Scalable Loop Self-Scheduling Schemes for Heterogeneous Clusters *

Anthony T. Chronopoulos[†], Satish Penmatsa, Ning Yu,
Dept. of Computer Science,
University of Texas at San Antonio,
6900 North Loop 1604 West, San Antonio, TX 78249,
atc@cs.utsa.edu

Abstract

Distributed 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 distributed system is how to deal with scheduling and load balancing of such a system which may consist of heterogeneous computers. Distributed scheduling schemes suitable for parallel loops with independent iterations on heterogeneous computer clusters have been proposed and analyzed in the past. Here, we implement the previous schemes in the CORBA (Orbix). We also present an extension of these schemes implemented in a hierarchical master-slave architecture. We present experimental results and comparisons.

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. Distributed systems are characterized by heterogeneity and large number of processors.

Self-scheduling schemes that take into account the characteristics of the different components of the system were devised, for example: 1) *Tree Scheduling*, 2) *Weighted Factoring* and 3) *Distributed Trapezoid Self-Scheduling*. See ([3],[6],[7]) and references there in.

*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.

†Senior member IEEE, Corresponding author

Notations:

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 PEs in the parallel or distributed system;
- P_1, P_2, ..., P_p represent the p 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: $i = 1, 2..$);
- N is the number of scheduling steps;
- t_j, $j = 1, .., p$, is the execution time of P_j to finish all its tasks assigned to it by the scheduling scheme;
- $T_p = \max_{j=1,..,p}(t_j)$, is the parallel execution time of the loop on p PEs;

CORBA is widely used in large-scale distributed simulations(see [1], [5], [9], [13]). Thus it is useful to study scheduling distributed computations using CORBA. Here, we implement the previous schemes in the CORBA (Orbix) and make comparisons. We also present an extension of these schemes implemented in a hierarchical master-slave architecture.

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.

Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02)
0-7695-1745-5/02 $17.00 © 2002 IEEE

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:15:14 UTC from IEEE Xplore. Restrictions apply.

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 distributed 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. Due 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.

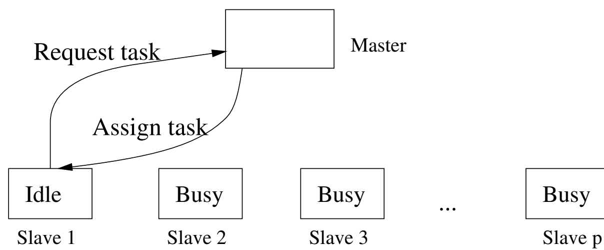

Figure 1. Self-Scheduling schemes: the Master-Slave model

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:

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

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 $j = 1, .., p$. 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)([11]) $C_i = C_{i-1} - D$, 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-input or $F = \left\lfloor \frac{I}{2p} \right\rfloor$, $L = 1$. The number of scheduling steps assigned: $N = \left\lceil \frac{2*I}{(F+L)} \right\rceil$. Note that $C_N = F - (N-1)D$ and $C_N \geq 1$ due 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 $R_i = R_{i-1} - pC_i$ after each stage.

Fixed Increase Self-Scheduling ($FISS$) $C_i = C_{i-1} + B$, where initially $C_0 = \left\lfloor \frac{I}{X*p} \right\rfloor$ (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 ($TFSS$)([3]) is a scheme which uses stages (as in FSS). In each stage the chunks for the PEs are computed by averaging the chunks of TSS.

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 Loop Scheduling Schemes for Distributed Systems

Load balancing in distributed systems 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 [3] we presented and studied distributed versions for the schemes of the previous section. We next review the distributed TSS (DTSS) scheme. The distributed version of the other schemes are similar and can be found in [3].

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:15:14 UTC from IEEE Xplore. Restrictions apply.

Terminology:

- $V_i = Speed(P_i)/min_{1 \le i \le p}\{Speed(P_i)\}$, is the virtual power of P_i, where Speed(P_i) is the CPU-Speed of P_i (computed by the master).
- $V = \sum_{i=1}^{p} V_i$ 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.

The assumption is made that a process 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, 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 Similarly, we understand "Send". 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 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 changed 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 $C_i = A_i * (F - D * (S_{i-1} + (A_i - 1)/2))$, where: $S_{i-1} = A_1 + .. + A_{i-1}$ (see [3]).

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 ($A_i = 0$) goto step 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.

4 Hierarchical distributed schemes

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

All the centralized policies, where a single node (the master) is in charge with 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 synchronized 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-centralized) policies usually do not perform as well as the centralized 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 (TS)

TS ([4],[7]) 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 neighbor 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 T for a cluster of 4 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 neighbor located on its left side, and a busy processor will always send work to the processor on its right. For example,

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:15:14 UTC from IEEE Xplore. Restrictions apply.

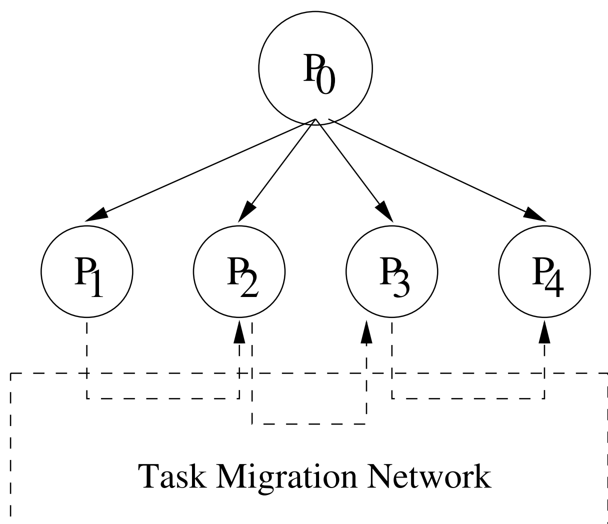

Figure 2. The Tree topology for load balancing

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 TS is the distributed communication, which leads to good scalability.

Note that in the distributed system there is still the need for a central processor which initially distributes the work and at the end it 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 which 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 processors 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 centralized schemes.

Architecture

We use the master-slave centralized 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 centralized 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* ($HDTSS$).

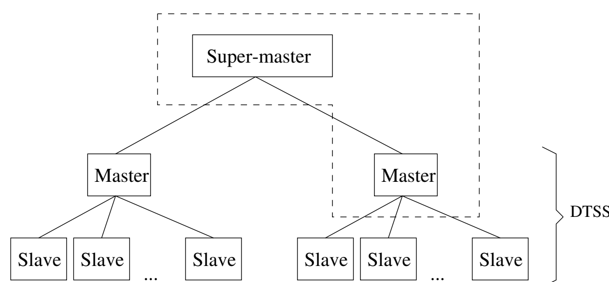

Figure 3. Hierarchical DTSS (two levels of masters)

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 $super - master \leftrightarrow master$ communication applies the master-slave algorithm with master replaced by super-master and slaves replaced by masters.The dotted lines surround processes that can be assigned to the same physical machine, for improved performance.

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 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),(c) and DTSS-Slave 1 (a),(b).

HSlave: Perform the DTSS-Slave steps.

Remark 2: The HM gathers all the messages from its ancestors/descendants and then it send them to the descendants/ancestors (merged) in one message.

5 Implementation

The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit. For the centralized schemes, the master accepts requests from the slaves and services them in the order of their arrival. It replies to each request with a pair of numbers representing

Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER'02)
0-7695-1745-5/02 $17.00 © 2002 **IEEE**
IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:15:14 UTC from IEEE Xplore. Restrictions apply.

the interval of iterations the slave should work on.

The slaves will attach (piggy-back) 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 seem 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 piggy-backing 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 ($TreeS$) ([7]) is different. The slaves do not contend for a central processor when making requests because they have predefined 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.

Table 2. Simple Schemes (CORBA), p = 8; PE_i: $T_{com+wait}$/T_{comp} (sec)

PE	TSS	FSS	$FISS$	$TFSS$	$TreeS$
1	5.8/3.1	5.6/2.8	3.4/2.7	3.5/2.5	1.2/4.5
2	0.9/8.4	5.6/3.0	4.3/2.9	3.9/2.8	0.9/5.6
3	1.0/8.1	5.6/3.2	4.5/2.7	4.0/2.8	1.2/4.5
4	1.3/8.2	5.5/3.3	4.7/2.8	5.3/2.6	0.3/6.1
5	5.3/4.3	1.4/10.3	1.8/12.7	1.4/11.8	1.1/5.1
6	6.3/3.2	1.8/9.9	2.2/13.4	2.1/11.9	0.5/6.2
7	6.1/3.5	1.6/10.3	2.5/13.4	2.7/12.1	1.1/5.6
8	0.7/9.2	1.3/11.1	3.1/12.9	1.7/13.6	0.7/6.5
T_p	10.3	12.5	16.0	15.3	11.8

Table 3. Distributed Schemes (CORBA), p = 8; PE_i: $T_{com+wait}$/T_{comp} (sec)

PE	$DTSS$	$DFSS$	$DFISS$	$DTFSS$	$TreeS$
1	7.4/3.2	8.7/3.6	5.2/4.0	8.9/3.4	1.1/4.4
2	5.5/4.9	9.1/3.4	5.7/3.3	5.7/6.5	0.4/5.6
3	4.6/5.4	9.1/3.4	6.0/3.2	5.6/6.8	1.0/4.9
4	3.8/6.4	9.1/3.2	8.1/4.3	9.4/3.7	0.8/5.3
5	3.8/6.4	4.8/6.5	7.8/4.4	5.7/6.5	1.1/4.9
6	5.2/5.6	4.8/6.7	8.4/4.4	9.2/4.0	1.2/4.9
7	6.7/4.1	4.7/6.7	7.7/3.0	3.9/8.5	0.4/6.1
8	6.2/4.8	3.9/7.6	8.1/4.5	9.4/3.9	1.2/4.7
T_p	9.3	11.1	13.1	12.6	10.3

6 Distributed Simulations

We use the Mandelbrot computation for a window size of 4000 $\times$ 2000,..,12000 $\times$ 6000 on a system consisting of p (= 1,2,4,8,16,32) slaves and one master. We used CORBA(orbix 2000)([12]). The workstations were Sun Ultra 10 with memory sizes 64Mb,128Mb and 384Mb and CPU speeds 300MHz,333MHz and 440MHz. We put an artificial load in the background (matrix by matrix product) on $p/2$ of the slaves. Thus p/2 of the slaves have V_i=1 and the other P_i/2 slaves have $V_i = 2$. We ran the simulations when no other user jobs existed on the workstations.

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 even 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 only implemented HDTSS, because DTSS was faster than the other master-slave schemes.

We present two cases, $dedicated$ and $nondedicated$. 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 on the fast slaves. For this we take $Q_i = 2$. For p $\leq$ 8 we ran only in-dedicated mode and for p $>$ 8 we ran also non-dedicated mode.

The times (Communication,Waiting/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: (1) Comparison between different schemes (TSS,FSS,FISS,TFSS,TreeS) both simple/distributed in CORBA. Results are in Tables 2-3 and Figures 4-7. We conclude that the best centralized scheme(according to our tests) is DTSS and compared it to TreeS (not centralized) in hierarchical implementation. (2) Comparison of DTSS, Hierarchical-DTSS and TreeS in CORBA (Figures 6-7). We conclude that for large sizes, the HDTSS is better than DTSS. Also, HDTSS is better than the TreeS.

The TreeS is slower than DTSS in our runs because in all runs the results are transmitted to a "Master" computer

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:15:14 UTC from IEEE Xplore. Restrictions apply.

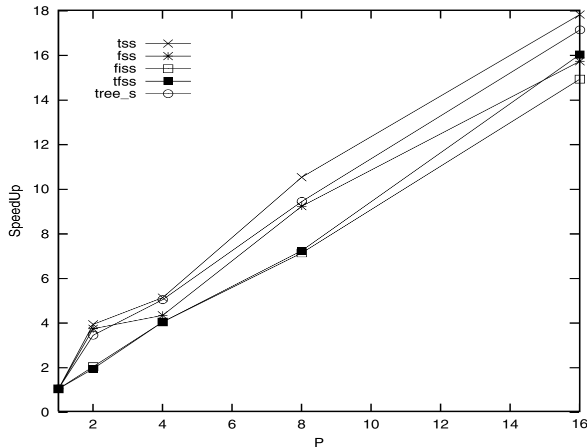

Figure 4. Speedup of Simple Schemes (CORBA)

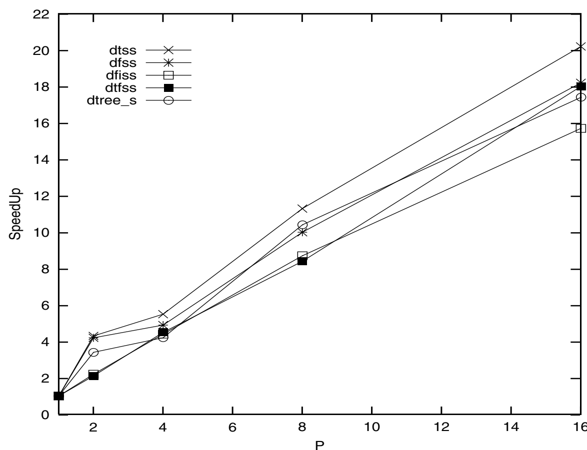

Figure 5. Speedup of Distributed Schemes (CORBA)

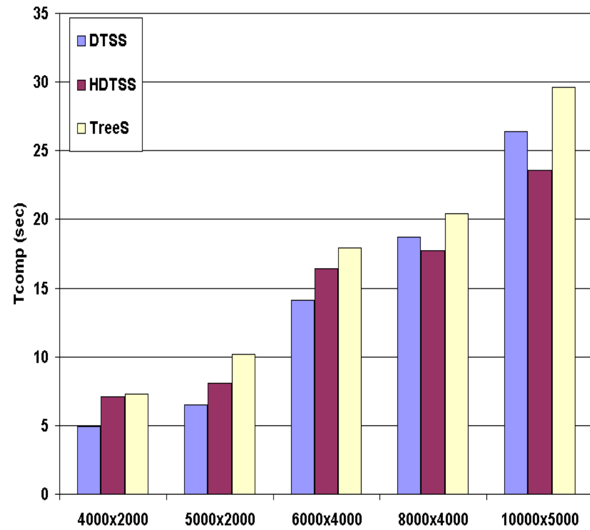

Figure 6. T_{comp}**, Dedicated,** $p = 24$

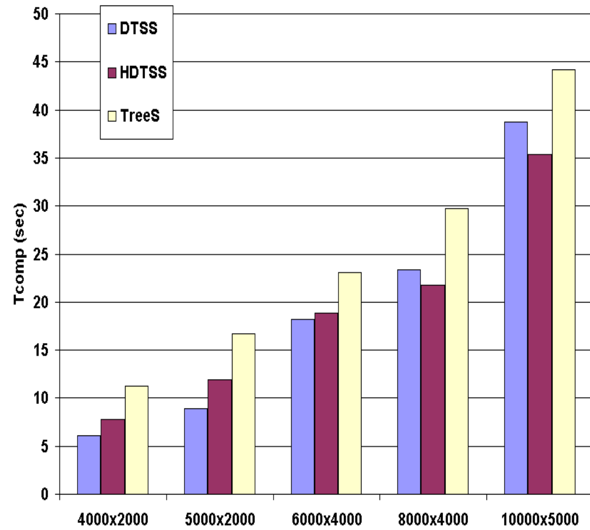

Figure 7. T_{comp}**, Non-Dedicated,** $p = 24$

(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 nondedicated runs since the Tree can not adapt to the load changes it is expected to be slower than the DTSS. The speedup plots in Figures 4-5 are superlinear because the run time on a single slave was made on a slow slave, whereas the parallel time was taken from a run with $p/4$ fast and $p/4$ slow slaves.

7 Conclusions

We studied and implemented (in CORBA Orbix) loop scheduling schemes for heterogeneous distributed systems. Our results show that that the Hierarchical-DTSS gave the best performance results.

References

[1] T.Barth, G.Flender, B.Freisleben and F.Thilo *Load Distribution in a CORBA Environment*, Proc. of the 1999 International Symposium on Distributed Objects and Applications, pp 158 - 166, Edinburg, Scotland, IEEE Press 1999.

IEEE COMPUTER SOCIETY
Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:15:14 UTC from IEEE Xplore. Restrictions apply.

[2] M. Cierniak, W. Li, M. J. Zaki. Loop Scheduling for Heterogeneity, *Proc. of the 4th IEEE Intl. Symp. on High Performance Distributed Computing*, 1995, pp 78 - 85.

[3] A. T. Chronopoulos, R. Andonie, M. Benche, D. Grosu, *A Class of Distributed Self-Scheduling Schemes for Heterogeneous Clusters* , Proc. of the 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001), October 8-11 2001, Newport Beach, California, USA.

[4] S. P. Dandamudi, T. K. Thyagaraj, *A Hierarchical Processor Scheduling Policy for Distributed-Memory Multicomputer Systems*, Proc. of the 4th International Conference on High-Performance Computing, 1997, pp 218 - 223.

[5] T.H. Harrison, C.O'Ryan, D.L.Levine and D.C.Schmidt *The Design and Performance of a Real-time CORBA Events Servic*, Proc. of the OOPSLA'97, Atlanta, GA, Oct 1997.

[6] S. F. Hummel, J. Schmidt, R. N. Uma and J. Wein. Load-Sharing in Heterogeneous Systems via Weighted Factoring, *Proc. of 8th Annual ACM Symp. on Parallel Algorithms and Architectures*, 1996.

[7] T. H. Kim, and J. M. Purtilo. Load Balancing for Parallel Loops in Workstation Clusters, *Proc. of Intl. Conference on Parallel Processing*, Vol III, pp 182 - 189, 1996.

[8] B. B. Mandelbrot. *Fractal Geometry of Nature*, W.H. Freeman & Co, August 1988.

[9] C.O'Ryan and D.L.Levine *Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations*, 5th International Workshop on Objects-oriented Real-time Dependable Systems (WORDS'99) ,Monterey, CA, Nov 15-18,1999 IEEE.

[10] T. Philip and C. R. Das. Evaluation of Loop Scheduling Algorithms on Distributed Memory Systems, *Proc. of Intl Conf. on Parallel and Distributed Computing Systems*, 1997.

[11] 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.

[12] Yong Yan, Canming Jin, Xiaodong Zhang. Adaptively Scheduling Parallel Loops in Distributed Shared-Memory Systems, *IEEE Trans. on Parallel and Distributed Systems*, Vol 8, No 1, Jan. 1997, pp 70 - 81.

[12] *IONA Orbix 2000 (CORBA with C++)*, 200 W. Street, Walthan, MA 02451.

[13] Istabrak Abdul-Fatah, Shikharesh Majumdar. Performance of CORBA-Based Client-Server Architectures, *IEEE Trans. on Parallel and Distributed S ystems*, Vol 13, No 2, 2002, pp 111 - 127.

0-7695-1745-5/02 $17.00 © 2002 **IEEE**

IEEE COMPUTER SOCIETY

Authorized licensed use limited to: University of Texas at San Antonio. Downloaded on May 13,2010 at 19:15:14 UTC from IEEE Xplore. Restrictions apply.