一個應用在異質型個人電腦叢集系統的平行迴圈排程方法

研究生：張順奇　　指導教授：楊朝棟博士

東 海 大 學

資 訊 工 程 與 科 學 系

摘要

具可擴充性的計算叢集(Scalable computing clusters)正快速變成高效能與大規模計算的標準平台。這是由於叢集系統具有低價、高效能等特質，並具備可使用現有硬體元件的高可用性。然而，目前似乎沒有為叢集系統量身訂做的排程方法。現有的 self-scheduling 方法其實是建立在 SMP 系統的，雖然也可以在叢集系統使用，但我們發現，在部分異質型叢集系統中，可能會有一些問題發生。在這篇論文中，我們提出一個可以在任何異質型叢集系統中，在規律性迴圈的情況下，得以有效率的運行。我們的方法是：把迴圈中 iteration 數目的 a%根據叢集系統工作電腦的處理器時脈來分配工作，剩下的(100-a)%再根據已知的 self-scheduling 方法來排程。在我們所架設的極度異質型環境中，a=75 時，使用我們的方法能比傳統的 self-scheduling 方法減少 13-61%的執行時間。

A Parallel Loop Self-Scheduling for Heterogeneous PC Clusters

Student: Shun-Chyi Chang Advisor: Dr. Chao-Tung Yang

Department of Computer Science and Information Engineering
Tunghai University
Taichung, 407, Taiwan, Republic of China

Abstract

Scalable computing clusters are rapidly becoming a standard platform for high performance and large-scale computing. This is due to their low cost, high performance, high availability of off-the-shelf hardware components and freely accessible software tools that can be used for developing applications. However, there is few scheduling scheme designed for cluster. Known scheduling schemes are based on SMP architecture. Although these schemes are function on cluster system also, there are some problems might happen in heterogeneous cluster system.

In this thesis, we revise known loop self-scheduling schemes to fit all heterogeneous PC clusters environment when loop is regular. We propose an approach to partition loop iterations and achieve good performance in any heterogeneous environment: partition a% of workload according to their performance weighted by CPU clock and the rest (100-a)% of workload according to known self-scheduling. Many various a values are applied to the matrix multiplication and a best performance is obtained with a=75. We also apply our schemes on both simulated increasing and decreasing workload loops and get obviously performance improvement. Therefore, our approach is suitable in all applications with regular loops.

Acknowledgement

I would like to thank all the people who have made writing this thesis a more pleasant task. In particular, I'd like to thank my principal advisor, Dr. Chao-Tung Yang, who introduced me to this topic and gave me broad support and guidance throughout my time at Tunghai. I'd kike to thank Professor Wuu Yang, Professor Nai-Wei Lin and Professor Yi-Min Wang for their valuable comments and advice given while serving on my reading committee.

There are many other people whom I would to thank. The administrator of Taichung Office Bureau of Consular Affairs Ministry of Foreign Affairs, Henrich Lin, allowed me to pursue further education in Tunghai. Many colleagues encouraged and supported me on studying. For them, I can make writing this thesis with no fear of disturbance in the rear.

Last, but certainly not the last, I'd like to thank my family and all my friends whose unconditional support made this thesis possible.

Content

Tables

Figures

Chapter 1

Introduction

1.1 Cluster Computing

To use supercomputer for high-performance computing has been growing. Parallel processing has been a most important technology in modern computing for several decades. Many powerful multiprocessor hardware systems have been developed to exploit parallelism for concurrent execution. Supercomputers that are single big expensive machines with a shared memory and one or more processors meet the professional need. A large-scale processing and storage system that provides high bandwidth at low cost is expected.

The use of loosely coupled, powerful and low-cost commodity components (PCs or workstations, typically) connected by high-speed network has resulted in the widespread usage of a technology popularly called *cluster computing*. Scalable computing clusters, ranging from a cluster of (homogeneous or heterogeneous) PCs or workstations, to SMPs (Symmetric MultiProcessors), as shown in Figure 1.1, are rapidly becoming the standard platforms for high-performance and large-scale computing.

A cluster, as shown in Figure 1.2, is a group of independent computer systems and thus forms a loosely coupled multiprocessor system. Usually, a cluster node contains its own disk and equipped with a complete operating systems, and therefore, it also can handle interactive jobs. Each node can function only as an individual resource while a cluster system presents itself as a single system to the user. A network is used to provide inter-processor communications. Applications that are distributed across the processors of the cluster use either message passing or network shared memory for communication. Cluster nodes work collectively as a single computing resource and fill the conventional role of using each node as an independent machine. [16, 17, 20]

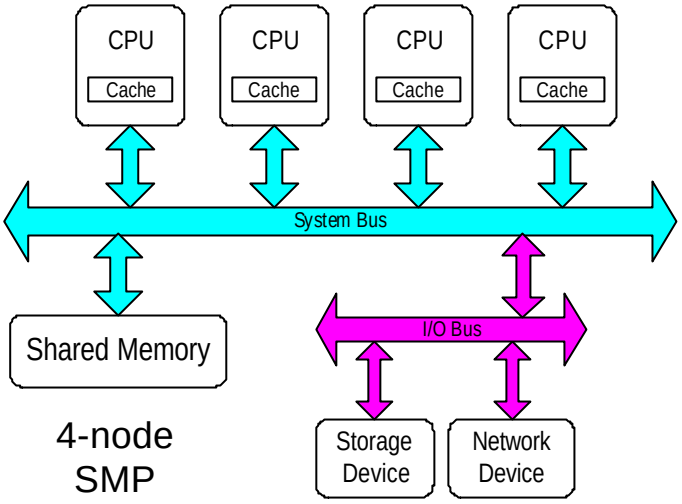

Figure 1.1 The structure of a typical SMP with four processors

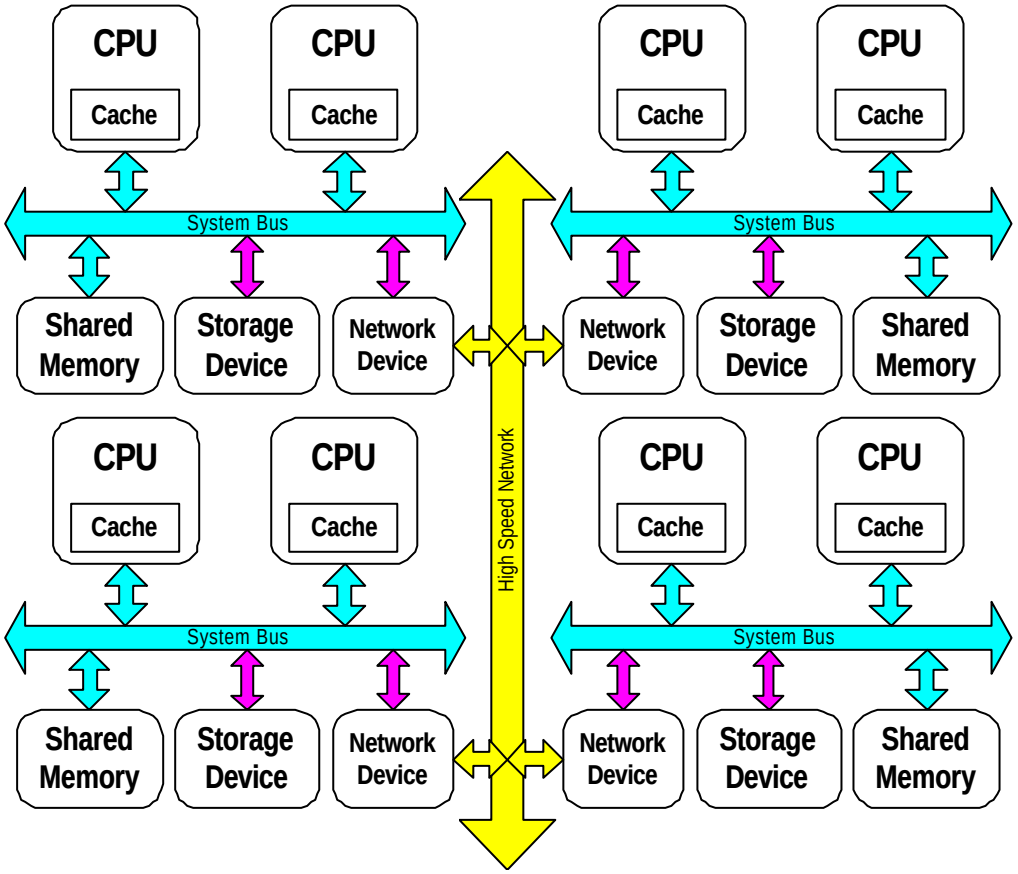

Figure 1.2 A typical cluster system with eight processors

Cluster computers can offer a number of specific benefits:

- **Cost-effective:** One of the main benefits of a cluster is its cost-effectiveness. Clusters are built from relatively inexpensive commodity components that are widely available.
- **Keeps pace with technologies:** Since Clusters only use mass-market components; it is easy to employ the latest technologies to maintain the cluster as a state-of-the-art system.
- **Flexible configuration:** Users can tailor a configuration that is feasible to them and allocate the budget wisely to meet the performance requirements of their applications. For example, a fine-grain parallel application (which exchange small messages frequently among processors) may motivate users to allocate a larger portion of their budget to high-speed interconnects.
- **Scalability:** When the processing power requirement increases, the performance and size of a cluster can be easily scaled up by adding more compute nodes.
- **High availability:** Each compute node of a cluster is an individual machine. The failure of a compute node will not affect other nodes or the availability of the entire cluster.
- **Compatibility and portability:** Due to the standardization and wide availability of message passing interface, such as MPI[18] and PVM[19], the majority of parallel applications use these standard middleware. A parallel application using MPI or PVM can be easily ported to a cluster. This is why clusters are rapidly replacing these expensive parallel computers in the low-end to midrange HPC market.

1.2 Performance Evaluation

Since cluster computer can work collectively as a single computing resource, how to evaluate the performance of cluster computer is an important issue. A measure of relative performance between a multiprocessor system and a single processor system is the speedup factor, *S(n)*, define as

$$S(n) = \frac{\text{Execution time using one processor (single processor system)}}{\text{Execution time using a multiprocessor with n processors}} = \frac{t_s}{t_p} \quad \text{——}$$

Where t_s is the execution time on a single processor and t_p is the execution time on a multiprocessor. $S(n)$ gives the increase in speed in using a multiprocessor.

Amdahl's law [14] gives another heuristic pointer. Assuming there will be some parts are only executed on one processor, the ideal situation would be for all the available processors to operate simultaneously for the other times. If the fraction of the computation that cannot be divided into concurrent tasks is f, and no overhead incurs when the computation is divided into concurrent parts, the time to perform the computation with n processors is given by $ft_s + (1-f)t_s / n$, as illustrated in Figure 1.3 [10]. Illustrated is the case with a single serial part at the beginning of the computation, but the serial part could be distributed throughout the computation. Hence, the speedup factor is given by

$$S(n) = \frac{t_s}{ft_s + (1-f)t_s / n} = \frac{n}{1+(n-1)f}$$

It is clearly that the less fraction of the computation that cannot be divided into concurrent tasks, the more speedup can be got.

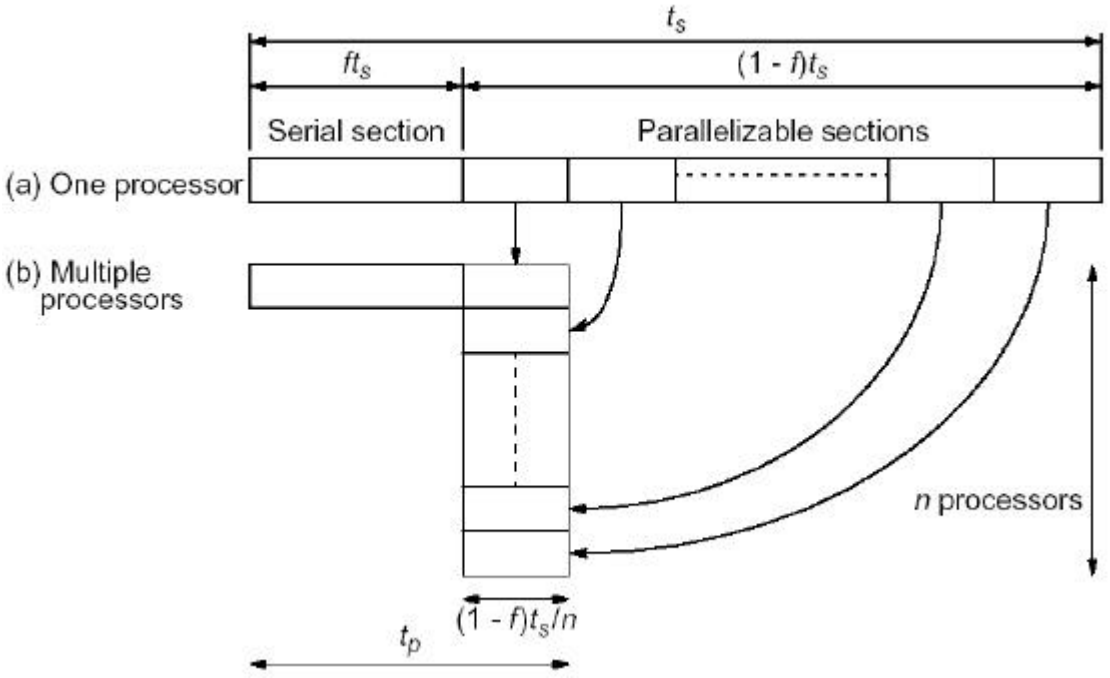

Figure 1.3 Parallelizing sequential problem-Amdahl's law

1.3 Motivation

To exploit the potential computing power of cluster computers, a key point is how to assign tasks to computers so that the computer loads are well balanced. That is how to assign the different parts of a parallel application to the computing resources to minimize the overall computing time and to efficiently use the resource.

Ideally, we want all the processors to be operating continuously on tasks that would lead to the minimum execution time. Achieving this goal by spreading the tasks evenly across the processors is called *load balancing*. Figure 1.4 is an illustration of load balancing on which the execution time of the program will be very different in (a) and (b).

Using a suitable scheduling approach is very important in the cluster computing system. Unfortunately, there are few schemes designed for cluster system. Known scheduling schemes are based on SMP architecture. Although these schemes are function on cluster system also, there are some problems might happen in heterogeneous cluster system.

On the other hand, parallel computers are becoming increasingly widespread, nowadays many of these parallel computers are no longer shared-memory multiprocessors, but rather follow the distributed memory model for scalable. These systems may consist of homogeneous workstations, where all the workstations have processors, memory and caches with exactly the identical specifications. However, more and more systems are now composed of a number of heterogeneous workstations clustered together, where each workstation may have different CPU performance capabilities, different amounts of memory and caches, and even different architectures and operating systems.

Moreover, scalability should be an important character of cluster. An additional PC might not help the performance if we use a wrong scheduling approach. We want to propose a heuristic scheduling approach which is suitable on heterogeneous environment and is scalability.

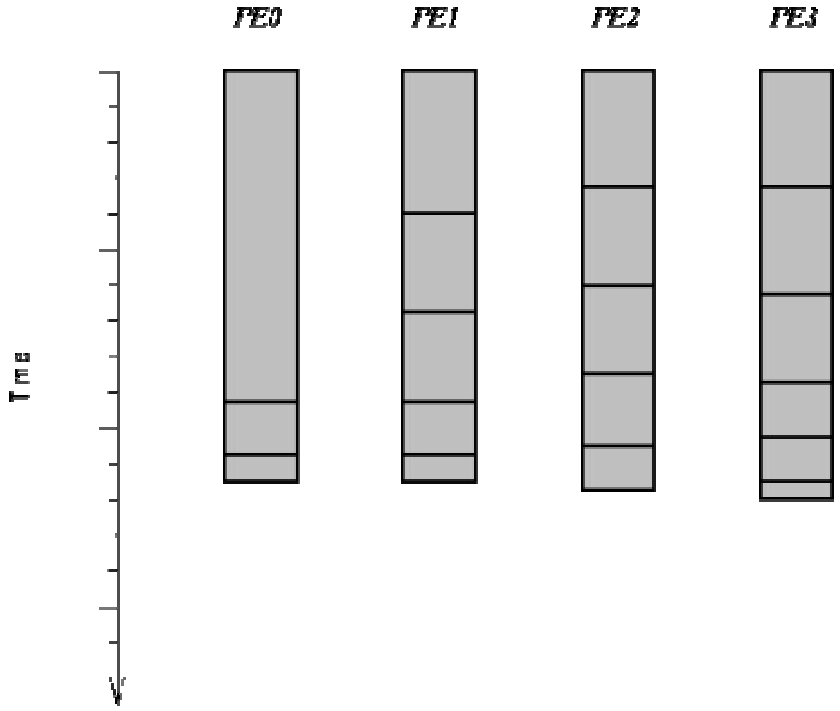

Figure 1.4(a) A suitable scheduling illustration

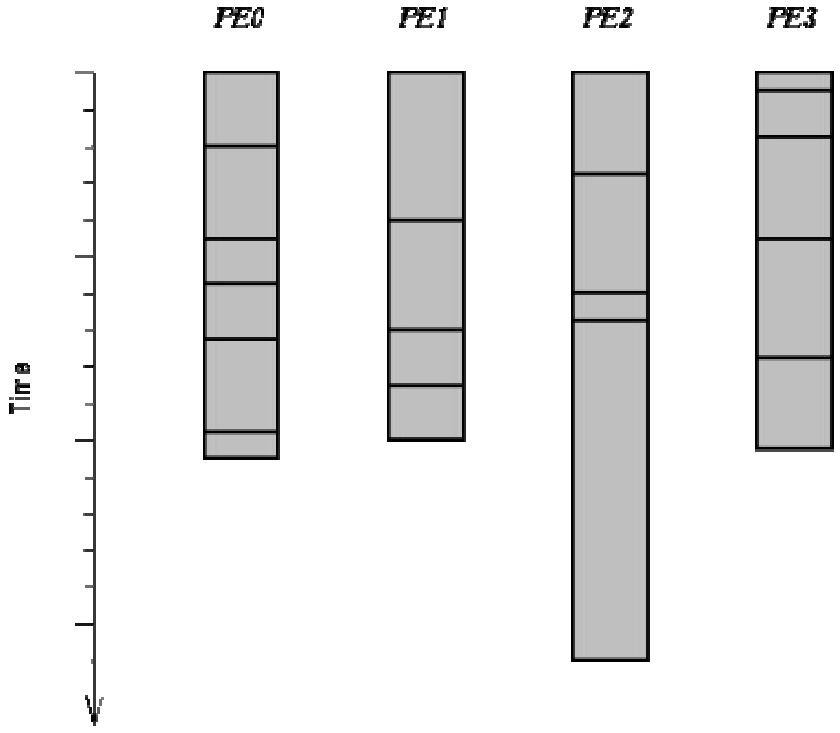

Figure 1.4(b) A unsuitable scheduling illustration

1.4 Thesis Organization

The rest of the thesis is organized as follows. In Chapter 2, a brief overview of self-scheduling is given. It provides the background necessary to understand this thesis effort. Chapter 3 states our approach and describes our system architecture. The experiments which are according to our approach will be shown in Chapter 4. We will also discuss the result and propose our suggestion at the same time. Chapter 5 has our conclusion remarks and future work.

Notation:

The following are common notations used throughout the whole paper:

- *PE* is a processor in the parallel or distributed system;
- *I* is the total number of iterations of a parallel loop;
- *p* is the number of *PE*s in the parallel or distributed system;
- A *chunk* is a collection of consecutive iterations. C_i is the chunk-size at the *i*-th scheduling step (where *i*=1, 2, 3, …);
- W_i is the workload of C_i;
- $W = \sum W_i$, the total workload in a loop;
- *N* is the number of scheduling steps;
- P_i is the CPU clock of processors *i*.
- $P = \sum P_i$, the sum of all CPU clock value in our cluster system.
- A_i is the Actual computing performance.
- $A = \sum A_i$, the total computing performance in our cluster system.

Chapter 2

Background

2.1 Loop Scheduling

Loops often comprise a large portion of a program's parallelism. An efficient approach to extract potential parallelism is to concentrate on the parallelism available in the loops. However, loops are not always easy to be paralleled. Data dependence is an obstacle. Data dependence is said to exist between two statements S_1 and S_2 if there is an execution path from S_1 to S_2, if both statements access the same memory location and if at least one of the two statements writes the memory location. There are three types of data dependences: True (flow) dependence occurs when S_1 writes a memory location that S_2 later reads. Anti-dependence occurs when S_1 reads a memory location that S_2 later writes. Output dependence occurs when S_1 writes a memory location that S_2 later writes [11]. There are many researches [12, 13] focus on how to deal with DOACROSS loop but we just concentrate on DOALL loop.

A loop is called a DOALL loop if there is no cross-iteration dependence in the loop; i.e., all the iterations of the loop can be executed in parallel. If all the iterations of a DOALL loop are distributed among different processors evenly, a high degree of parallelism can be exploited. Parallel loop scheduling is a method that attempts to evenly schedule a DOALL loop on multiprocessor systems.

In homogeneous environment, workload can be partitioned equally to each working computer, but in heterogeneous environment, this method will not work. Some researches were proposed to solve parallel loop scheduling problems on heterogeneous cluster environments by using self-scheduling schemes. However, these self-scheduling schemes might not in some situation.

In a parallel processing system, two kinds of parallel loop scheduling decisions can be made either statically at compile-time or dynamically at run-time.

Static scheduling is usually applied to uniformly distributed iterations on processors [6]. It has the drawback of creating load imbalances when the loop style is not uniformly distributed; when the loop bounds cannot be known at compile-time; when system is heterogeneous; or when locality management cannot be exercised. In contrast, dynamic scheduling is more appropriate for load balancing; however, the runtime overhead must be taken into consideration. In general, parallelizing compilers distribute loop iterations by using only one kind of scheduling algorithm, either static or dynamic.

2.1.1 Static Scheduling

Theoretically, workload can be partitioned according to their computer performance. Unfortunately, in heterogeneous system, it is difficult to evaluate each computer performance. Intuitively, CPU clock speed may be a good evaluation value. But it seems not enough. Many factors affect computer performance, such as the performance capability of the CPU, the amount of memory available, the cost of memory access, the communication medium between processors… etc [5].

Bohn and Lamont try to evaluate the performance of computer in compiler-time [4]. In their experiment, HINT is a good benchmark. It evaluates processor and memory perfo rmance for any data type and returns a single value, "QUIPS". Bohn and Lamont declared "QUIPS" can present the computer performance. It has the advantage of all computers being working computer - no control computer is needed. But, HINT requires hours to execute. It means this way will not be scaling well. It takes a long time to add one more computer and if we want to change the peripheral, for example to replace RAM from PC100 to PC133, we might have to rerun HINT.

Traditional static scheduling [6] is applied when each loop iteration takes roughly the same amount of time, and the compiler knows how many iterations will be run and how many processors are available for use at compile-time. It has the advantage of incurring the minimum scheduling overhead, but load imbalances may occur. These static scheduling schemes including Block Scheduling, Cyclic Scheduling, Block-D Scheduling, Cyclic-D Scheduling… etc [6].

- **Block Scheduling**

  In block scheduling, I iterations are divided into I/p round. Each round consists of consecutive iterations and is assigned to one processor. This is only suitable for uniformly distributed loop iterations.

- **Cyclic Scheduling**

  Instead of assigning a processor a consecutive block of iterations, iterations are assigned to different processors in a cyclic fashion, i.e., iteration i is assigned to processor (i mod p). This method may produce a more balanced schedule than block scheduling for some non-uniformly distributed parallel loops.

- **Block cyclic scheduling**

  It is a compromise between block scheduling and cyclic scheduling. This algorithm assigns blocks of a fixed size to processors in a round robin fashion. If the block size is equal to one, then it degenerates to cyclic scheduling. If the block size is I/p then it is same as block scheduling. Hence, block cyclic scheduling forms a continuum between block and cyclic scheduling algorithms.

An example shown the different between these approaches are given in Table 2.1. Nevertheless, these scheduling schemes were unsuitable in heterogeneous environment.

Approach	CPU	Partition Size
Block	1	1, 2, 3, …, 250
	2	251, 252, 253, …, 500
	3	501, 502, 503, …, 750
	4	751, 752, 753, …, 1000
Cyclic	1	1, 5, 9, 13, …
	2	2, 6, 10, 14, …
	3	3, 7, 11, 15, …
	4	4, 8, 12, 16, …
Block cyclic	1	1, 2, 3, 4, 17, 18, 19, 20, …
	2	5, 6, 7, 8, 21, 22, 23, 24, …
	3	9, 10, 11, 12, 25, 26, 27, 28, …
	4	13, 14, 15, 16, 29, 30, 31, 32, …

Table 2.1 A table of partition size using various approaches

2.1.2 Dynamic Scheduling

Dynamic scheduling adjusts the schedule during execution and is especially suitable whenever the number of iterations is uncertain or each iteration may take a different amount of time. Although it is more suitable for load balancing between processors, runtime overhead is the cost.

We use master/slave computation patterns to model problems, i.e., the master coordinate data distribution to the slaves, which perform computations and transmit the results back to the master. The master is not responsible for workload, the idle slave requests to the master for new loop iterations, and no communication occurs between slaves. How many iterations that a slave should be assigned is a critical issue. Improper assignment will cause bad system performance.

Self-scheduling is a large class of adaptive/dynamic centralized loop scheduling schemes. In a common self-scheduling scheme, p denotes the number of processors, I denotes the total iteration and $f()$ is a function to produce the chunk-size at each step. At the i-th scheduling step, the master computes the chunk-size C_i and the remaining number of tasks R_i,

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

where $f()$ possibly has more parameters than just i and p, such as R_{i-1}. The master assigns C_i tasks to an idle slave and the load imbalancing will depend on the execution time gap between t_j, for $j=1,\ldots,p$ [7, 15].

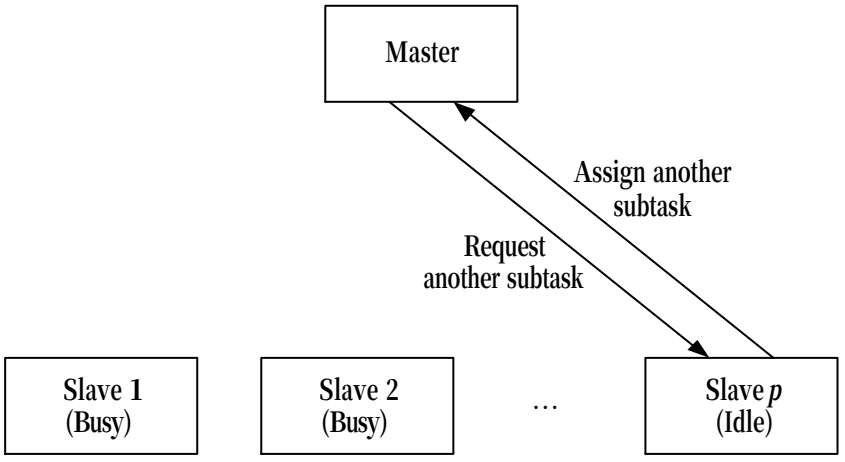

Figure 2.1 A master/slave model

Different ways to compute C_i have given rise to different scheduling schemes. The most notable examples are as following:

- **Pure Self-Scheduling (*PSS*)**

Formula: $C_i = 1$

This is the easiest and most straightforward dynamic loop scheduling algorithm [8]. Whenever a processor is idle, one iteration is assigned to it. This algorithm achieves good load balancing but also induces excessive overhead.

- **Chunk Self-Scheduling (*CSS*)**

Formula: $C_i = k$, where $k \geq 1$ (known as chunk size is chosen by the user).

Instead of assigning one iteration to an idle processor as in self-scheduling, *CSS* assigns *k* iterations each time, where *k*, called the chunk size, is fixed and must be specified by either the programmer or the compiler [8]. When the chunk size is one, this scheme is pure self-scheduling, as discussed above. If the chunk size is set to the bound of the parallel loop equally divided by the number of processors, the scheme becomes static scheduling. A large chunk size will cause load imbalancing while a small chunk is likely to produce too much scheduling overhead. For different partitioning schemes, we adopted *CSS*(*k*), which is a modified version of *CSS*, where *k* means the size of chunks.

- **Guided Self-Scheduling (*GSS*)**

Formula: $C_i = \lceil R_{i-1} / p \rceil$

This algorithm can dynamically change the number of iterations assigned to each processor [2]. More specifically, the next chunk size is determined by dividing the number of remaining iterations of a parallel loop by the number of available processors. The property of decreasing chunk size implies an effort is made to achieve load balancing and to reduce the scheduling overhead. By assigning large chunks at the beginning of a parallel loop, one can reduce the frequency of communication between master and slaves. The small chunks at the end of a loop partition serve to balance the workload across all the working processors.

- **Factoring (*FSS*)**

Formula: $C_i = \lceil R_{i-1} / ap \rceil$, where the parameter a is computed by a probability distribution or is suboptimally chosen $a = 2$.

In some cases, *GSS* might assign too much work to the first few processors, so that the remaining iterations are not time-consuming enough to balance the

workload. This situation arises when the initial iterations in a loop are much more time-consuming than the later iterations. The Factoring algorithm addresses this problem [1]. The assignment of loop iterations to working processors proceeds in phases. During each phase, only a subset of the remaining loop iterations (usually half) is divided equally among the available processors. Because Factoring assigns a subset of the remaining iterations in each phase, it balances loads better than *GSS* does when the computation times of loop iterations vary substantially. In addition, the synchronization overhead of Factoring is not significantly larger than that of *GSS*.

- **Trapezoid Self-Scheduling (*TSS*)**

Formula: $C_i = C_{i-1} - D$, with trunk size: $D = \lfloor (F - L)/(N - 1) \rfloor$, where the first and last chunk-size(F, L) are proposed that $F = \lfloor I / 2p \rfloor$, $L = 1$, $N = \lceil (2I)/(F + L) \rceil$.

This approach tries to reduce the need for synchronization while still maintaining a reasonable load balance [3]. This algorithm allocates large chunks of iterations to the first few processors and successively smaller chunks to the last few processors. The difference in the size of successive chunks is always a constant in *TSS* whereas it is a decreasing function in *GSS* and in Factoring.

- **Intelligent Parallel Loop Scheduling (*IPLS*)**

Fann, Yang, Tseng and Tsai propose a knowledge-based approach to solving loop-scheduling problems [9]. A rule-based system, called *IPLS*, is developed by combining a repertory grid and an attribute ordering table to construct a knowledge base. *IPLS* chooses an appropriate scheduling algorithm by inferring some features of loops and assigning parallel loops to multiprocessors to achieve significant speedup. However, this system is based on UMA architecture and not suitable on cluster architecture yet.

Table 2.2 shows the different chunk sizes for a problem with the number of iteration I=1000 and the number of processor p=4.

Scheme	Partition size
PSS	1, 1, 1, 1, 1, 1, 1…
CSS(125)	125, 125, 125, 125, 125, 125, 125, 125
GSS	250, 188, 141, 106, 79, 59, 45, 33, 25, 19, 14, 11, 8, 6, 4, 3, 3, 2, 1, 1, 1,1
FSS	125, 125, 125, 125, 63, 63, 63, 63, 31, 31, 31, 31, 16, 16, 16, 16, 8, 8, 8, 8, 4, 4, 4, 4, 2, 2, 2, 2, 1, 1, 1, 1
TSS	125, 117, 109, 101, 93, 85, 77, 69, 61, 53, 45, 37, 28
IPLS	Auto detect loop attributes and decide the loop partition strategy

Table 2.2 Sample partition sizes when *I*=1000 and *p*=4

Chapter 3

Our Approach and System Description

3.1 The Extreme Heterogeneous Environment

Known self-scheduling schemes according to formula to partition size of loop iteration. Unfortunately, if the ratio W_i/W is greater than the A_i/A in some slave computer, load imbalance happens. We call this slave computer "*the dominate computer*". If there exists a dominate computer in a cluster, we called this cluster in "*the extreme heterogeneous environment*". For example, there are two slaves. In *FSS*, every slave will get $I/4$ iterations at first step. If the performance difference between the fastest computer and the slowest computer is larger than 3, load imbalance happens.

In this condition, an additional slave computer may not lead to a better performance by these known self-scheduling schemes since they partition size of loop iteration according to formula instead of computer performance.

A combination of different machine types is used to test the behavior of these approaches in a heterogeneous computing environment, and the matrix multiplication is chosen as the test application to get a heuristic result due to its regular behavior. This experiment included four computers. One of them is assigned as master using some self-scheduling approach to partition size of loop iteration. The master is a PC with 300 MHz CPU speed and 208MB physical memory. Three slaves are PCs, respectively, with 1.6GHz CPU speed and 256MB physical memory, 233 MHz CPU speed and 96MB physical memory, and 200MHz CPU speed and 64MB physical memory. The slaves are added into the system sequentially in this order. We respectively use *GSS*, *FSS* and *TSS* approach to test matrix multiplication with different problem sizes from 512*512, 1024*1024 to 2048*2048 by floating point operations.

No. of slaves	Execution time(TSS)			Execution time(FSS)		
	512*512	1024*1024	2048*2048	512*512	1024*1024	2048*2048
1	0'12"066	1'44"357	17'12"483	0'12"136	1'44"688	17'11"402
2	0'17"520	2'49"652	19'34"016	0'18"371	3'16"561	23'48"723
3	0'13"339	1'53"202	16'30"651	0'14"543	2'00"491	16'36"007

Table 3.1 The result performance of number of slaves in extreme heterogeneous environment

Table 3.1 shows our experiment result. Note that just one slave in Table 3.1 means that all work is done by the fastest computer only. It shows the system having two slave computers gets worse performance than having only one slave computer. An additional PC does not help the performance.

According to Moore's law, CPU clock will double in 18 months. As the law still works today, to build clusters consisting of extreme different computer performance becomes demanding.

3.2 Our Approach

For the programs with regular loops, intuitively, we may want to partition problem size according to their CPU clock in heterogeneous environment. However, the CPU clock is not the only factor which affects computer performance. Many other factors also have dramatic influences in this aspect, such as the amount of memory available, the cost of memory accesses, and the communication medium between processors… etc[5]. Using this intuitive approach, the result will be degraded if the performance prediction is inaccurate. A computer with largest inaccurate prediction will be the last one to finish the assigned job.

Loops can be roughly divided into four kinds, as shown in Figure 1: uniform workload, increasing workload, decreasing workload, and random workload loops. They are the most common ones in programs, and should cover most case. These four kinds can be classified two types: regular and irregular. The first three kinds are regular and the last one is irregular.

Different loops may need to be handled in different ways in order to get the best performance. Since workload is predictable in regular loops, it is not necessary to process load balancing at beginning.

We propose to partition problem size in two stages. At first stage, partition the *a*% of total workload according to their performance weighted by CPU clock. In the way, the communication between master and slaves can be reduced efficiently. At second stage, partition following (100-*a*)% of total workload according to known self-scheduling scheme. In the way, load balancing can be archived. This approach can be suitable for all regular loops.

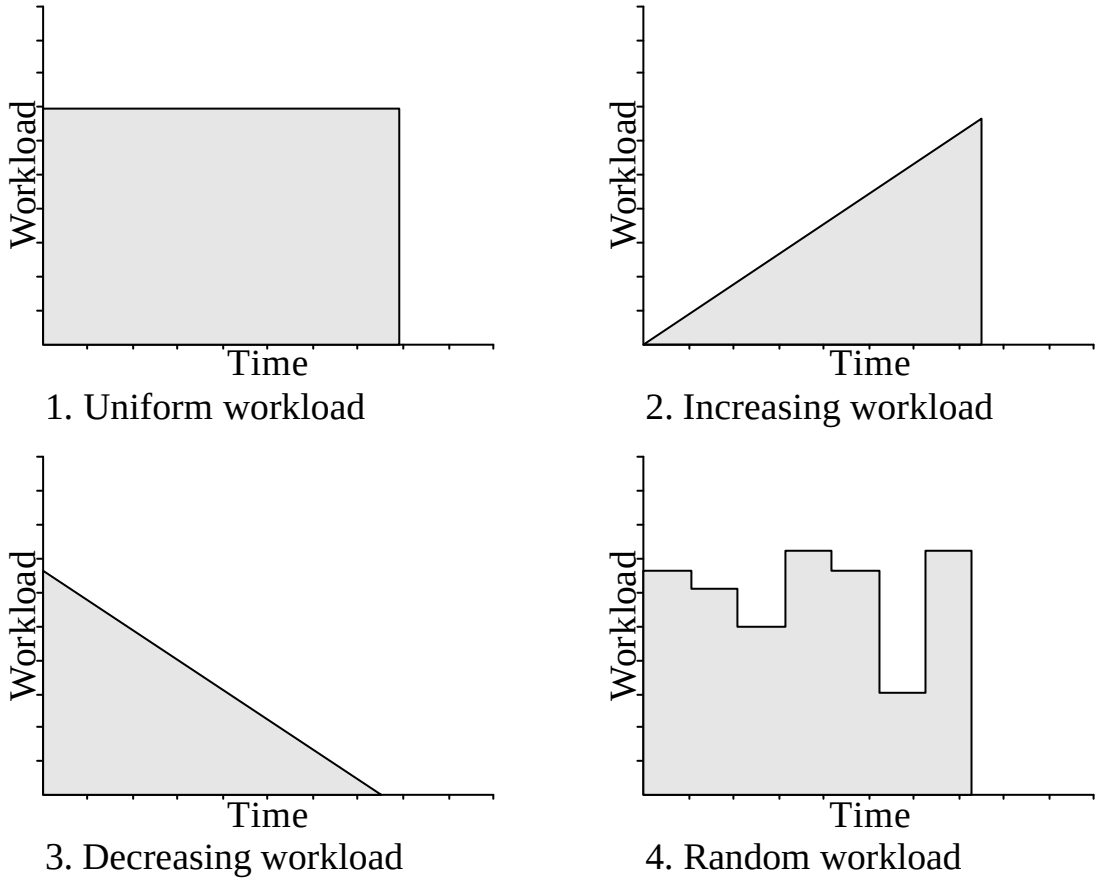

Figure 3.1 Four kinds of loops

With this approach, we don't need to know the real computer performance. The computer finishing its job early gets another larger job. The parameter *a* should not be too small or too big. In former case, the dominate computer will not finish its work. In the latter case, the dynamic scheduling strategy is rigid. In both cases, good performance can not be attained. An appropriate *a* value will lead to good performance.

Furthermore, dynamic load balancing approach should not be aware of the run-time behavior of the applications before execution. But in *GSS* and *TSS*, to achieve good performance, computer performance of each computer in the cluster has to be in order in extreme heterogeneous environment, which is not very applicable. With our schemes, this trouble will not exist.

In this thesis, the terminology "*FSS*-80" stand for "*a*=80, and remainder iterations use *FSS* to partition" and so on.

Example 3.1:

There is a cluster consist of five of the slaves. They are PCs respectively, with 200 MHz, 200 MHz, 233 MHz, 533MHz, and 1.5GHz CPU-clock. Table 3.2 shows the different chunk sizes for a problem with the number of iteration *I*=2048 in this cluster. The number of scheduling steps is parenthesized.

GSS	410, 328, 262, 210, 168, 134, 108, 86, 69, 55, 44, 35, 28, 23, 18, 14, 12, 9, 7, 6, 5, 4, 3, 2, 2, 2, 1, 1, 1, 1(N=30)
GSS-80	923, 328, 144, 123, 121, 82, 66, 53, 42, 34, 27, 21, 17, 14, 11, 9, 7, 6, 4, 4, 3, 2, 2, 1, 1, 1, 1, 1 (N=28)
FSS	205, 205, 205, 205, 205, 103, 103, 103, 103, 103, 51, 51, 51, 51, 51, 26, 26, 26, 26, 26, 13, 13, 13, 13, 13, 6, 6, 6, 6, 6, 3, 3, 3, 3, 3, 2, 2, 2, 2, 2, 1, 1, 1 (N=43)
FSS-80	923, 328, 144, 123, 121 41, 41, 41, 41, 41, 21, 21, 21, 21, 21, 10, 10, 10, 10, 10, 5, 5, 5, 5, 5, 3, 3, 3, 3, 3, 1, 1, 1, 1, 1, 1, 1, 1, 1 (N=39)
TSS	204, 194, 184, 174, 164, 154, 144, 134, 124, 114, 104, 94, 84, 74, 64, 38 (N=16)
TSS-80	923, 328, 144, 123, 121 40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12, 10, 8, 1 (N=23)

Table 3.2 Sample partition size of Example 3.1

To model our approach, we use following terminology:

- *T* is the total workload of all iterations in a loop.
- *W* is the *a*% of total workload.
- *b* is the fewest workload in an increasing/decreasing workload loop. It can be the workload of the first iteration (in an increasing workload loop) or the workload of the last iteration (in a decreasing workload loop).
- *h* is the different of workload between consequence iterations. *h* is a positive integer.
- *x* is the iteration number on which the *a* % accumulating workload is reached. *x* is positive real.
- Due to iteration is unpartitionable, we need a suitable integer to measure *x*. Let it be *m*. For achieving load balancing, we want m>x if possible.

Next, we will induce *m* in different types of regular loops.

3.2.1 Uniform Workload

The workload of each iteration is uniform in this loop type. h=0. Figure 3.2 is an illustration.

Assertion 1: In uniform workload loops, $m = \lfloor I \times a\% \rfloor$.

Proof:

$T = b \times I$

$W = T \times a\% = b \times x$

$x = I \times a\%$

$m = \lfloor I \times a\% \rfloor$

It is trivial. ?

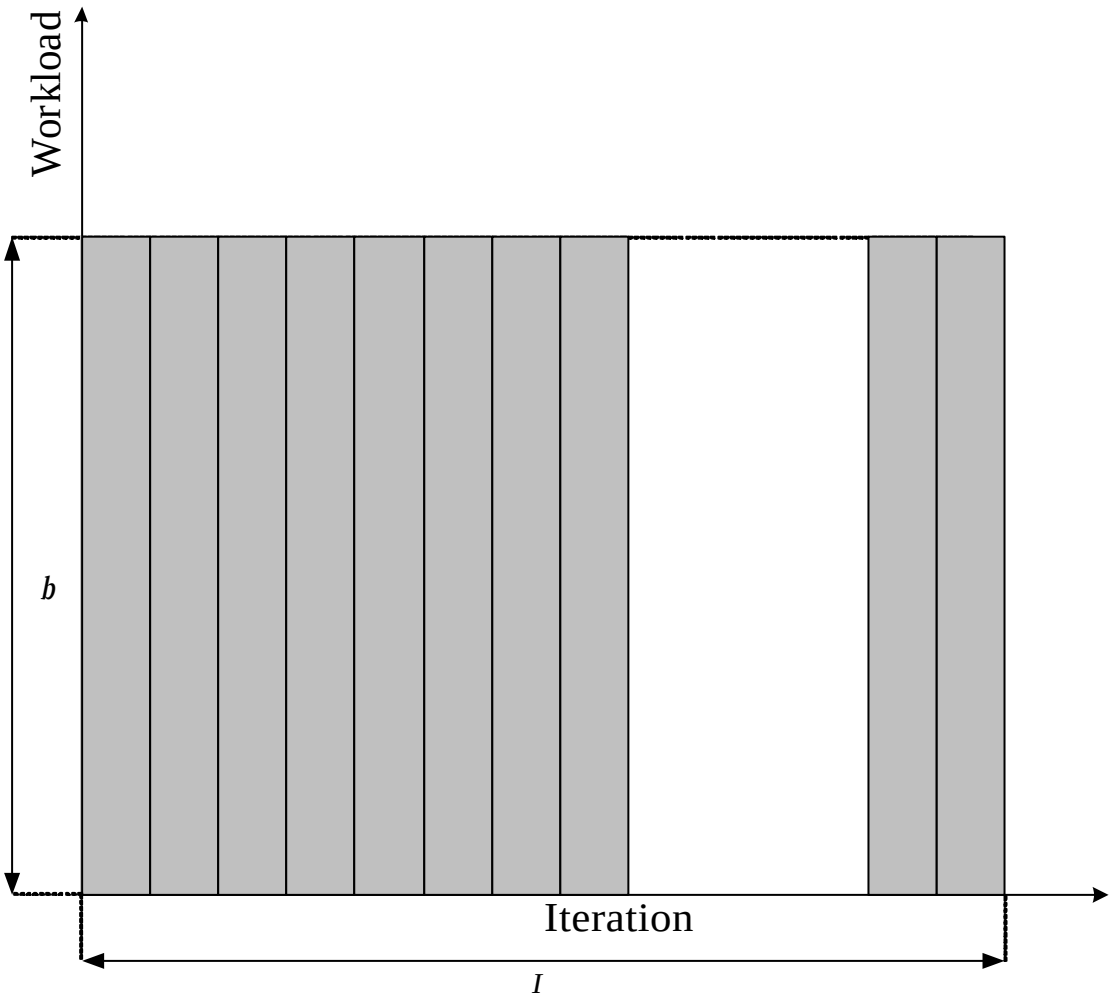

Figure 3.2 A uniform workload loop

3.2.2 Increasing and Decreasing Workload

The workload of each iteration increases (decreases) h units in this loop type. h is constant. Figure 3.3, 3.4 are illustrations.

Assertion 2: In increasing workload loops, $m = \left\lceil \frac{-b + \sqrt{b^2 + 2hW}}{h} \right\rceil$.

Proof:

$$T = \frac{[b + b + (I-1)h]I}{2}$$

$$W = \left\lceil \frac{(2b + (I-1)h)I}{2} \right\rceil \times a\% = \frac{[2b + (x-1)h]x}{2} \quad (1)$$

Let $y = x - 1$

$$\frac{[2b + yh]y}{2} < W < \frac{[2b + xh]x}{2} \quad (2)$$

By first two items of (2), we can get

$$hy^2 + 2by - 2W < 0$$

$$-1 < y < \frac{-2b + \sqrt{4b^2 + 8hW}}{2h}$$

That is

$$0 < x < \frac{-b + \sqrt{b^2 + 2hW}}{h} + 1 \quad (3)$$

By last two items of (2), we can get

$$hx^2 + 2bx - 2W > 0$$

Since x is a positive real,

$$x > \frac{-2b + \sqrt{4b^2 + 8hW}}{2h}$$

$$x > \frac{-b + \sqrt{b^2 + 2hW}}{h} \quad (4)$$

Conclude (3) and (4), we can get

$$\frac{-b + \sqrt{b^2 + 2hW}}{h} < x < \frac{-b + \sqrt{b^2 + 2hW}}{h} + 1 \quad (5)$$

Since $m \in \mathrm{N}$

$$m = \left\lceil \frac{-b + \sqrt{b^2 + 2hW}}{h} \right\rceil \qquad ?$$

Assertion 3: In decreasing workload loops, $m = \left\lfloor I - \frac{-b + \sqrt{b^2 + 2hW'}}{h} \right\rfloor$, where $W' = T - W$.

Proof:

Since we just focus on the DOALL loop, the increasing and decreasing workload loops are equivalence. The increasing workload loop done reversely will become the decreasing workload loop done.

Let $W' = T - W$, and we want to get $x' = I - x$.

According to inequality (5) of Assertion 2:

$$\frac{-b + \sqrt{b^2 + 2hW'}}{h} < x < \frac{-b + \sqrt{b^2 + 2hW'}}{h} + 1$$

$$I - \frac{-b + \sqrt{b^2 + 2hW'}}{h} > I - x > I - \left(\frac{-b + \sqrt{b^2 + 2hW'}}{h} + 1 \right)$$

Since $m \in \mathrm{N}$

$$m = \left\lfloor I - \frac{-b + \sqrt{b^2 + 2hW'}}{h} \right\rfloor \qquad ?$$

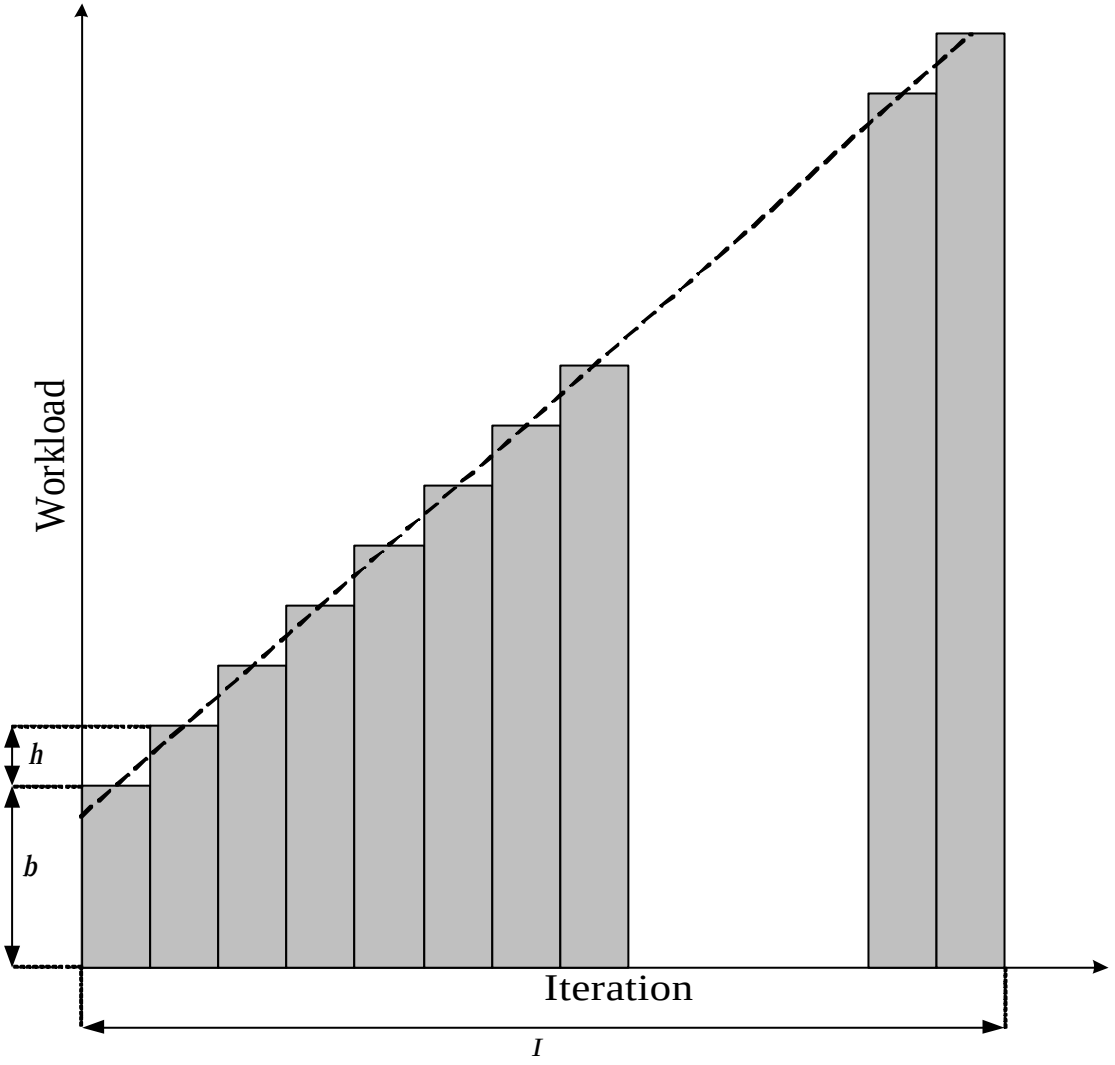

Figure 3.3 An increasing workload loop

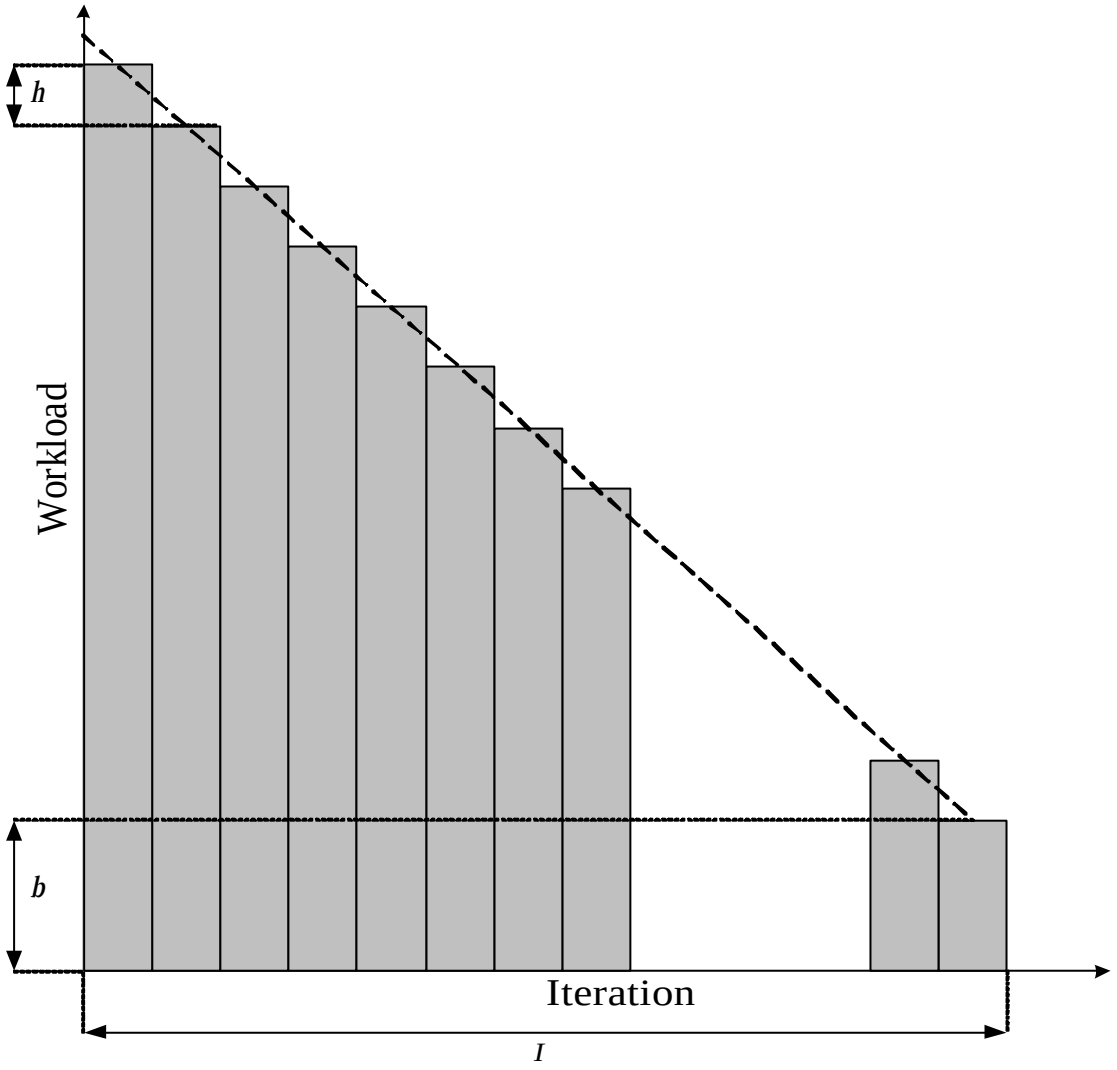

Figure 3.4 A decreasing workload loop

3.3 System Description and Experimental Design

3.3.1 System Description

The approach is applied in an extreme heterogeneous environment which includes six computers, shown as Table 3.3. All computers in this cluster run the RedHat Linux 7.1 operating system (Kernel 2.4.2-2). Program is developed using C language and LAM 6.5.1. The fastest computer is 7.5 times faster than the slowest ones in CPU-clock. HPC2 is assigned as the master and the other five computers are slaves. The host-name in 'lamhost' file is ordered by decreasing CPU-clock except the master computer. The master computer always is the first host-name in 'lamhost' file. Those computers may own various NIC and cost of memory access, regarding as part of computer performance. SWAP may occur in some computers. If SWAP does not occur often, this will not affect the result.

Another cluster is set up to show our proposed approach will still function well on moderate heterogeneous environment. Table 3.4 shows the characteristics of the experimental cluster. All computers in this cluster run the RedHat Linux 7.3 operating system (Kernel 2.4.18-3). The LAM/MPI Version is LAM 6.5.7. The difference of CPU time-clock between the fastest and the lowest computer is not obvious. We let HPC3 be the master and the other computer being the slaves.

Node	Processor	Memory	Operating System/MPI Library
HPC1	200MHz, Intel Pentium	96 MB	Linux Kernel 2.4.2-2/ LAM 6.5.1
HPC2	300MHz, Cyrix M2	208 MB	Linux Kernel 2.4.2-2/ LAM 6.5.1
HPC3	233MHz, Pentium	96 MB	Linux Kernel 2.4.2-2/ LAM 6.5.1
HPC4	600MHz, Intel Pentium II	192 MB	Linux Kernel 2.4.2-2/ LAM 6.5.1
HPC5	1.5GHz, Intel Pentium IV	128 MB	Linux Kernel 2.4.2-2/ LAM 6.5.1

Table 3.3: Characteristics of extreme heterogeneous environment in experiment cluster

Figure 3.5 Our extreme heterogeneous cluster

Figure 3.6 Our moderate heterogeneous cluster

Node	Processor	Memory	Operating System/MPI Library
HPC1	1.6G, AMD Athlon MP	1 GB	Linux Kernel 2.4.18-3/ LAM 6.5.7
HPC2	1.6G, AMD Athlon MP	512 MB	Linux Kernel 2.4.18-3/ LAM 6.5.7
HPC3	1.5G, AMD Athlon MP	512 MB	Linux Kernel 2.4.18-3/ LAM 6.5.7
HPC4	1.5G, AMD Athlon MP	512 MB	Linux Kernel 2.4.18-3/ LAM 6.5.7
HPC5	1.5G, AMD Athlon MP	512 MB	Linux Kernel 2.4.18-3/ LAM 6.5.7

Table 3.4: Characteristics of experimental cluster

3.3.2 Experimental Design

The matrix multiplication is chosen as the experimental application to get a heuristic result due to its regular behavior. The various *a* values are tested in 2048*2048 problem size to get the best performance. Then we use this value to evaluate performance in different problem size and different loop types.

Matrix multiplication is a program with typically uniform workload loop. For increasing workload, we simulate the behavior as following pseudo code.

```
For (i=1, i<=n,i=i+h)
   For (j=0, j<i, j++)
      donothing(msize);
```

`donothing` is a procedure to compute *msize***msize* matrix multiplex. We simulate the decreasing workload loop by reversing the performing order of the increasing ones. The main parameters are following: *b*=1, *h*=1, *I*=360, and *msize*=50 in extreme heterogeneous environment and *b*=1, *h*=1, *I*=360, and *msize*=100 in moderate heterogeneous environment.

All experiments will be tested in extreme heterogeneous environment and moderate heterogeneous environment.

3.4 An Example

An example of matrix multiplex using our approach with *GSS* is as followed:

```
#include <stdio.h>
#include <mpi.h>
#include <math.h>
#include <string.h>

struct ss
   {
```

```
   float cs;
   struct ss *ptrnext;
   };
struct ss *ptrfirst, *ptrthis, *ptrnew, *ptrsent;

int SIZE;
int prs;
float *cpuinfo;
void master(int);
void slave(void);
float get_cpu_clock_speed(void);
void wss(float, int);
void gss(float, int);

int main(int argc, char** argv)
{
   int myrank, numprocs;
   float cpuinfo_l;

   SIZE = atoi(argv[1]);
   prs = atoi(argv[2]);
   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
   cpuinfo = (float*)malloc(numprocs*sizeof(float));

   /* gather cpu information here */
   cpuinfo_l = get_cpu_clock_speed();
   MPI_Gather(&cpuinfo_l, 1, MPI_FLOAT, cpuinfo, 1,
      MPI_FLOAT, 0, MPI_COMM_WORLD);

   if (myrank == 0)
   master(numprocs);
   else
   slave();
   MPI_Finalize();
   return 0;
```

```
}

void master(int numprocs)
{
float *a, *buf;
int i, j, rowc, r, source, tag, count, r1, r2, recsource;
MPI_Status status;

/* get every trunk size */
ptrfirst=(struct ss *)NULL;
r1 = (SIZE*prs)/100;
r2 = SIZE - r1;
wss((float)r1, numprocs-1);

/* initial matrix */
a = (float*)malloc(SIZE*SIZE*sizeof(float));
for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
    a[i*SIZE+j]=2.0;

rowc=1; /* how many data be sent */
r=0;
ptrsent=ptrfirst;

for (i = 1; i < numprocs; i++) {
    MPI_Send(&a[(rowc-1)*SIZE], SIZE*(ptrsent->cs),
       MPI_FLOAT, i, rowc, MPI_COMM_WORLD);
    rowc = rowc + ptrsent->cs;
    ptrsent = ptrsent->ptrnext;
    r++;
    }

gss((float)r2, numprocs-1);

do {
    /* receive data from client */
    MPI_Probe(MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD,
```

```
      &status);
    source = status.MPI_SOURCE;
    tag = status.MPI_TAG;
    MPI_Get_count(&status, MPI_FLOAT, &count);
    buf = (float*)malloc(count*sizeof(float));
    MPI_Recv(&a[(tag-1)*SIZE], count, MPI_FLOAT, source, tag,
      MPI_COMM_WORLD, &status);
    r--;

    free(buf);

    /* sent another size to client */

    if (ptrsent!=(struct ss *)NULL) {
    MPI_Send(&a[(rowc-1)*SIZE], SIZE*(ptrsent->cs),
      MPI_FLOAT, source,
      rowc, MPI_COMM_WORLD);
    rowc = rowc + ptrsent->cs;
    ptrsent = ptrsent->ptrnext;
    r++;
    }
    else {
    MPI_Send(MPI_BOTTOM, 0, MPI_FLOAT, source, 0,
      MPI_COMM_WORLD);
    }

} while (r > 0);
}

void slave(void)
{
float *buf, *b, *c;
int i, j, k, l, f, row, myrank, count, tag, source;
MPI_Status status;
MPI_Request request;

/* initialize matrix */
```

```
b = (float*)malloc(SIZE*SIZE*sizeof(float));
for (i=0; i<SIZE; i++)
    for (j=0; j<SIZE; j++)
       b[i*SIZE+j]=1.0;

/* receive data from master at first time */

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

f=0;

while (status.MPI_TAG >0) {

for (i=0; i<(count/SIZE); i++)
    for (j=0; j<SIZE; j++)
       c[i*SIZE+j]=0.0;

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

    /* sent result*/

    MPI_Send(c, count, MPI_FLOAT, 0, tag, MPI_COMM_WORLD);
   free(buf);
   free(c);

    /* get another size */
```

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

/*********************************************************/
/*This procedure refer to                                 */
/* Advanced Linux Programming.                           */
/*********************************************************/
float get_cpu_clock_speed(void)
{
   FILE* fp;
   char buffer[1024];
   size_t bytes_read;
   char* match;
   float clock_speed;

   fp=fopen("/proc/cpuinfo", "r");
   bytes_read=fread(buffer, 1, sizeof(buffer), fp);
   fclose (fp);

   if (bytes_read==0 || bytes_read == sizeof(buffer))
   return 0;
   buffer[bytes_read]='\0';
   match=strstr(buffer, "cpu MHz");
   if (match==NULL)
   return 0;
   sscanf(match, "cpu MHz : %f", &clock_speed);
   return clock_speed;
}
```

```
void wss(float r, int n)
{
float c, b;
float cpu_total=0;
int i;
c=0;
b=r;
for (i=1; i<=n; i++)
    cpu_total = cpu_total + cpuinfo[i];

ptrfirst=(struct ss *)NULL;
for (i=1; i<=n; i++){

c=ceil((r*cpuinfo[i])/cpu_total);
    ptrnew=(struct ss *) malloc(sizeof(struct ss));
    if (ptrfirst == (struct ss *)NULL)
   ptrfirst=ptrthis=ptrnew;
   else
   {
   ptrthis->ptrnext=ptrnew;
   ptrthis=ptrnew;
   }
    if (b<c)
   c=b;
   ptrthis->cs=c;
   ptrthis->ptrnext=(struct ss *)NULL;
   b=b-c;
}
}

void gss(float r, int numprocs)
{
float c;
int j;
c=0;
j=0;
    while (r != 0)
```

```
/*for (i=1;i<=10;i++)*/
   {
   c=ceil(r/numprocs);
   ptrnew=(struct ss *) malloc(sizeof(struct ss));
       if (ptrfirst == (struct ss *)NULL)
          ptrfirst=ptrthis=ptrnew;
       else
   {
       ptrthis->ptrnext=ptrnew;
       ptrthis=ptrnew;
       if (j==0) ptrsent=ptrthis;
       j++;
   }
   ptrthis->cs=c;
/* numsent=numsent+1;*/
   ptrthis->ptrnext=(struct ss *)NULL;
   r=r-c;
   printf("Size\t%f\n",c);
   }
}
```

Chapter 4

Experimental Results and Discussion

4.1 Extreme Heterogeneous System

Many a values are applied to the experiments with different self-scheduling strategies, shown as Table 4.1 and Figure 4.1, and a=75 result in the best performance in all situations. Note that the column named “a=0” means the usage of known self-scheduling approaches.

Table 4.2 and Figure 4.2 show the result in a=75 with different self-scheduling strategies. The column name "None" stands for "none load-balancing" and workload be partitioned just by CPU clock. Note that in extreme heterogeneous environment, FSS and GSS get worse performance than scheme partitioning workload merely according to the CPU clock. Using our approach in 2048*2048 matrix multiplication will reduce 26.8%, 39.6% and 23.5% execution time than GSS, FSS and TSS respectively.

The a value should depend on system architecture. Different system architecture will have different a value. Using every a value from 60 to 90 will achieve a better performance than just using the known self-scheduling schemes in our system.

Applying a=75 to smaller problem size, 1024*1024 matrix multiplication, or larger problem size, 3072*3072 matrix multiplication, the result was shown in Table 4.3 and Figure 4.3. In 1024*1024 and 3072*3072 matrix multiplication, our approach will reduce execution time 19.5% and 13.1% than GSS, 31.1% and 27.8% than FSS, 14.9% and 23.7% in TSS, respectively.

The result of a=75 in simulated decreasing/increasing workload loop was shown in Table 4.4 and Figure 4.4. In decreasing case, 29.9% in GSS, 61.1% in FSS and 54.2% in TSS, execution time is reduced. Using our approach in simulated increasing workload loop will reduce 59.4%, 48.6.1%, 30.1% execution time than GSS, FSS and TSS respectively.

	a=0	a=60	a=65	a=70	a=75	a=80	a=85	a=90
GSS	853.1	731.1	719.0	681.4	624.1	650.4	690.7	731.5
FSS	1010.5	663.6	658.6	630.2	609.6	650.3	690.4	730.8
TSS	809.6	719.0	697.2	639.3	619.1	650.3	690.1	730.8

Table 4.1 Execution time for 2048*2048 matrix multiplication by various approaches in extreme heterogeneous environment

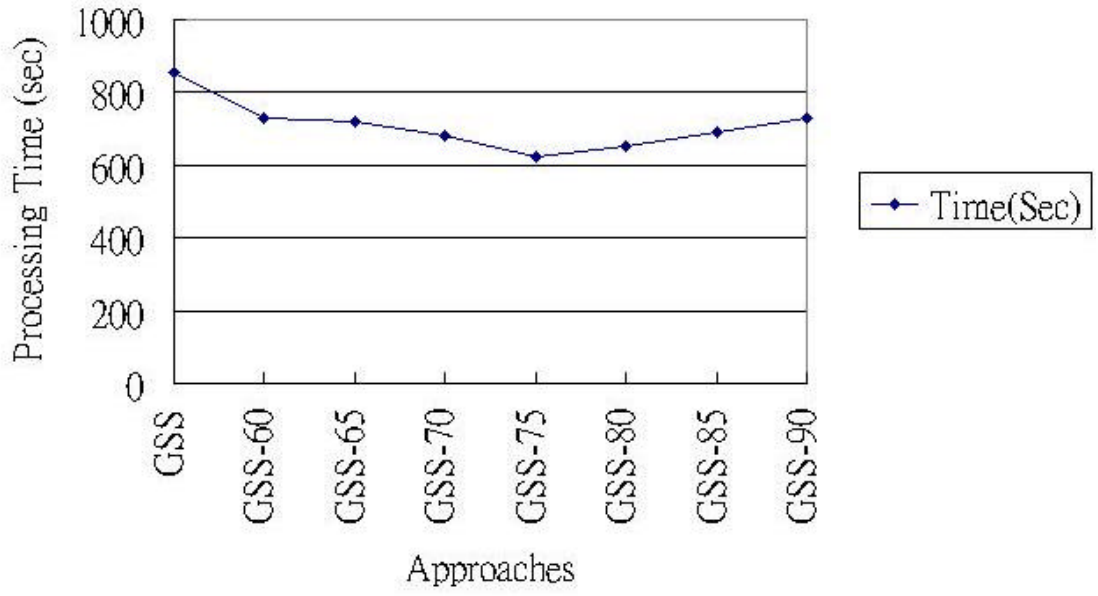

Figure 4.1(a) A chart of execution time of 2048*2048 matrix multiplication by GSS group approach in extreme heterogeneous environment

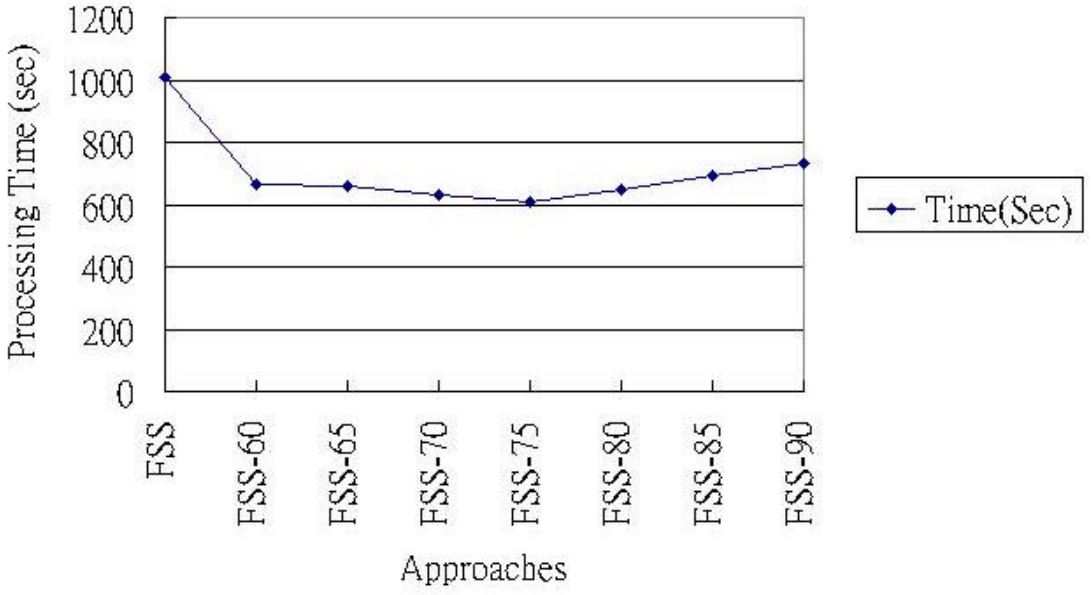

Figure 4.1(b) A chart of execution time of 2048*2048 matrix multiplication by FSS group approach in extreme heterogeneous environment

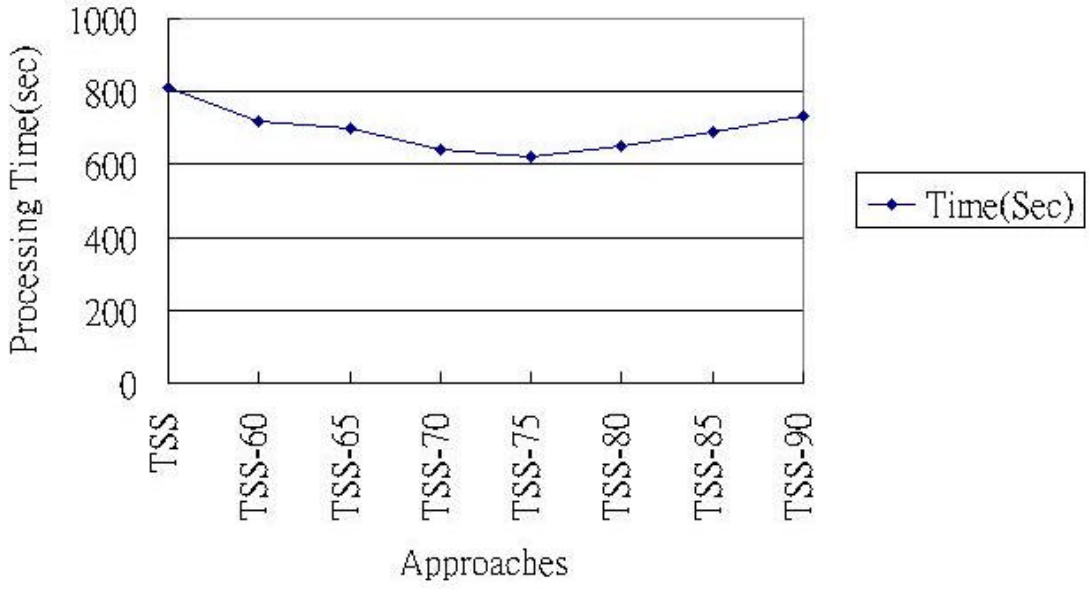

Figure 4.1(c) A chart of execution time of 2048*2048 matrix multiplication by TSS group approach in extreme heterogeneous environment

	None	GSS	GSS-75	FSS	FSS-75	TSS	TSS-75
Execution time	813.3	853.1	624.1	1010.5	609.6	809.6	619.1

Table 4.2 Execution time of 2048*2048 matrix multiplication by various self-scheduling approaches when a=75 in extreme heterogeneous environment

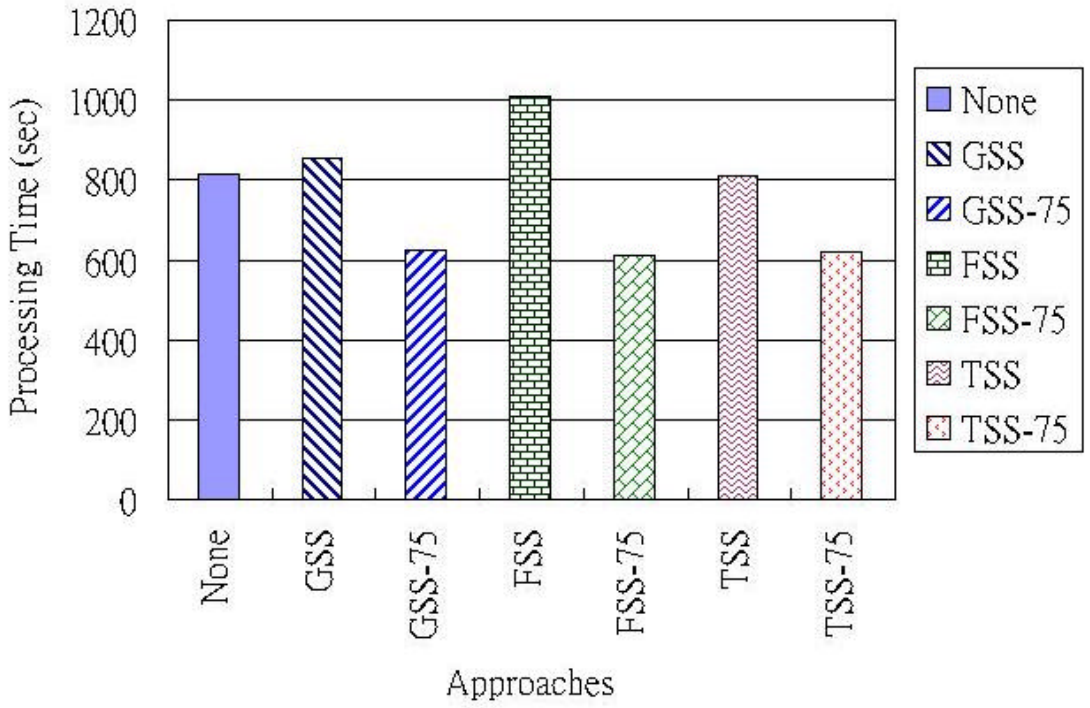

Figure 4.2 A chart of execution time of 2048*2048 matrix multiplication by various self-scheduling approaches when a=75 in extreme heterogeneous environment

	None	GSS	GSS-75	FSS	FSS-75	TSS	TSS-75
1024*1024	113.8	107.3	86.4	125.6	86.6	100.1	85.2
3072*3072	2730.9	2651.2	2305.8	3040.8	2311.7	2849.5	2313.6

Table 4.3 Execution time of different problem size by various self-scheduling approach when a=75 in extreme heterogeneous environment

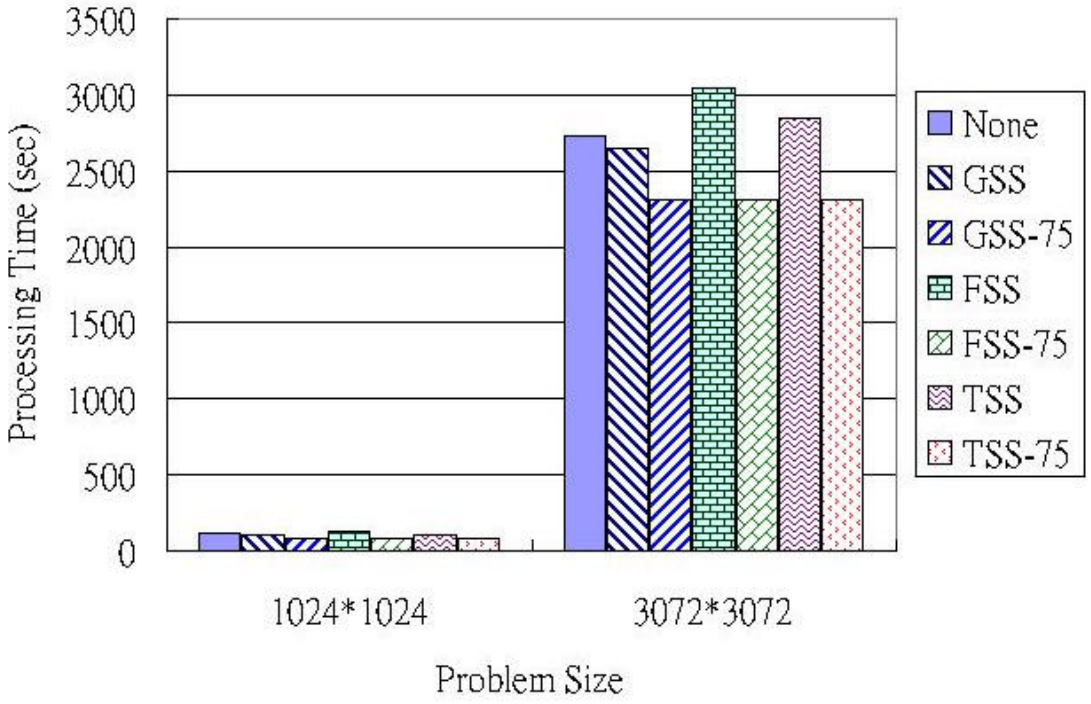

Figure 4.3 A chart of execution time of different problem size by various self-scheduling approach when a=75 in extreme heterogeneous environment

	GSS	GSS-75	FSS	FSS-75	TSS	TSS-75
Decreasing	336.6	244.7	610.3	244.7	552.4	244.7
Increasing	580.2	264.6	475.1	264.6	388.5	264.5

Table 4.4 Execution time of simulated increasing/decreasing workload loop by various self-scheduling approach when a=75 in extreme heterogeneous environment

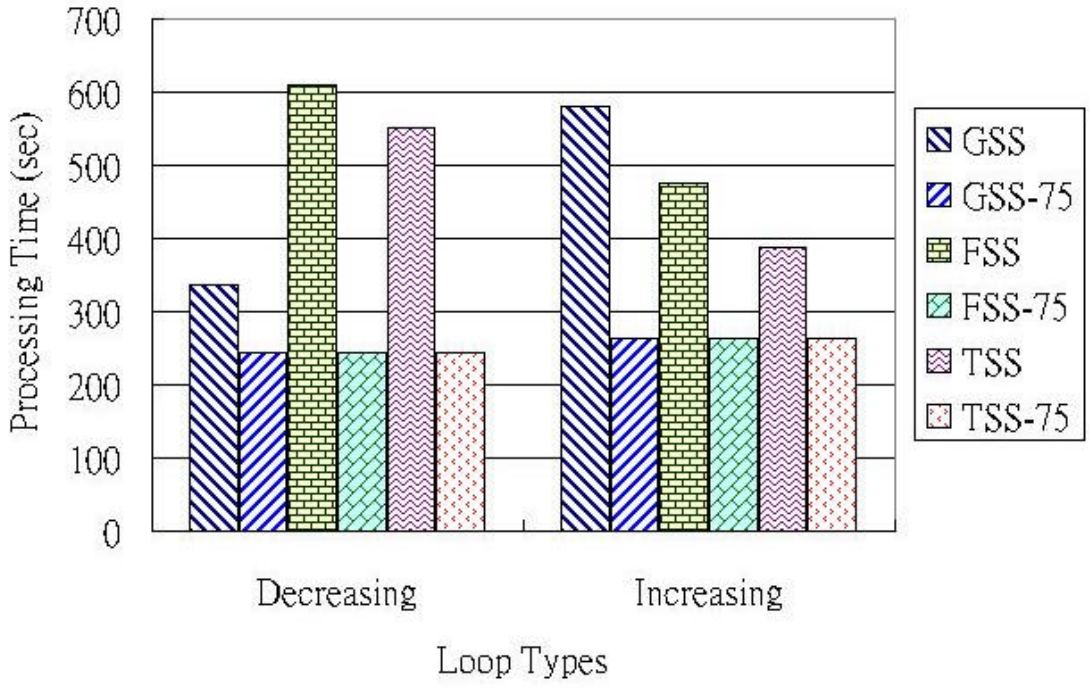

Figure 4.4 A chart of execution time of simulated increasing/decreasing workload loop by various self-scheduling approach when a=75 in extreme heterogeneous environment

4.2 Moderate Heterogeneous System

We want to prove our approach will function in moderate heterogeneous system also. Many a values are applied to the experiments with different self-scheduling strategies in 2048*2048 matrix multiplication, shown as Table 4.5 and Figure 4.5. Table 4.6 and Figure 4.6 show the result in a=75 with different self-scheduling strategies. The difference of system performance using various a value and various self-scheduling approaches is not obvious.

Applying a=75 to smaller problem size, 1024*1024 matrix multiplication, or larger problem size, 3072*3072 matrix multiplication, the result was shown in Table 4.7 and Figure 4.7. There is not obvious difference between various self-scheduling schemes.

The result of a=75 in simulated decreasing/increasing workload loop was shown in Table 4.8 and Figure 4.8. The execution time in decreasing loop and increasing loop should be same in theory, but there is obvious difference by GSS and FSS. That is because that the remainder 25% workload is processed by known self-scheduling get load imbalancing. Decreasing loop get better performance than increasing one due to getting smaller workload.

	a=0	a=60	a=65	a=70	a=75	a=80	a=85	a=90
GSS	485.6	478.9	479.6	480.6	483.5	479.5	479.4	481.1
FSS	479.1	474.3	475.7	476.3	478.2	474.5	476.8	474.4
TSS	489.3	495.6	492.6	490.4	486.8	494.6	495.2	491.2

Table 4.5 Execution time for 2048*2048 matrix multiplication by various approaches in moderate heterogeneous environment

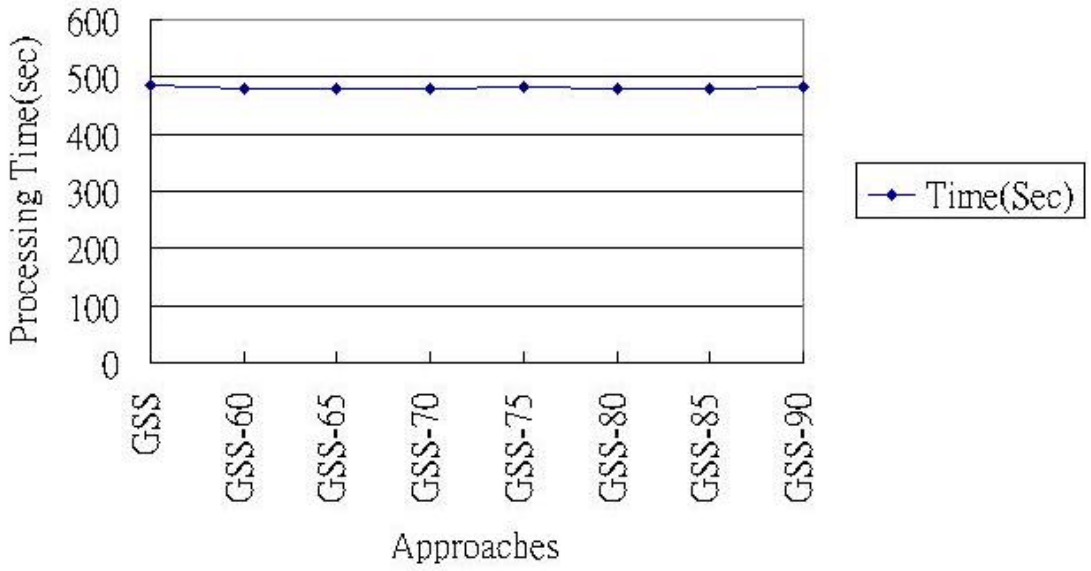

Figure 4.5(a) A chart of execution time of 2048*2048 matrix multiplication by GSS group approach in moderate heterogeneous environment

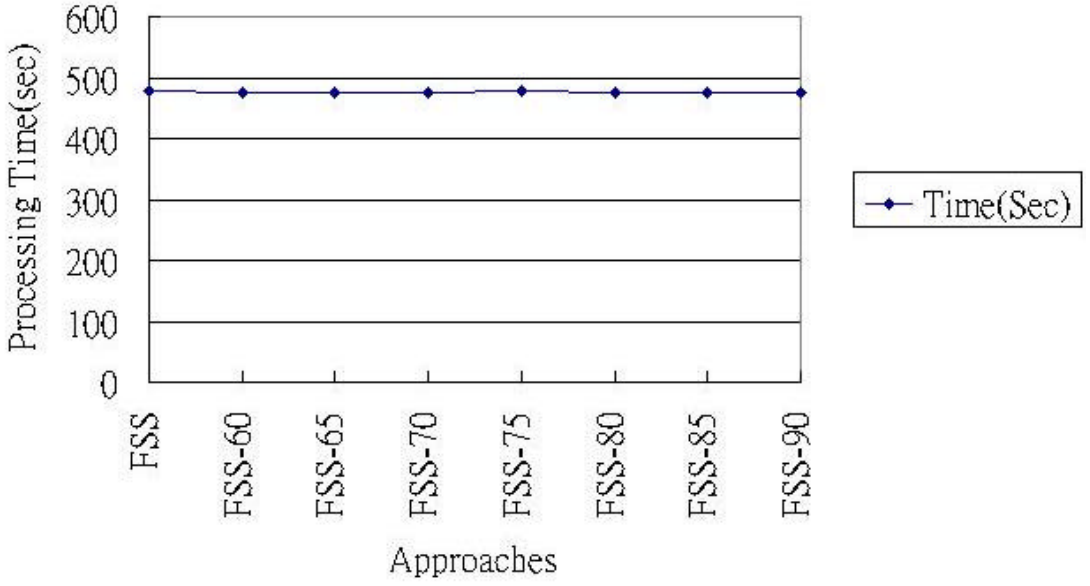

Figure 4.5(b) A chart of execution time of 2048*2048 matrix multiplication by FSS group approach in moderate heterogeneous environment

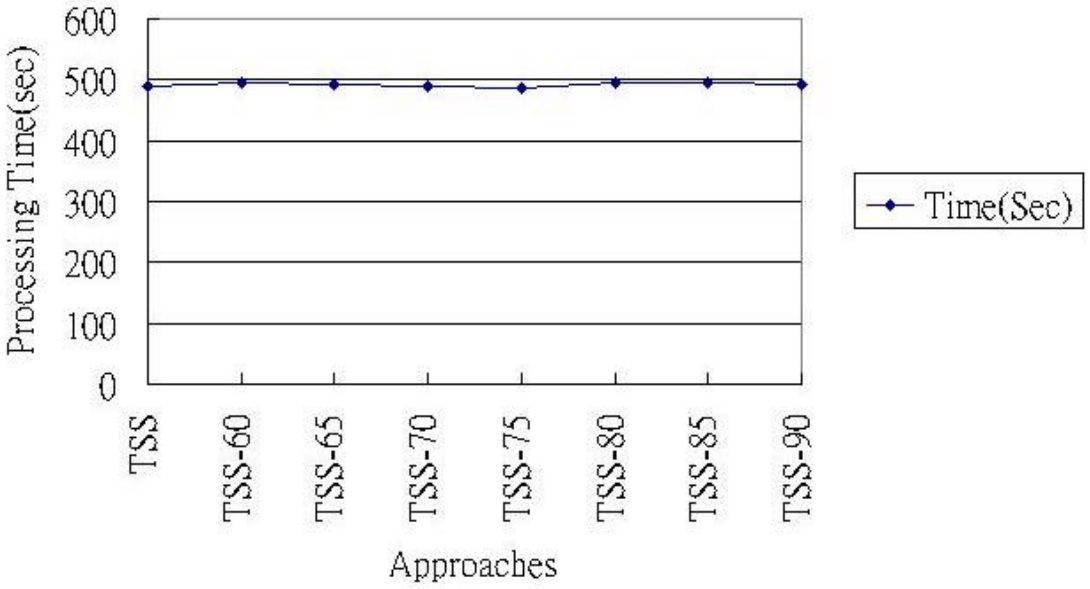

Figure 4.5(c) A chart of execution time of 2048*2048 matrix multiplication by TSS group approach in moderate heterogeneous environment

	None	GSS	GSS-75	FSS	FSS-75	TSS	TSS-75
Execution time	506.3	485.6	483.5	479.1	478.2	489.3	486.8

Table 4.6 Execution time of 2048*2048 matrix multiplication by various self-scheduling approaches when a=75 in moderate heterogeneous environment

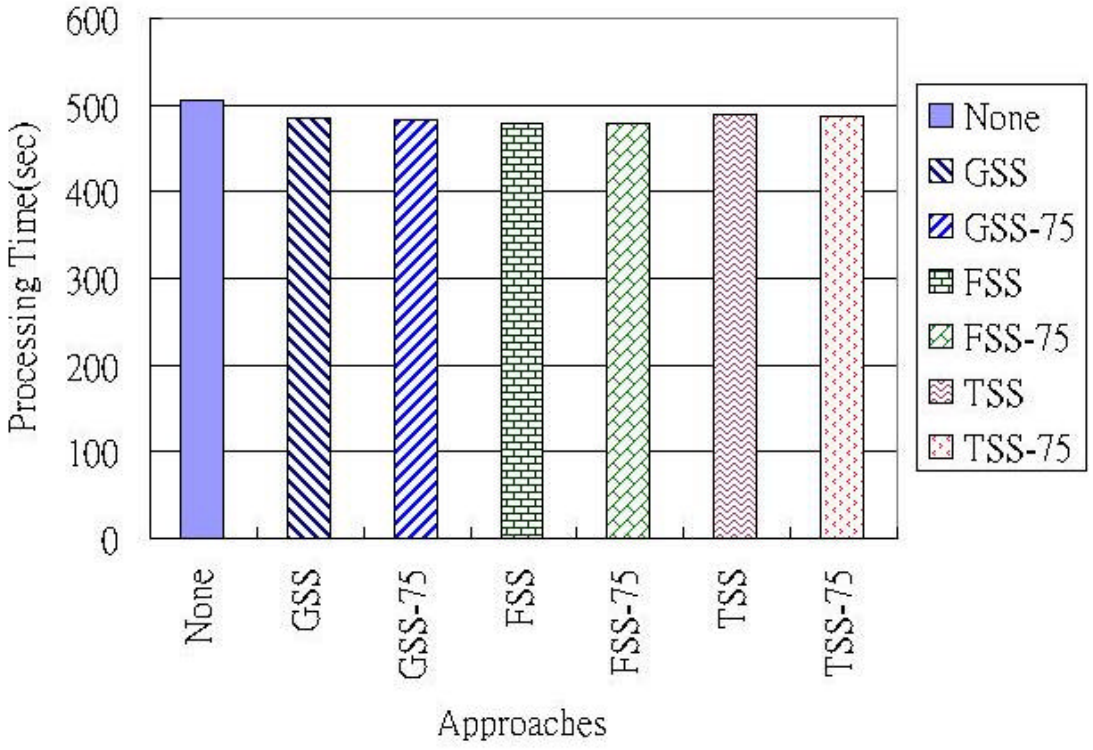

Figure 4.6 A chart of execution time of 2048*2048 matrix multiplication by various self-scheduling approaches when a=75 in moderate heterogeneous environment

	None	GSS	GSS-75	FSS	FSS-75	TSS	TSS-75
1024*1024	62.7	60.2	57.7	56.7	56.6	60.9	54.5
3072*3072	1726.2	1658.0	1643.2	1638.7	1632.7	1700.4	1657.3

Table 4.7 Execution time of different problem size by various self-scheduling approach when a=75 in moderate heterogeneous environment

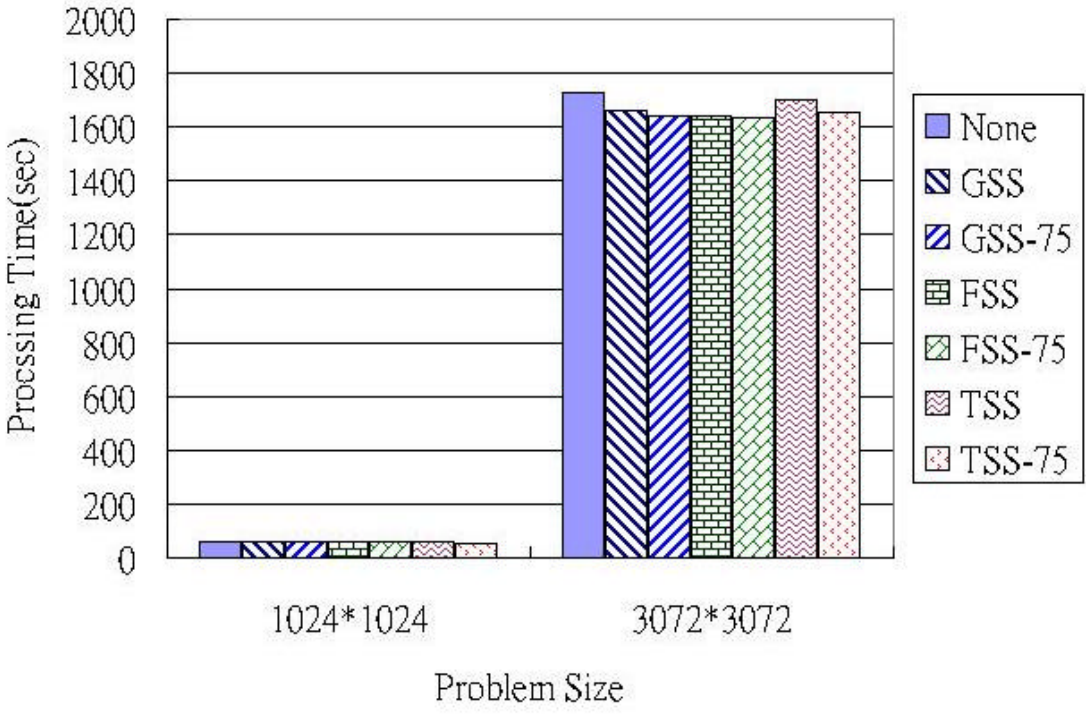

Figure 4.7 A chart of execution time of different problem size by various self-scheduling approach when a=75 in moderate heterogeneous environment

	GSS	GSS-75	FSS	FSS-75	TSS	TSS-75
Decreasing	495.1	347.0	252.8	280.3	304.3	254.4
Increasing	255.1	255.1	264.4	260.5	304.3	256.1

Table 4.8 Execution time of simulated increasing/decreasing workload loop by various self-scheduling approach when a=75

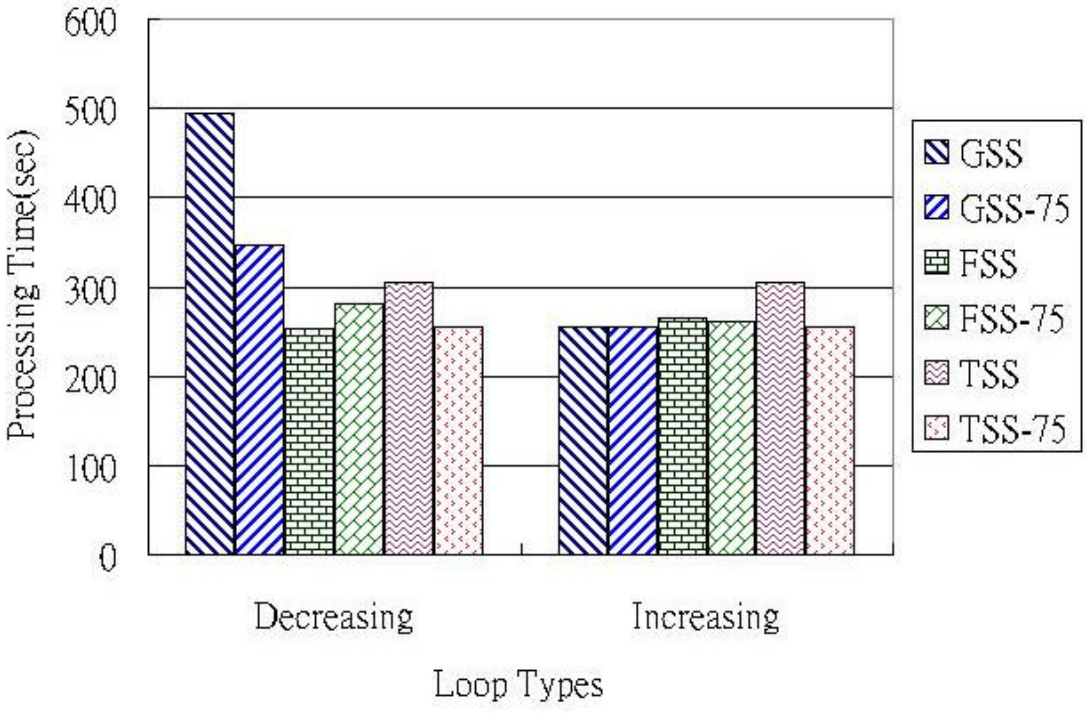

Figure 4.8 A chart of execution time of simulated increasing/decreasing workload loop by various self-scheduling approach when a=75

Chapter 5

Conclusion and Future Work

In this paper, we show that known self-scheduling schemes cannot achieve good load balancing in some situations. We propose an approach to partition loop iterations and achieve good performance in any heterogeneous environment: partition a% of workload according to their performance weighted by CPU clock and the rest (100-a)% of workload according to known self-scheduling. Many various a values are applied to the matrix multiplication and a best performance is obtained with a=75. We also applied our schemes on two simulated increasing/decreasing workload loops and get obviously performance improvement. Therefore, our approach is suitable in all applications with regular loops.

Our idea is just suitable for the regular workload loop. However, the irregular workload loop, such as displaying the Mandelbrot set problem, is more common bop type. We want to solve parallel loop scheduling problems with unpredictable loops on heterogeneous PC clusters. Furthermore, our approach is just ranged on DOALL loop; we want to expend our research field to DOACROSS loop and runtime scheduling.

Fann, Yang, Tseng and Tsai propose a knowledge-based approach to solving loop-scheduling problems [9]. A rule-based system, called IPLS, is developed by combining a repertory grid and an attribute ordering table to construct a knowledge base. IPLS chooses an appropriate scheduling algorithm by inferring some features of loops and assigning parallel loops to multiprocessors to achieve significant speedup. However, this system is based on UMA architecture. In near future, we will migrate IPLS to cluster architecture. Also, we will solve parallel loop scheduling problems with unpredictable loops on extreme heterogeneous PC clusters and integrate our approach into the new IPLS.

Reference

1. S. F. Hummel, E. Schonberg, L. E. Flynn, "*Factoring, a Scheme for Scheduling Parallel Loops*," Communications of the ACM, Vol 35, No 8, Aug. 1992.
2. C. D. Polychronopoulos and D. Kuck, "*Guided Self-Scheduling: a Practical Scheduling Scheme for Parallel Supercomputers*," IEEE Trans. on Computers, Vol 36, Dec. 1987, pp 1425 - 1439.
3. T. H. Tzen and L.M. Ni, "*Trapezoid Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers*," IEEE Trans. on Parallel and Distributed Systems, Vol 4, No 1, Jan. 1993, pp 87 - 98.
4. Christopher A. Bohn, Gary B. Lamont, "*Load Balancing for Heterogeneous Clusters of PCs*," Future Generation Computer Systems 18 (2002) 389–400.
5. E. Post, H. A. Goosen, "*Evaluating the Parallel Performance of a Heterogeneous System*," in the Proceedings of HPCAsia2001.
6. H. Li, S. Tandri, M. Stumm and K. C. Sevcik, "*Locality and Loop Scheduling on NUMA Multiprocessors*," in Proceedings of the 1993 International Conference on Parallel Processing, Vol. II, 1993, pp. 140-147.
7. A. T. Chronopoulos, R. Andonie, M. Benche and D.Grosu, "*A Class of Loop Self-Scheduling for Heterogeneous Clusters*," in Proceedings of the 2001 IEEE International Conference on Cluster Computing, pp. 282-291
8. P. Tang and P. C. Yew, "*Processor self-scheduling for multiple-nested parallel loops*," in Proceedings of the 1986 International Conference on Parallel Processing , 1986, pp. 528-535.
9. Yun-Woei Fann, Chao-Tung Yang, Shian-Shyong Tseng, and Chang-Jiun Tsai, "*An intelligent parallel loop scheduling for multiprocessor systems*," Journal of Info. Science and Engineering - Special Issue on Parallel and Distributed Computing, vol. 16, no. 2, pp. 169-200, March 2000.
10. Barry Wilkinson and Michael Allen, *Parallel Programming*, Prentice Hall PTR, 1999.
11. Michael Wolfe, *High Performance Compilers for Parallel Computing*, Addison-Wesley PTR, 1996.

12. Yung-Lin Liu and Chung-Ta King, "*EXPLORER: Supporting run-time parallelization of DOACROSS loops on general networks of workstations*" Parallel Computing 26 (2000), pp.355-375
13. S. Chen, J. Xue, "*Partitioning and scheduling loops on NOWs*", Computer Communications 22 (1999), pp. 1017–1033
14. Amdahl, G. "*Validity of the single processor approach to achieving large-scale computing capabilities*", In Proceedings of the AFIPS Conference (1967), pp. 483-485
15. Jianhua Xu and A. T. Chronopoulos, "*Distributed Self-Scheduling for Heterogeneous Workstation Clusters*", Proceedings of 12th International Conference on Parallel and Distributed Computing Systems, Fort Lauderdale, FL, Aug. 18-20, 1999, pp. 211-217.
16. R. Buyya, *High Performance Cluster Computing: System and Architectures*, Vol. 1, Prentice Hall PTR, NJ, 1999.
17. Jose C. Cunha, Peter Kacsuk and Stephen C. Winter, *Parallel Program Development for Cluster Computing*, Nova Science Publishers, NY, 2001
18. http://www.mpi-forum.org/, Message Passing Interface Forum.
19. http://www.epm.ornl.gov/pvm/, PVM – Parallel Virtual Machine.
20. T. L. Sterling, J. Salmon, D. J. Backer, and D. F. Savarese, *How to Build a Beowulf: A Guide to the* Implementation *and Application of PC Clusters*, 2nd Printing, MIT Press, Cambridge, Massachusetts, USA, 1999.