Distributed Self-Scheduling for Heterogeneous Workstation Clusters

Jianhua Xu
Department of Computer Science
Wayne State University
Detroit, MI 48202
jix@cs.wayne.edu

Anthony Chronopou los, Senior member, IEEE
Division of Computer Science
University of Texas at San Antonio
S an Antonio, TX 78249
atc@cs.utsa.edu

Abstract

Distributed Computing Systems are a viable and less expensive alternative to parallel computers. However, a serious difficulty in concurrent programming of a distributed system is how to deal with scheduling and load balancing of such a system which may consist of heterogeneous workstations. Kim and Purtilo[7] designed a distributed scheduling scheme suitable for parallel loops with independent iterations on heterogeneous workstation clusters.

In previous work, Self-Scheduling schemes for parallel loops with independent iterations have been applied to multiprocessor systems. We extend this type of schemes to heterogeneous distributed systems. We present experimental results showing that our scheme compares favorably to the scheduling in [7].

Keywords: Load balance, Scheduling, Self-scheduling, Loop scheduling, Network of workstations

1 Introduction

To exploit the potential computing power of workstation clusters, an important issue is how to assign tasks to workstations so that the workstation loads are well balanced, which is referred to as the scheduling problem. Several research results exist in scheduling and load balancing for distributed systems (see e.g., [1], [2], [4], [6], [7]). Loops are the most important source of concurrency in parallel/distributed computations. This paper focuses on scheduling parallel loops with independent iterations on heterogeneous distributed systems (with processors having different speeds).

There are basically two kinds of loop scheduling schemes, *static* scheduling and *dynamic* scheduling.

- *Static* scheduling assigns iterations (to processors) once before program execution starts. It is suitable for uniformly distributed loops (i.e. no dependencies exist between different loop iterations), when executed on a homogeneous multiprocessor system. For parallel loops static scheduling involves little overhead (because there is no run time data communication), but easily leads to load imbalance, for example if the loop iterations are of different execution times and the processors are homogeneous, or if the processors are heterogeneous (which is often true for workstation clusters).

- *Dynamic* scheduling assigns iterations at run time. A processor is assigned more iterations only after it finishes its current load. Obviously, dynamic scheduling introduces some scheduling overhead. Thus, to minimize overhead while maintaining good load balance becomes a critical problem which dynamic scheduling must solve.

Self-scheduling is a dynamic loop scheduling method for multiprocessor systems, in which an idle processor dynamically gets assigned loop iteration(s) for execution.

Various Self-scheduling schemes ([8], [10], [12], [13]) have been proven successful for shared memory multiprocessor systems. Since workstation clusters behave differently from multiprocessor systems, these schemes would not give satisfactory results when used directly on workstation clusters.

In this paper, we modify and improve self-scheduling schemes, so that they can be applied to heterogeneous workstation clusters. We propose a distributed self-scheduling scheme, which is suitable for heterogeneous workstation clusters. Both architecture heterogeneity and dynamic load variation are taken into account. A workstation is assigned a *virtual computing power*, which reflects its actual speed relative to the slowest one in the system. Before a workstation is assigned its next task, it checks its current load. We use the number of processes in the run-queue as a *load measure*. The virtual computing power divided by the load is called available computing power, and the next chunk size is determined by a workstation's *available computing power*. We implement our scheme using the distributed software environment PVM[5] on a heterogeneous workstation cluster and make comparisons with existing schemes.

In section 2 we review self-scheduling schemes. We address the heterogeneity of a workstation cluster and our motivation in section 3. In section 4 we present a loop iteration model. We present the Distributed Trapezoid Self-Scheduling(DTSS) in section 5. And we show our experimental results and conclusions in sections 6-7.

2 Self-Scheduling Schemes Review

The most often used self-scheduling schemes are *pure self-scheduling*[12], *chunk self- scheduling*[8], *guided self-scheduling*[10], and *trapezoid self-scheduling*[13].

Pure self-scheduling assigns one iteration to each idle processor at a time. Since the granularity size is one iteration, it is easy to achieve load balance, especially for non-uniformly distributed loops and heterogeneous systems such

as workstation clusters. On the other hand, the scheduling overhead is proportional to the total number of iterations, which may be high. In distributed computing, we are more concerned about minimizing the total execution time than pure load balancing. Thus pure self-scheduling is not an ideal scheme, especially for workstation clusters, where communication cost is high.

Chunk self-scheduling may be used to reduce the scheduling synchronization overhead. Instead of allocating one iteration at a time, chunk self-scheduling assigns a task (known as *chunk*) that consists of a number of iterations (k known as *chunk size*). Chunk self-scheduling requires $1/k$ as many synchronizations as pure self-scheduling does. A large chunk size reduces scheduling overhead, but also increases the chance of load imbalance. Thus, the key issue in chunk self-scheduling is how to determine the optimal chunk size, and this is left to the programmer.

To make scheduling less dependent on programmers, in terms of determining the chunk size, *guided self-scheduling* dynamically reduces the chunk size. Chunk size in guided self-scheduling is determined by the number of remaining (to be executed) iterations divided by the number of processors. The chunk size at the beginning is relatively large, and then progressively decreases , which indicates the intention of both achieving load balance and reducing the scheduling overhead.

While guided self-scheduling decreases the chunk size nonlinearly, *trapezoid self-scheduling(TSS)* decreases the chunk size (referred to as *Chore* in [13]) linearly. Let I be the total number of iterations in the loop and P the number of processors. In trapezoid self- scheduling, the first and last (assigned) chunk size pair (F, L) may be set by the programmer. In a conservative selection, *(F,L)* pair is determined as

$$F = I/2P \text{ and } L = 1 \ .$$

This ensures that the load of the first chunk is less than $1/P$ of the total load in most loop distributions, and reduces the chance of imbalance due to large size of the first chunk (see [13] for details).

The total number of steps needed for the scheduling process is

$$N = 2 * I/(F + L) \ .$$

Thus the decrease between consecutive chunks is

$$D = (F - L)/(N - 1).$$

Then the chunk sizes in TSS would be $F, F - D, F - 2 * D, \cdots$.

It was shown in [13] that trapezoid self-scheduling is better than the other schemes in most cases. Thus, we will use trapezoid self-scheduling in our examples to be presented in the following sections.

3 Motivation

Self-scheduling schemes were designed with multiprocessor systems in mind, where processors are homogeneous. When applying these schemes to a cluster of workstations, heterogeneity (i.e. speed and load variation) must be taken into account.

3.1 Speed

A workstation cluster may consist of workstations of different architecture, and even for the same kind of architecture, each workstation may have different computing speed. If a workstation slower than others gets a task assignment of the same size as others, it will probably become the critical (performance) workstation, resulting in severe load imbalance.

Let us assume there are four workstations W_1, W_2, W_3, and W_4, the total number of iterations is I, on average each workstation may compute $I/4$ iterations. According to TSS, the first chunk size would be $F = I/8$, which can not be the critical chunk if the four workstation have the same speed. However, if they have different speeds, for example, W_1's speed is 1 iteration per second, and the others, 9 iterations per second. Then, to achieve good

load balance, W_1 should be allocated about $I/(3*9+1) = I/28$ iterations. In TSS, W_1 may be assigned its first chunk, $I/8$ iterations, thus becoming the critical workstation, resulting in a computing time of $I/8$ seconds. On the other hand, if we use only W_2, W_3, and W_4, the time would be $I/27$ seconds. This example illustrates the case where computing an application using four workstations behaves much worse than using only three workstations. This anomaly was caused by the architecture heterogeneity.

3.2 Load Variation

Workstations may belong to different users, and may be used by multiple users at the same time (e.g. through *rlogin*), thus a workstation should not be treated as dedicated to one distributed application. If a workstation suddenly becomes overloaded, and it still gets task assignments as before, it will definitely become the critical workstation.

Assume four workstations W_1, W_2, W_3, and W_4 of the same speed are dedicated to a distributed computing application. TSS would balance well the computation if nothing changes during the execution. Assume it takes 4, 3, 2, and 1 seconds for W_1, W_2, W_3, and W_4 to finish their last (assigned) task, respectively. This will result in at most an imbalance of $4 - 1 = 3$ seconds in terms of execution time. However, if just before the last round, W_1 becomes overloaded, and its speed degrades 5 times, then it would take $4*5 = 20$ seconds to finish its last task, giving a load imbalance of $20-1 = 19$ seconds in terms of execution time. This shows that dynamic load variation may affect the computing time dramatically.

In distributed computing, the heterogeneity issues must be addressed.

3.3 Related Work

T. Kim and J M. Purtilo[7] implemented a decentralized load balancing scheme for heterogeneous workstation clusters. They establish a (virtual)

linear topology according to the workstation's static speed. Then load migration is only necessary between neighbors in one direction. The (virtual) topology is constructed through a *binary tree*, so we will refer to this scheme as *Tree-Scheduling* in the following discussion. This scheme is static, i.e. the situation where a workstation's load may change during program execution was not considered. Thus, Tree-Scheduling may not always be effective as it is demonstrated by our experiments.

4 A Parallel Loop Model

Parallel loops may be presented in different styles, and a scheduling scheme should work for any loop style. In this section, we outline various important loop styles.

4.1 Loop style

A loop may be presented in any of the styles in Figure 1. Figure 1 (a) represents a uniformly distributed loop; (b) and (c) are linearly distributed loops; (d) indicates a conditional loop, which may result from *IF* statements; (e) gives a irregular loop style, for which no simple formula or expression is available.

4.2 Pseudo-Uniformly Distributed Loops

Different loops may need to be handled in different ways in order to get the best performance. Linearly decreasing loops are easy to be balanced, while linearly increasing loops may cause severe load imbalance, if the iterations are assigned in a noncyclic fashion. While a scheduling scheme should work for all loop styles, we may be able to reorganize some parallel loops so that they appear uniformly distributed. We use the sampling methodology. For a loop with I iterations, a sampling frequency R is given, then we sample the loop R times, first the iterations whose index i satisfies $i (mod R) = 0$; second,

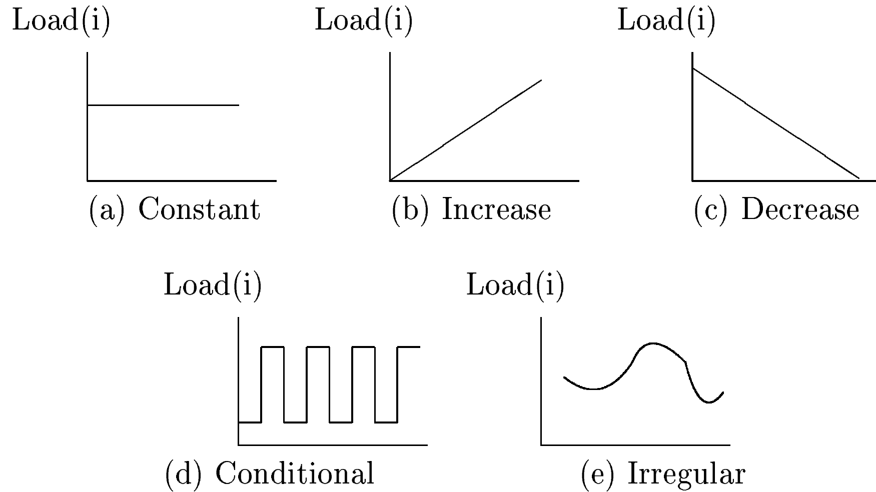

Figure 1: Parallel Loop Styles

the iterations with $i(modR) = 1$, and so on. After sampling, the R samples are placed in a sequence. Since no data dependency is assumed between iterations in parallel loops, computing the sampled loop will produce the same result as the original one. However, the sampled distribution appears uniformly distributed, if one sampling is treated as a unit. For example, the corresponding distribution for loops in Figure 1 (b) and (c) is shown in Figure 2. We used this method in our experiments.

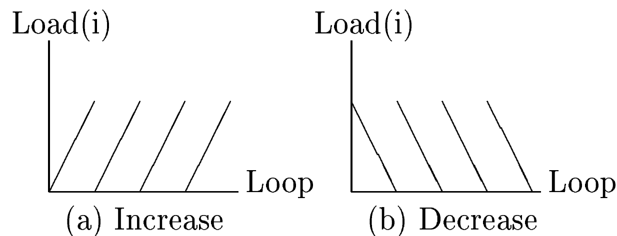

Figure 2: Pseudo Uniformly Distributed Loops

5 Distributed Self-scheduling

We next present a workstation-cluster model and our distributed self-scheduling scheme.

5.1 Workstation Cluster Model

As computing in a distributed workstation cluster is different from that in a multiprocessor system, the model for a workstation cluster should reflect the heterogeneity aspects of workstation clusters. A workstation cluster is modeled as follows:

- P: the cluster consists of P workstations, W_1, W_2, ..., W_P
- V_i: the virtual computing power of W_i, which represents the relative speed of W_i. We may assign $V_i = 1$ for the slowest workstation, V_i =2 to a workstation twice as fast, and so on. Then, a workstation with virtual computer computing power V_i=5 may be treated as if it has 5 virtual Processing Elements (PEs). The total virtual computing power (VCP) of the cluster is
$$V = \sum_i V_i.$$
- Q_i: the number of processes in the run-queue of W$_i$, which reflects the total load of a single workstation.
- A_i: the available computing power (ACP), $A_i = V_i/Q_i$, where integer division is applied. A_i is updated each time a new process is added to the run-queue of W$_i$. The total ACP of the cluster is
$$A = \sum_i A_i.$$

Remark: $A_i = 0$ is possible. If all the workstations are dedicated to a single application process then A_i equals V_i and the total ACP equals VCP.

There are several load measurement standards[9]. We use the number of processes in the run-queue as a load measure.

It must be noted that the available computing power indicates both a workstation's actual speed and how much of its power is available to a distributed application.

If a workstation has a virtual computing power $V_i = 6$, and there are three current processes in the run-queue, then $A_i = V_i/Q_i = 6/3 = 2$, which means each process can use only two out of the six virtual PEs.

5.2 Distributed Self-scheduling

We apply our model to trapezoid self-scheduling, which we call *Distributed Trapezoid Self- Scheduling* (DTSS). The same model can be used with other self-scheduling schemes.

The notations used for our scheduling are listed below:

- I: the total number of iterations in the loop.
- F: the proposed first chunk size
- L: the proposed last chunk size
- N: proposed total number of steps needed for the scheduling process.
- D: proposed decreasing step, which may also be dynamically adjusted
- $U_{i,j}$: the load to be assigned to a single virtual PE j of W_i, where $j = 1, .., A_i$.
- C_i: the next chunk size for workstation with A_i

We apply the TSS (see [13] for details) to the set of available virtual PEs in order to obtain the chunk sizes in DTSS. The programmer may determine

the pair (F, L), according to TSS; or the following formula may be used in the conservative selection approach (by default):

$$F = I/2A, \quad L = 1 \ . \tag{1}$$

Then, the proposed number of steps needed for the scheduling process (according to the virtual computing power) is :

$$N = 2I/(F + L) \ , \tag{2}$$

the step decrease for each task assignment is :

$$D = (F - L)/(N - 1) \ , \tag{3}$$

We apply TSS to each virtual processor to derive the chunk size for each workstation with available computing power $A_i > 0$. For $i = 1$

$$U_{1,1} = F, \tag{4}$$

and for $1 < j \leq A_1$

$$U_{1,j} = F - D * (j - 1) \ . \tag{5}$$

Therefore,

$$C_1 = \sum_{j=1}^{A_1} U_{1,j} = F * A_1 - D * \sum_{j=1}^{A_1} (j - 1) \ . \tag{6}$$

Which gives

$$C_1 = A_1 * F - D * A_1 * (A_1 - 1)/2 = A_1 * (F - D * (A_1 - 1)/2) \tag{7}$$

For $i > 1$ and $1 \leq j \leq A_i$

$$U_{i,j} = F - D * (A_1 + .. + A_{i-1} + j - 1) \ . \tag{8}$$

Let $S_{i-1} = A_1 + .. + A_{i-1}$, then

$$C_i = \sum_{j=1}^{A_i} U_{i,j} = F*A_i - D*A_i*(S_{i-1} + (A_i - 1)/2) = A_i*(F - D*(S_{i-1} + (A_i - 1)/2)) \ . \tag{9}$$

Remark: When all the workstations are dedicated to a single process then $A_i = V_i$. Also, when all the workstation have the same speed then $V_i = 1$ and the tasks assigned in DTSS is are the same as in TSS. The important difference between DTSS and TSS is that in DTSS we allocate the next chunk according to a workstation's available computing power, but in TSS all workstations are simply treated in the same way. Thus, faster workstations get more iterations than slower ones in DTSS.

5.3 Implementation

The implementation of self-scheduling in workstation clusters is different from (shared memory) multiprocessor systems. In multiprocessor systems, processors share the memory, thus each processor may fetch the next chunk by itself using the proper synchronization mechanism. On workstation clusters, there is no shared memory, a process must get the next chunk by means of communication. Thus we use a coordinator to distribute the iterations. We used the PVM distributed software environment in our implementation [5]. PVM helps avoid low level communication programming on workstation clusters and it provides process management.

The algorithm is described as follows:

Coordinator:

1. Wait for all workers with $A_i > 0$ to report their A_i; sort A_i in decreasing order and store them in a ACP Status Array(ACPSA); also for each A_i place a request in a queue in the sorted order. Calculate the total available computing power

   $$A = \sum_i A_i \ ,$$

   select a (F, L) pair, and compute N and D according to (1) - (3).

2. While there are unassigned iterations, if a request arrives, put it in the queue and store the newly received A_i if it is different from ACPSA; Pick a request from the queue, assign the next chunk according to (4)-(9).
   If more than half of the A_i's changed since the last time, update the ACPSA and go to step 1, with total number of iterations I set equal to the number of remaining iterations.

Worker:

1. Obtain the number of processes in the run-queue Q_i and calculate the available computing power

$$A_i = V_i/Q_i$$

   if($A_i > 0$) goto step 2
   else goto step 1

2. Send a request (containing its A_i) to the coordinator

3. Wait for a reply; if more tasks arrive
   { compute the new tasks; go to step 1; }
   else terminate.

6 Scheduling Testing

Most loops, in real life applications, can not be characterized by the styles (a) - (c) in Figure 1. We use the Mandelbrot set [13], which falls into category (e), in our experiment. The experiment was conducted on a four workstation cluster. We calculate Mandelbrot set on a (-2,-2) - (2,2) area with 1200 by 1200 pixels. The loop style figures for the original and Pseudo-Uniformly

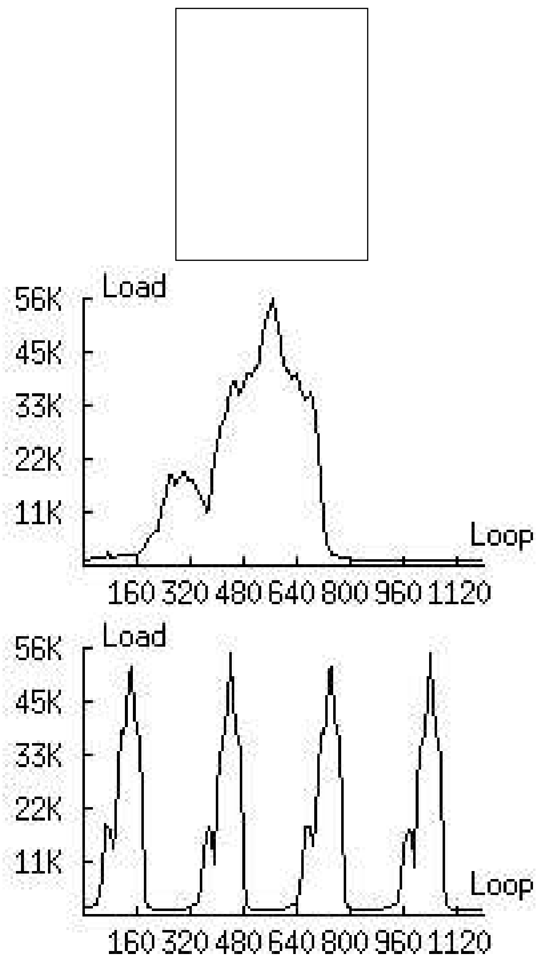

Figure 3: Mandelbrot Set, original distribution and pseudo-uniform distribution

Table 1: DTSS and TSS (seconds)

		W_1	W_2	W_3	W_4	Finish Time	Imbalance
TSS	Dedicated	27	28	27	26	29	2
	Not	64	47	47	45	65	19
DTSS	Dedicated	27	27	28	26	28	2
	Not	52	49	51	51	53	3

distributed loop (with frequency 4) iterations assignment are shown in Figure 3.

We present two experiments. The first one compares DTSS with TSS, and the second compares DTSS with Tree-Scheduling.

In the first experiment, we used four workstations(SunSparc Classic) of the same architecture and speed. We ran DTSS and TSS under two assumptions: (a) (*Dedicated Case*) all the workstations are dedicated to a single distributed computing application and (b) (*Nondedicated Case*) when other applications run at the same time. The results are shown in Tables 1.

We can see from Table 1, in (a), DTSS and TSS have almost the same results. However, in (b), for example when W_1, W_2, and W_3 have four, three, and two processes in their run-queue, respectively, then TSS produced a large load imbalance, while DTSS still balances very well.

In the second experiment (shown in Table 2), we also use four workstations. But this time, W_1, W_2, and W_3 (SunSparc Classic) are of the same actual speed, but W_4 (SGI Indigo) is 6 times faster than any of W_1, W_2, and W_3. Again we ran DTSS and Tree-Scheduling under the two assumptions (a) and (b) above.

In (a), Tree- Scheduling yields a very good result. However, such a result can rarely be reached in real life applications. The reason for this is that in Tree-Scheduling we assign iterations according to workstation speed, and there would be no load migration (thus theoretically no scheduling synchro-

Table 2: DTSS and Tree-Scheduling (seconds)

		W_1	W_2	W_3	W_4	Finish Time	Imbalance
Tree	Dedicated	11	11	11	11	12	0
	Not	12	12	11	86	86	75
DTSS	Dedicated	15	15	15	15	16	0
	Not	30	30	30	30	31	0

nization overhead) if the workstations' available computing powers do not change. This is a major weakness for Tree-Scheduling, which is evident in (b). For example, W_4, the fastest workstation becomes severely loaded (this is possible, when every application process wants to use the fastest machine), then the scheduling or waiting time becomes huge for Tree-Scheduling, which results in huge imbalance. However, DTSS still balances well, although with some scheduling overhead, and it yields overall a good performance.

7 Conclusion and Future work

In this paper we propose a model for Distributed Self-Scheduling, simple yet effective, which applies self-scheduling schemes to heterogeneous workstation clusters. And we present experimental results, which demonstrate that this scheduling is effective for distributed applications with parallel loops (i.e. loops without inter-iterations dependencies).

What we did not take into account is how communication can affect the efficiency of the proposed strategy, which may be conducted in future work.

References

[1] J. Casas, R. Konuru, S. W. Otto, R. Prouty, and J. Walpole. *Adaptive Load Migration Systems for PVM.* Proc. of Supercomputing 1994, Washington, DC, Nov., pp 40-49.

[2] M. Cermele, M. Colajanni. *Dynamic load balancing of distributed SPMD computations with explicit message passing*. Proc. Of IEEE Heterogeneous Computing Workshop, Geneva, pp 2-13, Apr., 1997.

[3] M. Cierniak, W. Li, and M. J. Zaki. *Loop scheduling for heterogeneity*. Proc. of the 4th International Symposium on High-Performance Distributed Computing, Aug., 1995.

[4] M. Colajanni, and M. Cermele. *DAME: An environment for preserving efficiency of data parallel computations on distributed systems*. IEEE Concurrency, 1997.

[5] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide*. The MIT Press, 1994.

[6] A. S. Grimshaw, J. B. Weissman, E. A. West, and Jr. E. C. Loyot. *Metasystem: An approach combining parallel processing and heterogeneous distributed computing systems*. Journal of Parallel and Distributed Computing, Vol 21, pp 257-270, 1994

[7] T. H. Kim, and J. M. Purtilo, *Load balancing for parallel loops in workstation clusters*, International Conference on Parallel Processing, Vol III, pp 182 - 189, 1996.

[8] C. P. Kruskal and A. Weiss. *Allocating independent subtasks on parallel processors*. IEEE Trans. on Software Engineering, Vol 11, No 10, pp 1001-1016, Oct., 1985.

[9] T. Kunz, *The influence of different workload description on a heuristic load balancing scheme*. IEEE Trans. On Software Engineering, Vol 17, No 7, pp 725-730, Jul., 1991.

[10] C. D. Polychronopoulos and D. J. Kuck. *Guided self-scheduling: A practical scheduling scheme for parallel supercomputers.* IEEE Trans. On Computer, Vol C-36(12), Dec. 1987, pp 1425 - 1439.

[11] J. Prichard. *The Chaos Cookbook: A practical programming guide.* Butterworth Heinemann Publ., 1996.

[12] P. Tang and P. C. Yew. *Processor self-scheduling for multi nested parallel loops.* Proceedings of '86 International Conference on Parallel Processing, Aug., 1986, pp 528- 535.

[13] 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.