A Class of Loop Self-Scheduling for Heterogeneous Clusters

Anthony T. Chronopoulos*
Dept. of Computer Science,
University of Texas at San Antonio,
6900 North Loop 1604 West, San Antonio, TX 78249,
atc@cs.utsa.edu

Razvan Andonie
Dept. of Electronics and Computers,
Transilvania University of Brasov,
Politehnicii 1-3, 2200, Brasov, Romania,
andonie@vega.unitbv.ro

Manuel Benche and Daniel Grosu
Dept. of Computer Science,
University of Texas at San Antonio,
6900 North Loop 1604 West, San Antonio, TX 78249,
{mbenche, dgrosu}@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 computers. Distributed scheduling schemes suitable for parallel loops with independent iterations on heterogeneous computer clusters have been designed in the past. In this work we consider a class of Self-Scheduling schemes for parallel loops with independent iterations which have been applied to multiprocessor systems. We extend this type of schemes to heterogeneous distributed systems. We present tests that the distributed versions of these schemes maintain load balanced execution on heterogeneous systems.

1 Introduction

To exploit the potential computing power of computer clusters, an important issue is how to assign tasks to computers so that the computer loads are well balanced. The problem is how to assign the different parts of a parallel application to the computing resources, so that to minimize the overall computing time and use efficiently the resources.

Loops are one of the largest sources of parallelism in scientific programs, and thus a lot of research work focused in this area ([1], [2], [8], [12], [13] and the references there in). If the iterations of a loop have no interdependencies, each iteration can be considered as a task and can be scheduled independently. For distributed systems, characterized by heterogeneity and large number of processors, the parallelization of loops is now a current research topic.

Self-scheduling is a large class of adaptive/dynamic centralized loop scheduling methods. Various self-scheduling schemes have been proven successful for shared memory multiprocessor systems: *Pure*, *Chunk*, *Guided* ([8]), *Trapezoid* ([9]), *Factoring* ([3]), *Fixed Increase* ([7]). These schemes will be referred to as *Simple* throughout this paper.

One of the characteristics of the distributed systems is their heterogeneity. Loop scheduling schemes that take into account the characteristics of the different components of the system were devised, for example: 1) *Tree Scheduling* ([5]), 2) *Weighted Factoring* ([4]) and 3) *Distributed Trapezoid Self-Scheduling* ([11]). Schemes 1), 2) and 3) take into account the speeds of the processors of the system in assigning loop iterations whereas 3) is a dynamic scheme that adapts to the actual load of the distributed system.

Here, we review loop scheduling problem solved via existing simple and distributed schemes mapped to a master-slave model. We then propose a new simple self-scheduling scheme, *Trapezoid Factoring Self-Scheduling (TFSS)*, a combination of two successful techniques, *Trapezoid* ([9]) and *Factoring* ([3]). When compared, experimentally, to the other self-scheduling schemes, the new scheme exhibits superior performance. Finally, we obtain distributed versions for the *Factoring*, *Fixed Increase* and $TFSS$. We map and test these simple and distributed schemes on a heterogeneous distributed system.

We chose the Mandelbrot set application as our test prob-

*Corresponding author

Proceedings of the 2001 IEEE International Conference on Cluster Computing (CLUSTER'01)
0-7695-1116-3/02 $17.00 © 2002 IEEE

lem because it has irregular size loop iterations. However, this is not a limitation. The distributed algorithms (considered) have two adaptive properties with respect to: (i) the task size and (ii) the available power of the processors. The first property exists because of the original method design for parallel systems and the second property is due to our design extension for distributed systems. Thus, these schemes are expected to perform well on other types of loop computations.

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 *chunk* is a collection of consecutive iterations. 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;

In section 2, we discuss parallel loop styles, a Master-slave model and the simple self-scheduling schemes. In section 3, we review an existing distributed self-scheduling scheme suitable for distributed computing systems. In section 4, we propose and analyze a new load balancing scheme for homogeneous systems. In section 5, we discuss the implementation of the simple schemes. In section 6, we extend a few schemes to distributed ones and implement them. In section 7, we draw conclusions.

2 Review of loop self-scheduling schemes for homogeneous parallel computers

2.1 Parallel loops distributions

Loops are one of the most important source of concurrency in parallel/distributed computations. A loop is called a *parallel loop* if there are no dependencies among iterations, i.e. iterations can be executed in any order or even simultaneously.

Parallel loops may be presented in any of the styles shown below. $L(i)$ represents the execution time for iteration i (see also [9]).

A parallel loop is *uniformly distributed* if the execution times of all iterations are the same, i.e. the iterations have the same $L(i)$. The following is an example where the same instruction is executed in each iteration:

```
DOALL K = 1 TO I
      X[K] = X[K] + A
END DOALL
```

The following code fragments corresponds to *linearly distributed* loops (increasing and decreasing, respectively).

```
/* increasing */
DOALL K = 1 TO I
      Serial DO J = 1 TO K
                  Serial Loop Body
      End Serial DO
END DOALL

/* decreasing */
DOALL K = 1 TO I
      Serial DO J = 1 TO I-K+1
                  Serial Loop Body
      End Serial DO
END DOALL
```

A *conditional* loop, which may result from IF statements is presented below:

```
DOALL K = 1 TO I
      IF(Expression1) THEN
                  Block1
      ELSE
                  Block2
      ENDIF
END DOALL
```

Figure 1 gives an example of an *irregular* loop style representing the loop distribution required by the Mandelbrot set computation (see [6]).

The loop style can be manipulated in order to make it easier to schedule for parallel execution. For example, there are parallelizing compiler techniques ([8], [10]), such as loop splitting, expression splitting, loop interchange, and loop collapsing, which are used for this purpose.

The more information is available about the loop style, the easier it is to load balance the computation in an efficient manner. The simplest loops for scheduling are those for which the required amount of computation for each iteration is known at compile time. Another class of loops are the *predictable* loops for which we cannot determine the iteration sizes, but they can be ordered. The most difficult class of loops are the *irregular* loops that cannot be ordered.

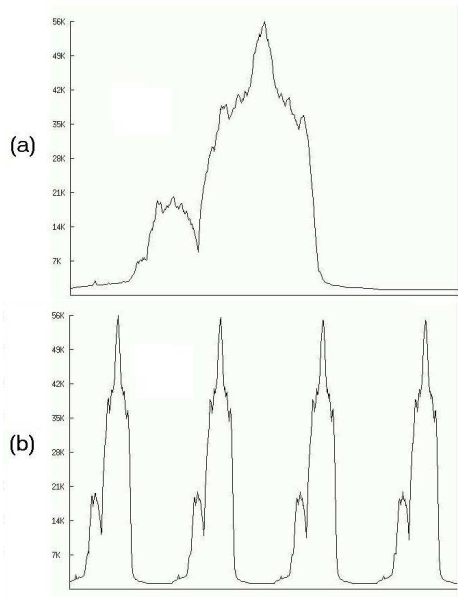

Figure 1. Mandelbrot Set, (a) original and (b) reordered distribution

This class of loops is the most severe test for a scheduling scheme. We use, in our tests, the Mandelbrot fractal computation algorithm ([6]) on the domain [-2.0, 1.25] $\times$ [-1.25, 1.25], for different window sizes (for example 4000 $\times$ 2000, 5000 $\times$ 2000, and so on). The algorithm uses unpredictable irregular loops.

We use a *sampling* technique to reorder loop iterations so that the loop appears more uniform. For a loop with I iterations, a sampling frequency S_f is given. We sample the loop S_f times, taking first the iterations whose index i satisfies $i \, mod \, S_f = 0$, then the iterations with $i \, mod \, S_f = 1$, and so on. After sampling, the S_f samples are placed in a sequence. Since no data dependency is assumed between iterations, computing the sampled loops will produce the same result as the original one. If one sampling is treated as a task, then for some loops we obtain an almost uniform distribution of tasks.

Figure 1 shows the loop distribution for the Mandelbrot set computation, in its original form, and in the reordered form with a $S_f = 4$. The picture corresponds to a windows size of 1200 $\times$ 1200. The X coordinate holds the iteration (column) number, and the Y coordinate shows the number of basic computations associated with that column (ranging from 1200 to 56,000). Figure 2 shows the graph generated by the Mandelbrot set.

In our tests, the computation of one column is considered the smallest unit that can be scheduled independently (i.e. a task). Thus, every iteration corresponds to the computation of the data associated with one column. Because of this, the

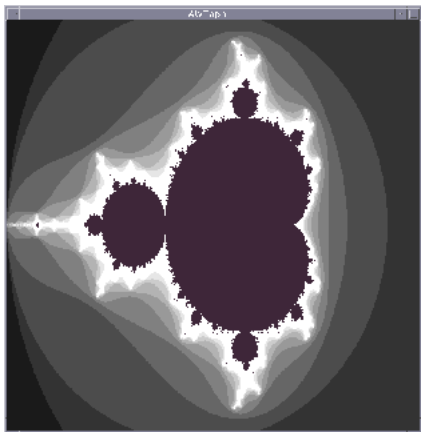

Figure 2. Mandelbrot fractal

tasks are still not uniformly distributed, but they seem more uniform than if no reordering were taking place.

2.2 The Master-Slave model

Self-scheduling is a dynamic loop scheduling method in which idle PEs dynamically request new loop iterations to be assigned to them. The self-scheduling methods we are going to discuss were initially designed for shared memory multiprocessor computers, where requesting PE acquire a lock on the loop index variable in order to be assigned new iterations. This model does not assume a master PE to control lock access.

We will study these methods from the perspective of distributed systems. For this, we use the Master-Slave architecture model, presented in Figure 3. 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.

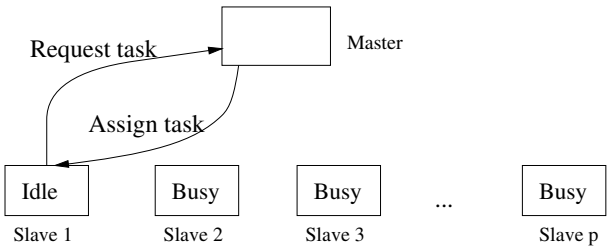

Figure 3. Self-Scheduling schemes: the Master-Slave model

In the remaining of this section we present the following most often used simple self-scheduling schemes and we discuss their advantages and disadvantages: *Pure Self-Scheduling*, *Chunk Self-Scheduling*, *Guided Self-Scheduling* ([8]), *Trapezoid Self-Scheduling* ([9]), *Factoring Self-Scheduling* ([3]) and *Fixed Increase Self-Scheduling* ([7]). These schemes will be called *simple schemes* to be distinguished from their distributed versions.

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.

Chunk Self-Scheduling (CSS) $C_i = k$, where $k \geq 1$ (known as *chunk size* is chosen by the user). For $k = 1$ CSS is the so-called (pure) Self-Scheduling. *Weaknesses*: Increased chance of load imbalance due to difficulty to predict an optimal k, nonadaptive. *Strengths*: Reduced communication/scheduling overheads.

Guided Self-Scheduling (GSS) $C_i = \lceil R_{i-1}/p \rceil$. *Weaknesses*: At the last steps too many small chunks are assigned. *Strengths*: Adaptive. Large chunks initially, implies reduced communication/scheduling overheads in the beginning. A modified version $GSS(k)$ with minimum assigned chunk-size k (chosen by the user) attempts to improve on the weaknesses of GSS.

Trapezoid Self-Scheduling (TSS) $C_i = C_{i-1} - D$, 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 $F = \left\lfloor \frac{I}{2p} \right\rfloor$, $L = 1$. *Weaknesses*: Still many synchronizations may occur. One can improve this by choosing $L > 1$. *Strengths*: Improves the GSS by decreasing the chunk-size linearly. We can calculate the number of tasks 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 $\alpha = 2$ ([3]). 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. *Weaknesses*: Difficult to determine the optimal parameters. *Strengths*: Tests show improvement on previous adaptive schemes (possibly) due to fewer adaptations of the chunk-size.

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) ([7]). Weaknesses/Strengths similar to FSS. However the authors claim is that $FISS$ reduces the communication/scheduling overheads of earlier adaptive schemes which assign chunks with too small sizes.

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 $p = 4$. S stands for the static scheduling scheme, which divides equally all the iterations to the number of PEs. For CSS, k represents the fixed chunk size.

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

Scheme	Chunk size
S	250 250 250 250
SS	1 1 1 1 1 ...
CSS	k k k k k ...
GSS	250 188 141 106 79 59 45 33 25 19 14 11 8 6 4 3 3 2 1 1 1 1
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

Remark: In the rest of the paper, we only consider adaptive schemes. We also do not consider GSS because we use (its linearized approximation) the TSS, which has been reported to have better performance.

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.

3.1 An existing distributed scheme

One characteristic of the distributed systems is their heterogeneity. The load balancing methods adapted to distributed environments usually take into account the processing speeds of the computers forming the cluster. The relative computing powers are used as weights that scale the size of the sub-problem each process is assigned to compute. This is shown to improve sometimes significantly the total execution time when a heterogeneous computing environment is used.

To illustrate this, let us consider the example shown in Table 1 above, with $I = 1000$ and $p = 4$. We assume that the relative processing powers of the four PEs are $w_1 = 1/2$, $w_2 = 1/2$, $w_3 = 1$ and $w_4 = 2$. The first stage of 500 iterations will be divided as 75, 75, 125 and 250 among PEs.

Distributed Trapezoid Self-Scheduling (DTSS): $DTSS$ ([11]) dynamically selects the chunk size for each request, according to the computing power and the actual load of the PE making the request. For this purpose, [11] proposes a model for the computers cluster that includes the number of processes in the run-queue of each PE.

Terminology:

- V_i is the virtual power of P_i (e.g. $V_i = 1$ for the slowest PE).
- $V = \sum_{i=1}^{p} V_i$ 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).
- $A = \sum_{i=1}^{p} A_i$ 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. The DTSS algorithm is described as follows:

Master:

1. **(a)** 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). For each A_i place a request in a queue in the sorted order. Calculate A. **(b)** Use $p = A$ to obtain F, L, N, D as in TSS.

2. **(a)** 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 the ACPSA entry.
   (b) Pick a request from the queue, assign the next chunk with $C_i = A_i * (F - D * (S_{i-1} + (A_i - 1)/2))$, where: $S_{i-1} = A_1 + .. + A_{i-1}$ (see [11]).
   (c) 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.

Slave:

1. Obtain the number of processes in the run-queue Q_i and recalculate A_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.

Remark: (1) To determine chunk sizes, $DTSS$ now applies the same technique as TSS ([9]), but using A instead of p. Each idle processor will be assigned a number of iterations according to its power. (2) To adjust the algorithm for the dynamic changes in the running queues of the processors, $DTSS$ proposes that the slaves report their A_i with every request for work to the master. The master recomputes the scheduling parameters each time more than half of the A_i's have changed. This will ensure good performance when computer loads change unexpectedly (e.g. a new user logs in to the system and starts a computational resources expensive task on some of the processors). This adjustment can be viewed as a change in the slope of the trapezoid function according to the up-to-date state of the system.

4 A new simple Trapezoid scheme with stages

We propose in this section a new self-scheduling scheme for homogeneous systems, *Trapezoid Factoring Self-Scheduling* ($TFSS$), designed by combining the characteristics of two of the most successful load balancing

schemes (see section 2), *Trapezoid Self-Scheduling* ([9]) and *Factoring Self-Scheduling* ([3]).

We use the idea of *stages* introduced by FSS meaning that the iterations are scheduled in groups of p equal-sized chunks. The analysis in [3] suggests that the chunk-size of a stage be computed as half of the remaining number of iterations.

We propose a different approach for determining the number of stages and their chunk-size. Our scheme decreases linearly the chunk size similar to TSS, hopefully producing little synchronization overhead (because of the large chunks at the beginning, and thus, few scheduling steps) and good load balancing (because of the small chunks at the end). The size of a the next chunk is the sum of the next p chunks that would have been computed by the TSS algorithm. The chunk is then equally divided among the p processors, as in FSS. Thus the $TFSS$ chunk-size is computed:

$$C_j^{TFSS} = \sum_{i=k}^{k+p} C_i^{FSS}$$

Example 2: Table 1, illustrates this scheme for a problem with $I = 1000$ iterations to be solved with $p = 4$ processors.

In this example, the first chunks of *TFSS*, containing 113 iterations, were computed as the sum of 125, 117, 109 and 101 (the first four chunks from TSS) divided by the number of processors, 4. In a similar way we obtain 81, 49 and 17 for the next stages. It can be observed that $TFSS$ follows the pattern of FSS (creates groups of p chunks of equal size), but the number of scheduling steps (chunks) and the linear decreasing evolution of the chunk size function is similar to TSS.

5 Implementation of the simple schemes

Our implementation relies on the distributed programming framework offered by the `mpich.1.2.0` implementation of the Message Passing Interface (MPI).

The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit. We reordered the loop with $S_f = 4$. 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 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 ($TreeS$) ([5]) 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.

5.1 Test results

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.

This experiment included 9 computers, one of them being assigned the role of master. We wanted to test the behavior of these techniques in a heterogeneous computing environment, so we used a combination of machines types. The master is a Sun UltraSPARC 10 with 440 MHz CPU speed and 384 MB of physical memory. Three of the slaves are also Sun UltraSPARC 10, but with 128 MB of physical memory, and the remaining of five slaves are Sun UltraSPARC 1 with 166 MHz CPU speed and 64 MB of physical memory. The LAN bandwidth (connecting the Master to the Slaves) is also heterogeneous. It is 10Mbits/sec for the slow slaves and 100Mbits/sec for the fast slaves.

We present two cases, *dedicated* and *nondedicated*. In the first case, processors are dedicated to running our program. In the second, we started resource expensive processes on some slaves. Two such processes are started. Each one adds two random matrices of size 1000. In the nondedicated case, the 'overloaded' processors are as follows: 1) $p = 1$: 1 fast slave; 2) $p = 2$: 1 fast and 1 slow slave; 3) $p = 4$: 1 fast and 1 slow slave; 4) $p = 8$: 1 fast and 3 slow slaves;

The times (Communication/Waiting/Computation) of the slave processors (PE_i) are tabulated for 8 slaves. T_p is the total time measured on the Master PE. Table 2 shows the results. PE_i, for $i = 1, 2, 3$ are the fast PEs. TSS

performed best, followed by $TFSS$. The execution is not well-balanced.

We also plot the speedup of various schemes for $p = 1, .., 8$ in Figures 4 - 5. The following configurations were used. For $p = 1$: 1 fast PE. For $p = 2$: 1 fast and 1 slow PE. For $p = 4$: 2 fast and 2 slow PEs. The 'dip', for $p = 2$, is due to the communication cost and load imbalance. The TSS scales well in the nondedicated case. The results show that in the simple schemes the communication and waiting time is larger than in the distributed schemes (as expected by the theory).

5.2 Improvements to DTSS

The $DTSS$ Algorithm has a couple of difficulties in the parameter computation, which we try to correct here.

(I) Let us assume that we want to solve a problem using two processors, P_1 with $V_1 = 1$ and P_2 with $V_2 = 3$. Moreover, let us say that at the time the computation is started, P_1 will have $Q_1 = 2$ processes and P_2 will have $Q_2 = 3$ processes in the run-queue. Using $DTSS$, there is no available computing power ($A_1 = A_2 = 0$), and the solving of the problem will have to wait.

We propose the use of decimal division in the computation of the available computing power of each processor A_i. and its scaling by a constant integer value (e.g. 10 or 100). Then we get:

$$A_{dec_i} = \frac{V_i}{Q_i}, \;\; A_i = \lfloor 10 \times A_{dec_i} \rfloor \qquad (2)$$

Thus, for our example, P_1 will have $A_1 = \lfloor (1/2) \times 10 \rfloor = \lfloor 0.5 \times 10 \rfloor = 5$, and P_2 will have $A_2 = \lfloor (3/4) \times 10 \rfloor = \lfloor 0.75 \times 10 \rfloor = 7$. Now we have $A = 12$ and we can start solving the problem like in the old $DTSS$ version.

Also, with this approach it is easy to define a lower bound for the load of a processor that will make it unavailable for another computation. We can, for our example, set a limit of $A_{min} = 6$ for which a machine is declared not available. This would mean that only the quick processor can be used for our computation. This allows flexibility in searching for the best configuration of computers to be used.

(II) $DTSS$ assumes that the virtual computing powers of the participating processors are integer numbers. But in actual distributed systems we will never find a computer whose performance can be evaluated exactly as an integer multiple of another's computer performance.

One solution is to use decimal numbers to represent the virtual computing powers. For example, assume that V_2 is 3.4 and $Q_i = 4$. Then $A_2 = (3.4/4) \times 10 = 0.85 \times 10 = 8$. If we did not use decimal numbers, A_2 would be 7, which is an under-estimation of the computing power of P_2. So this solution provides a more precise estimation of the relative processing powers of the computing elements.

6 New distributed self-scheduling schemes

Using our enhanced model of $DTSS$ we observe that any self-scheduling scheme discussed in section 2 can become a Master-Slave centralized *distributed* scheme. We will regard as *distributed* the methods that follow the pattern of $DTSS$ (see section 3.1), i.e. use for load balancing the initial computing power of the elements *and* the runtime information of the number of processes each process is running. For example, *Weighted Factoring* ([4]) is *not* distributed, by this definition, because the actual state of the system is not considered.

We obtained *distributed* versions for *Distributed Trapezoid Factoring Self-Scheduling* ($DTFSS$), *Distributed Factoring Self-Scheduling* ($DFSS$) and *Distributed Fixed Increase Self-Scheduling* ($DFISS$).

The algorithms are very similar to the algorithm for $DTSS$. We must only modify parts 1.(a) and 2.(b) because each of these schemes use different parameters in computing the chunk-size.

Let SC_k denote the sum of the chunk-sizes at the k-th stage of these schemes. Also, let C_i^k denote the chunk-sizes at the k-th stage. For all these schemes, we observe that the chunk size for PE P_j, $j = 1, .., p$ is given by $C_j^k = SC_k * (A_j / A)$.

Modifications of the DTSS algorithm part 1.(b):
(i) $DFTSS$: same as $DTSS$ 1.(b); (ii) $DFSS$: 1.(b) not needed ; (iii) $DFISS$: 1.(b) Compute: $SC_0 = \lfloor \frac{I}{X} \rfloor$ and $B = \left\lceil \frac{2I(1-\sigma/X)}{\sigma(\sigma-1)} \right\rceil$.

Modifications of the DTSS algorithm part 2.(b):
(i) $DFTSS$: 2.(b) Compute $SC_k = \sum_{j=1}^{p} C_j^{TSS}$ and C_j^k; (ii) $DFSS$: 2.(b) Compute $SC_k = \left\lfloor \frac{2R_{i-1}}{A} \right\rfloor$ and C_j^k; (iii) $DFISS$: 2.(b) Compute $SC_k = SC_k + B$ and C_j^k.

6.1 Implementation and test results

Again, we use the Mandelbrot computation for a window size of 4000×2000, on a system consisting of eight heterogeneous slave machines and one master. The machines used for the *dedicated* and the *non-dedicated* case are the same as in the tests described above. For the $TreeS$ the master assigns a number of tasks to the slaves (according to their virtual power) in the initial allocation stage. For the rest of machines the Distributed versions of the schemes are used.

The times (Communication/Waiting/Computation) of the slave processors (PE_i) are tabulated for 8 slaves. T_p is the total time measured on the Master PE. Table 3 shows the results. PE_i, for $i = 1, 2, 3$ are the fast PEs. TSS performed best, followed by $DFISS$. The execution is well-balanced, in terms of the computation times. Also, the communication/waiting times are much reduced compared to the Simple schemes.

We also plot the speedup of various schemes for $p = 1, .., 8$ in Figures 4 - 5. The following configurations were used. For $p = 1$: 1 fast PE. For $p = 2$: 1 fast and 1 slow PE. For $p = 4$: 2 fast and 2 slow PEs.

For nondedicated runs, the 'overloaded' slaves were chosen as in the 'simple' case discussed above. The 'dip', for $p = 2$, is only due to the communication cost. The $DTSS$ scales the best.

Figures 4 - 5 are expected to have low speedup because of the long T_{com}/T_{wait} times. In Figure 3 $TSSS$ shows high speedup (e.g. for $p = 8$) because 2 fast slaves are without extra load. However all other schemes show low speedup because all processors in these simple schemes get assigned the same number of tasks at each stage.

In Figure 6 the PE speeds are according to virtual power and we have 3 fast and 5 slow PEs. The fast PEs are about 3 times faster than slow ones. Thus, without T_{com}/T_{wait} we expect $S_p \leq 4.5$.

In Figure 7 two fast PEs are dedicated and each is 3 times faster than a slow PE. Thus, we expect $S_p \leq 6$.

7 Conclusions

In this paper we obtain distributed extensions for some important loop self-scheduling schemes. We compare the new schemes against their counterparts on a heterogeneous workstation cluster. The main feature of the new schemes is that they take into account the computer processing speeds and their actual loads. Thus the master adapts the assigned load accordingly in order to maintain load balancing. Our test results demonstrate that the new schemes are effective for distributed applications with parallel loops (i.e. loops without inter-iterations dependencies). The $DTSS$ and $DFISS$ were the most efficient amongst all the distributed schemes.

Acknowledgements

The CS graduate student Yu Du did some of the programming. Some reviewer comments helped enhance the quality of presentation. This research was supported, in part, by research grants from (1) NASA NAG 2-1383 (1999-2000), (2) State of Texas Higher Education Coordinating Board through the Texas Advanced Research/Advanced Technology Program ATP 003658-0442-1999 (3) Air Force grant F49620-96-1-0472.

References

[1] 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.

[2] E. H. D'Hollander. Partitioning and Labeling of Loops by Unimodular Transformations, *IEEE Trans. on Parallel and Distributed Systems*, Vol 3, No 4, July 1992, pp 465 - 476.

[3] S. F. Hummel, E. Schonberg, L. E. Flynn. Factoring, a Method for Scheduling Parallel Loops, *Communications of the ACM*, Vol 35, No 8, Aug. 1992.

[4] 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.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] M. J. Wolfe. *Languages and Compilers for Parallel Computing*, MIT Press, July 1990.

[11] J. Xu and A. T. Chronopoulos. Distributed Self-Scheduling for Heterogeneous Workstation Clusters, *Proc. of the 12th Intl. Conf. on Parallel and Distributed Computing Systems*, 1999, pp. 211-217.

[12] E. P. Markatos and T. J. LeBlanc. Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessor, *IEEE Trans. on Parallel and Distributed Systems*, Vol 5, No 4, April 1994, pp 379 - 400.

[13] 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.

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

PE	TSS	FSS	$FISS$	$TFSS$	$TreeS$
Dedicated:					
1	2.7/17.5/3.5	0.2/0.8/3.2	1.5/18.5/3.7	2.3/18.7/3.2	0.6/0.0/3.3
2	0.9/18.8/3.7	4.4/15.1/3.3	1.5/19.8/3.4	0.1/0.9/3.2	6.3/12.9/5.4
3	1.3/18.3/3.7	2.8/16.6/3.3	1.8/19.3/3.5	2.1/18.4/3.2	6.1/12.1/5.6
4	1.0/17.5/4.4	1.6/17.6/8.9	1.1/19.8/9.0	0.6/16.7/8.8	6.6/9.2/7.3
5	0.9/12.3/8.0	4.5/9.1/9.3	2.2/7.9/9.6	2.8/9.5/9.7	7.6/6.0/6.5
6	2.4/7.5/10.4	4.2/9.2/9.8	3.8/6.2/8.9	5.0/8.7/9.9	4.9/2.1/9.7
7	4.0/5.7/10.7	3.8/5.9/9.9	3.8/4.4/9.3	4.9/5.9/10.1	3.5/0.0/7.4
8	3.6/4.8/11.8	8.1/4.0/10.4	2.2/4.3/8.9	5.3/4.2/10.2	5.1/0.0/6.0
T_p	23.6	28.1	30.0	26.2	25.0
NonDedicated:					
1	6.0/12.3/9.5	7.0/13.9/9.0	2.3/1.1/9.1	2.9/21.4/11.6	1.8/0.0/9.4
2	8.8/13.4/5.5	0.4/0.8/3.1	2.7/1.0/3.4	2.8/23.0/3.6	9.7/20.8/6.9
3	6.1/14.3/7.2	2.9/2.6/3.4	2.4/1.4/3.5	1.6/18.7/4.0	19.7/19.9/5.7
4	3.0/11.3/13.3	4.6/13.2/28.2	2.8/8.8/27.4	3.9/17.4/26.9	14.0/14.5/15.9
5	4.7/10.4/9.2	9.8/10.7/9.1	3.7/6.0/9.3	3.2/7.4/9.2	23.2/9.0/7.2
6	5.1/8.2/10.3	10.1/10.0/9.6	3.2/7.6/9.1	3.0/5.7/8.9	19.7/4.1/9.8
7	1.1/5.5/17.3	2.8/10.3/28.9	1.5/12.5/27.6	2.7/9.1/28.4	4.6/4.1/15.8
8	4.2/3.8/15.9	6.1/8.9/30.3	2.9/13.7/26.6	3.3/7.1/25.6	15.9/0.0/12.7
T_p	27.8	46.0	48.1	45.8	46.8

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

PE	$DTSS$	$DFSS$	$DFISS$	$DTFSS$	$TreeS$
Dedicated:					
1	2.2/1.8/6.3	4.1/7.8/5.6	2.5/6.4/5.6	1.4/9.3/5.6	3.1/9.0/5.7
2	2.7/1.2/6.6	2.9/8.8/5.8	2.5/6.2/5.8	1.9/8.5/5.6	3.4/7.7/6.1
3	2.1/1.6/7.0	1.7/10.5/5.3	1.5/7.9/5.4	1.4/9.6/5.6	8.9/0.0/10.2
4	2.5/2.2/5.9	3.3/7.3/6.8	2.3/6.0/6.5	2.4/7.9/6.5	3.1/0.0/5.6
5	2.4/4.2/4.4	2.5/8.3/6.0	1.9/7.3/6.1	1.6/9.2/6.3	2.4/0.0/5.8
6	2.0/5.7/3.7	2.1/8.5/6.3	0.9/9.2/5.6	2.0/9.4/6.0	4.9/0.0/6.1
7	0.5/7.7/4.2	1.6/9.8/5.7	1.9/8.4/5.9	1.5/10.1/6.0	5.2/2.1/5.7
8	1.3/9.5/2.6	2.8/8.3/6.1	3.5/7.3/6.0	3.4/8.3/6.0	4.3/0.0/10.4
T_p	13.4	17.6	16.9	17.6	18.1
NonDedicated:					
1	1.2/3.0/8.5	0.9/9.8/10.6	0.9/8.4/6.5	1.9/9.3/10.7	10.8/14.1/6.7
2	2.2/1.5/8.5	2.5/13.6/7.0	1.6/5.8/7.8	3.4/13.3/6.7	11.1/15.3/6.3
3	1.3/2.4/8.5	2.1/14.7/6.2	0.8/7.3/7.4	1.3/15.7/6.5	7.5/16.2/9.8
4	0.7/5.5/6.6	5.8/0.8/14.4	2.5/3.3/9.8	4.7/2.3/15.1	3.9/0.0/13.6
5	0.4/6.6/7.7	3.0/5.3/13.4	1.3/6.0/8.9	3.2/4.6/14.7	5.9/1.3/14.6
6	1.6/3.9/7.1	0.8/13.7/7.3	1.4/7.3/7.6	0.9/15.6/7.1	9.0/4.0/12.2
7	0.9/8.2/7.2	1.0/8.0/13.0	2.6/5.5/8.4	1.3/7.8/13.1	14.0/7.5/7.0
8	1.8/4.5/6.3	1.1/13.5/7.4	2.1/7.0/8.0	1.8/13.7/7.2	12.6/10.4/8.8
T_p	16.6	23.3	17.7	23.6	33.3

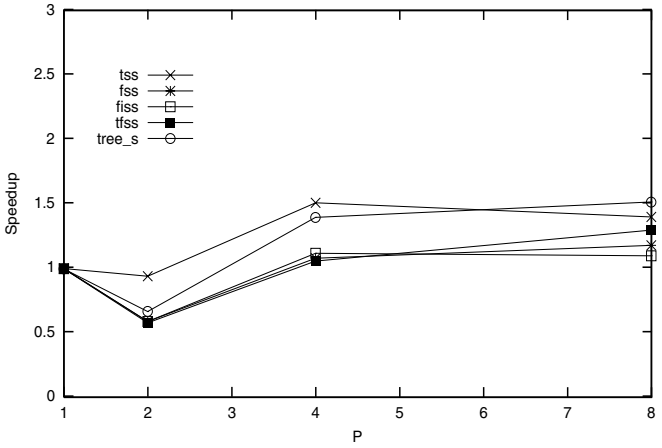

Figure 4. Speedup of Simple Schemes - Dedicated

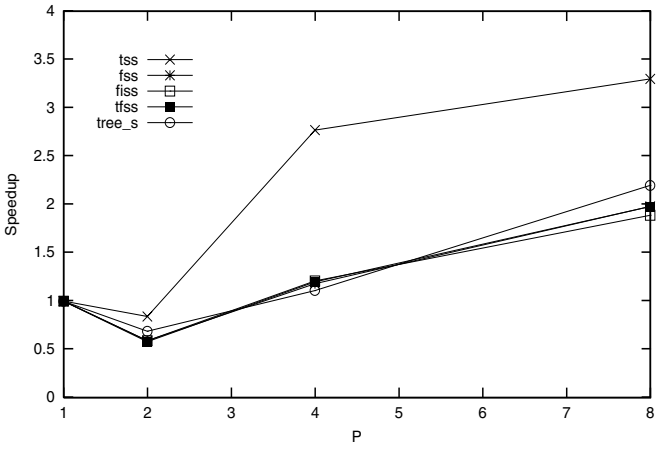

Figure 5. Speedup of Simple Schemes - NonDedicated

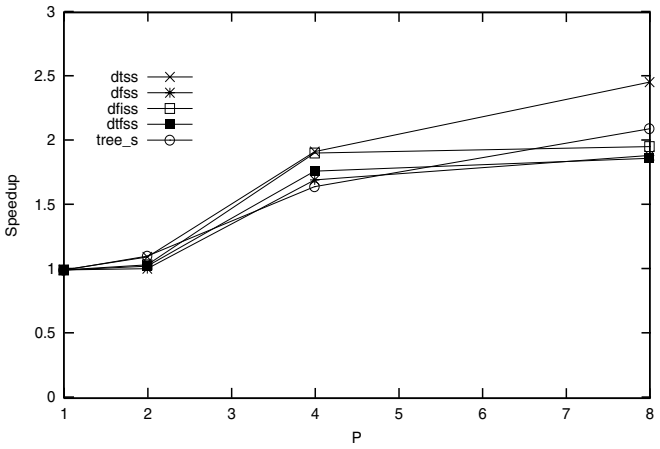

Figure 6. Speedup of Distributed Schemes - Dedicated

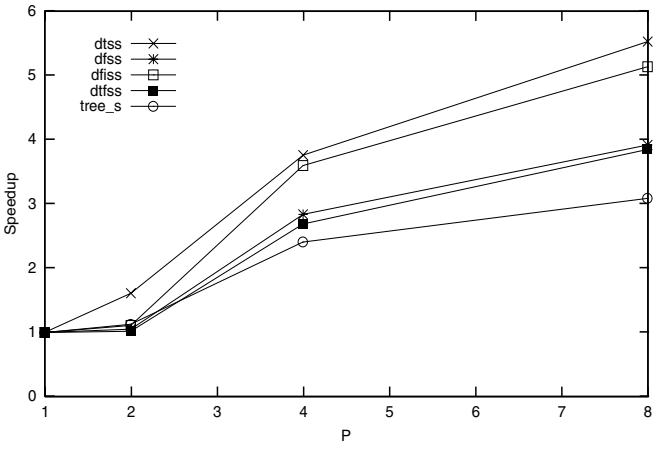

Figure 7. Speedup of Distributed Schemes - NonDedicated