CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2005; (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe.960

Distributed loop-scheduling schemes for heterogeneous computer systems

Anthony T. Chronopoulos[1,*,†], Satish Penmatsa[1], Jianhua Xu[2] and Siraj Ali[1]

[1]*Department of Computer Science, University of Texas at San Antonio, 6900 N. Loop 1604 West, San Antonio, TX 78249, U.S.A.*
[2]*Lucent Technologies, 101 Crawfords Corner Rd, Holmdel, NJ 07733, U.S.A.*

SUMMARY

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. Some distributed scheduling schemes suitable for parallel loops with independent iterations on heterogeneous computer clusters have been designed in the past. In this work we study self-scheduling schemes for parallel loops with independent iterations which have been applied to multiprocessor systems in the past. We extend one important scheme of this type to a distributed version suitable for heterogeneous distributed systems. We implement our new scheme on a network of computers and make performance comparisons with other existing schemes. Copyright © 2005 John Wiley & Sons, Ltd.

KEY WORDS: heterogeneous distributed systems; loop scheduling; master–slave model

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 [1] (and references therein) and some recent results are presented in [2,3]. Research results also exist

*Correspondence to: Anthony T. Chronopoulos, Department of Computer Science, University of Texas at San Antonio, 6900 N. Loop 1604 West, San Antonio, TX 78249, U.S.A.
†E-mail: atc@cs.utsa.edu

Contract/grant sponsor: National Science Foundation; contract/grant number: CCR-0312323

Received 30 June 2004
Revised 19 March 2005
Accepted 29 March 2005

on scheduling loops and linear algebra data parallel computations on message-passing parallel systems and on heterogeneous systems; see [4–19].

Loops can be scheduled statically at compile-time. This type of scheduling has the advantage of minimizing the scheduling time overhead, but it may cause load imbalancing when the loop is not uniformly distributed. Examples of such scheduling are block, cyclic, etc. [1]. Dynamic scheduling adapts the assigned number of iterations whenever the size of the loop tasks are not known in advance. An important class of dynamic scheduling are the self-scheduling schemes [1]. In these schemes, each idle processor accesses a few loop iterations to execute next. In parallel computer systems, these schemes can be implemented using a critical section for the processors to access the loop iteration index as a shared variable. This is why these schemes are called self-scheduling schemes. On distributed systems we can implement these schemes (as shown later) using a master–slave model.

Heterogeneous systems are characterized by heterogeneity and large number of processors. Some distributed schemes that take into account the characteristics of the different components of the heterogeneous system have devised in the past, for example: (1) *tree scheduling*; and (2) *weighted factoring* (see [12,14]).

In Section 2, we review loop distributions and simple parallel loop self-scheduling schemes. In Section 3, we present distributed self-scheduling schemes. In Section 4, an implementation, results and comparisons are presented. In Section 5, conclusions are drawn.

Notation

The following is a list of common notation used throughout the paper.

- PE (processor element) 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 which execute the computational tasks.
- $P_1, P_2, \ldots, P_p$ represent the p slave PEs in the system.
- N is the number of scheduling steps (equal to the total number of chunks).
- A few consecutive iterations are called a *chunk*. C_i is the ith chunk size (where $i = 1, 2, \ldots, N$). The ith chunk is assigned to the slave PE making the ith request.
- t_j, $j = 1, \ldots, p$, is the execution time of P_j to finish all the 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.

2. REVIEW OF LOOP SELF-SCHEDULING SCHEMES FOR HOMOGENEOUS PARALLEL COMPUTERS

2.1. Parallel loop distributions

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 loop styles shown below. $L(i)$ represents the execution time for iteration i (see also [20]).

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 [21]).

The loop style can be manipulated in order to make it easier to schedule for parallel execution. For example, there are parallelizing compiler techniques [22], such as loop splitting, expression splitting, loop interchange and loop collapsing, which are used for this purpose.

The greater the availability of information 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. This class of loops is the most severe test for a scheduling scheme. We use, in our tests, the Mandelbrot fractal computation

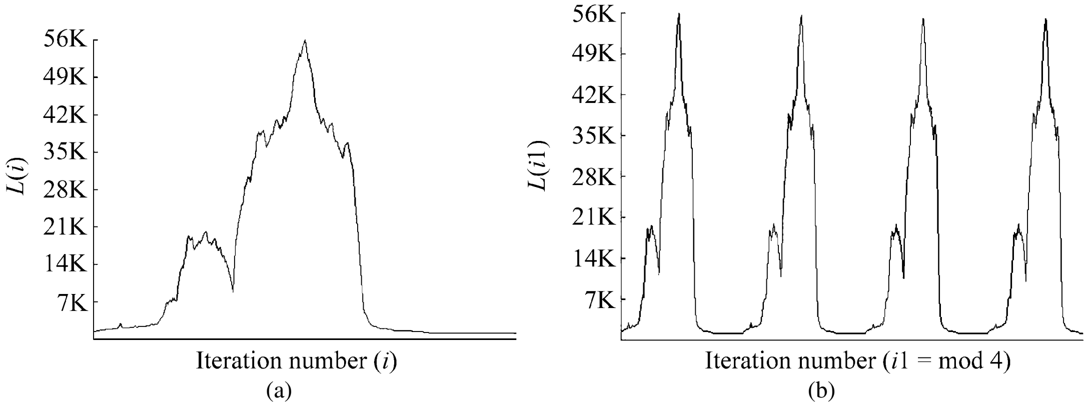

Figure 1. Mandelbrot Set: (a) original ($L(i)$ = number of computations for ith iteration); and (b) reordered distribution ($L(i1)$ = number of iterations for $i1$th iteration).

algorithm [21] on the domain $[-2.0, 1.25] \times [-1.25, 1.25]$, for different window sizes (for example, 4000×2000, 5000×2000, and so on). The algorithm uses unpredictable irregular loops.

We use a *sampling* technique (or cyclic distribution) 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 $i \bmod S_f = 0$, then the iterations with $i \bmod S_f = 1$, 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. 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 window size of 1200×1200. 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).

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

2.2. 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 study these methods from the perspective of heterogeneous systems. For this, we use the master–slave architecture model (Figure 2). 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

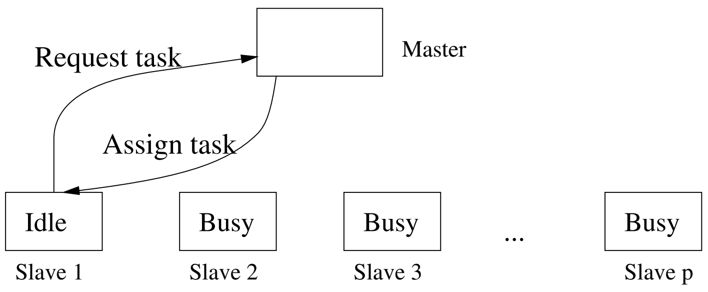

Figure 2. Self-scheduling schemes: the master–slave model.

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.

In a generic self-scheduling scheme, at the ith 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 $f(\cdot, \cdot)$ is a function possibly of more inputs than just R_{i-1} and p. Then the master assigns C_i tasks to a slave PE. Imbalance depends on the execution time difference between t_j, for $j = 1, \dots, p$. This difference may be large if the first chunk is too large. There is a detailed theoretical analysis of these schemes in [15]. Amongst other results, it is shown that trapezoid self-scheduling (TSS) performs better than guided self-scheduling (GSS) with loops of varying task sizes.

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) [20] $C_i = C_{i-1} - D$, with (chunk) decrement $D = \lfloor (F - L)/(N-1) \rfloor$, where the first and last chunk-sizes (F, L) are user/compiler-input or $F = \lfloor I/2p \rfloor$, $L = 1$. The number of scheduling steps assigned is $N = \lceil 2 * I/(F + L) \rceil$. Note that $C_N = F - (N-1)D$ and $C_N \geq 1$ due to integer divisions.
- *Chunk self-scheduling* (CSS) $C_i = k$, where $k \geq 1$ (known as the *chunk size* and is chosen by the user). For $k = 1$, CSS is the so-called (pure) self-scheduling. Weaknesses include increased chance of load imbalance due to the difficulty in predicting an optimal k and that it is non-adaptive. Its strengths include reduced communication/scheduling overheads.
- *Guided self-scheduling* (GSS) [23,24] $C_i = \lceil R_{i-1}/p \rceil$. The weaknesses of this method is that too many small chunks are assigned to the last steps. Its strength is that it is adaptive. Having 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.

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

Table I. 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

scheduling scheme, which divides all of the iterations equally between the number of PEs. For CSS, k represents the fixed chunk size.

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.

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 sometimes shown to significantly improve the total execution time when a heterogeneous computing environment is used.

3.1. Tree scheduling (TreeS)

TreeS [10,14] 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 3 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 input/output (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, in Figure 3, 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.

Note that in the heterogeneous system there is still the need for a central processor which initially distributes the work and collects the results at the end, unless the problem is of such a nature that the

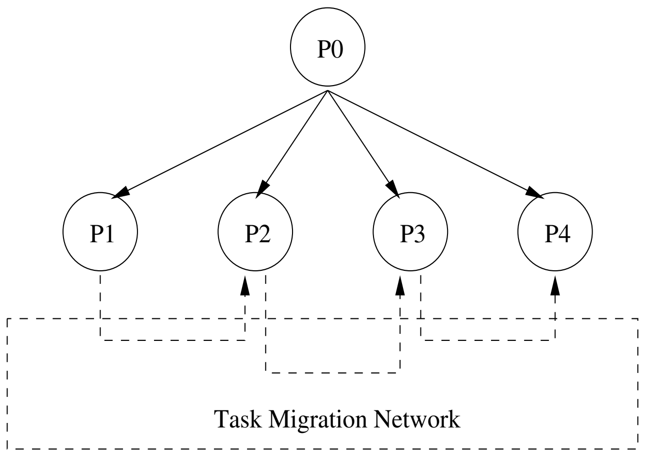

Figure 3. The tree topology for load balancing.

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 PE 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.

3.2. Derivation of a distributed self-scheduling

We apply our model to TSS, which we then call *distributed trapezoid self-scheduling* (DTSS) [8]. The same model can be used with other self-scheduling schemes.

Terminology

- $U_{i,j}$: the load to be assigned to a single virtual PE j (where $j = 1, \ldots, A_i$) of computer P_i.
- C_i: the next chunk size for PE P_i.
- 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.
- $A_i = \lfloor V_i/Q_i \rfloor$ 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 ACP of the cluster.

We apply the TSS (see [20] for details) to the set of available virtual PEs in order to obtain the chunk sizes in DTSS. Let I, F, L, D be as defined for TSS in Section 2. The programmer may determine the pair (F, L), according to TSS; or the following formula may be used in the conservative selection approach (by default):

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

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

$$N = 2I/(F + L) \tag{3}$$

the step decrease for each task assignment is

$$D = (F - L)/(N - 1) \tag{4}$$

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

$$U_{1,1} = F \tag{5}$$

and for $1 < j \leq A_1$,

$$U_{1,j} = F - D * (j - 1) \tag{6}$$

Therefore,

$$C_1 = \sum_{j=1}^{A_1} U_{1,j} = F * A_1 - D * \sum_{j=1}^{A_1} (j - 1) \tag{7}$$

which gives

$$C_1 = A_1 * F - D * A_1 * (A_1 - 1)/2 = A_1 * (F - D * (A_1 - 1)/2) \tag{8}$$

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

$$U_{i,j} = F - D * (A_1 + \cdots + A_{i-1} + j - 1) \tag{9}$$

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

$$C_i = \sum_{j=1}^{A_i} U_{i,j} = F * A_i - D * A_i * (S_{i-1} + (A_i - 1)/2) = A_i * (F - D * (S_{i-1} + (A_i - 1)/2)) \tag{10}$$

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

3.3. DTSS

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.
   (b) Calculate A. 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 $S_{i-1} = A_1 + \cdots + A_{i-1}$ (see [8]).
   (c) If more than half of the A_i have changed since the last time, update the ACPSA and go to step 1(b), 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.

Remarks. (1) To determine chunk sizes, DTSS now applies the same technique as TSS [20], 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 slave PEs 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 have changed. This will ensure good performance when computer loads change unexpectedly (e.g. a new user logs into 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. IMPLEMENTATION AND TEST RESULTS

4.1. Implementation

Our implementation relies on the distributed programming framework offered by the `mpich.1.2.0` implementation of the message-passing interface (MPI) [25].

The slave PEs will attach (piggy-back) to every request, except the first, 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 slave PEs stored the results of their requests locally). This technique produced longer finishing times because when all of the slave PEs 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 slave PEs will work on their requests while few slave PEs communicate data to the master.

The implementation for TreeS [14] is different. The PEs do not contend for a central processor when making requests because they have pre-defined partners. However, the data still have to be collected on a single central processor. When we used the approach described above (sending all the results at the end of the computation), we observed a lot of idling time for the PEs, thus degrading the performance. We also implemented an alternative: the PEs send their results to the central coordinator (master) 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.

The distributed system consisted of nine computers, one of them being assigned the role of master. We wanted to test the behavior of these schemes in a heterogeneous computing environment. So, we put an artificial load (a continuously running matrix multiplication process on 100×100 matrices) on half of the PEs on a homogeneous system. Then the PEs are in two subsets (each containing half of all the machines): (i) machines with three loads, which are assumed to have a virtual power of one; and (ii) machines with no load, which are assumed to have a virtual power of four. This was verified by timing any program running on a single machine of type (i) or (ii) above. The actual machines used were Sun Blade 100 with 502 MHz CPU speed and 512 MB of physical memory. The machines are connected by an Ethernet bus. The bus bandwidth is equal to 100 Mbits/s.

We present two cases, *dedicated* and *non-dedicated*. In the dedicated case, processors are dedicated to running our program and their loads do not change during the execution of our program. Recall that half of the PEs are fast and half are slow. In the non-dedicated case, we started three continuously running processes (doing matrix multiplication on 100×100 matrices) on half of the fast PEs in the middle of the application program execution. After doing this the fast PEs become overloaded and their virtual power is equal to one. Thus, $3p/4$ of the PEs have a virtual power of one and $p/4$ of the PEs have a virtual power four.

We ran TSS (dedicated), DTSS ((non-)dedicated) and TreeS ((non-)dedicated). We also ran DTSS and TreeS (dedicated) for $p = 1, 2, 4, 8, 16$ PEs in order to compute the speedup. Recall that, in the dedicated case, half of the PEs have a virtual power of four and the other half have a virtual power of one. In order to plot the speedup curve, we consider each fast PE in the system as a single virtual processor and the slow PEs are considered to be fractions of one virtual processor. For example, if we have 16 PEs, half with virtual power four and half with virtual power one, then the total number of virtual processors (V_p) equals 10 ($= 8 \times 1 + 8 \times 0.25$). We computed the speedup according to the following equation:

$$S_p = \frac{\min\{T_{p_1}, T_{p_2}, \ldots, T_{p_p}\}}{T_p} \tag{11}$$

where T_{p_i} is the execution time on one PE and T_p is the execution time on p PEs. However, in the plotting of S_p, we use V_p instead of p on the x-axis.

4.2. Test problems

We use the Mandelbrot computation for a window size of 6000×4000 [21], the Jacobi method computation for a size of 5000×5000, and the Gaussian elimination for a size of 2000×2000, on a system consisting of eight heterogeneous slave machines and one master. The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit. We reordered the loop with $S_{\mathrm{f}} = 4$. For the master–slave schemes, the master accepts requests from the slave PEs 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 PE should work on. For the TreeS, the master assigns a number of tasks to the PEs (according to their virtual power) in the initial allocation stage. We next give the pseudocodes for the Jacobi and Gaussian elimination problems.

```
/* Jacobi */
Serial DO K = 1, MAXITER
      DOALL I = 1, N
                X[K][I] = b[I]
                Serial DO J = 1, I-1
                     X[K+1][I] = X[K][I] - A[I][J]*X[K][J]
                End Serial DO
                Serial DO J = I+1, N
                     X[K+1][I] = X[K][I] - A[I][J]*X[K][J]
                End Serial DO
                X[K+1][I] = X[K+1][I]/A[I][I]
                IF 'CONVERGED' THEN
                     EXIT
                ENDIF
      END DOALL
End Serial DO

/* Gaussian Elimination */
Serial DO K = 2, N
      DOALL I = K, N
                Serial DO J = K-1, N+1
                A[I][J] = A[I][J] - A[K-1][J]*A[I][K-1]/A[K-1][K-1]
                End Serial DO
      END DOALL
End Serial DO
```

The outermost loop in these pseudocodes is serial and it is controlled by the master, which assigns chunks of the parallel loop to the slave PEs.

4.3. Results

Tables II–V show the results. The times (computation/communication) of the slave PEs are tabulated for eight PEs. T_p is the total parallel time measured on the master PE. T_p includes the sum of communication times of all PEs (because a single shared Ethernet bus is used), the concurrent PE computation times and any PE idle times.

Table II. Mandelbrot computation, dedicated case, $p = 8$; PE_i: $T_{\text{comp}}/T_{\text{com}}$ (in seconds).

PE	TSS	DTSS	TreeS
1	42.4/1.2	18.3/0.5	17.2/0.4
2	44.0/1.4	19.6/0.5	25.5/0.6
3	51.5/1.1	19.5/0.5	24.9/0.6
4	14.2/1.3	19.1/0.5	21.8/0.7
5	13.4/1.2	22.3/1.8	18.0/1.4
6	13.6/1.6	21.2/1.6	18.0/1.0
7	14.3/0.8	16.1/1.7	18.5/1.0
8	13.5/1.9	16.5/1.7	17.0/1.6
T_p	53.1	28.0	32.8

Table III. Mandelbrot computation, non-dedicated case, $p = 8$; PE_i: $T_{\text{comp}}/T_{\text{com}}$ (in seconds).

PE	DTSS	TreeS
1	26.0/0.6	17.3/0.7
2	25.8/0.9	28.2/0.8
3	25.3/1.2	53.0/0.5
4	26.0/0.8	21.9/0.7
5	25.3/1.7	20.8/1.1
6	27.3/0.8	70.8/2.0
7	28.1/1.5	20.7/1.0
8	25.6/2.0	17.8/1.0
T_p	39.0	73.6

Table IV. Jacobi computation, distributed schemes, $p = 8$; PE_i: $T_{\text{comp}}/T_{\text{com}}$ (in seconds).

	Dedicated		Non-dedicated	
PE	DTSS	TreeS	DTSS	TreeS
1	70.1/4.2	34.2/3.8	74.1/4.4	36.1/4.8
2	71.2/4.8	49.4/4.5	73.1/4.8	49.9/5.2
3	70.7/4.3	49.1/4.4	73.6/4.3	48.1/4.6
4	70.3/4.2	35.2/3.2	74.8/4.1	53.3/4.1
5	65.3/3.1	63.0/3.0	68.4/3.2	69.7/4.8
6	66.7/3.1	74.1/3.4	72.1/3.2	95.2/5.3
7	67.3/3.1	86.6/3.4	73.3/3.2	79.7/4.9
8	71.4/3.1	96.6/4.3	75.4/3.7	109.9/4.1
T_p	80.7	125.1	95.3	146.9

Table V. Gaussian elimination, dedicated case, $p = 8$; PE_i: $T_{\mathrm{comp}}/T_{\mathrm{com}}$ (in seconds).

PE	DTSS
1	77.6/2.6
2	83.3/2.7
3	74.2/2.6
4	74.4/2.6
5	90.5/2.4
6	74.7/2.7
7	87.7/3.1
8	80.2/2.9
T_p	110.5

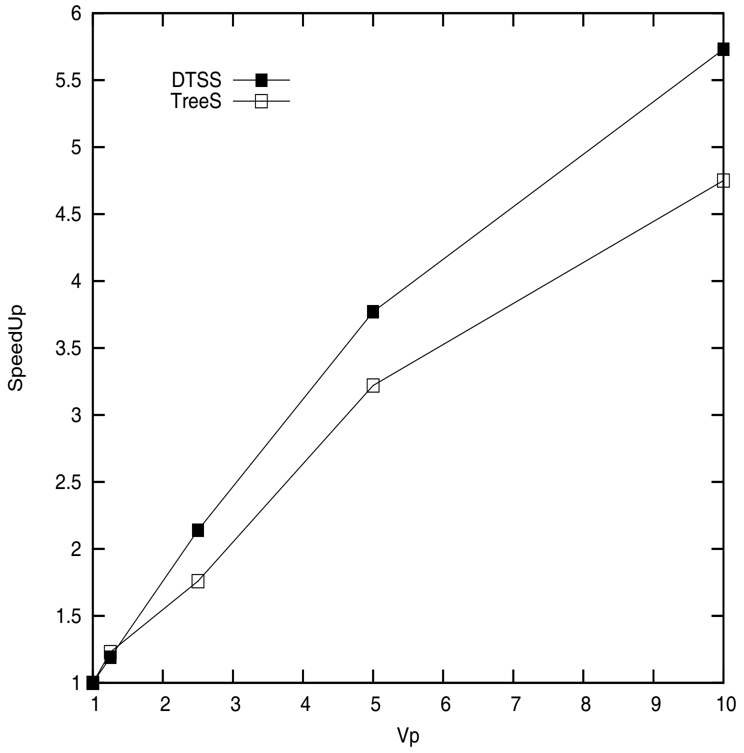

Figure 4. Speedup versus number of virtual processors.

Table II compares the simple TSS and the distributed schemes (DTSS and TreeS) for the Mandelbrot problem in the dedicated case. From this comparison we observe that the distributed schemes exhibit better load balance and performance. Table III compares the DTSS and TreeS for the Mandelbrot problem, in the non-dedicated case. The DTSS shows better load balance and performance than the TreeS.

Table IV contains the results for the Jacobi computation for eight PEs. It can be seen again that DTSS performs better and it is better balanced than TreeS. Since DTSS outperforms TreeS, we only implemented DTSS for Gaussian elimination for the dedicated case. The results (Table V) show that DTSS is also well balanced for this case. The superior performance of the DTSS in the non-dedicated case is due to its adaptiveness to the loads of the PEs, a property missing in the TreeS.

In Figure 4, we present the speedup of DTSS and TreeS for the dedicated case. We can see that the schemes are scalable.

5. CONCLUSIONS

In this paper we study existing self-scheduling schemes for parallel loops. We obtain distributed extensions for an important loop self-scheduling scheme (DTSS). We compare the new scheme against: (1) its counterpart (simple TSS) on a heterogeneous network of computers; (2) an existing distributed scheme (TreeS). The main feature of the new scheme is that it takes 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 scheme is effective for distributed applications with parallel loops. The DTSS is the best balanced and efficient scheme.

ACKNOWLEDGEMENTS

This work was supported in part by National Science Foundation under grant CCR-0312323. Some reviewers' comments helped enhance the quality of presentation.

REFERENCES

1. Fann YW, Yang CT, Tseng SS, Tsai CJ. An intelligent parallel loop scheduling for parallelizing compilers. *Journal of Information Science and Engineering* 2000; **16**:169–200.
2. Bull JM. Feedback guided dynamic loop scheduling: Algorithms and experiments. *Proceedings of the 4th International Euro-Par Conference*, Southampton, U.K., 1998 (*Lecture Notes in Computer Science*, vol. 1470). Springer: Berlin, 1998; 377–382.
3. Hancock DJ, Bull JM, Ford RW, Freeman TL. An investigation of feedback guided dynamic scheduling of nested loops. *Proceedings of the International Workshops on Parallel Processing*, Toronto, Canada, August 2000, Sadayappan P (ed.). IEEE Press: Piscataway, NJ, 2000; 315–321.
4. Banicescu I, Liu Z. Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes. *Proceedings of the High Performance Computing Symposium 2000*, Washington, DC, 2000. The Society for Modeling and Simulation International (SCS): San Diego, CA, 2000; 122–129.
5. Banicescu I, Velusamy V, Devaprasad J. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing* 2003; **6**:215–226.

6. Barbosa J, Tavares J, Padilha AJ. Linear algebra algorithms in a heterogeneous cluster of personal computers. *Proceedings of the 9th Heterogeneous Computing Workshop (HCW 2000)*, Cancun, Mexico, May 2000. IEEE Computer Society Press: Los Alamitos, CA, 2000; 147–159.
7. Cierniak M, Li W, Zaki MJ. Loop scheduling for heterogeneity. *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, August 1995. IEEE Computer Society Press: Los Alamitos, CA, 1995; 78–85.
8. Chronopoulos AT, Andonie R, Benche M, Grosu D. A class of distributed self-scheduling schemes for heterogeneous clusters. *Proceedings of the 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001)*, Newport Beach, CA, 8–11 October 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001.
9. Dandamudi SP. The effect of scheduling discipline on dynamic load sharing in heterogeneous distributed systems. *Proceedings of the 5th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'97)*, Haifa, Israel, January 1997. IEEE Press: Piscataway, NJ, 1997.
10. Dandamudi SP, Thyagaraj TK. A hierarchical processor scheduling policy for distributed-memory multicomputer systems. *Proceedings of the 4th International Conference on High-Performance Computing* 1997; 218–223.
11. Goumas G, Drosinos N, Athanasaki M, Koziris N. Compiling tiled iteration spaces for clusters. *Proceedings of the IEEE International Conference on Cluster Computing*, Chicago, IL, 2002. IEEE Computer Society Press: Los Alamitos, CA, 2002; 360–369.
12. Hummel SF, Schmidt J, Uma RN, Wein J. Load-sharing in heterogeneous systems via weighted factoring. *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, Padua, Italy, 1996. ACM Press: New York, 1996; 318–328.
13. Kee Y, Ha S. A robust dynamic load-balancing scheme for data parallel application on message passing architecture. *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'98)*, vol. 2, Las Vegas, NM, 1998. CSREA Press: Athens, GA, 1998; 974–980.
14. Kim TH, Purtilo JM. Load balancing for parallel loops in workstation clusters. *Proceedings of the International Conference on Parallel Processing*, vol. III, Bloomingdale, IL, 1996. IEEE Press: Piscataway, NJ, 1996; 182–190.
15. Markatos EP, LeBlanc TJ. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 1994; **5**(4):379–400.
16. Philip T, Das CR. Evaluation of loop scheduling algorithms on distributed memory systems. *Proceedings of the International Conference on Parallel and Distributed Computing Systems*, Washington, DC, 13–16 October 1997. IASTED: Calgary, AB, 1997.
17. Yan Y, Jin C, Zhang X. Adaptively scheduling parallel loops in distributed shared-memory systems. *IEEE Transactions on Parallel and Distributed Systems* 1997; **8**(1):70–81.
18. Yang CT, Chang SC. A parallel loop self-scheduling on extremely heterogeneous PC clusters. *Proceedings of the International Conference on Computational Science (ICCS 2003)*, Melbourne, Australia and St. Petersburg, Russia, 2003 (*Lecture Notes in Computer Science*, vol. 2660). Springer: Berlin, 2003; 1079–1088.
19. Xu J, Chronopoulos AT. Distributed self-scheduling for heterogeneous workstation clusters. *Proceedings of the 12th International Conference on Parallel and Distributed Computing Systems*, Fort Lauderdale, FL, 18–20 August 1999. ISCA: Cary, NC, 1999; 211–217.
20. Tzen TH, Ni LM. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems* 1993; **4**(1):87–98.
21. Mandelbrot BB. *Fractal Geometry of Nature*. W. H. Freeman: New York, 1988.
22. Wolfe M. *High Performance Compilers for Parallel Computing*. Addison-Wesley: Reading, MA, 1996.
23. Polychronopoulos CD, Kuck D. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers* 1987; **36**:1425–1439.
24. Freeman TL, Hancock DJ, Bull JM, Ford RW. Feedback guided scheduling of nested loops. *Proceedings of the 5th International Applied Parallel Computing (PARA) Workshop*, Bergen, Norway, 2000 (*Lecture Notes in Computer Science*, vol. 1947). Springer: Berlin, 2001; 149–159.
25. Pachecho P. *Parallel Programming with MPI*. Morgan Kaufmann: San Mateo, CA, 1997.