The Performance Impact of Different Master Nodes on Parallel Loop Self-scheduling Schemes for Rule-Based Expert Systems

Chao-Chin Wu[1], Lien-Fu Lai[1], Liang-Tsung Huang[2], Chao-Tung Yang[3], and Chung Lu[1]

[1] Department of Computer Science and Information Engineering National Changhua University of Education, Taiwan
{ccwu,lflai}@cc.ncue.edu.tw, m98612005@mail.ncue.edu.tw
[2] Department of Biotechnology, MingDao University, Taiwan
larry@mdu.edu.tw
[3] Department of Computer Science and Information Engineering, Tunghai University

Abstract. The technique of parallel loop self-scheduling has been successfully applied to auto-parallelize rule-based expert systems previously. In a heterogeneous system, different compute nodes have different computer powers. Therefore, we have to choose a node to run the master process before running an application. In this paper, we focus on how different master nodes influence the performances of different self-scheduling schemes. In addition, we will investigate how the file system influences the performance. Experimental results give users the good guidelines on how to choose the master node, the self-scheduling scheme, and the file system for storing the results.

Keywords: Parallel computing, cluster system, heterogeneous system, self scheduling, file system.

1 Introduction

A cluster system consists of multiple computers connected by network; it divides a big task into lots of small tasks and has them processed by different computers in parallel. Now most cluster systems use Local Area Network and Ethernet to connect computers, which are often cheap personal computers rather than expensive workstations. Thus, such cluster systems have a better cost-performance ratio.

Traditional cluster systems are homogeneous ones. In this kind of system, all compute nodes own identical system resources, including network bandwidth, I/O storage equipments, CPU clock speed, memory capacity, etc. On the other hand, a cluster system consisting of compute nodes with distinct system resources is called a heterogeneous cluster system. To this kind of cluster, assigning equal amount of work to each computing nodes is not proper. Nodes with better performance take less time to complete their jobs than those with weaker performance. The need of synchronization or data transmission would then have some nodes be idle and wait for others to finish their jobs; the performance of the whole system deteriorates. Therefore, Load Balancing, to properly dispatch works to each node to achieve best system performance, becomes a significant issue to heterogeneous cluster systems.

MapReduce is one of the frequent referred technologies for cloud computing, which is a programming model introduced by Google Inc. for processing and generating large data sets [1]. Programmers use map and reduce functions to process data and merge results. Programs written based on the MapReduce model are automatically parallelized and executed on a large cluster of commodity machines. Data partitioning, task scheduling and inter-process communication are all handled by the run-time system. Programmers have no need to learn the complicated techniques for parallel computation for efficient resource utilization in a large distributed system.

We have proposed a method that can be easily parallelize FuzzyCLIPS-based expert system based on the MapReduce programming model [2, 3, 4]. The programmer only has to use our proposed directives to specify which facts can be parallelized, how to infer these facts, which inferred results should be sent back to the master and which rule has to be applied to infer the returned results. The modified inference engine will use a conventional well-known self-scheduling scheme to parallelize the inference. However, none of conventional scheduling schemes consider the matching feature of FuzzyCLIPS language. The unique feature of the matching of facts and rules for FuzzyCLIPS language is that allocating too many or too few data facts in a chunk cannot have the best processing efficiency. We use the feature to design a self-scheduling scheme especially for parallel FuzzyCLIPS-based expert systems. In addition to load balancing, the proposed self-scheduling scheme takes the processing efficiency into account and gives it the higher priority. The proposed $PC^{SL}SS$ scheme relies on a finite state machine to learn the proper chunk size for each worker node.

In a heterogeneous cluster system, different compute node has different compute power. Therefore, in this paper we investigate how different master compute node influence the execution time of applications. To alleviate the burden of the master node when the weak node is adopted, every worker node can writes the results of their allocated tasks directly to the output file via Network File System (NFS). Therefore, we also study how the performance is affected if every work node help write the results to the output file. Experimental results give the users and the system resources brokers a good guidance of how to select compute nodes from a heterogeneous cluster system.

2 Related Work

A parallel loop is a loop having no cross-iteration data dependencies. If a parallel loop has *N* iterations, it can be executed at most by *N* processors in parallel without any interaction among processors. However, because the number of available processors in a system is always much smaller than *N*, each processor has to execute more than one loop iteration. Static scheduling schemes decide how many loop iterations are assigned for each processor at compile time. The advantage of this kind of scheduling schemes is no scheduling overhead at runtime. However, it is hard to estimate the computation power of every processor in the heterogeneous computing system and to predict the amount of time each iteration takes for irregular programs, resulting in load imbalance usually. Dynamic scheduling is more suitable for load balance because it makes scheduling decisions at runtime. No estimations and predictions are required. Various self-scheduling schemes have been proposed to achieve better load balance with less scheduling overhead.

Chunk self-scheduling (CSS) assigns k consecutive iterations each time [5]. 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 contrast, a small chunk size is likely to result in too much scheduling overhead. If k is equal to 1, CSS will be degraded to PSS. Thus, it is important to choose the proper chunk size.

Guided self-scheduling (GSS) assigns decreasing-sized chunks for requests [5]. Initially, the master allocates large chunks to workers and later uses the smaller chunks to smoothen the unevenness of the execution times of the initial larger chunks. 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]. It tries to address the following problem of GSS. Because GSS might assign too much work to the first few processors in some cases, the remaining iterations are not time-consuming enough to balance the workload. During each phase of FSS, only a subset of remaining loop iterations (usually half) is equally distributed to available processors. FSS can prevent from assigning too much workload to the first few processors like GSS does. 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 [5]. The chunk sizes decrease linearly in TSS, in contrast to the geometric decrease of the chunk sizes in GSS. A TSS is represented by TSS(N_s, N_f), where N_s is the number of the first iterations to be assigned to the processor starting the loop and N_f is the number of the last iterations to be assigned to the processor performing the last fetch. The two parameters, N_s and N_f, have to be specified in TSS either by the programmer or by the compiler. According to the values of N_s and N_f, the number of iterations to be assigned at each step is decreased in a constant ratio. Tzen and Ni have proposed TSS($N/2p$, 1) as a general selection, where N is the number of iterations and p is the number of processors.

3 Parallel FuzzyCLIPS

FuzzyCLIPS is an extended version of CLIPS (C Language Integrated Production System) [6] that is a tool for helping the developer to design the expert system. FuzzyCLIPS extends CLIPS by adding the concept of fuzzy logic, i.e. fuzziness and uncertainty. The extension let the FuzzyCLIPS inference engine be able to do the inference with the facts and rules with fuzzy expressions. Due to the rule-based characteristics of CLIPS, it makes the execution very time-consuming.

We proposed a SPMD-based programming model that hides message passing subroutine calls from the programmers [2, 3]. In other words, there are no Send or Recv function calls needed in the parallel code of a fuzzyCLIPS expert system. Instead, several simple directives are employed for parallelization. The programmers only have to identify the following information. (1) Which facts will be processed by worker processes in parallel? These facts are called the parallel facts and they will be stored in the input files. (2) Which rules will be applied to the parallel facts? These rules are called the mapping rules. (3) What kinds of facts should be sent back after each worker process finishes its work? These facts are called the intermediate facts. (4) Which rules the master process has to apply to the intermediate facts returned from worker processes? These rules are called the reduce rules.

The execution of a parallel FuzzyCLIPS program consists of the following four steps. (1) A master process and multiple worker processes will be created. The master process tells each worker process about the information about the assigned facts. The allocation is based on one of well-known self-scheduling schemes. (2) Each worker process reads assigned parallel facts from the input file and the uses the map-ping rules to match with these facts. (3) The intermediate facts are sent back to the master process by each worker process. (4) The master process matches the intermediate facts with the reduce rules and write the result to the output files.

We also proposed a self-scheduling scheme called PC^{SL}SS especially for parallel FuzzyCLIPS-based expert systems [2]. In addition to load balancing, we also focus on the processing efficiency when designing a self-scheduling scheme. Furthermore, the processing efficiency plays more important role than load balancing in our design. In this way, the system can have better performance. In order to make each worker having better processing efficiency, the PCSLSS scheme was designed based on the following strategy. When the worker's performance with current chunk size is better than that with previous chunk size, the next assigned chunk size for this worker is increased, otherwise decreased. However, the adjustment of the chunk size is different depending on whether the worker has adequate workload or not.

In a heterogeneous cluster system, different compute node has different compute power. Consequently, which compute node acts as the master node might influence the application performance. We will investigate the issue in the next section.

4 The Impact of Different Master Node

We have constructed a cluster system with the configuration shown in Table 1 to evaluate how different master node influences the application performance. The cluster system consists of 16 compute nodes and 48 processor cores. The compute nodes are partitioned into five types and each type is labeled in Table 1.

In the cluster system, some of the compute nodes are multicore architecture and some are single-core architecture. To figure out how much performance difference between any two processor cores in different compute nodes, the compute power of each processor core in a compute node is normalized to the slowest processor core in the system, as shown in Fig. 1. The compute power of Type 1 compute nodes is 11.13 times than that of Type 5 compute node.

Table 1. The cluster configuration

Type 1: *Intel Core Quad (2 Quad-core PCs)*	
CPU	Intel Core 2 Quad-Core Q6600, 2394MHz
Cores	4
Memory	2048MB DDR2-553 × 1
Swap	2048
HD	SATA 160GB
OS	Slackware 12.1, kernel 2.6.24.smp
Type 2: *AMD Quad-Core (8 Quad-core PCs)*	
CPU	Phenom™ 9850 Quad-Core, 2499MHz
Cores	4
Memory	1024MB DDR2-533×1
Swap	1024
HD	SATA 300G
OS	Slackware 13.0, kernel 2.6.29.6.smp
Type 3: *Intel Core Duo (2 Duo-core PCs)*	
CPU	Inter Core 2 Duo E2160, 1809Mhz
Cores	2
Memory	512MB DDR2-667 × 1
Swap	1024 MB
HD	SATA 80GB
OS	CentOS 4.4, kernel 2.6.9-42.ELsmp
Type 4: *AMD Athlon XP (3 single-core PCs)*	
CPU	AMD Athlon XP 2800+, 2083MHz
Cores	1
Memory	256 MB DDR400 × 1
Swap	2048
HD	SATA 80G
OS	Slackware 12.0, kernel 2.6.21.5.smp
Type 5: *AMD Athlon XP (1 single-core PCs)*	
CPU	AMD Athlon XP 2800+, 1243MHz
Cores	1
Memory	256 MB DDR400 × 1
Swap	2048
HD	SATA 80G
OS	Slackware 12.0, kernel 2.6.21.5.smp

A human resources website has been implemented to evaluate the performance. Because users' query requirements are usually imprecise and uncertain, instead of matching the input phrases with the records in the database, the search engine uses fuzzy logic to find the records with different levels of fitness. The core of the human resources website is implemented by FuzzyCLIPS language and parallelized by our proposed method.

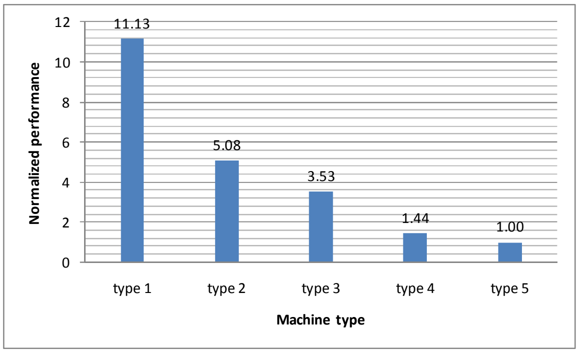

Fig. 1. The execution times normalized to Type-5 processor core

First, we evaluate the impact of different master compute node when we use five processor cores to run the application in parallel, as shown in Fig. 2. We select one processor core from each type of compute nodes and let each different processor core take turns running the master process. Fig. 2 shows that CSS(50), meaning CSS with the chunk size of 50, has the shortest execution time no matter which processor core runs the master process. The second best self-scheduling scheme is TSS, the third is GSS, and the last is FSS. The reason that GSS and FSS have poor performances is the chunk size is so large at the beginning of scheduling that the weakest processor core becomes the performance bottleneck if it is assigned a huge chunk of tasks. The situation is much severer if a fast processor core is responsible for running the master process because the fast processor core cannot help process the tasks. On the contrary, if a weak processor core runs the master process, fast compute nodes can accelerate the processing of tasks even though they are assigned large chunk of tasks, resulting in better performance.

Next, we evaluate the impact of different master compute node when we use eight processor cores to run the application in parallel, as shown in Fig. 3. Among the eight processor cores, at least one processor core from each type of compute nodes is selected. Each different type of processor core takes turns running the master process. Fig. 3 shows that CSS(50) still has the shortest execution time if Type-4 compute node is not selected to runs the master process. More importantly, it is better to choose a fast processor core to run the master processor because it can have better performance. When there are more processor cores to run the same application, the master process becomes busier because of more task requests from worker nodes. As for the performance ranking of the four kinds of self-scheduling schemes, both the cases of five-process and eight-process have almost the same results. However, it is hard to tell which kind of processor cores plays the role of the master can provide better performance. In particular, for GSS and FSS, the performance is more unpredictable because we cannot make sure if the weakest processor core will be assigned the largest chunk of tasks and then become the performance bottleneck. In other words, the request message from the weakest processor core might be the first one arriving at the master process.

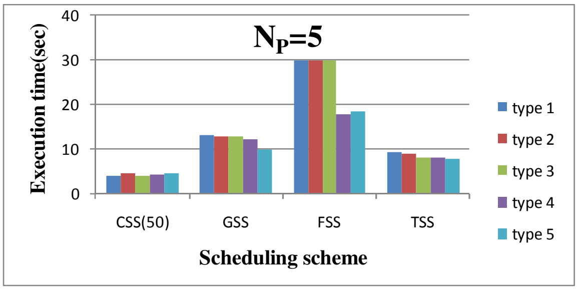

Fig. 2. The impact of different master processor core when the total number of cores is 5

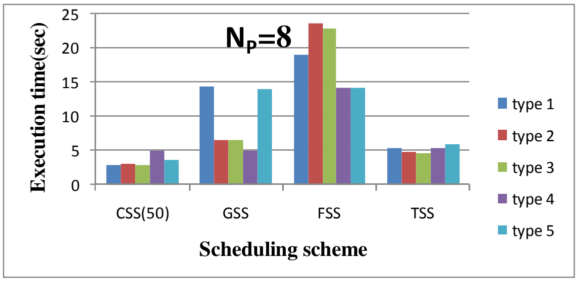

Fig. 3. The impact of different master processor core when the total number of cores is 8

When the total number of processor cores for running the same application in parallel becomes as large as 16, as shown in Fig. 4, the more powerful the master processor core is, the better the performance we have. The exceptions happen in two cases: when Type-1 processor core is adopted as the master core in GSS and FSS. However, when the total number of processor cores becomes as large as 32 or 48, as shown in Fig. 5 and Fig. 6, we can always have the best performance if the fastest processor core is chosen to run the master process. The only issue is which self-scheduling scheme can provide the best performance. For the application of human resources Website, CSS(50) can provide the best performance.

Since CSS(50) provides the best performance, we will investigate how much performance improvement CSS(50) can provide when different type of processor core is adopted as the master core and when the total number of processor core is increased, as shown in Fig. 7. For each specific total number of processor cores used to run the application in parallel, the execution time when each different type of processor core acts as the master core is normalized to the execution time when Type-5 processor core behaves as the master core. The impact of different master processor cores on the performance is increased when the total number of processor cores becomes larger. Although the compute power of Type-1 processor core is 11.13 times faster than Type 5, the performance improvement provided by Type 1 over Type 5 is as high as 2.9. Because the performance difference is very large, it is recommended to adopt the fastest processor core to be the master core, especially for the case when there are many worker nodes.

When an application needs many cores for its parallel execution, we have different core selection methods. For instance, if we need four cores, we can choose the four cores from the same compute node, or two cores from one node and two cores from the other node. To evaluate the impact of different core selection methods, we conduct the following experiment. We need 16 cores and they are from five types of nodes, where four cores are Type 1. For the configuration of one node, the four Type-1 cores are all form the same node while for the configuration of tow nodes, the four Type-1 cores are form two Type-1 nodes. Experimental result shows the configuration of two nodes is better and has speedup of 1.28. The reason is that each processor core can be allocated more system resources.

Because the compute power of the master core will influence the performance when the total number of cores is small, we are interested in whether we can alleviate the burden of the weak master processor cores by letting each worker node write the results of their allocated tasks directly to the output file via NFS. Fig. 8 shows the results when each worker nodes writes the results directly to the output file. Comparing the results with previous results shown in Fig. 3 to Fig. 6, letting worker nodes write results via NFS is not a good solution even though the total number of cores is small. Furthermore, how the master core influences the performance is not so regular. Adopting a weak core rather than a strong core to be the master core is better for GSS and FSS.

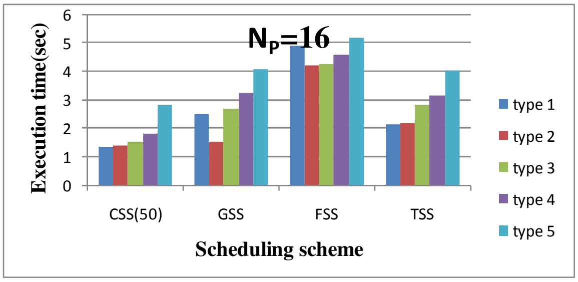

Fig. 4. The impact of different master processor core when the total number of cores is 16

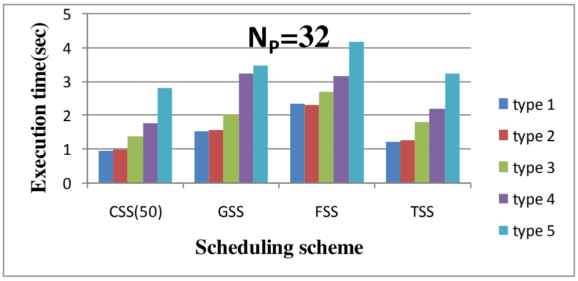

Fig. 5. The impact of different master processor core when the total number of cores is 32

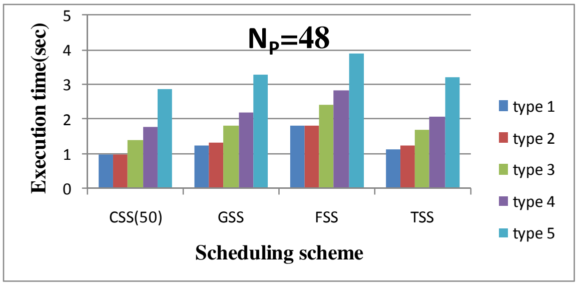

Fig. 6. The impact of different master processor core when the total number of cores is 48

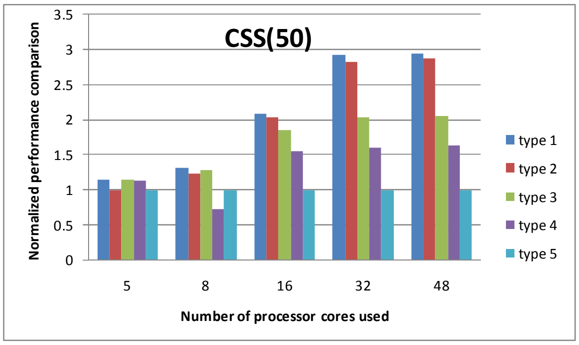

Fig. 7. Normalized performance comparison for different number of processor cores for CSS

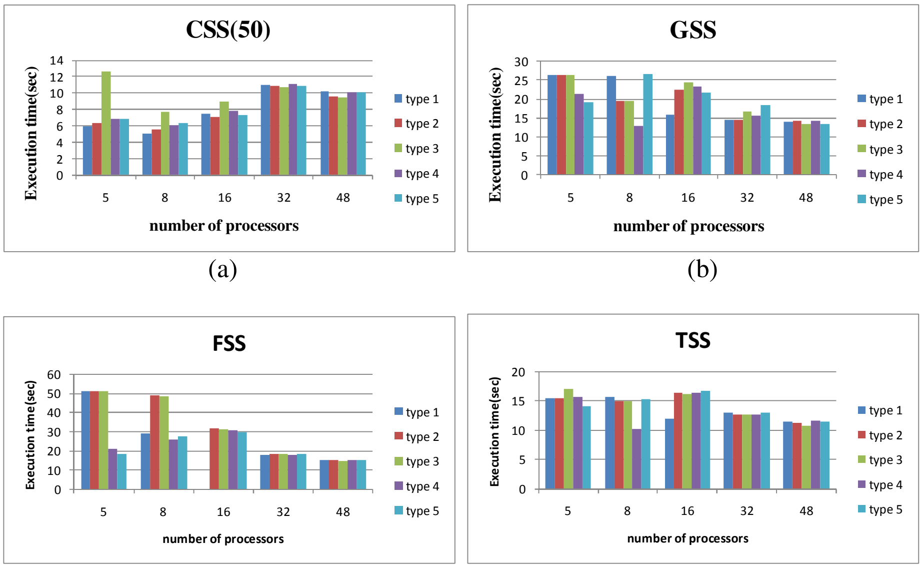

Fig. 8. The impact of different master processor core when each work writes results via NFS

5 Conclusions

In this paper, we study how different master cores influence application performance. When we need a large number of cores, a stronger master core always lead to better performance. On the other hand, if we need a small number of cores, adopting more powerful master nodes is recommended for only CSS and TSS. However, CSS is always the best choice for these two cases. Furthermore, if we need several processor cores of the same type, it is recommended to choose them from different compute nodes because each processor core can be allocated more system resources. Finally, it is better to let the master core be the only one writing the results to its local output file. Although allow each worker node to write their responsible results to the output file via NFS helps alleviate the burden of the master core, the performance is worsen because of write contention to the NFS. Experimental results give the users and the system resources brokers a good guidance of how to select compute nodes from a heterogeneous cluster system.

Acknowledgement. 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.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, vol. 6, p. 10 (2004)
2. Wu, C.-C., Lai, L.-F., Ke, J.Y., Jhan, S.-S., Chang, Y.-.: Designing a Parallel Fuzzy Expert System Programming Model with Adaptive Load Balancing Capability for Cloud Computing. Journal of Computers 21(1), 38–48 (2010)
3. Wu, C.-C., Lai, L.-F., Chang, Y.-S.: Towards Automatic Load Balancing for Programming Parallel Fuzzy Expert Systems in Heterogeneous Clusters. Journal of Internet Technology 10(2), 179–186 (2009)
4. Wu, C.-C., Lai, L.-F., Yang, C.-T., Chiu, P.-H.: Using Hybrid MPI and OpenMP Programming to Optimize Communications in Parallel Loop Self-Scheduling Schemes for Multicore PC Clusters. Journal of Supercomputing (2009), doi:10.1007/s11227-009-0271-z
5. Wu, C.-C., Lai, L.-F., Chang, Y.-S.: Extending FuzzyCLIPS for Parallelizing Data-Dependent Fuzzy Expert Systems. Journal of Supercomputing, doi:10.1007/s11227-010-0542-8
6. FuzzyCLIPS, http://www.iit.nrc.ca/IR_public/fuzzy/fuzzyClips/fuzzyCLIPSIndex2.html