A Robust Dynamic Load-balancing Scheme for Data Parallel Application on Multicomputer Systems

Yangsuk Kee and Soonhoi Ha

Abstract *Multicomputer systems based on message passing draw attractions in the field of high performance computing, where loop or data parallelism is a main source of parallel processing. When exploiting parallelism on multicomputers, however, we confront several challenging problems. First, the nodes of multicomputer are likely to be heterogeneous with respect to computing power and workload. To minimize the elapsed time of program, loads should be balanced according to the performance of nodes. In addition, various dynamic failures should be managed appropriately. Lastly, selecting a proper number of nodes in a node pool is also desirable to achieve a good performance. We propose a new dynamic load balancing scheme called RAS to simultaneously deal with the three issues. RAS solves the load-balancing problem and dynamic failures by a work stealing mechanism, and the processor selection problem by data distribution based on a reservation scheme. According to our experiments on an IBM SP2 with matrix multiplication and ray tracing, RAS has shown better performance than other algorithms such as static chunk and weighted factoring under a shared running environment and even under the dedicated running environment.*

Keywords : data parallelism, multicomputer, dynamic load balancing, processor selection

1 Introduction

As the performance of computer networks improves and the software supports of interprocess communication such as PVM[1] and MPI[2] prevail, multicomputers based on message passing emerge as a viable platform for high performance computing. Multicomputer has a wide spectrum of systems from MPP to a cluster of workstations. A node of multicomputer is an autonomous computer which communicates with other nodes by explicit message passing through a high performance network.

A main drawback of message passing is its high communication overhead, which includes software overhead, hardware latency and delay caused by network and memory contention[3][4]. The high communication overhead implies that coarse-grain parallelism is the most adequate for the message passing communication; coarse-grain parallelism can amortize the overhead. We can find candidates for coarse-grain parallelism in the fields of scientific simulations and computer graphics algorithms.

When exploiting parallelism on multicomputers, we confront several challenging problems regardless of the form of workload. The first problem is load-balancing. Many prior load balancing algorithms are devised assuming that the nodes of system are homogeneous with respect to computing power and workload[5][6][7]. However, in a shared running environment, the working condition of nodes changes dynamically and unpredictably due to the interference of the operating system and other processes. Therefore, any load balancing algorithms for multicomputers should be adaptive to heterogeneity of processor speed, network latency and workload.

In addition, a multicomputer system, especially NOW/COW, may experience various dynamic failures. A node can be down, very heavily loaded with other tasks, or disconnected due to network

problems. Few previous dynamic load balancing approaches, if any, do allow such dynamic failures. A robust algorithm should deal with those failures even under an extremely dynamic environment.

Finally, it is useful to consider the node selection problem. As discussed in [8], it is not always true that the performance of algorithm increases with the number of nodes. The overall elapsed time of a parallel program is totally dependent on the finish-time of the slowest node. In addition, superfluous nodes may enlarge the scheduling overhead. Selecting an appropriate number of faster nodes can lead an algorithm to better performance and give other tasks chances to use the resources of the slower nodes.

1.1 Applications

There are two types of parallelism to be exploited on multicomputers: task parallelism and data parallelism. While many load balancing algorithms focus on task parallelism to minimize the average response time of tasks[7][9][10][11], our goal is to exploit data parallelism within a task, that is, to minimize the total execution time of a program.

Data parallel applications can be classified according to their characteristics. When it comes to data dependency, they can be categorized as independent, adjacently dependent and totally dependent. Independent is a data parallel application, if the decomposed data has no relation to each other for computing(e.g. matrix multiplication), adjacently dependent, if it has a relation to only its neighbors(e.g. macro block in a MPEG frame), and totally dependent, if it has a chaining relation(e.g. fibonacci number).

From the point of view of execution time, they can be classified into uniform, semi-uniform and non-uniform[12]. A data parallel application is uniform if the computation times of data are uniform(e.g. matrix multiplication), semi-uniform, if they depend on the indices of data(e.g. adjoint convolution), and non-uniform, if they depend on data themselves(e.g. ray tracing).

Specially, in this paper, we deal with two applications of independent type. First, we implement a matrix multiplication program of two different dimensions as a benchmark, since matrix multiplication is one of the most common routines of scientific computing. Next, we implement a simple ray tracing algorithm drawing a 320x200 sized image . Ray tracing is a time consuming rendering algorithm to create photo realistic images[13]. We carry out the experiments on an IBM SP2 using MPI, a standard of the message passing communication.

The multicomputer does not have a shared address space, which prevents a node from displaying a part of image independently. Hence, we need a node to display an image on the screen. Due to this constraint, we adopted the master/slave programming model. The master takes charge in serial execution, data distribution and image display, while the slaves determine the value of pixels.

[Place for Figure 1: The master/slave program model]

We propose a new dynamic load balancing algorithm called RAS to exploit loop or data parallelism on multicomputers. RAS stands for Reservation And work Stealing. RAS is proven superior to other algorithms such as static chunk and weighted factoring under a shared running environment and even under the dedicated running environment. Furthermore, we show that RAS deals with one of dynamic failures as long as the master is alive.

The rest of paper is organized as follows. In section 2, we discuss relevant prior works on load balancing. Then, we carry out preliminary experiments in section 3 to determine a good communication strategy. We also give the details of our algorithm by phases in this section. In section 4, we analyze the experimental results and finally in section 5, we conclude with a discussion of future work.

2 Previous Work

Load distribution in distributed environments has been a challenging issue in the parallel computing society. Regardless of the form of workload being task or data, load balancing algorithms can be categorized according to their characteristics. First, algorithms can be classified as centralized and decentralized. Centralized algorithms have one special purpose node called coordinator or master, which manages global load information and distributes loads according to the information. However, in case of decentralized algorithms, there is no coordinator, but each node manages its local(global) load information and balances the workload according to the information interacting with other nodes.

In addition, algorithms can be characterized as relocatable and irrelocatable. Relocatable methods distribute loads at an early stage of distribution, and monitor the workloads of nodes. When the degree of load imbalance exceeds a certain threshold, an algorithm directs loads to migrate from a heavily loaded node to a lightly loaded one. On the other hand, irrelocatable schemes try to schedule or partition data without a significant load imbalance. As irrelocatable schemes have no mechanism of load migration, the scheduling should anticipate the performance of nodes accurately.

Finally, algorithms can be categorized into static and dynamic. Shivaratri, Krueger and Singhal[9] divided algorithms into three types; static, dynamic and adaptive. For simplicity, however, both dynamic and adaptive are called dynamic in this paper. Dynamic algorithms use the information reflecting the state of nodes to distribute loads. However, static algorithms just use the hard-wired prior information of system. Table 1 shows the classification of algorithms referred to in this paper.

[Place for Table 1: Algorithm classification]

2.1 Load Balancing on Multiprocessors

Given N data and P nodes, we wish to schedule data among nodes to minimize the overall finish-time. Static Chunking algorithm(SC) is the simplest static algorithm with minimal scheduling overhead. SC simply assigns $\frac{N}{P}$ data to each P slaves. Even though SC seems an intuitive choice under a perfectly balanced condition, it results in very poor performances under shared running environments. Moreover, due to the interference of the operating system and the communication overhead, SC is not a good choice even under the dedicated environment. Zhiwei Xu stated the relation between the communication overhead and the message size for the point to point communication on the SP2 as

$$O(m) = 46 + 0.035m \ (us) \tag{1}$$

in [3][4], where O denotes the communication overhead and m does the size of data in bytes. Generally, the value of $\frac{N}{P}$ for data parallel application is large. The gap of start-times between the first slave and the last one is $O(\frac{N}{P} * (P-1))$, which is not negligible.

As a progressed form of variable chunks approaches, Guided Self Scheduling(GSS)[5] was proposed. In addition that GSS was mainly proposed to balance non-uniform loads per item on shared memory multiprocessors, it can deal with the discrepancy in the start-time. GSS decreases the amount of data to schedule as the scheduling progresses. Larger chunks at the beginning reduce the scheduling overhead of the master, while smaller chunks assigned dynamically towards the end are sufficient to achieve a good load balance. In GSS, the chunk size scheduled on the next idle slave is $\frac{1}{P}$ of the remaining data. Thus, the ith chunk size(G_i) is determined as

$$G_i = \left\lceil (1 - \frac{1}{P})^i \frac{N}{P} \right\rceil \tag{2}$$

. Allocation of too many loads in early chunks, however, does not balance the loads specially when large chunks of data are allocated to poor slaves[6].

As a variant of decreasing chunk method, factoring[6] was designed by Hummel. Factoring schedules data in batches of P chunks sized equally. The total number of data per batch is a fixed proportion of the remaining data. Where F_i is the size of a chunk in the ith batch, F_i is determined according to formula 3 :

$$F_i = \left\lceil (1 - \frac{1}{x})^i \frac{N}{xP} \right\rceil = \left\lceil (\frac{1}{2})^{i+1} \frac{N}{P} \right\rceil \tag{3}$$

In practice, it is impossible to determine the optimal ratio. Therefore, they chose a magic value for x as 2, meaning that the total number of data per batch is half the remaining data. However, factoring still allocates too many data at the beginning of computation. If one slave is approximately twice slower than others, the slow one may become a bottleneck. In Table 2, we illustrate the sample chunk sizes according to these algorithms.

[Place for Table 2 : Sample chunk sizes according to algorithms]

2.2 Load-balancing on Multicomputers

There is less research carried out on load balancing for multicomputers than for multiprocessors. Weighed factoring(WF)[8] is a variant of factoring which was proposed for a cluster of workstations. WF assigns weights(W) to nodes in proportion to their computing powers determined experimentally. The master decides the size of chunks in a batch in proportion to W instead of the fixed ratio of factoring. Where F_{ij} is the size of the jth chunk in the ith batch, F_{ij} is determined according to formula 4 :

$$F_{ij} = \left\lceil (1 - \frac{1}{x})^i \frac{N}{x} \frac{W_j}{\sum_{k=1}^{k=p} W_k} \right\rceil = \left\lceil (\frac{1}{2})^{i+1} N \frac{W_j}{\sum_{k=1}^{k=p} W_k} \right\rceil \tag{4}$$

What is noticeable is the effect of slow superfluous nodes. By ignoring several slowest nodes, they achieved a better performance. However, WF has no automatic mechanism to exclude slow slaves in a node pool.

Many algorithms for multicomputers adapt a kind of load migration scheme[12][14][15]. In addition, they consist of four typical steps: initial data distribution, load monitoring, decision of balancing, and load migration. At first, algorithms distribute initial data statically and evenly according to the number of nodes. After that, they balance loads by their unique solutions.

Nedeljkovic and Quinn[14] introduced the notion of virtual processor as the unit of workload. They adopted the average execution time per virtual processor as the index of node performance. The nodes exchange load information periodically and when the cost of load migration is less than some portion of the performance penalty due to load imbalance, the algorithm redistributes virtual processors. Their algorithm is a distributed approach but uses global information to determine the load imbalance.

Hybrid Scheduling[15] combines a static scheduling method with a dynamic one. The master evenly distributes data to the slaves at the beginning. Then, the slaves divide the given data into chunks using an algorithm like GSS. After the execution of each chunk, all the slaves send messages to the master to inform of their performance. The master periodically counts the number of messages received and uses it as the index of node performance. When the master detects a certain imbalance of workload, it sends a direction to the heavily loaded nodes to send chunks to the lightly loaded ones.

Hamdi and Lee[12] introduced the notion of region as the unit of parallelism. The master periodically polls the slaves to collect load information and partitions data into several regions. To avoid unnecessary communication, a slave sends a request for the data in the region to the slave who has the data, when the requesting slave has no data to compute.

Maeng[16] used the average idle time of an iteration as the load index. She focused on how to detect a load imbalance, while the aforementioned algorithms focused on the mechanism to balance workload. The algorithm took into account small difference of the idle time an iteration, which may have a bad effect on the performance, when the difference is accumulated through several iterations.

Schnekenburget and Huber[17] expanded the existing homogeneous domain decomposition algorithms, which are discussed in [18] in detail, and applied them to a network of heterogeneous workstations. To gather load information they introduced the term of process weight, which is a normalized measure of node speed. What is noticeable is they do not use an explicit load index, because the behavior of virtual memory management can have a considerable influence on the execution time.

3 RAS Algorithm

We propose a new dynamic load-balancing algorithm, RAS, for the multicomputers. To reduce or hide the communication overhead, RAS adopts an overlapped communication. In message passing architectures, sending a large message is more desirable than multiple small messages, because it amortizes the high start-up cost. However, as shown in Figure 2, a large message delays the slaves' start-time.

[Place for Figure 2 : Comparison between large message and small message]

In order to achieve high performance, there has to be a trade-off between the start-up cost and the start-time. The communication overhead is proportional to the size of message, if the size is large enough. From an experiment shown in Figure 3, we can find the point where linearity is broken; the message larger than 1KB gives us relatively constant overhead per unit message. As a result, a good compromise is to use multiple small messages of size larger than 1KB. We estimated the communication overhead using a kind of blocking operation, MPI_Send.

[Place for Figure 3 : The communication overhead per 1KB of MPI_Send on the SP2]

RAS consists of three phases. In the first phase, the master estimates the performance of the slaves in terms of their computation time. Then, the master distributes data on the basis of the estimate using the reservation scheme. In the distribution phase, the master can select faster slaves in a slave pool as many as profitably used. In the last phase, the master monitors the workload of slaves and redistributes data when a load imbalance is detected. As the master is responsible for scheduling data, RAS allows several slaves to redundantly compute a data. This mechanism makes RAS survive under a dynamic node failure.

[Place for Figure 4 : The flowchart of RAS algorithm]

3.1 Estimate for Slaves' Performance

The estimate for the performance of slaves should be made in advance to schedule data precisely. We divide the whole data into *basic chunk*s which are the unit of computation. In case of matrix multiplication, one column(row) can be a basic chunk and in case of ray tracing, a pixel can be one. If the size of a basic chunk is smaller than 1KB, several basic chunks are packed into a *basic message* to reduce the communication overhead. The size of a basic message is determined

as

$$\left\lceil \frac{1024}{B_{basic}} \right\rceil B_{basic}, \tag{5}$$

where B_{basic} denotes the size of a basic chunk in bytes. When RAS starts up, the master sends one basic message to each slave for the estimate. The first basic message sent to each slave is called *probe*.

The average execution time of a basic message is used to represent the performance. The master wants the slaves to report their performance every computation of a basic message for the accurate estimate. However, frequent reports will be burdensome to both the slaves and the network. As a compromise, a slave reports the result of computation piggybacked with its average execution time, when it meets the following conditions.

- When a slave finishs computing the probe(initial estimate).
- When a slave receives the End Of Data Distribution message.
- When a slave produces results larger than 1KB.
- When the master interrupts a slave to steal data.
- When a slave has no data to compute.

The estimate is used both in the distribution phase and in the work stealing phase. The master continues to check the arrival of reply to the probe until the end of the distribution phase, if not all the slaves report their performance.

3.2 Data Distribution on the Basis of Reservation

To distribute data in proportion to the performance of slaves, RAS uses the reservation mechanism. The main idea of the reservation is that the master sends a basic message to each slave periodically. The sending period is determined by the communication overhead of the master and the execution time of the slave. In the distribution phase, the master sends data in units of basic message. Therefore, the master converts the average execution time into the sending period of basic messages and distributes basic messages on the basis of the period. The sending period of the ith slave(P_i) is determined as

$$P_i = \left\lceil \frac{T_{exec}(i)}{T_{overhead}} \right\rceil, \tag{6}$$

where $T_{exec}(i)$ denotes the average execution time of a basic message in the ith slave and $T_{overhead}$ does the overhead of sending operation for a basic message. The number of MPI_Send operations is used to represent the sending period. When the ith basic message is sent to the jth slave, the (i+P_j)th basic message is reserved for the slave.

During this reservation process, there can be two possible ambiguous cases. One is the case that several slaves try to reserve the same basic message. In this case, the master selects the fastest slave and sends the basic message to that slave. The others that fail to get the basic message reserve the next basic message.

The other is the case that no slave reserves a basic message. This case is divided into two sub-cases. When there is no slave who replies the result of the probe, the master sends the basic messages in a round-robin fashion to all the slaves. When there are more than one slaves that

responded to the result of the probe, the master sends the basic messages in a round-robin fashion to the responded slaves. These two cases make some slaves receive superfluous data. The master does not count these superfluous data for future reservation.

This distribution scheme enables the master to select faster slaves. When the number of slaves exceeds a certain threshold, all the messages will be reserved by some faster slaves. Since the master sends messages to faster slaves, slower ones repeatedly fail to get data.

A scenario of the data distribution based on the reservation is shown in Figure 5, where the sending periods of slaves are 6,9,5 and 7, respectively. The first four messages M1, M2, M3 and M4 are the probes for the estimate. The master sends M5, M6, and M7 to S1, S2, and S3, respectively in a round-robin fashion, since they are not reserved by any slave. As the master receives a reply from S1 while sending M7, it can decide the sending period of S1 as 6(=7-1). Then, the master sends M8 to S1 and reserves M14 for S1. Similarly, the master sends M9 to S3 and reserves M14. M10 and M11 are sent to S1 and S3 in a round-robin fashion with the replied slaves. While sending M11, the master receives two responses. This is the same case as the next data, M12, is reserved by the two slaves. The master selects the fastest slave, S4 and sends M12 to S4. S2 who fails to get the message, reserves the next message M13.

[Place for Figure 5 : A scenario of data distribution based on the reservation]

3.3 Work Stealing

Even though the master tries to distribute data in proportion to the performances of slaves, the dynamic change of running condition makes the finish-times of slaves inconsistent. Superfluous data distributed in a round-robin style can be another source of load imbalance. The master redistributes data by work stealing when it detects any significant load imbalance.

To monitor the workload of slaves, the master maintains the global information about the number of chunks assigned and computed. The slaves also maintain the local information about the number of chunks assigned and computed. Synchronizing both information is carried out when the master gathers the results of computation. When a slave requests additional chunks after computing all the assigned chunks, the master determines whether a meaningful load imbalance occurs. At first, the master chooses the richest slave on the basis of its global information. Then, the master determines the amount of chunks to steal. When the amount of chunks to steal is more than 1, the master sends the data to the requesting slave. Otherwise, the master redundantly sends one basic chunk of $S_{richest}$ to $S_{requesting}$ without interrupting $S_{richest}$ to avoid the interrupt overhead. The amount of basic chunks to steal is given by

$$\left\lfloor \frac{T_{exec}(richest)}{T_{exec}(requesting) + T_{exec}(richest)} D_{richest} \right\rfloor, \tag{7}$$

where $D_{richest}$ denotes the amount of the remaining basic chunks of the richest slave.

To implement the work stealing mechanism, a software interrupt is required. The master should notify the richest slave of the number of chunks deprived using an interrupt. However, the MPI standard does not define any interrupt mechanism between processes running on different nodes[2]. Therefore, we implement an interrupt using the asynchronous I/O with a BSD socket. After a socket is set up between two processes, one can interrupt the other by just writing data on the socket.

This stealing mechanism enables RAS to survive under dynamic failures, since the master can reallocate data to any slaves. When a node is very heavily loaded with other tasks, the slave in the node can not reply the result of computation due to long computation time. In this case, the

master meets a situation of virtual node failure. When the master receives data requests from other slaves, it gradually steals data from the failed slave until the failed slave is deprived of all data.

4 Experimental Results

RAS is compared with static chunk and weighted factoring. We take weighted factoring as a counterpart, because it is simple and shows a better performance than a simple distributed work stealing algorithm[8]. In this paper, we only focus on the applications with independent data type. We implement matrix multiplication programs for the uniform type and a ray tracing algorithm for the non-uniform type.

[Place for Figure 6 : A sample image created by ray tracing]

Experiments are carried out under two running environments: dedicated and shared. The booking facility of the SP2 easily has the whole system dedicated to a program. However, creating a shared running condition is somewhat ad hoc. We have to execute a background process using the IP protocol, because the SP2 does not allow several processes to simultaneously use the high performance switch under the dedicated environment. The background process is a parallel version of matrix multiplication and ray tracing programs respectively, adopting static chunk algorithm. The background process interferes the foreground process, changing the load of node dynamically but not significantly on average. Both the foreground and the background processes use the same number of processors.

We check the net elapsed time of programs excluding the initialization time of MPI. We take the mean value of the elapsed time after 10 executions of programs. When we refer to the number of processors, it means the total number of processors allocated including the master.

The results show that RAS is superior to weighted factoring under the shared running condition and a little better even under the dedicated running condition. The superiority under the dedicated condition is due to the communication strategy. In the distribution phase, communication overlaps computation with multiple small messages.

[Place for Figure 7 : The elapsed time of 100x100 matrix multiplication under the dedicated running environment]

[Place for Figure 8 : The elapsed time of 200x200 matrix multiplication under the dedicated running environment]

Note the results of the experiments with 16 processors shown in Figure 9 and Figure 10. RAS balances the loads very adaptively under the dynamic environment, while weighted factoring suffers from load imbalance. The main reasons are that the master sends too many data to the slower slaves and the number of nodes is too large compared to the problem size. In addition, in Figure 11 and Figure 12, it is confirmed that SC methods give poor results for non-uniform applications.

[Place for Figure 9 : The elapsed time of 100x100 matrix multiplication under the shared running environment]

[Place for Figure 10 : The elapsed time of 200x200 matrix multiplication under the shared running environment]

[Place for Figure 11 : The elapsed time of ray tracing under the dedicated running environment]

[Place for Figure 12 : The elapsed time of ray tracing under the shared running environment]

It is useful to observe the number of stealing, since the number of stealing is highly related to the performance. Table 3, Table 4, Table 5, and Table 6 show the number of stealing under the various configurations. The number of interrupts excluding the number of redundant computations is used to represent the number of stealing. The interrupt is burdensome to slaves due to the context switch. Nevertheless, if computation time is large enough, the wrong placement of data is more

serious, since the performance of a program is determined by the performance of the slowest.

The number of processors and the variance of environment have an effect on the number of stealing. When the master schedules data incorrectly, it will try to balance loads with additional stealing. As expected, the number of stealing increases as the number of processors grows and the environment becomes dynamic. If the initial estimate is accurate, the number of stealing is small and RAS takes advantage of the overlapping scheme. The tables show that the number of stealing does not increase severely regardless of running conditions. The small number of stealing wipes out this worry and leads to good performance.

[Place for Table 3 : The number of stealing in case of matrix multiplication under the dedicated running environment]

[Place for Table 4 : The number of stealing in case of matrix multiplication under the shared running environment]

[Place for Table 5 : The number of stealing in case of ray tracing under the dedicated running environment]

[Place for Table 6 : The number of stealing in case of ray tracing under the shared running environment]

To demonstrate the merit of the work stealing, we forced RAS to undergo a surge of loads in a node. For simplicity, we modeled a surge of loads with an infinite loop. We let a slave fall into an infinite loop after program starts. Then, the master can not obtain any information about the slave. As the master is responsible for distributing data, it redundantly sends the data sent to the virtually failed slave to others. RAS was successfully terminated without significant performance degradation, while weighted factoring was not terminated. Even though this experiment does not guarantee fault-tolerance of algorithm, it has simulated a node failure and given a hint to solve the fault tolerance.

5 Conclusions

To write a portable and efficient parallel program on multicomputer systems, user must take into account of the three problems: load balancing, processor selection and dynamic node failures. We showed that RAS can be a good solution to exploit parallelism with the experiments on the SP2. RAS showed good performance under both the dedicated and the shared running environments. Furthermore, RAS was adaptive to the number of slaves and survived under the virtual node failure.

To rigorously evaluate our algorithm, however, it is necessary to apply RAS to various applications with different data types which are not tested in this paper. In fact, many application programmers experience hard time in writing parallel programs taking into account of the issues. Hence, it is required to build up a user friendly library easy to develop parallel programs without the knowledge of the issues in detail. In addition, to be tolerant to dynamic failures, we need more sophisticated error handling routines. RAS will be not only applicable, but also user friendly in the future.

The users in the application fields suffer from harsh economic cost in using MPP systems. We expect multicomputer systems to evolve into a cluster of PCs, as the network-based operating systems appear. Since PCs will experience more dynamics in a multiprogrammed environment, fault tolerance will be a major issue in parallel programming. We conjecture that the proposed algorithm will be a promising load balancing scheme to PC clustering environments.

References

[1] A. Beguelin, J. Dongarra, A. Geist, R. Manchek, and V. Sunderam. "A Users' Guide to PVM(Parallel Virtual Machine)". Technical Report ORNL/TM-11826, Oak Ridge National Laboratory, 1991.

[2] *MPI : A Message-Passing Interface Standard*. Message Passing Interface Forum, May, 1994.

[3] Zhiwei Xu and Kai Hwang. "Modeling Communication Overhead: MPI and MPL Performance on the IBM SP2". *IEEE Parallel and Distributed Technology*, Spring, pp.9-23, 1996.

[4] Kai Hwang, Zhiewei Xu, and Masahiro Arakawa. "Benchmark Evaluation of the IBM SP2 for Parallel Signal Processing". *IEEE Transactions on Parallel and Distributed Systems*, Vol.7, No.5, pp.522-536, MAY 1996.

[5] Constantine D. Polychronopoulos and David J. Kuck. "Guided Self-scheduling: A Practical Scheduling Scheme for Parallel Supercomputers". *IEEE Transactions on Computers*, Vol.C-36, No.12, pp.1425-1439, DEC. 1987.

[6] Susan Flynn Hummel, Edith Schonberg, and Lawrence E.Flynn. "Factoring: A Method for Scheduling Parallel Loops". *Communication of the ACM*, Vol.35, No.8, pp.90-101, Aug. 1992.

[7] Yuet-Ning Chan, Sivarama P. Dandamudi, and Shikharesh Majumdar. "Performance Comparison of Processor Scheduling Strategies in a Distributed-Memory Mutlicomputer System". *International Parallel Processing Symposium*, pp.139-145, April 1997.

[8] Susan Flynn Hummel, Jeanette Schmidt, R.N.Uma, and Joel Wein. "Load-Sharing in Heterogeneous Systems via Weighted Factoring". *ACM Symposium on Parallel Algorithms and Architectures*, pp.318-328, 1996.

[9] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. "Load Distributing for Locally Distributed Systems". *IEEE Computer*, pp. 33-44, Dec. 1992.

[10] Sajal K. Das, Daniel J. Harvey, and Rupak Biswas. "Adaptive Load-Balancing Algorithms using Symmetric Broadcast Networks:Performance Study on an IBM SP2". *International Conference on Parallel Processing*, pp.360-367, August 1997.

[11] Clemens H. Cap and Volker Strumpen. "Efficient Parallel Computing in Distributed Workstation Environemnts". *Parallel Computing*, Vol.19, pp.1221-1234, 1993.

[12] Mounir Hamdi and Chi-Kin Lee. "Dynamic Load Balancing of Data Parallel Applications on a Distributed Network". *International Conference on Supercomputer*, pp.170-179, 1995.

[13] Feiner Foley, van Dam and Hughes. *Computer Graphics*. Addison Wesley, pp.777-793, 1990.

[14] Nenad Nedeljkovic and Michael J. Quinn. "Data-Parallel Programming on a Network of Heterogeneous Workstations". *High-Performance Distributed Computing*, pp.23-36, 1992.

[15] Oscar Plata and Francisco F. Rivera. "Combining Static and Dynamic Scheduling on Distributed-Memory Multiprocessors". *International Conference on Supercomputer*, pp.186-195, 1994.

[16] Hye-Seon Maeng, Hyoun-Su Lee, Tack-Don Han, Sung-Bong Yang, and Shin-Dug Kim. "Dynamic Load Balancing of Iterative Data Parallel Problems on a Workstation Clustering". *High Performance Computing, Asia*, pp.563-567, 1997.

[17] Thomas Schnekenburger and Martin Huber. "Heterogeneous Partitioning in a Workstation Network". *Heterogeneous Computing Workshop*, pp.72-77, April 1994.

[18] Reinhard VG. Hanxleden and L. Ridgway Scott. "Load Balancing on Message Passing Architectures". *Journal of Parallel and Distributed Computing*, Vol.13, pp.312-324, 1991.

Irrelocatable		Relocatable	
Static	Dynamic	Dynamic	
SC	Guided Self Scheduling[5] Factoring[6] Weighted Factoring[8] Maeng's method[16] Schnekenburger's method[17]	Hybrid Scheduling[15] Hamdi's method[12]	Nedeljkovic's method[14]
Centralized			Decentralized

Table 1: Algorithm classification

Algorithm	Chunk Size
Static Chunk	25 25 25 25
GSS	25 19 14 11 8 6 5 3 3 2 1 1 1 1
Factoring	13 13 13 13 6 6 6 6 3 3 3 3 2 2 2 2 1 1 1 1
Weighted Factoring	17 13 13 9 8 6 6 4 4 3 3 2 2 2 2 1 1 1 1 1 1

(P=4, N=100, Weights are 2 1.5 1.5 1.)

Table 2: Sample chunk sizes according to algorithms

Matrix Size	Number of Processors		
	4	8	16
100x100	2.93	4.50	8.00
200x200	1.39	3.30	4.15

Table 3: The number of stealing in case of matrix multiplication under the dedicated running environment

Matrix Size	Number of Processors		
	4	8	16
100x100	3.67	4.67	8.00
200x200	3.73	8.53	9.20

Table 4: The number of stealing in case of matrix multiplication under the shared running environment

Image Size	Number of Processors		
	4	8	16
320x200	13.1	33.1	51.9

Table 5: The number of stealing in case of ray tracing under the dedicated running environment

Image Size	Number of Processors		
	4	8	16
320x200	14.5	33.7	53.0

Table 6: The number of stealing in case of ray tracing under the shared running environment

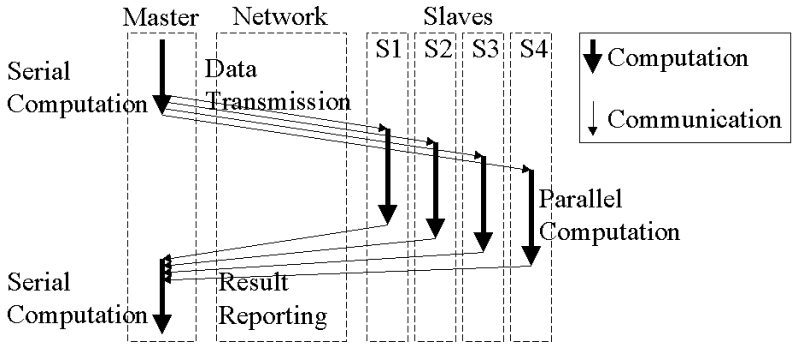

Figure 1: The master/slave programming model

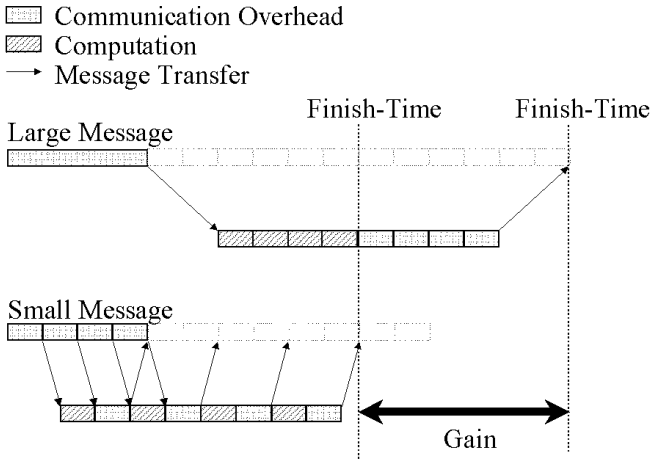

Figure 2: Comparison between large message and small message

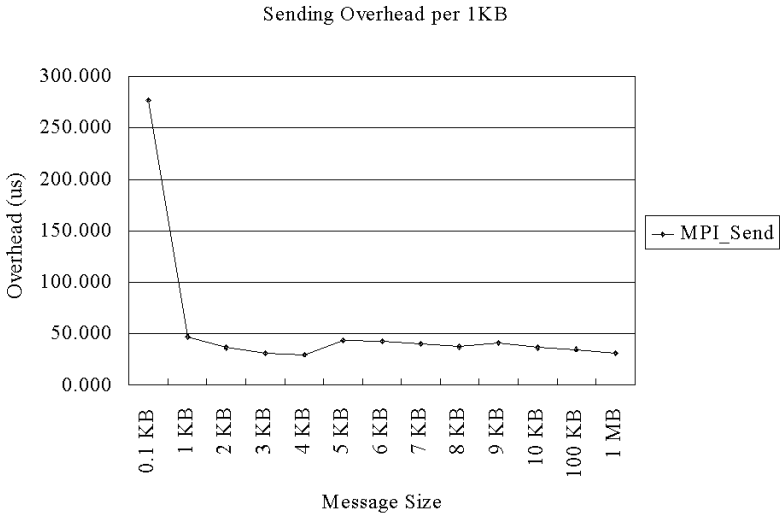

Figure 3: The communication overhead per 1KB of MPI_Send on the SP2

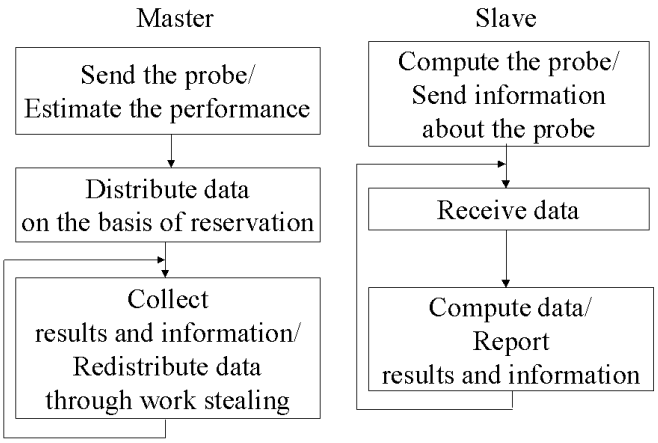

Figure 4: The flowchart of RAS algorithm

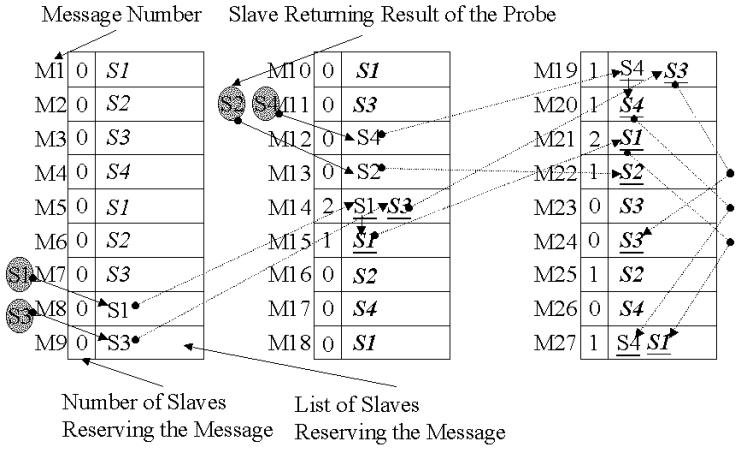

Figure 5: A scenario of the data distribution based on the reservation

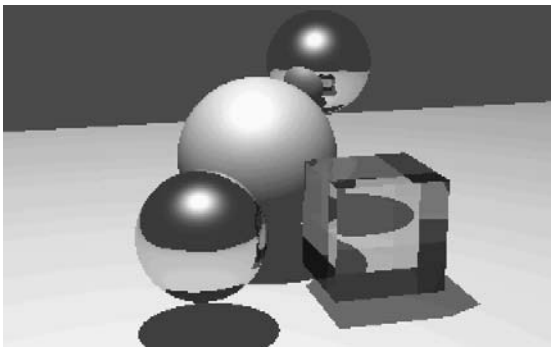

Figure 6: A sample image created by ray racing

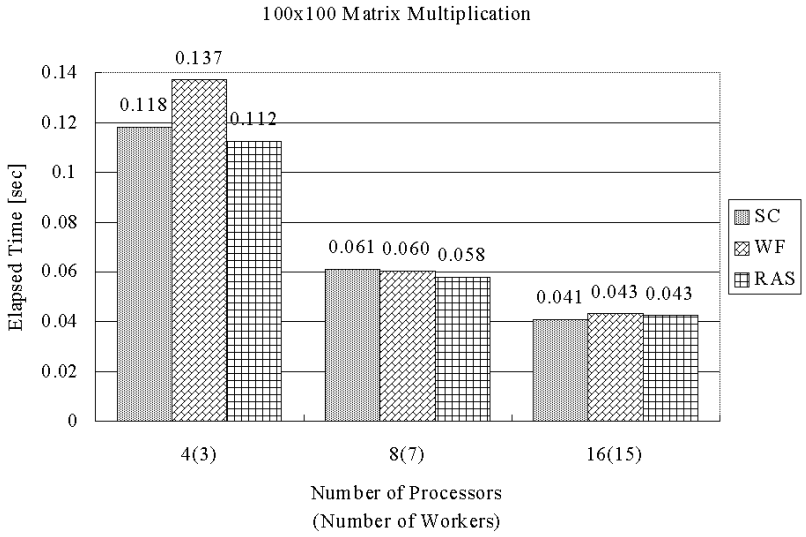

Figure 7: The elapsed time of 100x100 matrix multiplication under the dedicated running environment

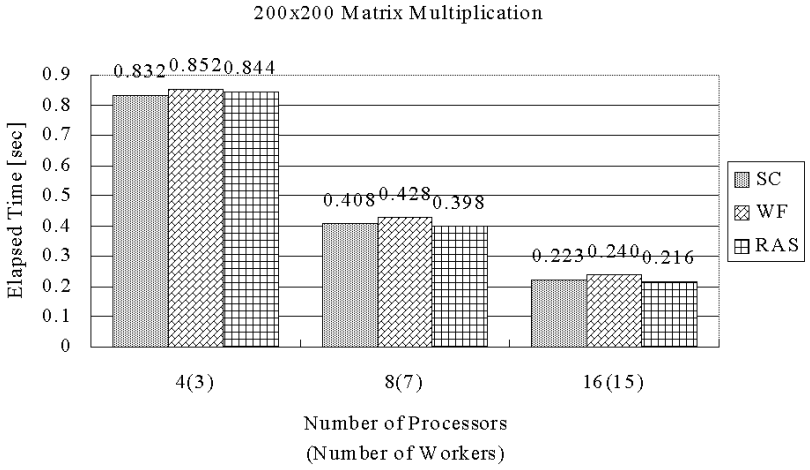

Figure 8: The elapsed time of 200x200 matrix multiplication under the dedicated running environment

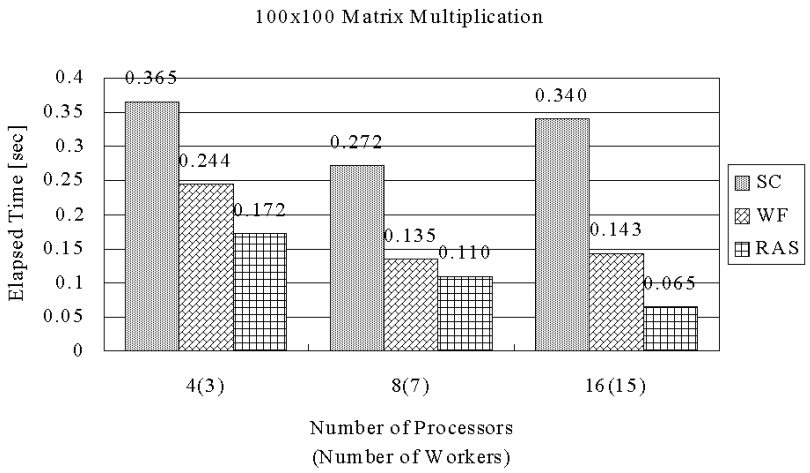

Figure 9: The elapsed time of 100x100 matrix multiplication under the shared running environment

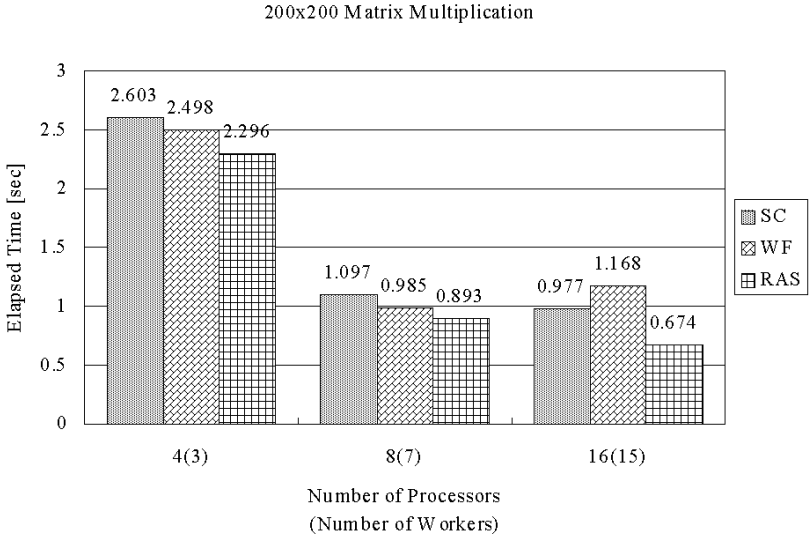

Figure 10: The elapsed time of 200x200 matrix multiplication under the shared running environment

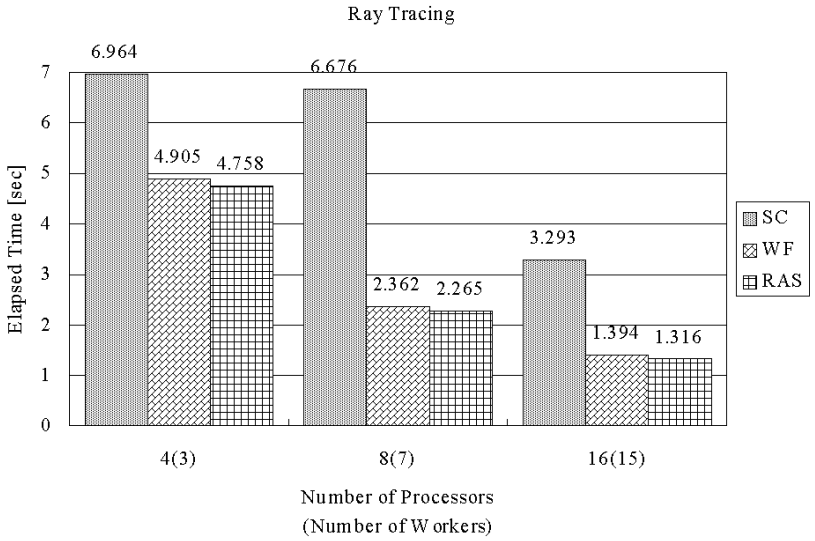

Figure 11: The elapsed time of ray tracing under the dedicated running environment

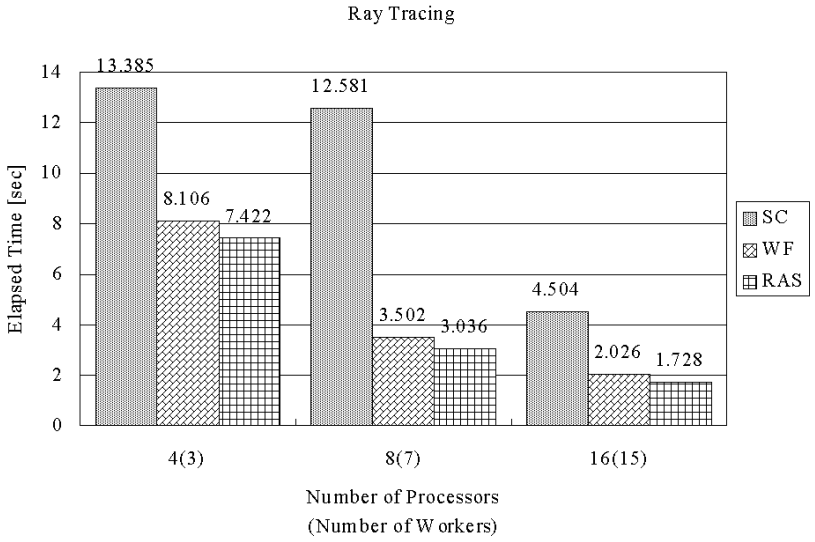

Figure 12: The elapsed time of ray tracing under the shared running environment