Scalable loop self-scheduling schemes for heterogeneous clusters

Anthony T. Chronopoulos*, Satish Penmatsa, Ning Yu and Du Yu

Department of Computer Science, The University of Texas,
6900 N Loop, 1604 W, San Antonio, Texas 78249, USA
E-mail: antony.tc@gmail.com
E-mail: atc@cs.utsa.edu E-mail: spenmats@cs.utsa.edu
*Corresponding author

Abstract: Heterogeneous cluster systems (e.g., a LAN of computers) can be used for concurrent processing for some applications. However, a serious difficulty in concurrent programming of a heterogeneous system is how to deal with scheduling and load balancing of such a system that may consist of heterogeneous computers. Distributed scheduling schemes suitable for parallel loops with independent iterations on heterogeneous computer clusters have been proposed and analysed in the past. Here, we implement the previous schemes in MPI. We present an extension of these schemes implemented in a hierarchical Master–Slave architecture and include experimental results and comparisons.

Keywords: scalable; distributed; loops; scheduling schemes.

Reference to this paper should be made as follows: Chronopoulos, A.T., Penmatsa, S., Yu, N. and Yu, D. (2005) 'Scalable loop self-scheduling schemes for heterogeneous clusters', *Int. J. Computational Science and Engineering*, Vol. 1, Nos. 2/3/4, pp.110–117.

Biographical notes: A.T. Chronopoulos received his PhD in Computer Science, University of Illinois, Urbana-Champaign, 1987. He is an Associate Professor in Department of Computer Science, University of Texas at San Antonio.

S. Penmatsa received his MS in Computer Science, University of Texas at San Antonio, 2003. He is a PhD candidate in Computer Science and Instructor at the University of Texas at San Antonio.

N. Yu received his MS in Computer Science, University of Texas San Antonio, 2002. He is a Network System Administrator at the University of Texas Health Sciences, San Antonio, TX.

D. Yu received her MS in Computer Science, University of Texas San Antonio, 2002. She is a Network System Administrator at the University of Texas at Dallas, TX.

1 Introduction

LOOPS are one of the largest sources of parallelism in scientific programs. If the iterations of a loop have no interdependencies, each iteration can be considered as a task and can be scheduled independently. A review of important loop-scheduling algorithms for parallel computers is presented in Fann et al. (2000) (and references therein), and some recent results are presented in Bull (1998) and Hancock et al. (2000). Research results also exist on scheduling loops and linear algebra data parallel computations on message passing parallel systems and on heterogeneous systems. (See Banicescu and Liu, 2000; Banicescu et al., 2003; Barbosa et al., 2000; Cierniak et al., 1995; Chronopoulos et al., 2001; Dandamudi, 1997; Dandamudi and Thyagaraj, 1997; Goumas et al., 2002; Hummel et al., 1996; Kee and Ha, 1998; Kim and Purtilo, 1996; Markatos and LeBlanc, 1994; Philip and Das, 1997; Yan et al., 1997; Yang and Chang, 2003; Freeman et al., 2000).

Loops can be scheduled statically at compile-time. This scheduling has the advantage of minimising the scheduling-time overhead, but it may cause load imbalancing when the loop style is not uniformly distributed. Examples of such scheduling are Block, Cyclic, etc. (Fann et al., 2000). Dynamic scheduling adapts the assigned number of iterations whenever it is unknown in advance how large the loop tasks could be. An important class of dynamic scheduling schemes is the self-scheduling schemes (Chronopoulos et al., 2001; Fann et al., 2000). In these schemes, each idle processor accesses a few loop iterations to execute next. In UMA (Uniform Memory Access) parallel system, these schemes can be implemented using a critical section for the loop iterations and no need exists for dedicating a processor to do the scheduling. That is why these schemes are called self-scheduling schemes.

Heterogeneous systems are characterised by heterogeneity and large number of processors. Some significant distributed schemes that take into account the characteristics of the different components of the heterogeneous system were devised, for example:

- tree scheduling
- weighted factoring
- distributed trapezoid self-scheduling.

See Chronopoulos et al. (2001), Hummel et al. (1996), Kim and Purtilo (1996) and references therein.

1.1 Notations

The following are common notations used throughout the whole paper:

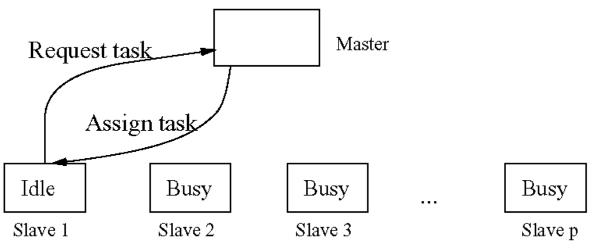
- *PE* is a processor in the parallel or heterogeneous system
- *I* is the total number of iterations of a parallel loop
- *p* is the number of slave PEs in the parallel or heterogeneous system that execute the computational tasks
- $P_1, P_2, \ldots, P_p$ represent the *p* slave PEs in the system
- a few consecutive iterations are called a *chunk*. C_i is the chunk-size at the *i*th scheduling step (where $i = 1, 2, \ldots$)
- *N* is the number of scheduling steps
- $t_j, j = 1, \ldots, p$, is the execution time of P_j to finish all tasks assigned to it by the scheduling scheme
- $T_p = \max_{j=1,\ldots,p}(t_j)$ is the parallel execution time of the loop on *p* slave PEs.

In Section 2, we review simple loop self-scheduling schemes. In Section 3, we review distributed self-scheduling schemes. In Section 4, we describe the hierarchical distributed schemes. In Section 5, an implementation is presented. In Section 6, distributed simulations are presented. In Section 7, conclusions are drawn.

2 Simple loop-scheduling schemes

Self-scheduling is an automatic loop-scheduling method in which idle PEs request new loop iterations to be assigned to them. We will study these methods from the perspective of heterogeneous systems. For this, we use the Master-Slave architecture model (Figure 1). Idle slave PEs communicate a request to the master for new loop iterations. The number of iterations a PE should be assigned is an important issue. Owing to PEs' heterogeneity and communication overhead, assigning the wrong PE a large number of iterations at the wrong time may cause load imbalancing. Also, assigning a small number of iterations may cause too much communication and scheduling overhead.

Figure 1 Self-scheduling schemes: the Master-Slave model



In a generic self-scheduling scheme, at the *i*th scheduling step, the master computes the chunk-size C_i and the remaining number of tasks R_i:

$$R_0 = I, \quad C_i = f(R_{i-1}, p), \quad R_i = R_{i-1} - C_i \tag{1}$$

where, $f(.,.)$ is a function possibly of more inputs than just R_{i-1} and *p*. Then the master assigns to a slave PE C_i tasks. Imbalance depends on the execution time gap between t_j, for $j = 1, \ldots, p$. This gap may be large if the first chunk is too large or (more often) if the last chunk (called the *critical chunk*) is too small.

The different ways to compute C_i has given rise to different scheduling schemes. The most notable examples are the following:

Trapezoid Self-Scheduling (TSS) (Tzen and Ni, 1993)

$C_i = C_{i-1} - D$, with (chunk) decrement:

$$D = \left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor,$$

where the first and last chunk-sizes (F, L) are user/compiler-inputs or $F = \left\lfloor \frac{I}{2p} \right\rfloor$, $L = 1$. The number of scheduling steps assigned:

$$N = \left\lceil \frac{2 \times I}{(F+L)} \right\rceil.$$

Note that $C_N = F - (N - 1)D$ and $C_N \geq 1$ owing to integer divisions.

Factoring Self-Scheduling (FSS)

$$C_i = \lceil R_{i-1}/(\alpha p) \rceil,$$

where, the parameter α is computed (by a probability distribution) or is suboptimally chosen, $\alpha = 2$. The chunk-size is kept the same in each *stage* (in which all PEs are assigned one task) before moving to the next stage. Thus $R_i = R_{i-1} - pC_i$ (where $R_0 = I$) after each stage.

Fixed Increase Self-Scheduling (FISS)

Based on the number of iterations, the user or the compiler selects the number of stages (σ) (Philip and Das, 1997). $C_i = C_{i-1} + B$, where initially

$$C_0 = \left\lfloor \frac{I}{X \times p} \right\rfloor$$

(with X a compiler/user chosen parameter) and the (chunk increase or 'bump')

$$B = \left\lceil \frac{2I(1-\sigma/X)}{p\sigma(\sigma-1)} \right\rceil$$

(where σ, the number of stages, must be a compiler/user chosen parameter; $X = \sigma + 2$ was suggested).

Trapezoid Factoring Self-Scheduling (TFSS)(Chronopoulos et al., 2001)

This is a scheme that uses stages (as in FSS). In each stage, the chunks for the PEs are computed by averaging the chunks of TSS.

We next give an example to illustrate these schemes.

Example 1: We show the chunk-sizes selected by the self-scheduling schemes discussed above. Table 1 shows the different chunk-sizes for a problem with $I = 1000$ and $p = 4$.

Table 1 Sample chunk-sizes for $I = 1000$ and $p = 4$

Scheme	*Chunk-size*
TSS	125 117 109 101 93 85 77 69 61 53 45 37 29 21 13 5
FSS	125 125 125 125 62 62 62 62 32 32 32 32 16 16 16 16 8 8 8 8 4 4 4 4 2 2 2 2 1 1 1 1
FISS	50 50 50 50 83 83 83 83 117 117 117 117
TFSS	113 113 113 113 81 81 81 81 49 49 49 49 17 17 17 17

3 Distributed loop-scheduling schemes for heterogeneous systems

Load balancing in heterogeneous system is a very important factor in achieving near optimal execution time. To offer load balancing, loop-scheduling schemes must take into account the processing speeds of the computers forming the system. The PE speeds are not precise, since memory, cache structure and even the program type will affect the performance of PEs. However, one must run simulations to obtain estimates of the throughputs, and one must show that these schemes are quite effective in practice.

In past work (Chronopoulos et al., 2001), we presented and studied distributed versions for the schemes of the previous section. In the distributed schemes, when the master assigns new tasks, it takes into account the available virtual powers of the slaves (Chronopoulos et al., 2001). These schemes resemble some past results that existed for some of these distributed schemes. Notably, the distributed FSS for the dedicated case (i.e., when available powers equal virtual powers) is identical to the *Weighted Factoring* scheduling scheme (Banicescu et al., 2003; Hummel et al., 1996). However, in the distributed FSS for the non-dedicated case, the master uses the available power of the slaves in assigning the tasks (Chronopoulos et al., 2001). Thus, distributed FSS differs from the *Adaptive Weighted Factoring* (Banicescu and Liu, 2000; Banicescu et al., 2003), where the master changes the weight factors according to slaves' execution times in the preceding stage.

We next review the distributed TSS (DTSS) scheme. The distributed versions of the other schemes are similar and can be found in Chronopoulos et al. (2001).

3.1 Terminology

- $V_i = Speed(P_i)/\min_{1 \le i \le p}\{Speed(P_i)\}$, is the virtual power of P_i (computed by the master), where Speed(P_i) is the CPU-Speed of P_i
- $V = \sum_{i=1}^{p} V_i$ is the total virtual computing power of the cluster
- Q_i is the number of processes in the run-queue of P_i, reflecting the total load of P_i
- $A_i = \left\lfloor \frac{V_i}{Q_i} \right\rfloor$ is the available computing power (ACP) of P_i (needed when the loop is executed in non-dedicated mode. In dedicated mode, $Q_i = 1$ and $A_i = V_i$)
- $A = \sum_{i=1}^{p} A_i$ is the total available computing power of the cluster.

We note that Q_i may contain other processes (un)related to the problem to be scheduled.

The assumption is made that different processes running on a computer will take an equal share of its computing resources. Even if this is not entirely true, other factors being neglected (memory, process priority and program type), this simple model appears to be useful and efficient in practice. Note that at the time A_i is computed, the parallel loop process is already running on the computer. For example, if a processor P_i with $V_i = 2$ has an extra process running, then $A_i = 2/2 = 1$, which means that p_i behaves just like the slowest processor in the system. In order to simplify the algorithm for the hierarchical case, we clarify 'Receive' and 'Send' (in the Master-Slave tree).

Remark 1: 'Receive' (i) for Master means that it receives (information) from the descendant nodes. (ii) for Slave means that it receives (information) from the parent node. We understand 'Send' similarly. Also, note that each message from the slave contains a 'request' and the current A_i.

The DTSS algorithm is described as follows:

Master:

1) (a) Receive all *Speed* (P_i);

   (b) Compute all V_i;

   (c) Send all V_i;

2) Repeat: (a) Receive one of A_i, sort A_i in decreasing order and store them in a temporary ACP Status Array (ACPSA). For each A_i, place a request in a queue in the sorted order. Calculate A.

   (b) If more than 1/2 of A_i changes since the last time, update ACPSA and set I = remaining iterations. Use $p = A$ to obtain (new) F, L, N, D as in TSS.

3) (a) While there are unassigned iterations, if a request arrives (in 2(a)), put it in the queue.

   (b) Pick a request from the queue, assign the next chunk $C_i = A_i \times (F - D \times (S_{i-1} + (A_i - 1)/2))$, where: $S_{i-1} = A_1 + \cdots + A_{i-1}$ (see Chronopoulos et al., 2001).

Slave:

1) (a) Send *Speed* (P_i);

   (b) Receive V_i.

2) Obtain the number of processes in the run-queue Q_i and recalculate A_i.

   If ($A_i \leq$ Threshold) repeat 2.

3) Send a request (containing its A_i).

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

Remark 2: (i) The necessary data is either replicated or locally generated on all participating slaves. (ii) Since the run-queue Q_i changes as other user application jobs may start or terminate, step 2 is executed by the slave before each new request to the master. (iii) The master process runs on a dedicated processor.

4 Hierarchical distributed schemes

When comparing a centralised scheme using the Master-Slave model (Figure 1), to a physically distributed scheme, several issues must be studied: the scalability, the communication and synchronisation overhead and the fault tolerance.

All the centralised policies, where a single node (the master) is in charge of the load distribution, will present degradation in performance when the problem size increases. This means that for a large problem (and a large number of processors), the master becomes a bottleneck. The access to the synchronised resources (variables) will take a long time, during which many processors will idle waiting for service instead of doing useful work. This is an important problem for a cluster of heterogeneous computers, where long communication latencies can be encountered.

It is known that distributed (or non-centralised) policies usually do not perform as well as the centralised policies, for small problem sizes and small number of processors. This is because the algorithm and the implementation of distributed schemes usually add a non-trivial overhead.

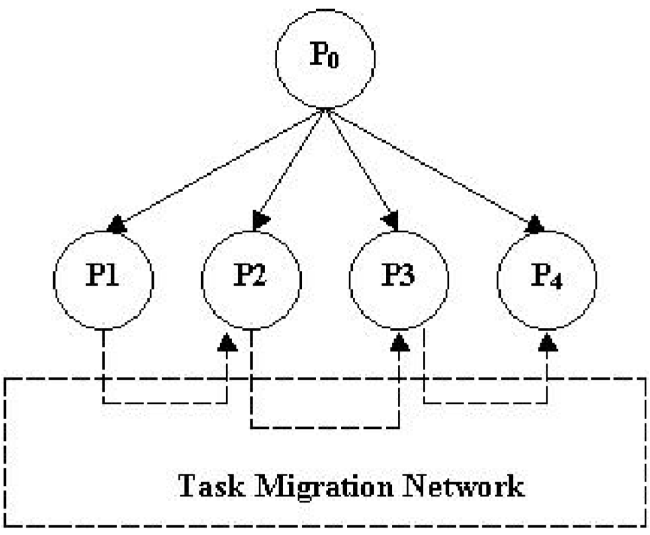
4.1 Tree Scheduling (TreeS)

TreeS (Dandamudi and Thyagaraj, 1997; Kim and Purtilo, 1996) is a distributed load-balancing scheme that statically arranges the processors in a logical communication topology based on the computing powers of the processors involved.

When a processor becomes idle, it asks for work from a single, pre-defined partner (its neighbour on the left). Half of the work of this processor will then migrate to the idling processor. Figure 2 shows the communication topology created by *TreeS* for a cluster of four processors. Note that P_0 is needed for the initial task allocation and the final I/O. For example, P_0 can be the same as the fastest P_i.

An idle processor will always receive work from the neighbour located on its left side, and a busy processor will always send work to the processor on its right. For example, in Figure 2, when P_2 is idle, it will request half of the load of P_1. Similarly, when P_3 is idle, it will request half of load of P_2 and so on. The main success of *TreeS* is the distributed communication, which leads to good scalability.

Figure 2 The Tree topology for load balancing



Note that in the heterogeneous system, there is still the need for a central processor that initially distributes the work and at the end collects the results, unless the problem is of such a nature that the final results are not needed for I/O. Thus, the Master-Slave model still has to be used initially and at the end.

The main disadvantage of this scheme is its sensitivity to the variation in computing power. The communication topology is statically created and might not be valid after the algorithm starts executing. If, for example, a workstation that was known to be very powerful becomes severely overloaded by other applications, its role of taking over the excess work of the slower processors is impaired. This means that the excess work has to travel more until reaching an idle processor or that more work will be done by slow processors, producing a large finish time for the problem.

4.2 A hierarchical DTSS

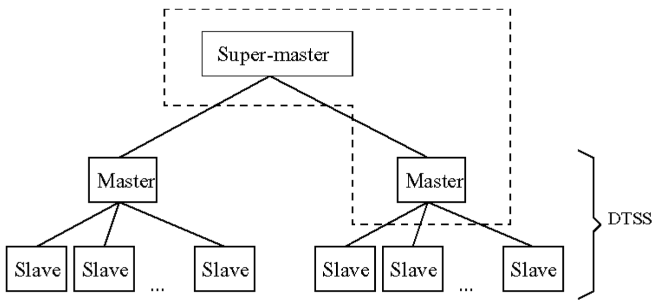
We see that the logical hierarchical architecture is a good foundation for scalable systems. In the following, we propose a new hierarchical method for addressing the bottleneck problems in the centralised schemes.

4.2.1 Architecture

We use the Master-Slave centralised model (which is known to be very effective for small problem sizes), but instead of making one master process responsible for all the workload distribution, new master processes are introduced. Thus, the hierarchical structure contains a lower level, consisting of slave processes, and several superior levels, of master processes. On top, the hierarchy has an overall *super-master.* The level of slaves will use for load balancing the best-centralised self-scheduling method for the problem that is to be solved. We used for our experiments the *Distributed Trapezoid Self-Scheduling.* We named the new scheme, *Hierarchical DTSS* (HDTSS).

Figure 3 shows this design for two levels of master processes. The slaves are using DTSS when communicating with their master. We note that the s*uper-master* ↔ *master* communication applies the master-slave algorithm with master replaced by super-master and slaves replaced by masters. We note that the HMasters do not perform any computation (i.e., they are not slaves). However, they assign tasks to slaves from the pool of tasks that they obtain periodically from the super-master. They communicate with the super-master only when they run out of tasks for their cluster. The dotted lines surround processes that can be assigned to the same physical machine, for improved performance.

Figure 3 Hierarchical DTSS (two levels of masters)



We can describe the algorithm for the HDTSS by making reference to the (Master–Slave) DTSS Algorithm and the Remark 1. Let us use the abbreviated notations: HSlave for a Slave node and HMaster for a Master node in the hierarchical Master–Slave architecture tree. The HDTSS algorithm can be concisely described as follows:

SuperMaster: Perform the DTSS-Master steps.

HMaster: Perform the DTSS-Master 1((a) and (c)) and DTSS-Slave 1 ((a) and (b)).

HSlave: Perform the DTSS-Slave steps.

Remark 3: The HMaster gathers all the messages from its ancestors/descendants and then sends them to the descendants/ancestors (merged) in one message.

5 Implementation

The Mandelbrot computation (Mandelbrot, 1988) is a doubly nested loop without any dependencies. The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit. For the centralised schemes, the master accepts requests from the slaves and serves them in the order of their arrival. It replies to each request with a pair of numbers representing the interval of iterations the slave should work on.

The slaves will attach (piggyback) to each request, except for the first one, the result of the computation due to the previous request. This improves the communication efficiency. An alternative we tested was to perform the collection of data at the end of the computation (the slaves stored locally the results of their requests). This technique produced longer finishing times because when all the slaves finished, they seemed to contend for master access in order to send their results. During this process, they will have to idle instead of doing useful work. By piggybacking the data produced by the previous request to the actual request, we achieve some degree of overlapping of computation and communication. There will be still some contention for the master access, but mostly the slaves will work on their requests, while few slaves communicate data to the master.

The implementation for the Tree Scheduling (TreeS) (Kim and Purtilo, 1996) is different. The slaves do not contend for a central processor when making requests because they have pre-defined partners. But the data still has to be collected on a single central processor. When we used the approach described above, of sending all the results at the end of the computation, we observed a lot of idling time for the slaves, thus degrading the performance. We implemented a better alternative: the slaves send their results to the central coordinator from time to time, at predefined time intervals. The contention for the master cannot be totally eliminated, but this appears to be a good solution.

6 Distributed simulations

We use the Mandelbrot computation (Mandelbrot, 1988) for a window size of $4000 \times 2000, \ldots, 10000 \times 5000$ on a system consisting of p (=1, 2, 4, 8, 16, 24) slaves and one master. We used MPI (Pachecho, 1997). The workstations were Sun UltraSPARC 10 with memory sizes 128 Mb and 512 Mb and CPU speeds 440 MHz and 502 MHz. In our experiments, the size of the problem is such that it does not cause any memory swaps. So the virtual powers depend only on the CPU speeds. We put an artificial load in the background (matrix by matrix product) on $p/4$ of the slaves which have $V_i = 2$ and three loads on the other $p/4$ slaves which have $V_i = 1$. The remaining $p/2$ of the slaves have no load in the background and have $V_i = 4$. The window size used for Tables 2–3 and Figures 4–5 is 4000 × 2000. For the Figures 6–7, the window size ranges from 4000 × 2000 to 10000 × 5000. We ran the simulations when no other user jobs existed on the workstations.

Table 2 Simple Schemes (MPI), $p = 8$; PE_i: $T_{\text{com}}/T_{\text{comp}}$ (SEC)

PE	*TSS*	*FSS*	*FISS*	*TFSS*	*TreeS*
1	0.1/5.8	0.7/4.3	0.1/4.8	0.1/4.9	5.4/6.1
2	0.2/5.5	0.2/4.1	0.5/5.1	1.1/5.5	4.8/6.8
3	0.1/5.6	0.8/5.5	0.9/5.8	0.2/4.5	5.6/6.3
4	0.1/5.8	0.3/4.8	0.8/5.2	0.8/4.9	3.0/6.1
5	0.4/6.5	1.6/7.3	1.9/11.2	0.9/5.1	0.1/8.6
6	0.3/10.5	2.5/9.8	1.3/8.4	1.3/7.6	0.2/8.6
7	0.7/6.6	1.3/7.1	1.7/8.1	2.2/10.8	0.3/9.9
8	0.1/7.4	1.9/10.8	1.7/8.3	1.7/8.9	3.2/7.4
T_p	9.8	14.9	13.4	14.3	10.9

Table 3 Distributed schemes (MPI), $p = 8$; PE_i: $T_{\text{com}}/T_{\text{comp}}$ (SEC)

PE	*DTSS*	*DFSS*	*DFISS*	*DTFSS*	*TreeS*
1	0.1/3.7	0.1/4.1	0.8/3.9	0.8/5.1	0.3/6.6
2	0.1/4.3	0.3/3.9	0.1/3.8	0.2/4.0	0.9/6.7
3	0.1/4.5	0.5/4.4	0.8/4.9	0.9/4.7	0.9/6.7
4	0.1/4.3	0.1/5.3	0.3/4.2	1.1/4.6	0.8/6.8
5	0.3/6.9	0.9/5.9	0.9/5.0	0.9/7.1	0.3/7.1
6	0.1/4.9	1.2/6.4	1.1/7.1	1.2/6.9	0.7/7.1
7	0.1/6.4	1.1/6.3	0.1/5.5	1.3/7.6	0.4/7.2
8	0.2/6.9	0.9/7.6	1.6/6.9	1.1/7.3	0.5/7.9
T_p	9.2	13.7	12.9	13.2	10.5

Figure 4 Speed-up of simple schemes (MPI)

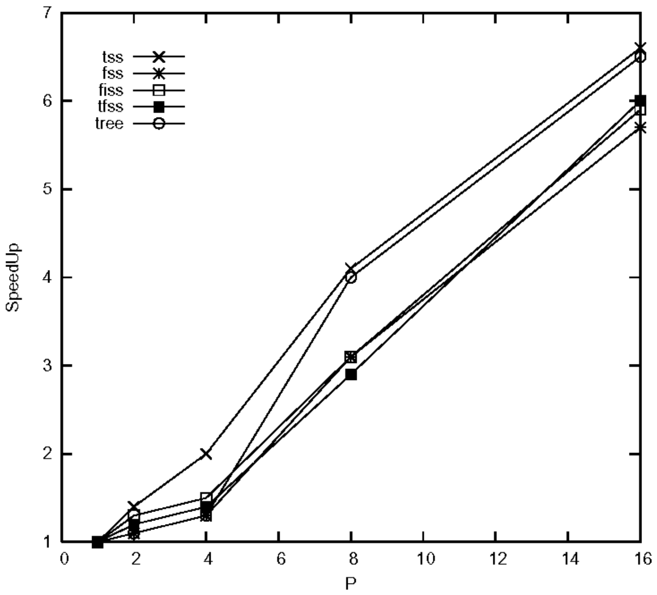
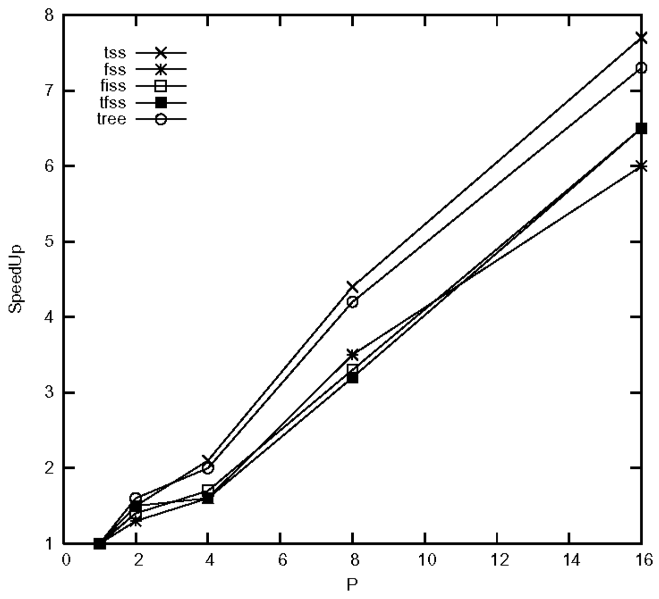


Figure 5 Speed-up of distributed schemes (MPI)



We test the *simple* schemes (i.e., those described in Section 2) on a heterogeneous cluster. All slaves (PEs) are treated (by the schemes) as having the same computing power. For the *TreeS*, the master assigns an equal number of tasks to all slaves in the initial allocation stage. The *distributed* schemes take into account V_i or A_i of the slave. For hierarchical schemes, we implemented only HDTSS, because DTSS was faster than the other master-slave schemes.

We present two cases, *dedicated* and *non-dedicated.* In the first case, processors are dedicated to running our program (i.e., $Q_i = 1$). In the second, we started a resource-expensive process (load) on half of the slaves. This load is forked by these slaves with their first request for work to the master. For this we take $Q_i = 2$. For $p \leq 8$, we ran only in dedicated mode and for p > 8, we ran also in non-dedicated mode.

The times (communication/computation) of the slave (PE_i) are tabulated for p slaves. T_p is the total time measured on the Master PE. All times are measured in *seconds* (sec). We performed the following tests:

- Comparison between different schemes (TSS, FSS, FISS, TFSS and TreeS), both simple/distributed in MPI (Pachecho, 1997). Tables 2–3 contain the communication/computation times ($T_{\text{com}}/T_{\text{comp}}$) for the various schemes. It can be seen that the slaves' computation times are more balanced in Table 3 than in Table 2, as expected. Figures 4–5 contain the speed-ups of these schemes vs. one slave times. We conclude that the best-centralised scheme (according to our tests) is DTSS and compared it to TreeS (not centralised) in hierarchical implementation.

- Comparison of DTSS, Hierarchical-DTSS and TreeS in MPI (Figures 6–7). We conclude that for large sizes, HDTSS is better than DTSS. Also, HDTSS is better than TreeS.

Figure 6 $T_{comp,}$ dedicated, $p = 32$

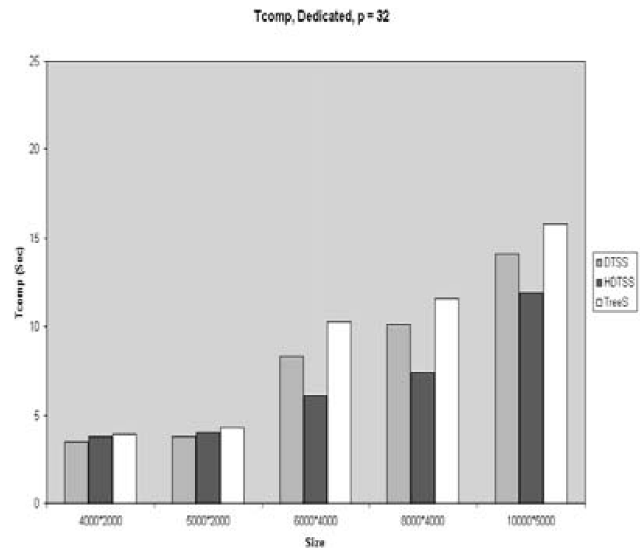
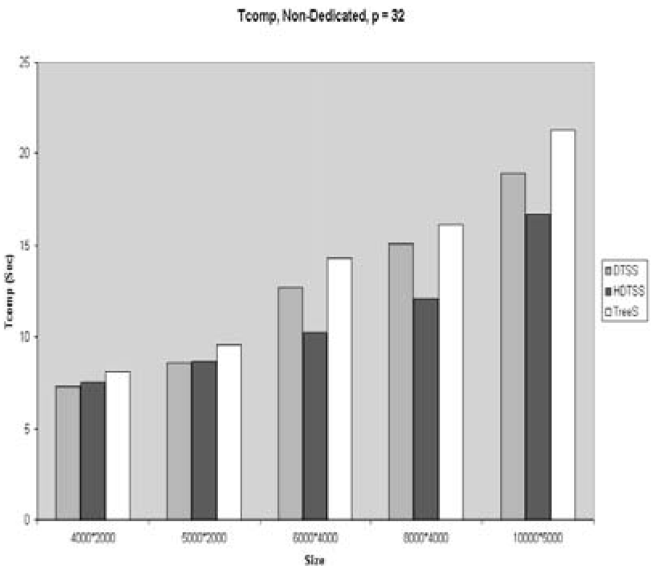


Figure 7 $T_{comp,}$ non-dedicated, $p = 32$



The TreeS is slower than DTSS in our runs because in all runs, the results are transmitted to a 'Master' computer (which distributes the initial data and collects the final results). Also, the TreeS performance is expected to be higher than presented if two additional computers (i.e., $p + 2$ computers are used in HDTSS as Masters) are also used in the TreeS. Otherwise, the TreeS gives similar performance results as the DTSS in the dedicated runs. In the non-dedicated runs, since the Tree cannot adapt to the load changes, it is expected to be slower than the DTSS.

The speed-ups in Figures 4–5 are low because the run time on a single slave was made on a fast slave, whereas the parallel time was taken from a run with $p/2$ fast, $p/4$ slower and $p/4$ slowest slaves.

7 Conclusion

We studied and implemented (in MPI) loop-scheduling schemes for heterogeneous systems. Our results show that the hierarchical schemes are scalable and compare well to Tree scheduling that is a scalable scheme.

Acknowledgements

This research was supported, in part, by research grants from: (1) NASA NAG 2-1383 (1999–2001); (2) State of Texas Higher Education Coordinating Board through the Texas Advanced Research/Advanced Technology Program ATP 003658-0442-1999. Some reviewers' comments helped enhance the quality of presentation.

References

Banicescu, I. and Liu, Z. (2000) 'Adaptive factoring: a dynamic scheduling method tuned to the rate of weight changes', *Proc. High Performance Computing Symposium*, Washington, USA, pp.122–129.

Banicescu, I., Velusamy, V. and Devaprasad, J. (2003) 'On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring', *Cluster Computing 6*, pp.215–226.

Barbosa, J., Tavares, J. and Padilha, A.J. (2000) 'Linear algebra algorithms in a heterogeneous cluster of personal computers', *Proc. 9th Heterogeneous Computing Workshop (HCW 2000)*, May, Cancun, Mexico, pp.147–159.

Bull, J.M. (1998) 'Feedback guided dynamic loop scheduling: Algorithms and experiments', *Proc. 4th Intl Euro – Par Conference*, Southampton, UK, pp.377–382.

Chronopoulos, A.T., Andonie, R., Benche, M. and Grosu, D. (2001) 'A class of distributed self-scheduling schemes for heterogeneous clusters', *Proc. 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001)*, October, Newport Beach, California, USA.

Cierniak, M., Li, W. and Zaki, M.J. (1995) 'Loop scheduling for heterogeneity', *Proc. 4th IEEE Intl. Symp. on High Performance Distributed Computing*, August, Washington, DC, USA, pp.78–85.

Dandamudi, S.P. (1997) 'The effect of scheduling discipline on dynamic load sharing in heterogeneous distributed systems', *Proc. MASCOTS'97*, January, Haifa, Israel.

Dandamudi, S.P. and Thyagaraj, T.K. (1997) 'A hierarchical processor scheduling policy for distributed-memory multicomputer systems', *Proc. 4th International Conference on High-Performance Computing,* Nagoya, Japan, pp.218–223.

Fann, Y.W., Yang, C.T., Tseng, S.S. and Tsai, C.J. (2000) 'An intelligent parallel loop scheduling for parallelizing compilers', *Journal of Information Science and Engineering*, pp.169–200.

Freeman, T.L., Hancock, D.J., Bull, J.M. and Ford, R.W. (2000) 'Feedback guided scheduling of nested loops', *Proc. the 5th Intl Workshop*, PARA, Bergen, Norway, pp.149–159.

Goumas, G., Drosinos, N., Athanasaki, M. and Koziris, N. (2002) 'Compiling tiled iteration spaces for clusters', *Proc. IEEE International Conference on Cluster Computing,* Chicago, Illinois, pp.360–369.

Hancock, D.J., Bull, J.M., Ford, R.W. and Freeman, T.L. (2000) 'An investigation of feedback guided dynamic scheduling of nested loops', *Proc. Intl Workshops on Parallel Processing*, August, Toronto, Canada, pp.315–321.

Hummel, S.F., Schmidt, J., Uma, R.N. and Wein, J. (1996) 'Load-sharing in heterogeneous systems via weighted factoring', *Proc. 8th Annual ACM Symp. on Parallel Algorithms and Architectures*, Padua, Italy, pp.318–328.

Kee, Y. and Ha, S. (1998) 'A robust dynamic load-balancing scheme for data parallel application on message passing architecture', *Intl Conf. on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, Vol. 2, pp.974–980.

Kim, T.H. and Purtilo, J.M. (1996) 'Load balancing for parallel loops in workstation clusters', *Proc. Intl. Conference on Parallel Processing,* Bloomingdale, IL, USA, pp.182–190.

Mandelbrot, B.B. (1988) *Fractal Geometry of Nature*, W.H. Freeman & Co., New York, USA.

Markatos, E.P. and LeBlanc, T.J. (1994) 'Using processor affinity in loop scheduling on shared-memory multiprocessors', *IEEE Trans. on Parallel and Distributed Systems*, April, Vol. 5, No. 4, pp.379–400.

Pachecho, P. (1997) *Parallel Programming with MPI*, Morgan Kauffman, San Francisco, CA, USA.

Philip, T. and Das, C.R. (1997) 'Evaluation of loop scheduling algorithms on distributed memory systems', *Proc. Intl Conf. on Parallel and Distributed Computing Systems*, Nagoya, Japan.

Tzen, T.H. and Ni, L.M. (1993) 'Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers', *IEEE Trans. on Parallel and Distributed Systems*, January, Vol. 4, No. 1, pp.87–98.

Yan, Y., Jin, C. and Zhang, X. (1997) 'Adaptively scheduling parallel loops in distributed shared-memory systems', *IEEE Trans. on Parallel and Distributed Systems*, January, Vol. 8, No. 1, pp.70–81.

Yang, C.T. and Chang, S.C. (2003) 'A parallel loop self-scheduling on extremely heterogeneous pc clusters', *Proc. Intl Conf. on Computational Science,* Melbourne, Australia, pp.1079–1088.