Multi-Dimensional Dynamic Loop Scheduling Algorithms

Anthony T. Chronopoulos $^{\#1}$, Lionel M. Ni ^{*2}, Satish Penmatsa $^{\#3}$

$^{\#}$*Department of Computer Science, University of Texas at San Antonio*
One UTSA Circle, San Antonio, TX 78249, USA
^{1}atc@cs.utsa.edu
^{3}spenmats@cs.utsa.edu

**Department of Computer Science and Engineering*
The Hong Kong University of Science and Technology
Clear Water Bay, Kowloon, Hong Kong
^{2}ni@cse.ust.hk

***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. Loop scheduling schemes for parallel computers and computer clusters have been proposed in the past. All these schemes are one-dimensional because they partition only the outermost loop of a nested loop construct. In this work, we consider scheduling nested loops with many dimensions. We propose a new methodology which partitions many levels (or dimensions) of nested loops. These new schemes show superior performance over the existing schemes. We implement our new schemes on a network of computers and make performance comparisons with other existing schemes. We expect the new schemes to be particularly useful for multi-core systems because of the fine granularity of the generated tasks.**

I. 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. Such parallel loops are often called DOALL loops. The loops that have interdependencies are often called DOACROSS loops. Loop scheduling schemes for parallel and distributed systems have been proposed and studied in the past. For example, See ([1], [2], [3], [4], [5], [6], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22]) and references therein.

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 were devised in the past; for example: 1) *Tree Scheduling* and 2) *Weighted Factoring* ([7], [3]). Distributed loop scheduling schemes, which can be applied to DOALL and DOACROSS loops and take into account the available computing powers of the computers, have also been devised ([4], [23], [24], [13]).

This work was supported in part by the National Science Foundation under grant CCR-0312323. A.T.Chronopoulos is an IEEE Senior Member, L.M.Ni is an IEEE Fellow and S.Penmatsa is an IEEE Student Member.

In this article, we review some well known scheduling schemes for DOALL loops. The *'simple'* versions of these schemes are the versions suitable for homogeneous systems with single-user-job (dedicated) execution mode. The 'distributed' versions are suitable for heterogeneous systems. A key issue in achieving high (delivered) performance in concurrent processing lies in scheduling nested program loops to execute as efficiently as possible. All the dynamic scheduling schemes (previously proposed) partition only the outermost loop of a program loop structure and assign tasks (chunks) to the processors. This is not efficient for multi-dimensional nested loops. All the previous 'multi-dimensional' loop scheduling schemes for nested loops (e.g. [25]) are static. Thus, these methods are inefficient when the loop tasks sizes are unequal. This calls for devising new 'multi-dimensional' dynamic loop scheduling methods. To our knowledge this has not been attempted before.

Here, we propose new dynamic loop scheduling schemes for computing nested DOALL loops on parallel and distributed systems. We implemented the new schemes (in C++ and MPI) on a network of computers. We show that the new schemes are superior to the previous schemes by simulation results on nested loops with irregular iterations task sizes.

The following are common notations used throughout the entire paper:

- PE is a processor in the parallel or heterogeneous system.
- I is the total number of iterations of a parallel loop.
- p is the number of worker PEs in the parallel or heterogeneous system which execute the computational tasks.
- P_1, P_2,..., P_p represent the p worker PEs in the system.
- N is the number of scheduling steps (= the total number of chunks).
- A few consecutive iterations are called a *chunk*. C_i is the i-th chunk-size (where: $i = 1, 2, \ldots, N$). The i-th chunk is assigned to the k_i (where: $k_i \in \{1, 2, \ldots, p\}$) worker PE making the i-th request.
- R_i is the remaining number of tasks after scheduling the i-th chunk.

- $L \geq 1$ is a 'threshold' chunk-size chosen by the user or the system. If the chunk-size in a scheduling scheme drops below this threshold (i.e. $C_i < L$) then it is set equal to it (i.e. $C_i = L$). If no threshold is set then the chunk-size can become 1 for several scheduling steps.
- t_j, $j = 1, .., p$, is the execution time of P_j to finish all the tasks assigned to it by the scheduling scheme.
- $T_p = \max_{j=1,..,p}(t_j)$, is the parallel execution time of the loop on p worker PEs.

In Section II, we present some examples of nested (DOALL) loops. In Section III, we present simple loop scheduling schemes for DOALL loops. In Section IV, we present distributed loop scheduling schemes. In Section V, we present new multi-dimensional loop scheduling schemes. In Section VI, we present our implementation and simulation results. In Section VII, we discuss conclusions and future work.

II. Nested Loop Examples

In this section, we give examples of nested *parallel* loops or *DOALL* loops. We consider the following nested loop model, which we call m-dimensional (mD) loop.

```
Do j1 = $J_1^1$, $J_1^2$
   Do j2 = $J_2^1$, $J_2^2$
     ..
       Do jm = $J_m^1$, $J_m^2$
         program statements
       ENDO
     ..
   ENDO
ENDO
```

where J_n^1, J_n^2 are the lower and upper loop bound for the n-th ($n = 1, \ldots, m$) loop. For simplicity we assume that the loop is one-way nested (i.e. each loop contains only one immediately inner loop). We assume that the upper and lower bounds are constant. We use the following notation: $I^n = J_n^2 - J_n^1$ (for $n = 1, \ldots, m$) to denote the number of loop iterations in each loop. For the 1D loops we drop the subscript in I^n and we simply write I for the number of loop iterations.

We next present examples of DOALL loops, with $m = 2$ (i.e. 2-dimensional). $L(i)$ represents the execution time for iteration i (see also [26]). A parallel loop is *uniformly distributed* if the execution times of all the 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
  DOALL L = 1 TO J2
        X[K,L] = X[K,L] + A
  END DOALL
END DOALL
```

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

```
/* increasing */
DOALL K = 1 TO I
  DOALL L = 1 TO J2
      Serial DO J = 1 TO K
       Serial Loop Body
      End Serial DO
  END DOALL
END DOALL


/* decreasing */
DOALL K = 1 TO I
  DOALL L = 1 TO J2
      Serial DO J = 1 TO I-K+1
       Serial Loop Body
      End Serial DO
  END DOALL
END DOALL
```

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

```
DOALL K = 1 TO I
  DOALL L = 1 TO J2
      IF(Expression1) THEN
                Block1
      ELSE
                Block2
      ENDIF
  END DOALL
END DOALL
```

Following is an example of an *irregular* loop style representing the loop distribution required by the Mandelbrot set computation [27].

ALGORITHM MSetLSM(MSet,nx,ny,xmin,xmax,ymin,ymax,maxiter)

```
BEGIN
 FOR iy = 0 TO ny-1 DO
  cy = ymin+iy*(ymax - ymin)/(ny - 1)
  FOR ix = 0 TO nx-1 DO
     cx = xmin+ix*(xmax - xmin)/(nx - 1)
     MSet[ix][iy]=MSetLevel(cx,cy,maxiter)
  END FOR
 END FOR
END
```

ALGORITHM MSetLevel(cx,cy,maxiter)

```
BEGIN
 x = y = x2 = y2 = 0.0, iter = 0
 WHILE(iter < maxiter)AND(x2 + y2 < 2.0)DO
   temp = x2 - y2 + cx
   y = 2*x*y + cy
   x = temp
   x2 = x*x
   y2 = y*y
   iter = iter + 1
 END WHILE
 RETURN(iter)
END
```

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. This class of loops is the most severe test for a scheduling scheme.

We use, in our tests, the Mandelbrot fractal computation algorithm [27] on the domain [-2.0, 2.0] $\times$ [-2.0, 2.0], for different window sizes (for example 4000 $\times$ 4000, 8000 $\times$ 8000, and so on). The algorithm uses unpredictable irregular loops. 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.

III. Simple DOALL Loop Scheduling Schemes

Dynamic or self-scheduling is an automatic loop scheduling method in which idle PEs request new loop iterations to be assigned to them. We will also study these methods from the perspective of both parallel and distributed systems. For this, we use the Master-Worker architecture model (Fig. 1). Idle worker 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 possible worker 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.

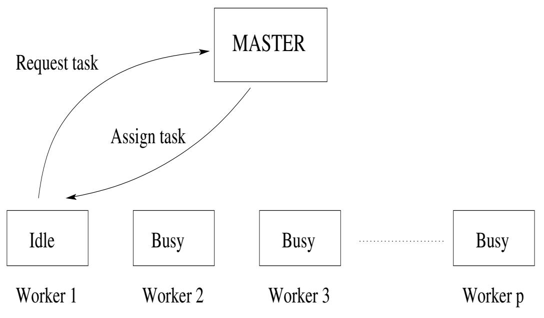

Fig. 1. Self-Scheduling schemes: the Master-Worker model

In a generic self-scheduling scheme, at the i-th scheduling step, the master computes the chunk-size C_i, a starting (iteration) index $istart$, and the remaining number of tasks R_i as follows.

Initially, we have $R_0 = I$, $istart = J_1^1$. The Master computes the chunk-size for the i-th scheduling step:

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

where $f(.,.)$ is a function possibly of more inputs than just R_{i-1} and p. Then the master assigns to a worker PE C_i tasks and a starting (iteration) index $istart$. Then the $istart$ and R_i for the next scheduling step are updated:

$$istart = istart + C_i, \qquad R_i = R_{i-1} - C_i. \qquad (2)$$

When the user or the system has chosen a 'threshold' last chunk-size $L > 1$, then the computation of C_i must be modified by adding: If $(C_i < L)$ then $C_i = L$.

Algorithm (Simple Scheme):

Master:

- Receive a new request from a worker for tasks.
- If($R_i > 0$) then
  - Compute C_i, $istart$ and R_i from eqs. (1) - (2) above.
  - Send a new task (starting index $istart$ and size C_i) to the worker.

  Else
  - Send 'terminate' signal to (requesting) workers.

Worker:

- Send a request to Master.
- Receive new tasks or 'terminate' signal.
- Perform tasks or terminate.

The different ways to compute C_i has given rise to different scheduling schemes. The most notable examples are the following (e.g. See ([3], [4], [14], [18], [24]) and references therein).

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 (by default) $F = \left\lfloor \frac{I}{2p} \right\rfloor$, and $L = 1$. The number of scheduling steps assigned: $N = \left\lceil \frac{2*I}{(F+L)} \right\rceil$. Note that (i) The chunks sizes are given by the formula $C_N = F - (N-1)D$ and $C_N \geq 1$ due to integer divisions; (ii) The number of scheduling steps are: $2p \leq N \leq 4p$.

Chunk Self-Scheduling (CSS): $C_i = l$, where $l \geq 1$ (known as *chunk size* is chosen by the user). For $l = 1$, CSS is the so-called (pure) Self-Scheduling (SS) and for $l = I/p$, CSS is the so-called Fixed-size (FS) scheme. There is an increased chance of load imbalance due to difficulty to predict an optimal l. It is static. It has reduced communication/scheduling overheads. Note that $N = I/l$.

Guided Self-Scheduling (GSS): $C_i = \lceil R_{i-1}/p \rceil$. This is a dynamic scheme with a non-linear chunk-size function. It assigns large chunks initially, which implies reduced communication/scheduling overheads in the first scheduling steps. A modified version $GSS(l)$ with minimum assigned chunk-size l attempts to improve on the weaknesses of GSS. Note that the number of scheduling steps satisfies $p \leq N \leq pH_n$, where $H_n = ln(n) + \gamma/(2n)$, $\gamma = 0.5772157$ and $n = \lceil I/p \rceil$.

TABLE I
SAMPLE CHUNK SIZES FOR $I = 1000$ AND $p = 4$

Scheme	Chunk size
FS	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
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
TSS	125 117 109 101 93 85 77 69 61 53 45 37 28

Factoring Self-Scheduling (FSS)**:** FSS consists of rounds of p scheduling steps. In each round i_r the master distributes $\lceil R_{i_r-1}/2\rceil$ iterations to the p workers. Thus, $C_{p*i_r+n} = \lceil R_{i_r-1}/2p\rceil$, for $n = 1,\ldots,p$ and the remaining iterations are $R_{i_r} = R_{i_r-1}/2$. We note that the number of scheduling steps satisfies $p \leq N \leq 1.44p\ ln(I/p)$.

Since each scheduling step involves master-worker communication, the number of scheduling steps plays an important role in the overall communication cost. The load imbalance depends on the execution time difference between t_j, for $j = 1,\ldots,p$. This difference may be large if the first chunk is too large. There is a detailed theoretical analysis of these schemes in [8].

Example 1: We show here 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 $p = 4$. For CSS, k represents the fixed chunk size.

IV. Distributed Dynamic Loop Scheduling Schemes

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

A. Tree Scheduling (TREES)

TREES ([5], [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. Fig. 2 shows the communication topology created by TREES for a cluster of 4 processors. Note that P_0 is needed for the initial task allocation and the final I/O.

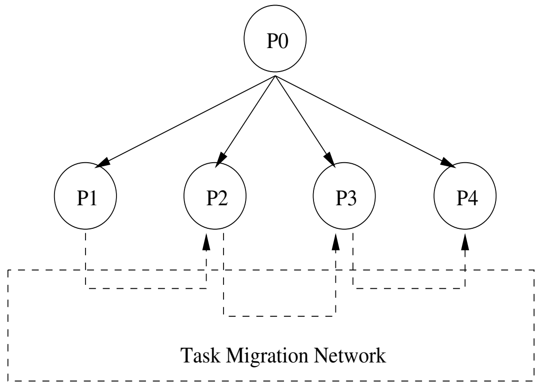

Fig. 2. The Tree topology

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. The main success of TREES is the distributed communication, which leads to good scalability. 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. For example, if 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 processor or that more work will be done by a slow processor, producing a large finish time for the problem.

B. Distributed Trapezoid Self-Scheduling (DTSS)

Distributed versions of the self-scheduling schemes have been derived and studied in ([4], [28], [24]) for (non)dedicated heterogeneous distributed environments (i.e. the application shares the system with other users jobs running at the same time). Here for simplicity we only consider the dedicated (to a single user job) system case. Also, we consider only the Trapezoid scheme. Our study extends easily to the other distributed self-scheduling schemes. In the dedicated environment case, DTSS takes into account the processing speeds of the worker PEs in assigning tasks to them. The virtual power V_j ($j = 1,\ldots,p$) of a worker PE P_j ($j = 1,\ldots,p$) is computed by the master PE as: $V_j = Speed(P_j)/min_{1\leq j\leq p}\{Speed(P_j)\}$, where Speed($P_j$) is the CPU-Speed of P_j. Thus, the total virtual computing power of the system, V, is given by: $V = \sum_{j=1}^{p} V_j$. In our algorithms implementation we only consider virtual powers which are integers. Thus, $V_j = 1, 2, \ldots$. We note that for homogeneous systems : $V_j = 1$ ($j = 1,\ldots,p$).

The chunk sizes (C_i) (for $i = 1,\ldots$) are calculated using a chunk decrement. The decrement, D, is given by $D =$

$\left\lfloor \frac{(F-L)}{(N-1)} \right\rfloor$, where F is the first chunk given by $\left\lfloor \frac{I}{2V} \right\rfloor$, L is the last chunk which can be set to a *threshold* and N is a value given by $\left\lceil \frac{2*I}{(F+L)} \right\rceil$.

We note that, in DTSS, the computation of chunk-size C_i is derived as follows. Let V_{k_i} be the virtual power of the PE (P_{k_i}), which will be assigned the chunk of size C_i. We only consider integer virtual powers. We can think of P_{k_i} as a set of V_{k_i} virtual homogeneous PEs. Thus, we apply V_{k_i} consecutive TSS steps to compute the chunk size C_i as follows: Compute (intermediate chunk-sizes) $C_{i,1} = C_{i-1} - D$, $C_{i,2} = C_{i,1} - D, \ldots, C_{i,V_{k_i}} = C_{i,V_{k_i-1}} - D$; Summing up these chunk-sizes we get the chunk size $C_i = C_{i,1} + \ldots + C_{i,V_{k_i}}$ for the i-th scheduling step (in DTSS). Thus, we obtain for $i = 1$:

$$C_1 = V_{k_1} * F - V_{k_1} * (V_{k_1} - 1)/2 * D \qquad (3)$$

and for $i = 2, \ldots$:

$$C_i = V_{k_i} * C_{i-1} - V_{k_i} * (V_{k_i} - 1)/2 * D \qquad (4)$$

We next use this formula in the DTSS algorithm.

Algorithm (DTSS Scheme):

Master:

```
1) (a) Receive $Speed(P_j)$ of PEs $P_j$, $j = 1, \ldots, p$.
   (b) Compute all $V_j$.
   (c) Send $V_j$ to PEs $P_j$, $j = 1, \ldots, p$.
2) Calculate $F$ and $D$.
3) (a) While there are unassigned iterations (i.e. $R_i > 0$),
   if a request arrives, put it in the queue.
   (b) Compute the chunk-size, remaining tasks and
   starting index (for $i = 1, \ldots$) by using eqs. (2), (3),
   and (4).
   (c) Pick a request from the queue and assign the next
   chunk to a worker.
```

Worker:

```
1) (a) Send $Speed(P_j)$ to Master.
   (b) Receive $V_j$ from Master.
2) Send a request.
3) Wait for a reply.
   • If more tasks arrive, compute the new tasks, go to
     step 2.
   • Else Terminate.
```

V. Multi-Dimensional Loop Scheduling Schemes

We consider multi-dimensional (one-way) nested parallel loop constructs defined in Section II. We consider the self-scheduling schemes presented above. In this section we will derive multi-dimensional versions for these schemes. We will focus our attention on TSS and DTSS. The derivation of other multi-dimensional schemes is analogous.

At first, we consider the case of the simple schemes (e.g. TSS). We use the index n $(= 1, 2, \ldots, m)$ to denote the dimension of the loop. Here we restrict our attention (without loss of generality) to the study of 2-dimensional (2D) schemes (i.e. $m = 2$). We will derive the 2-dimensional schemes by applying the simple scheme in each dimension of the loop. We denote by i_n $(= 1, \ldots, N^n)$ the index of the scheduling step and by N^n the number of scheduling steps in dimension n. We denote by $C^n_{i_n}$, $R^n_{i_n}$, the chunk-sizes, the number of remaining iterations in each dimension. The chunks of the 2D scheme will be rectangular with sizes $C^1_{i_1}$x$C^2_{i_2}$ at scheduling step (i_1, i_2). We define $(istart1, istart2)$ as the origin of the rectangular chunk at scheduling step (i_1, i_2).

We assume that the number of iterations in each dimension are much greater than the number of workers: i.e. $I^1 >> p$ and $I^2 >> p$. In order to derive the 2D scheme, we will apply the simple (1D) scheme with p PEs in each loop and construct 2D chunks which are the Cartesian products of the 1D chunks. If we do not have $I^1 >> p$ and $I^2 >> p$ then we could consider a decomposition of $p = p1 \times p2$ and use two 1D schemes (one with $p1$ PEs and another with $p2$ PEs) to construct the 2D chunks. Our 2D algorithm derivation also works for this case.

The computation of the chunk-size and starting index point (using eqs. (1) - (2)) is performed in 2 dimensions (i.e. loops: J_1 and J_2) to compute $C^1_{i_1}$ and $C^2_{i_2}$ (for i_1, $i_2 = 1, 2, \ldots$). Also, we initialize $istart1 = J^1_1$ and $istart2 = J^1_2$. The chunks are rectangular regions of the index space (i_1, i_2) of $N^1 \times N^2$ scheduling steps. Thus, each chunk is a rectangle which has as its origin the index point $(istart1, istart2)$ and a width and a height: $C^1_{i_1}$ and $C^2_{i_2}$, respectively. We will often refer to the 2D (or rectangular) chunk by its width and a height i.e. $C^1_{i_1} \times C^2_{i_2}$.

We compute the rectangular chunks in order along 'wave-front diagonals' (shown in example in Fig. 3, below). These diagonals of rectangles in the index space start from the bottom left (index point (J^1_1, J^1_2)) and end up at the top right of the region (index point (J^2_1, J^2_2)). We note that we need to compute the new chunk-size $C^1_{i_1}$ or $C^2_{i_2}$ by eq. (1) only the first time when needed in a rectangular chunk. After $C^n_{i_1}$ or $C^n_{i_2}$ have been computed, they can be stored (by the Master) and they can be used to assign to workers the upcoming rectangular chunks along the wavefront diagonals (see Fig. 3, below).

The starting index point is updated as follows: (1) $istart1 = istart1 + C^1_{i_1}$ or $istart1 = istart1 - C^1_{i_1}$ and (2) $istart2 = istart2 + C^2_{i_2}$ or $istart2 = istart2 - C^2_{i_2}$, because the rectangular chunks are computed along the wavefront diagonals. We denote by F^n and D^n $(n = 1, 2)$ the first chunk and decrement of TSS in dimension n.

Taking these considerations into account we formulate the 2-dimensional Simple Scheme as follows.

Algorithm (2-dimensional Simple Scheme e.g. TSS-2D):

Master:

```
• Receive a new request from a worker for tasks.
• If (there exist still unassigned rectangular chunks) then
    – Compute a new task (2D chunk $C^1_{i_1} \times C^2_{i_2}$ and
      $(istart1, istart2)$) along the wavefront diagonals
      (see Fig. 3).
    – Send the new task to the worker.
```

Else
- Send a ‘terminate’ signal to (requesting) workers.

Worker:
- Send a request to the Master.
- Receive new tasks or a ‘terminate’ signal.
- Perform tasks or terminate.

Table II shows the chunk-sizes computed by TSS-2D discussed above. We show the different chunk-sizes (width/height for the rectangular chunk separated by commas) for a problem with $I^1 = I^2 = 1000$ iterations and $p = 4$. Fig. 3 shows the iterations (C^1_{i1}xC^2_{i2}) that will be assigned during each scheduling step ($i1, i2 = 1, 2, \dots$) to the worker PEs using TSS-2D. It also shows the order of assignment of the chunks to the requesting worker PEs.

We now show how to derive the 2-dimensional DTSS. In the standard (i.e. 1D) DTSS algorithm described above, we derived a formula for computing chunk of size C_i. We used V_{k_i}, the virtual power of the worker (P_{k_i}) scheduled (by the Master) at the i-th step. In the 2D DTSS, we must also use the virtual power of the worker to compute the rectangular chunks which will be assigned by the master. This computation of the rectangular chunks is similar to the 2-dimensional TSS (e.g. shown in Table II). However, the chunk sizes are computed using $F^n = \lfloor \frac{I}{2V} \rfloor$ instead of $F^n = \left\lfloor \frac{I}{2p} \right\rfloor$ in TSS. Since $V > p$ for heterogeneous systems, we can create a table for the 2D DTSS which is analogous to Table II. Such a table for 2D DTSS would contain more rectangular chunks than the TSS-2D (Table II). Then we assign a number of rectangular chunks along a wavefront diagonal (see Fig. 3) equal to the virtual power of the worker. Next, we present the 2-dimensional DTSS algorithm.

Algorithm (DTSS-2D Scheme):

Master:

```
1) (a) Receive $Speed(P_j)$ of PEs $P_j$, $j = 1, \dots, p$.
   (b) Compute all $V_j$.
   (c) Send $V_j$ to PEs $P_j$, $j = 1, \dots, p$.
2) Calculate $F^n$ and $D^n$.
3) (a) While there are unassigned iterations (rectangular
   chunks), if a request arrives, put it in the queue.
   (b) Compute the rectangular chunks as in TSS-2D and
   compute $(istart1, istart2)$.
   (c) Pick a request from the queue of a worker with
   virtual power $V_j$ and assign the next $V_j$ rectangular
   chunks along the same or adjacent wavefront diagonals.
```

Worker:

```
1) (a) Send $Speed(P_j)$ to the Master.
   (b) Receive $V_j$ from the Master.
2) Send a request.
3) Wait for a reply.
   • If more tasks arrive, compute the new tasks, go to
     step 2.
   • Else Terminate.
```

TABLE II
SAMPLE CHUNK SIZES FOR $I^1 = I^2 = 1000$ AND $p = 4$

Scheme	Chunk size (width/height)
$TSS-2D$	125/125,117/125,125/117,109/125,117/117,125/109, 101/125,109/117,117/109,125/101,93/125,101/117, 109/109,117/101,125/93,85/125,93/117,101/109, 109/101,117/93,125/85,77/125,85/117,93/109, 101/101,109/93,117/85,125/77,69/125,77/117,85/109, 93/101,101/93,109/85,117/77,125/69,61/125,69/117, 77/109,85/101,93/93,101/85,109/77,117/69,125/61, 53/125,61/117,69/109,77/101,85/93,93/85,101/77, 109/69,117/61,125/53,45/125,53/117,61/109,69/101, 77/93,85/85,93/77,101/69,109/61,117/53,125/45, 37/125,45/117,53/109,61/101,69/93,77/85,85/77, 93/69,101/61,109/53,117/45,125/37,28/125,37/117, 45/109,53/101,61/93,69/85,77/77,85/69,93/61,101/53, 109/45,117/37,125/28,117/28,109/37,101/45,93/53, 85/61,77/69,69/77,61/85,53/93,45/101,37/109,28/117, 109/28,101/37,93/45,85/53,77/61,69/69,61/77,53/85, 45/93,37/101,28/109,101/28,93/37,85/45,77/53,69/61, 61/69,53/77,45/85,37/93,28/101,93/28,85/37,77/45, 69/53,61/61,53/69,45/77,37/85,28/93,85/28,77/37, 69/45,61/53,53/61,45/69,37/77,28/85,77/28,69/37, 61/45,53/53,45/61,37/69,28/77,69/28,61/37,53/45, 45/53,37/61,28/69,61/28,53/37,45/45,37/53,28/61, 53/28,45/37,37/45,28/53,45/28,37/37,28/45,37/28, 28/37,28/28.

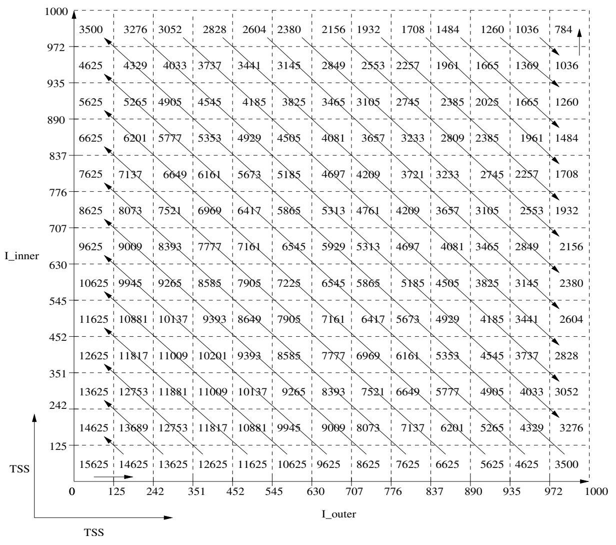

Fig. 3. Chunk allocation with TSS-2D

VI. Implementation and Test Results

A. Implementation

The scheduling schemes are implemented using the distributed programming framework offered by MPI [29] on a cluster of Sun workstations. The workstations have a CPU Speed of 502 MHz. The test problem used is the Mandelbrot computation [27] for a matrix size (problem size) ranging from 4000 $\times$ 4000 to 16000 $\times$ 16000. The Mandelbrot computation is a doubly nested loop without any dependencies. The computation of one column of the Mandelbrot matrix is considered the smallest schedulable unit.

We performed experiments with the number of workers (p)

ranging from 1 to 24. The size of the test problem is such that it does not cause any memory swaps. So, the virtual powers (V_i, $i = 1, \ldots, p$) depend only on the processor speeds. To create a heterogeneous environment, we put an artificial load (one continuously running matrix multiplication process) in the background on $p/2$ workers. The workers with one load in the background are assumed to have V_i = 1 and the workers without any load are assumed to have V_i = 2. Thus, we have $p/2$ fast and $p/2$ slow workers. This was verified by timing any program running on a single machine of the above two types. The simulations are done when no other user jobs existed on the workstations.

B. Results

In this section, we present the results of the experiments performed. *Tp* denotes the total execution time measured on the master PE. All timings are in seconds (sec). We denote TSS and DTSS (one-dimensional schemes) as TSS-1D and DTSS-1D respectively.

Table III shows the *Tp* of various schemes with 8 workers for a problem size of 4000 × 4000. The table also shows the computation times (T_{comp}) and communication times (T_{comm}) of each worker. It can be observed that the 2-dimensional schemes (2D) show substantial performance improvement over the one-dimensional (1D) schemes. Also, DTSS-1D and DTSS-2D, which take into account the processor speeds, show better performance than TSS-1D and TSS-2D respectively. It can also be observed that the worker computation times in the case of 2D schemes are very well load balanced compared to the 1D schemes. The communication times of the workers in all the schemes are very small. This low value in communication time is because the workers does not send the computed results back to the master. They only send a request for work and receive a reply. These messages are small in size.

Fig. 4 shows the time *Tp* of all the schemes with increasing problem size. The number of worker PEs are fixed to 16. It can be observed that for all the problem sizes, TSS-2D and DTSS-2D shows superior performance compared to the other schemes. The 2D algorithms provide better load balancing and thus the overall performance is enhanced.

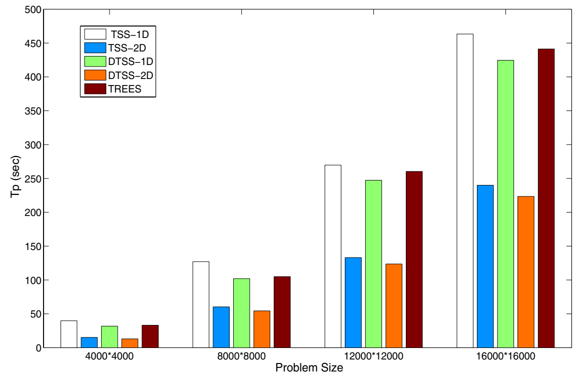

Fig. 4. Total execution time of various schemes for various problem sizes

TABLE III
COMPUTATION, COMMUNICATION AND TOTAL EXECUTION TIMES OF VARIOUS SCHEMES FOR 8 WORKERS AND A PROBLEM SIZE OF 4000 × 4000 (PE_i: T_{comp}/T_{comm} (SEC))

PE	$TSS-1D$	$TSS-2D$
1	35.6/0.01	27.3/0.03
2	26.6/0.01	27.3/0.05
3	21.7/0.01	27.3/0.07
4	14.5/0.02	27.3/0.07
5	24.7/0.16	27.1/0.38
6	25.2/0.01	29.1/0.32
7	51.1/0.16	26.9/0.59
8	58.8/0.13	29.6/0.83
T_p	58.9	30.4

PE	$DTSS-1D$	$DTSS-2D$	$TREES$
1	45.2/0.01	24.0/0.05	40.9/0.01
2	33.2/0.01	24.0/0.06	45.9/0.01
3	18.6/0.01	24.0/0.05	39.4/0.01
4	17.7/0.02	24.0/0.04	33.4/0.02
5	19.1/0.01	23.5/0.48	37.2/0.01
6	33.0/0.01	23.6/0.37	45.9/0.01
7	36.0/0.17	23.4/0.60	45.4/0.01
8	39.1/0.01	23.3/0.66	43.6/0.01
T_p	45.2	24.0	46.3

In Fig. 5, we present the speedup of all the schemes with increasing number of worker PEs for a problem size of 4000 × 4000. We computed the speedup (S_p) according to the equation:

$$S_p = min\{T_{P_1}, T_{P_2}, \ldots, T_{P_p}\}/T_p$$

where T_{P_i} is the execution time on one PE and T_p is the execution time on p PEs.

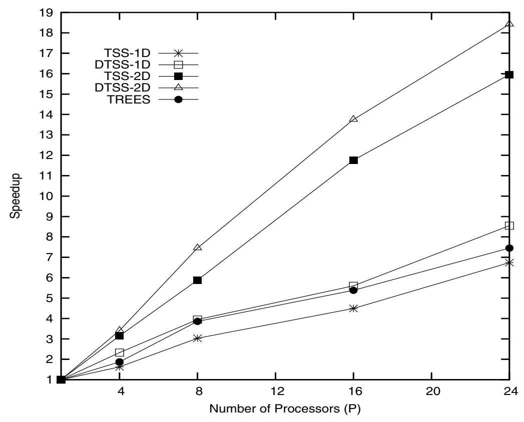

Fig. 5. Speedup of various schemes with number of processors

It can be observed that, as the number of worker PEs increases, the speedup of all the schemes improves which shows that the schemes are scalable. However, the speedup of TSS-2D and DTSS-2D is substantially high for large number of PEs compared to TSS-1D and DTSS-1D respectively. Also, DTSS-2D shows better speedup than TSS-2D.

Observations: (i) The distributed schemes (e.g. DTSS) take into consideration the virtual computing power (VCP) of the (worker) processors. One could use the raw CPU speeds as virtual powers. However, it is more realistic to run some benchmark problems and obtain the actual (or delivered) VCP. In particular, VCP takes into account other processor characteristics (e.g. cache, communication network etc).

(ii) Although, our experiments were performed in a distributed environment of workers with two different VCPs, we expect that similar performance will be observed for workers with significantly varying VCPs.

(iii) In both TSS-2D and DTSS-2D, the workers are well-balanced based on their computation times. However, the difference in performance between TSS-2D and DTSS-2D is not very big. This difference possibly depends on the problem (irregular loop) and also on the the VCPs of the workers.

VII. Conclusions and future work

We derived new multi-dimensional loop scheduling schemes for computing nested DOALL loops on distributed systems. We implemented the new schemes (in C++ and MPI) on a network of computers. We showed that the new schemes are superior to the previous schemes by simulation results. The nested loop that we tested has irregular iterations task sizes. We expect the new schemes to be particularly useful for multi-core systems because of the fine granularity of the generated tasks.

In the past we have derived scalable loop scheduling schemes by considering a hierarchical Master-Worker model. We plan to study the extension of the hierarchical Master-Worker (1D) schemes to multi-dimensional schemes. We also plan to test our schemes in distributed environments with several groups of workers with varying speeds. We plan to analyze theoretically the properties of the new schemes. We also plan to derive new multi-dimensional versions of other existing schemes, study the new schemes for nested loops with dependences, and to implement and test the new schemes on multi-core systems.

Acknowledgements

Some of the reviewers comments which helped improve the quality of the paper are gratefully acknowledged.

References

[1] Y. W. Fann, C. T. Yang, S. S. Tseng, and C. J. Tsai, "An intelligent parallel loop scheduling for parallelizing compilers," *Journal of Information Science and Engineering*, vol. 16(2), pp. 169–200, 2000.

[2] J. M. Bull, "Feedback guided dynamic loop scheduling: Algorithms and experiments," in *Proc. of 4th Intl Euro - Par Conference*, Southampton, UK, 1998, pp. 337–382.

[3] I. Banicescu, V. Velusamy, and J. Devaprasad, "On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring," *Cluster Computing*, vol. 6, pp. 215–226, 2003.

[4] A. T. Chronopoulos, R. Andonie, M. Benche, and D. Grosu, "A class of distributed self-scheduling schemes for heterogeneous clusters," in *Proc. of the 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001)*, Newport Beach, California, Oct 2001.

[5] S. P. Dandamudi and T. K. Thyagaraj, "A hierarchical processor scheduling policy for distributed-memory multicomputer systems," in *Proc. of the 4th International Conference on High-Performance Computing*, 1997, pp. 218–223.

[6] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris, "Compiling tiled iteration spaces for clusters," in *Proc. of IEEE International Conference on Cluster Computing*, Chicago, Illinois, 2002, pp. 360–369.

[7] T. H. Kim and J. M. Purtilo, "Load balancing for parallel loops in workstation clusters," in *Proc. of Intl. Conference on Parallel Processing*, Bloomingdale, IL, Vol. 3 1996, pp. 182–190.

[8] E. P. Markatos and T. J. LeBlanc, "Using processor affinity in loop scheduling on shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed systems*, vol. 5(4), pp. 379–400, April 1994.

[9] T. Philip and C. R. Das, "Evaluation of loop scheduling algorithms on distributed memory systems," in *Proc. of Intl Conf. on Parallel and Distributed Computing Systems*, 1997.

[10] C. T. Yang and S. C. Chang, "A parallel loop self-scheduling on extremely heterogeneous pc clusters," in *Proc. of Intl Conf. on Computational Science*, Melbourne, Australia and St. Petersburg, Russia, 2003, pp. 1079–1088.

[11] I. Banicescu, R. Carino, J. P. Pabico, and M. Balasubramaniam, "Design and implementation of a novel dynamic load balancing library for cluster computing," *Parallel Computing*, vol. 31, no. 7, July 2005.

[12] S. Chen and J. Xue, "Partitioning and scheduling loops on nows," *Computer Communications*, vol. 22, no. 11, July 1999.

[13] F. Ciorba, T. Andronikos, I. Riakiotakis, A. Chronopoulos, and G. Papakonstantinou, "Dynamic multi phase scheduling for heterogeneous clusters," in *IEEE International Parallel and Distributed Processing Symposium*, Rhodes Island, Greece, April 2006.

[14] A. Kejariwal, A. Nicolau, and C. Polychronopoulos, "History-aware self-scheduling," in *International Conference on Parallel Processing*, Columbus OH, Aug 2006.

[15] M. E. Wolfe, *Optimizing Compilers for Supercomputers*. Redwood City, CA: Addison-Wesley Publication Co., 1996.

[16] C.-T. Yang, K.-W. Cheng, and K.-C. Li, "An enhanced parallel loop self-scheduling scheme for cluster environments," in *19th International Conference on Advanced Information Networking and Applications*, Vol 2, March 2005, pp. 207–210.

[17] T. Hagerup, "Allocating independent tasks to parallel processors: An experimental study," *Journal of Parallel and Distributed Computing*, vol. 47, pp. 185–197, 1997.

[18] A. R. Hurson, J. T. Lim, K. K. Kavi, and B. Lee, "Parallelization of doall and doacross loops: A survey," *Advances in computers*, vol. 45, pp. 53–103, 1997.

[19] E. M. Rosenvinge, A. C. Elster, and C. Banino, "Online task scheduling on heterogeneous clusters: An experimental study," in *PARA*, 2004, pp. 1141–1150.

[20] H. Bast, "On scheduling parallel tasks at twilight," *Theory Comput. Systems*, vol. 33, pp. 489–563, 2000.

[21] D. J. Lilja, "Exploiting the parallelism available in loops," *IEEE Computer*, vol. 27, no. 2, pp. 13–26, Feb 1994.

[22] C. Xu, "Effects of parallelism degree on run-time parallelization of loops," in *Proc. of the 31st Hawaii Intl. Conference on System Sciences*, Volume 7, Jan 6-9 1998, pp. 86–95.

[23] A.T.Chronopoulos, S. Penmatsa, and N. Yu, "Scalable loop self-scheduling schemes for heterogeneous clusters," in *IEEE Intl. Conference on Cluster Computing*, Chicago, Illinois, Sept 2002, pp. 353–359.

[24] A. T. Chronopoulos, S. Penmatsa, J. Xu, and S. Ali, "Distributed loop-scheduling schemes for heterogeneous computer systems," *Concurrency and Computation: Practice and Experience*, vol. 18, no. 7, pp. 771–785, 2006.

[25] C. D. Polychronopoulos, D. J. Kuck, and D. A. Padua, "Utilizing multidimensional loop parallelism on large scale parallel processor systems," *IEEE Trans. on computers*, vol. 38, no. 9, pp. 1285–1296, Sept 1989.

[26] T. H. Tzen and L. M. Ni, "Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers," *IEEE Transactions on Parallel and Distributed systems*, vol. 4(1), pp. 87–98, Jan 1993.

[27] M. F. Bransley, R. L. Devaney, B. B. Mandelbrot, H. O. Peitgen, D. Saupe, R. F. Voss, Y. Fisher, and M. McGuire, *The Science of Fractal Images*. NY: Springer-Verlag, 1988.

[28] A. T. Chronopoulos, S. Penmatsa, N. Yu, and D. Yu, "Scalable loop self-scheduling schemes for heterogeneous clusters," *Intl. Jrnl. of Computational Science and Engineering*, vol. 1, no. 2/3/4, pp. 110–117, 2005.

[29] P. Pachecho, *Parallel Programming with MPI*. Morgan Kauffman, 1997.