed is not the only factor which affects node performance. Many other factors also have dramatic influences in this respect, such as the amount of memory available, the cost of memory accesses, and the communication bandwidth between nodes, etc. Using this intuitive approach, the result will be degraded if the performance estimation is not accurate. To address this problem, we also proposed a general approach called PLS (Performance-based Loop Scheduling) [14]. This approach utilizes performance functions to estimate the performance of each node.

Although our previous approaches improve the system performance, they did not take the feature of multicore architecture into account. Recently, more and more cluster systems include multicore computers because almost all the commodity personal computers are multicore architecture. The primary feature of multicore architecture is that multiple processors on the same chip can communicate with each other by directly accessing the data in shared memory. Unlike multicore computers, each computer in the distributed system has its own memory system and thus it relies on the 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 language. However, MPI is not the best programming language for multicore computers. Instead, OpenMP is very suitable for multicore computers because it is a shared-memory programming language. Therefore, in this paper we propose to use hybrid MPI and OpenMP programming mode to design the loop self-scheduling scheme for the grid system with multicore computers. Preliminary experimental results show that the proposed approach outperforms the previous work with the average speedup of 3.39.

2. Related Work

Pure self-scheduling (PSS) is the first straightforward dynamic loop scheduling algorithm [3]. Whenever a processor becomes idle, the master will assign a loop iteration to it. This algorithm achieves good load balancing because the maximum waiting time for the last processor is the execution time of a loop iteration. However, it induces excessive runtime overhead because it requires N times to dispatch the iterations one by one by the master if there are N iterations totally.

Chunk self-scheduling (CSS) assigns k consecutive iterations each time [3]. The chunk size, k, is fixed and must be specified by either the programmer or by the compiler. A large chunk size will cause load imbalance because the maximum waiting time for the last processor is the execution time of k loop iterations. In contrary, a small chunk size is likely to result in too much runtime overhead. If k is equal to 1, CSS will be degraded to PSS. Thus, it is important and difficult to choose the proper chunk size.

Guided self-scheduling (GSS) dispatches iterations decreasingly [4]. More specifically, the next chunk size is calculated by dividing the number of the remaining iterations by the number of available processors. It aims at reducing the dispatch frequency to minimize the scheduling overhead and reducing the number of iterations assigned to the last few processors to achieve better load balancing.

Factoring Self-Scheduling (FSS) assigns loop iterations to processors in phases [5]. During each phase, only the half of remaining loop iterations is equally divided among available processors. FSS can prevent from assigning too much workload to the first few processors. As a result, it balances workloads better than GSS when loop iteration computation times vary substantially.

Trapezoid Self-Scheduling (TSS) reduces the scheduling frequency while still providing reasonable load balancing [6]. Two parameters have to be specified either by the programmer or by the compiler: the number of the first iterations to be assigned to the processor starting the loop, N_s; and the number of the last iterations to be assigned to the processor performing the last fetch, N_f. According to the values of N_s and N_f, the number of iterations to be assigned in each step is decreased in a constant ratio.

3. The proposed method

A grid system is comprised of multiple computational nodes connected by the Internet. Each computational node has its own memory system and the address space. Parallel processes running in different computational nodes communicate with each other by explicit message transmissions. Therefore, message-passing programming languages, such as the MPI de facto standard, are used to design parallel programs for grid systems. On the other hand, a multicore computer is a shared-memory multiprocessor. Because all the cores share the same physical main memory modules, parallel processes communicate with each other by accessing to data in the shared memory. In addition, because every process has completely separate program

with its own variables and memory allocation while threads share the same memory space and global variables between routines, it is more cost effective if processes are replaced with threads. Therefore, it is more suitable to use shared-memory programming languages, such as the OpenMP library, to develop parallel programs for multicore computational nodes. Therefore, we propose to adopt the hybrid parallel programming model to combine both the advantages of message-passing programming and shared-memory programming for the grid system with multicore computational nodes. MPI message-passing programming is adopted for the communications among different computational nodes and OpenMP shared-memory programming is adopted for the communications among different cores in the same computational node.

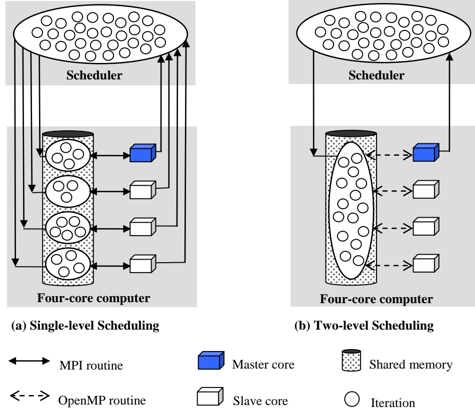

Fig. 1. Single-level and two-level scheduling schemes

We give an example to explain the idea more detailed as shown in Figure 1. Assume that we have a 4-core computational node. In the pure MPI programming model, there will be four parallel MPI processes running on the four cores as shown in Figure 1(a). Every process has to request the iterations from the scheduler directly. The iterations assigned to a process cannot be shared by the other three processes although the assigned iterations are in the shared memory. On the other hand, if the hybrid MPI and OpenMP programming model is employed, there will be only one MPI process running in one of the four processor cores as shown in Figure 1(b). The MPI process will communicate with the scheduler to request new iterations. Whenever receiving the assigned iterations, the MPI process will fork four parallel OpenMP threads to process the assigned iterations. The four parallel threads will adopt the OpenMP built-in self-scheduling function to process the iterations. As soon as the assigned iterations have been processed by OpenMP threads, the MPI process returns the result to the scheduler and asks for new iterations.

Because only one MPI process will be created for each multicore computational and the assigned iterations will be processed by all the processor cores in parallel using OpenMP threads, the number of iterations assigned at each scheduling step must be modified. We describe how to extend our previous work [14] for the hybrid MPI and OpenMP programming model.

Let M denote the number of computing nodes, P denote the total number of processor cores. Computing node i is represented by m_i, and the total number of processor cores in computing node m_i is represented by p_i, where $1 \leqq i \leqq M$. In consequence, $P = \sum_{i=1}^{M} p_i$. The jth processor core in computing node i is represented by c_{ij}, where $1 \leqq i \leqq M$ and $1 \leqq j \leqq p_i$. 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 computing node i and the remaining number of tasks R_s,

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

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 [10–12] 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 computing node, we define a performance function (PF) for a computing node i as

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

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

$$\mathrm{PF}_i = \frac{\sum_{k=1}^{p_i} \frac{\mathrm{CS}_{ik}}{CL_{ik}}}{\sum_{q=1}^{M} \sum_{k=1}^{p_i} \frac{\mathrm{CS}_{qk}}{CL_{qk}}}\ , \qquad (3)$$

where CS_{ij} is the CPU clock speed of processor core j in computing 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 computing 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 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 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.

We propose to use a parameter, SWR (Static-Workload Ratio), to alleviate the effect of irregular workload. In order to take advantage of static scheduling, SWR percentage of the total workload is dispatched according to Performance Ratio. If the workload of the target application is regular, SWR can be set to be 100. However, if the application has irregular workload, it is efficient to reserve some amount of workload for load balancing. We propose to randomly take five sampling iterations, and compute their execution time. Then, the SWR of the target application i is determined by the following formula.

$$SWR_i = \frac{\min_i}{MAX_i} \quad (4)$$

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. 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, or another application, the five sampling execution time might be 7, 7.5, 8, 8.5 and 10 seconds, respectively. Then the SWR is $7/10$, i.e. a percentage of 70. Therefore, 70 percentages of the iterations would be scheduled statically according to PR, while 30 percentages of the iterations would be scheduled dynamically by any one of the well known self-scheduling scheme such as *GSS*. In the second phase, when an MPI process requests for new iterations at each scheduling step, we have to take the number of processor cores into consideration when the master process determines the number of iterations to be allocated for the process 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 base on the applied well known self-scheduling scheme 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 the chunk size of 128 iterations is adopted, the master will assign 512 iterations whenever the MPI process running on the computational node asks for new iterations.

4. Performance evaluations

Table 2. The configuration of our grid system

Bao-Shan Campus Of NCUE (5 PCs)	
Intel Pentium Dual-Core × 4	
CPU	Pentium Dual-core E2160, 1.8GHz
RAM	512 MB DDR 667 × 1
Hard Disk	80 GB
Swap Space	1 GB
Bandwidth	400Mbps
Intel Pentium Quad-Core × 1	
CPU	Intel Core 2 Quad Q6600, 2.4G/8M/1066FSB
Memory	1GB DDR2 × 2
Hard Disk	SATAII 160GB
Swap Space	2GB
Bandwidth	400Mpbs
Jin-Der Campus Of NCUE (3 SMPs)	
CPU	Dual-Core AMD Opteron Processor 270/2.0/2M × 2
Memory	1GB DDR 400 Registered ECC × 4
Hard Disk	SATAII 320GB
Swap Space	2GB
Bandwithd	1000Mpbs
National Taichung University (1 PC)	
CPU	Intel Core 2 Quad Q6600 2.4G/8M/1066FSB
Memory	1GB DDR2 × 2
Hard Disk	SATAII 160GB
Swap Space	2GB
Bandwidth	400Mbps
Lin Tung University (3 PCs)	
CPU	Intel Pentium III 1GHz @ 997MHz
Memory	256MB PC-133 × 1
Hard Disk	ATA66 30GB
Swap Space	512MB
Bandwidth	1000Mbps

To verify the proposed approach, we have constructed a grid system consisting of twelve computational nodes, where nine nodes are multicore architecture. Totally, there are 31 processor cores in the system. The configuration of our grid is listed in Table 2. We compare our approach with the *PLS (Performance-based Loop Scheduling)* scheme [14]. The speedup is obtained by dividing the execution time of the proposed scheme by the execution time of the PLS scheme when the same well known dynamic self-scheduling is employed in the second scheduling phase.

The benchmark program is the sparse matrix multiplication that is a fundamental operation in many numerical linear algebra applications. The input matrix A is a sparse matrix. We 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. Therefore, the workloads of different iterations in sparse matrix multiplication are irregular. In addition, when a row is assigned for an MPI process, the corresponding element values of matrix A are sent to that process.

First, we compare the chuck self-scheduling based schemes as shown in Fig. 2. The label PCSS in the legend denotes the PLS scheme with the CSS used in the second scheduling phase. The label LCSS in the legend denotes the proposed scheme with the CSS used in the second scheduling phase. Our scheme has the better performance for any matrix sizes. The speedups are all about 2.

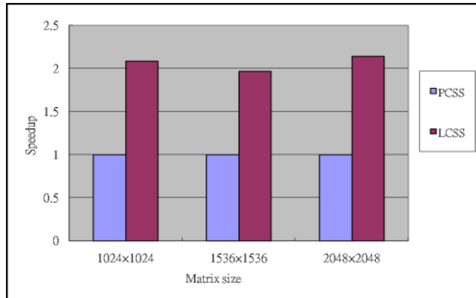

Fig. 2. Comparison for CSS-based schemes. The chunk size is 128.

Second, we compare the guided self-scheduling based schemes with static self-scheduling scheme as shown in Fig. 3. The label PGSS in the legend denotes the PLS approach with the GSS employed in the second scheduling phase. The label LGSS in the legend denotes the proposed scheme with the GSS used in the second scheduling phase. The speedups obtained by the hybrid MPI and OpenMP model range from 5.03 to 2.65, which is better than that for CSS-based schemes as shown in Figure 2. However, the speedup is decreased when the matrix size is increased. The reason is that the larger amount of data communication will influence the performance significantly as follows. In the hybrid MPI and OpenMP programming mode, all the processor cores in the same computation node have to wait for the completion of the data transmission at each scheduling step because the data will be processed in parallel by these processor cores using OpenMP threads. The total amount of data to be transmitted at each scheduling step is the sum of the data that will be transmitted at several continuous scheduling steps in the original GSS scheme. However, in the pure MPI model, each processor core in the same computational node only needs to wait for the completion of the data transmission at each scheduling step and the amount of data to be transmitted is equal to that in the original GSS scheme. Therefore, our approach needs to wait longer than the previous approach before the data are available at each scheduling step.

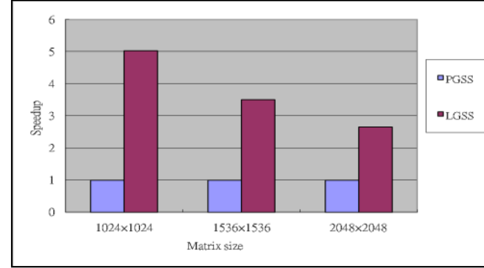

Fig. 3. Comparison for GSS-based schemes.

Third, we compare the self-scheduling based schemes as shown in Fig. 4. The label PGSS in the legend denotes the PLS approach with the GSS employed in the second scheduling phase and the label LGSS denotes the proposed scheme with the GSS used in the second scheduling phase. Our approach outperforms the previous approach. The speedups range from 4.56 to 3.32. Unlike the GSS scheme, FSS can prevent from assigning too much workload to the first few processors. As a result, it balances workloads better than GSS when loop iteration computation times vary substantially. Moreover, the influences caused by the data communication overhead will be lessened.

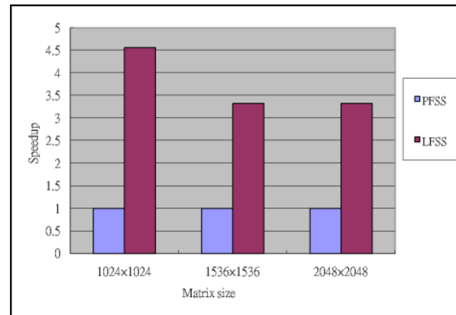

Fig. 4. Comparison for FSS-based schemes.

Finally, we compare trapezoid self-scheduling based schemes as shown in Fig. 5. The labels PGSS and LGSS in the legend denote the PLS approach and our approach with the GSS employed in the second scheduling phase, respectively. The speedups range from 4.58 to 3.04. Unlike GSS and FSS whose speedups are decreased when the matrix size becomes larger, for TSS, the speedup is decreased substantially only when the matrix size is as large as 2048×2048.

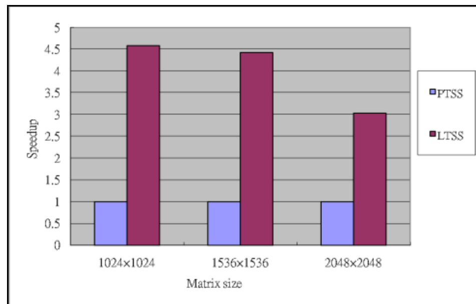

Fig. 5. Comparison for TSS-based schemes.

5. Conclusions

This paper uniquely investigate how to employ the hybrid MPI and OpenMP programming mode to design parallel loop self-scheduling schemes for emerging grid systems with multicore computational nodes. The proposed scheduling approach 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. We have constructed a grid system to verify our proposed approach. The benchmark is the sparse matrix multiplication, which has the irregular workload distribution among iterations and requires data communication at each scheduling step. Preliminary experimental results show that the proposed approach outperforms the previous work with the average speedup of 3.39.

References

[1] I. Foster, C. Kesselman, “Globus: a metacomputing infrastructure toolkit”, Int’l J. Supercomputing Applications and High Perform Computing, Vol. 11, No. 2, pp.115–128, 1997.

[2] Foster, I. and Kesselman, C., The Grid 2: Blueprint for a New Computing Infrastructure, Morgan Kaufmann Publishers Inc., 2003.

[3] H. Li, S. Tandri, M. Stumm, and K. C. Sevcik, “Locality and Loop Scheduling on NUMA Multiprocessors”, *Proceedings of the 1993 International Conference on Parallel Processing*, Vol II, 1993, pp. 140–147.

[4] C. D. Polychronopoulos, and D. Kuck, “Guided Self-Scheduling: a Practical Scheduling Scheme for Parallel Supercomputers,” IEEE Trans. on Computers, vol. 36, no. 12, pp. 1425-1439, 1987.

[5] S. F. Hummel, E. Schonberg, and L. E. Flynn “Factoring: A Method Scheme for Scheduling Parallel Loops”, *ACM Communications*, Vol. 35, 1992, pp. 90–101.

[6] T. H. Tzen, and L. M. Ni, “Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers,” *IEEE Transactions on Parallel and Distributed Systems*, Vol. 4, 1993, pp. 87-98.

[7] I. Banicescu, R.L. Carino, J. P. Pabico, M.Balasubramaniam, “Overhead analysis of a dynamic load balancing library for cluster computing”, Proceedings of the 19th IEEE international parallel and distributed processing symposium, 2005.

[8] A. T. Chronopoulos, S. Penmatsa, J. Xu, S. Ali, “Distributed loop-self-scheduling schemes for heterogeneous computer systems,” Concurrent Computing: Practice and Experience, Vol. 18, pp.771–785, 2006.

[9] C.-T. Yang, and S.-C. Chang, “A Parallel Loop Self-Scheduling on Extremely Heterogeneous PC Clusters”, *Journal of Information Science and Engineering*, Vol. 20, No. 2, 2004, pp. 263–273.

[10] C.-T. Yang, K.-W. Cheng, and K.-C. Li, “An Enhanced Parallel Loop Self-Scheduling Scheme for Cluster Environments”, *The Journal of Supercomputing*, Vol. 34, No. 3, 2005, pp. 315-335.

[11] C.-T. Yang, W.-C. S., and S.-S. Tseng, “Dynamic Partitioning of Loop Iterations on Heterogeneous PC Clusters”, *The Journal of Supercomputing*, Vol. 44, 2008, pp. 1-23.

[12] C.-T. Yang, K.-W. Cheng, and W.-C. Shih, “On Development of an Efficient Parallel Loop Self-Scheduling for Grid Computing Environments”, *Parallel Computing,* Vol. 33, No. 7-8, 2007, pp. 467-487.

[13] HINT performance analyzer. http://hint.byu.edu/

[14] W.-C. Shih, C.-T. Yang, and S.-S. Tseng, “A Performance-Based Parallel Loop Self-scheduling on Grid Computing Environments”, *The Journal of Supercomputing*, Vol. 41, 2007, pp. 247-267.