Dynamic Multi Phase Scheduling for Heterogeneous Clusters

Florina M. Ciorba¹, Theodore Andronikos¹, Ioannis Riakiotakis¹, Anthony T. Chronopoulos² and George Papakonstantinou¹

¹Computing Systems Laboratory
Dept. of Electrical and Computer Engineering
National Technical University of Athens
Zografou Campus, 15773, Athens, Greece
{cflorina,tedandro,iriak}@cslab.ece.ntua.gr

²Dept. of Computer Science
University of Texas at San Antonio
6900 N. Loop 1604 West, San Antonio, TX 78249
atc@cs.utsa.edu

Abstract

Distributed computing systems are a viable and less expensive alternative to parallel computers. However, concurrent programming methods in distributed systems have not been studied as extensively as for parallel computers. Some of the main research issues are how to deal with scheduling and load balancing of such a system, which may consist of heterogeneous computers. In the past, a variety of dynamic scheduling schemes suitable for parallel loops (with independent iterations) on heterogeneous computer clusters have been obtained and studied. However, no study of dynamic schemes for loops with iteration dependencies has been reported so far. In this work we study the problem of scheduling loops with iteration dependencies for heterogeneous (dedicated and non-dedicated) clusters. The presence of iteration dependencies incurs an extra degree of difficulty and makes the development of such schemes quite a challenge. We extend three well known dynamic schemes (CSS, TSS and DTSS) by introducing synchronization points at certain intervals so that processors compute in pipelined fashion. Our scheme is called dynamic multi-phase scheduling (DMPS) and we apply it to loops with iteration dependencies. We implemented our new scheme on a network of heterogeneous computers and studied its performance. Through extensive testing on two real-life applications (the heat equation and the Floyd-Steinberg algorithm), we show that the proposed method is efficient for parallelizing nested loops with dependencies on heterogeneous systems.

1. Introduction

Loops are one of the largest sources of parallelism in scientific programs. The iterations within a loop nest are either independent (called parallel loops) or precedence constrained (called dependence loops). Furthermore, the precedence constraints can be uniform (constant) or non-uniform throughout the execution of the program. A review of important parallel loop scheduling algorithms is presented in [7] (and references therein) and some recent results are presented in [4]. Research results also exist on scheduling parallel loops on message passing parallel systems and on heterogeneous systems [1],[2],[3],[5],[6],[8],[9],[10],[11],[15],[16]. Static scheduling schemes for dependence loops have been studied extensively for shared memory and distributed memory systems [18] (and references therein), [27],[26],[28], [30] [25],[19],[20], [31].

Loops can be scheduled statically at compile-time or dynamically at run-time. Static scheduling is applicable to both parallel and dependence loops. It has the advantage of minimizing the scheduling time overhead, and achieving near optimal loop balancing when the execution environment is homogeneous with uniform and constant workload. Examples of such scheduling are Block [29], Cyclic [18], etc. However, most cluster nowadays are heterogeneous and non-dedicated to specific users, yielding a system with variable workload. When static schemes are applied to heterogeneous systems with variable workload the performance is severely deteriorated. Dynamic scheduling algorithms adapt the assigned number of iterations to match the workload variation of both homogeneous and heterogeneous systems. An important class of dynamic scheduling algorithms are the self-scheduling schemes.
(such as CSS [23], GSS [24], TSS [13], Factoring [1], and others [11]). On distributed systems these schemes are implemented using a Master-Slave model.

Another very important factor in achieving near optimal execution time in distributed systems is load balancing. Distributed systems are characterized by heterogeneity. To offer load balancing loop scheduling schemes must take into account the processing power of each computer in the system. The processing power depends on CPU speed, memory, cache structure and even the program type. Furthermore, the processing power depends on the workload of the computer throughout the execution of the problem. Therefore, load balancing methods adapted to distributed environments take into account the relative powers of the computers. These relative computing powers are used as weights that scale the size of the sub-problem assigned to each processor. This significantly improves the total execution time when a non-dedicated heterogeneous computing environment is used. Such algorithms were presented in [4],[9]. A recent algorithm that improves TSS by taking into account the processing powers of a non-dedicated heterogeneous system is DTSS (Distributed TSS) [6]. All dynamic schemes proposed so far apply only to parallel loops without dependencies.

When loops without dependencies are parallelized with dynamic schemes, the index space is partitioned into chunks, and the master assigns these chunks to processors upon request. Throughout the parallel execution, every slave works independently and upon chunk completion sends the results back to the master. Obviously, this approach is not suitable for dependence loops because, due to dependencies, iterations in one chunk depend on iterations in other chunks. Hence, slaves need to communicate. Inter-processor communication is the foremost important reason for performance deterioration when parallelizing loops with dependencies. No study of dynamic algorithms for loops with dependencies on homogeneous or heterogeneous clusters has been reported so far.

In this paper, we study the problem of dynamic scheduling of uniform dependence loops on heterogeneous distributed systems. We extend three well known dynamic schemes (CSS, TSS, DTSS) and apply them to dependence loops. After partitioning the index space into chunks (using one of the three schemes), we introduce synchronization points at certain intervals so that processors compute chunks in pipelined fashion. Synchronization points are carefully placed so that the volume of data exchange is reduced and the pipeline parallelism is improved. Our scheme is called dynamic multi-phase scheduling (DMPS(\text{x})), where \( \text{x} \) stands for one of the three algorithms, considered as an input parameter to DMPS. We implement our new scheme on a network of heterogeneous (dedicated and non-dedicated) computers and evaluate its performance through extensive simulation and empirical testing. Two case studies are examined: the Heat Equation and the Floyd-Steinberg dithering algorithm. The experimental results validate the presented theory and corroborate the efficiency of the parallel code.

Section 2 gives the algorithmic model and some notations. In section 3, we thoroughly present our algorithm and motivation. In section 4, the implementation, the case studies we used and the experimental results are presented. In Section 5, conclusions are drawn.

2. Notation

Parallel loops have no dependencies among iterations and, thus, the iterations can be executed in any order or even simultaneously. In dependence loops the iterations depend on each other, which imposes a certain execution order. The depth of the loop nest, \( n \), determines the dimension of the iteration index space \( J = \{ j \in \mathbb{N}^n \mid l_r \leq l_r \leq u_r, 1 \leq r \leq n \} \). Each point of this \( n \)-dimensional index space is a distinct iteration of the loop body. \( L = (l_1, \ldots, l_n) \) and \( U = (u_1, \ldots, u_n) \) are the initial and terminal points of the index space.

![Figure 1. Algorithmic model.](image)

Without loss of generality we assume that \( L = (1, \ldots, 1) \) and that \( u_1 \geq \ldots \geq u_n \). \( DS = \{d_1, \ldots, d_p\} \), \( p \geq n \), is the set of the \( p \) dependence vectors, which are uniform, i.e., constant throughout the index space. The index space of the dependence loop is divided into chunks, using one of the three dynamic schemes, giving preference to the smallest dimension (here \( u_n \)). The following notation is used throughout the paper:

- PE stands for processing element.
- \( P_1, \ldots, P_m \) are the slaves.
• $N$ is the number of scheduling steps, $i = 1, \ldots, N$.
• $C_i$: A few consecutive iterations of the loop are called a chunk; $C_i$ is the chunk size at the $i$-th scheduling step.
• $V_i$ is the size (in number of iterations) of chunk $i$ along dimension $u_i$.
• $SP$: In each chunk we introduce $M$ synchronization points ($SP$) uniformly distributed along $u_1$.
• $H$ is the interval (number of iterations along dimension $u_1$) between two $SP$s ($H$ is the same for every chunk).
• The current slave is the slave assigned with the chunk $i$, whereas the previous slave is the slave assigned with the chunk $i - 1$.
• $VP_k$ is the virtual computing power of slave $P_k$.
• $VP = \sum_{k=1}^{m} VP_k$ is the total virtual computing power of the cluster.
• $Q_k$ is the number of processes in the run-queue of $P_k$, reflecting the total load of $P_k$.
• $ACP$: $A_k = \left\lceil \frac{VP_k}{Q_k} \right\rceil$ is the available computing power (ACP) of $P_k$ (needed when the loop is executed in non-dedicated mode).
• $A = \sum_{k=1}^{m} A_k$ is the total available computing power of the cluster.
• $SC_{i,j}$ is the set of iterations of chunk $i$, between $SP_{j-1}$ and $SP_j$.

Figure 3 below illustrates $C_i$, $V_i$ and $H$. Note that $C_i$ is the number of iterations in the rectangular region, i.e. $C_i = V_i \times M \times H$.

3. A dynamic scheduling scheme for uniform dependence loops

This section gives the motivation for this work and describes our proposed method.

3.1. Motivation

Existing dynamic scheduling algorithms cannot cope with uniform dependence loops. Consider, for instance, the heat equation, with its pseudocode below:

```c
/* Heat equation */
for (i=1; i<width; i++) {
    for (j=1; j<height; j++) {
    }
}
```

When dynamic schemes are applied to parallelize this problem, the index space is partitioned into chunks, that are assigned to slaves. These slaves then work independently. But due to the presence of dependencies, the slaves have to communicate. However, existing dynamic schemes do not provide for inter-slave communication, only for master-to-slaves communication. Therefore, in order to apply dynamic schemes to dependence loops, one must provide an inter-slave communication scheme, such that problem’s dependencies are not violated or ignored.

In this work we bring dynamic scheduling schemes into the field of scheduling loops with dependencies. We propose an inter-slave communication scheme for three well known dynamic methods: CSS [23], TSS [13] and DTSS [6]. In all cases, after the master assigns chunks to slaves, the slaves synchronize by means of synchronization points. This provides the slaves with a unified communication scheme. This is depicted in Fig. 2 and 3, where chunks $i - 1, i, i + 1$ are assigned to slaves $P_{k-1}, P_k, P_{k+1}$, respectively. The shaded areas denote sets of iterations that are computed concurrently by different PEs. When $P_k$ reaches the synchronization point $SP_{j+1}$ (i.e. after computing $SC_{i,j+1}$) it sends $P_{k+1}$ only the data $P_{k+1}$ requires to begin execution of $SC_{i+1,j+1}$. The data sent to $P_{k+1}$ designates only those iterations of $SC_{i+1,j+1}$ imposed by the dependence vectors, on which the iterations of $SC_{i+1,j+1}$ depend on. Similarly, $P_k$ receives from $P_{k-1}$ the data $P_k$ requires to proceed with the execution of $SC_{i,j+2}$. Note that slaves do not reach a synchronization point at the same time. For instance, $P_k$ reaches $SP_{j+1}$ earlier than $P_{k+1}$ and later than $P_{k-1}$. The existence of synchronization points leads to pipelined execution, as shown in Fig. 2 by the shaded areas.

3.2. Dynamic scheduling for dependence loops

This section gives a brief description of the three dynamic algorithms we used. Chunk Self-Scheduling (CSS) assigns a chunk that consists of a number of iterations (known as $C_i$) to a slave. A large chunk size reduces scheduling overhead, but also increases the chance of load imbalance. The Trapezoid Self-Scheduling (TSS) [13] scheme linearly decreases the
chunk size $C_i$. Considering $|J|$ the total number of iterations of the loop, in TSS the first and last (assigned) chunk size pair $(F,L)$ may be set by the programmer. In a conservative selection, the $(F,L)$ pair is determined as: $F = \frac{|J|}{2 \times m}$ and $L = 1$. This ensures that the load of the first chunk is less than 1/m of the total load in most loop distributions and reduces the chance of imbalance due to large size of the first chunk. Then, the proposed number of steps needed for the scheduling process is $N = \frac{2 \times |J|}{F - L}$. Thus the decrease between consecutive chunks is $D = (F - L)/(N - 1)$. Then the chunk sizes in TSS are $C_1 = F, C_2 = F - D, C_3 = F - 2 \times D, \ldots$. Distributed TSS (DTSS) [6] improves on TSS by selecting the chunk sizes according to the computational power of the slaves. DTSS uses a model that includes the number of processes in the run-queue of each PE. Every process running on a PE is assumed to take an equal share of its computing resources. 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): $F = \frac{|J|}{2 \times m}$ and $L = 1$. The total number of steps is $N = \frac{2 \times |J|}{F - L}$ and the chunk decrement is $D = (F - L)/(N - 1)$. The size of a chunk in this case is $C_i = A_k \times (F - D \times (S_k - 1 + (A_k - 1)/2))$, where: $S_k - 1 = A_1 + \ldots + A_k - 1$. When all PEs are dedicated to a single process then $A_k = V_k$. Also, when all the PEs have the same speed then $V_k = 1$ and the tasks assigned in DTSS are the same as in TSS. The important difference between DTSS and TSS is that in DTSS the next chunk is allocated according to a PE’s available computing power, but in TSS all PEs are simply treated in the same way. Thus, faster PEs get more iterations than slower ones in DTSS. Table 3.2 shows the chunk sizes computed with CSS, TSS and DTSS for an index space size of $5000 \times 10000$ and $m = 10$ slaves. CSS and TSS obtain the same chunk sizes in dedicated clusters as in non-dedicated clusters; DTSS adapts the chunk size to match the different computational powers of slaves. These algorithms have been evaluated for parallel loops and it has been established that the DTSS algorithm improves on the TSS, which in turn outperforms CSS [6].

**Table 1. Sample chunk sizes given for $|J| = 5000 \times 10000$ and $m = 10$**

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Chunk sizes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSS</td>
<td>300 300 300 300 300 300 300 300</td>
</tr>
<tr>
<td>TSS</td>
<td>277 270 263 256 249 242 235 228 221</td>
</tr>
<tr>
<td>DTSS (dedicated)</td>
<td>392 383 368 354 344 334 324 314 304</td>
</tr>
<tr>
<td>DTSS (non-dedicated)</td>
<td>263 253 243 233 223 213 203 193 183</td>
</tr>
</tbody>
</table>

For the sake of simplicity we consider a 2D dependence loop with $U = (u_1, u_2)$, and $u_1 \geq u_2$. The index space of this loop is divided into chunks along $u_2$ (using

![Figure 2. Synchronization points](image1.png)

![Figure 3. Chunks are formed along $u_2$ and SP are introduced along $u_1$.](image2.png)
one of the three algorithms). Along $u_1$ synchronization points are introduced at equal intervals. The interval length ($H$), chosen by the programmer, determines the number of synchronization points.

3.3. The $DMPS(x)$ algorithm

The following notation is essential for the inter-slave communication scheme: the master always names the slave assigned with the latest chunk ($C_i$) as current and the slave assigned with the chunk $C_{i-1}$ as previous. Whenever a new chunk is computed and assigned, the current slave becomes the (new) previous slave, whereas the new slave is named (new) current. Fig. 4 below shows the state diagram related to the (new) current — (new) previous slaves. The state transitions are triggered by new requests for chunks to the master.

The $DMPS(x)$ algorithm is described in the following pseudocode:

**INPUT**
(a) An $n$-dimensional dependence nested loop, with terminal point $U$.
(b) The choice of algorithm CSS, TSS or DTSS.
(c) If CSS is chosen, then chunk size $C_i$.
(d) The synchronization interval $H$.
(e) The number of slaves $m$; in case of DTSS the virtual power of every slave.

**Master:**
*Initialization:* (a) Register slaves. In case of DTSS, slaves report their $ACP$.
(b) Calculate $F, L, N, D$ for TSS and DTSS. For CSS use the given $C_i$.

1. While there are unassigned iterations do:
   (a) If a request arrives, put it in the queue.
   (b) Pick a request from the queue, and compute the next chunk using CSS, TSS or DTSS.
   (c) Update the current and previous slave ids.
   (d) Send the id of the current slave to the previous one.

**Slave** $P_i$:
*Initialization:* (a) Register with the master; in case of DTSS report $ACP$.
(b) Compute $M$ according to the given $H$.

1. Send request to the master.
2. Wait for reply; if received chunk from master goto step 3 else goto OUTPUT.
3. While the next synchronization point is not reached compute chunk $i$.
4. If id of the send-to slave is known goto step 5 else goto step 6.
5. Send computed data to send-to slave.

**OUTPUT** Master: If there are no more chunks to be assigned to slaves, terminate.

**Slave** $P_i$: If no more tasks come from master, terminate.

**Remark:** (1) Note that the synchronization intervals are the same for all chunks. For remarks (2)–(5) below refer to Fig. 4 for an illustration. (2) Upon completion of $SC_{i,0}$, slave $P_k$ requests from the master the identity of the send-to slave. If no reply is received, then $P_k$ is still the current slave, and it proceeds to receive data from the previous slave $P_{k-1}$, and then it begins $SC_{i,1}$. (3) Slave $P_k$ keeps requesting the identity of the send-to slave, at the end of every $SC_{i,j}$ until either a (new) current slave has been appointed by the master or $P_k$ has finished chunk $i$. (4) If slave $P_k$ has already executed $SC_{i,0}, \ldots, SC_{i,j}$ by the time it is informed by the master about the identity of the send-to slave, it sends all computed data from $SC_{i,0}, \ldots, S_{i,j}$. (5) If no send-to slave has been appointed by the time slave $P_k$ finishes chunk $i$, then all computed data is kept in the local memory of slave $P_k$. Then $P_k$ makes a new request to the master to become the (new) current slave.
4. Implementation and Test Results

Our implementation relies on the distributed programming framework offered by the mpi.ch.1.2.6 implementation of the Message Passing Interface (MPI) [12], and the 1.2.6 version of the gcc compiler.

We used a heterogeneous distributed system that consists of 10 computers, one of them being the master. More precisely we used: (a) 4 Intel Pentiums III 1266MHz with 1GB RAM (called zealots), assumed to have \( V_P = 1.5 \) (one of these was chosen to be the master); and (b) 6 Intel Pentiums III 500MHz with 512MB RAM (called kids), assumed to have \( V_P = 0.5 \). The virtual power for each machine type was determined as a ratio of processing times established by timing a test program on each machine type. The machines are interconnected by a Fast Ethernet, with a bandwidth of 100 Mbits/sec.

We present two cases, dedicated and non-dedicated. In the first case, processors are dedicated to running the program and no other loads are interposed during the execution. We take measurements with up to 9 slaves. We use one of the fast machines as a master. In the second case, at the beginning of the execution of the program, we start a resource expensive process on some of the slaves. Due to the fact that scheduling algorithms for loops with uniform dependencies are usually static and no other dynamic algorithms have been reported so far, we cannot compare with similar algorithms. We ran three series of experiments for the dedicated and non-dedicated case: (1) DMPS(CSS), (2) DMPS(TSS), and (3) DMPS(DTSS) and compare the results for two real-life case studies. We ran the above series for \( m = 3, 4, 5, 6, 7, 8, 9 \) slaves in order to compute the speedup. We compute the speedup according to the following equation:

\[
S_p = \frac{\text{min}(T_{P_1}, T_{P_2}, \ldots, T_{P_m})}{T_{PAR}}
\]

where \( T_{P_i} \) is the serial execution time on slave \( P_i \), \( 1 \leq i \leq m \), and \( T_{PAR} \) is the parallel execution time (on \( m \) slaves). Note that in the plotting of \( S_p \), we use \( V_P \) instead of \( m \) on the \( x \)-axis.

4.1. Test Problems

We used the heat equation computation for a domain of \( 5000 \times 10000 \) points, and the Floyd-Steinberg error diffusion computation for a image of \( 10000 \times 20000 \) pixels, on a system consisting of 9 heterogeneous slave machines and one master, with the following configuration: zealot1 (master), zealot2, kid1, zealot3, kid2, zealot4, kid3, kid4, kid5, kid6. For instance, when using 6 slaves, the machines used are: zealot1 (master), zealot2, kid1, zealot3, kid2, zealot4, kid3. The slaves in italics are the ones loaded in the non-dedicated case. As mentioned previously, by starting a resource expensive process on these slaves, their ACP is halved.

4.2. Heat Equation

The heat equation computation is one of the most widely used case studies in the literature, and its loop body is similar to the majority of the numerical methods used for solving partial differential equations. It computes the temperature in each pixel of its domain based on two values of the current time step \((A[i-1][j], A[i][j+1])\) and two values from the previous time step \((A'[i+1][j], A'[i][j+1])\), over a number of loop time steps. The dependence vectors are: \( d_1 = (1, 0) \) and \( d_2 = (0, 1) \). The pseudocode is given below:

```c
/* Heat equation */
for (i=1; i<width; i++) {  
  for (j=1; j<height; j++) {  
                  + A[i+1][j] + A[i][j+1];
    A'[i+1][j] = A'[i][j-1] + A'[i][j+1];
  }
}
```

An illustration of the dependence patterns is given in Fig. 5. The iterations in a chunk are executed in the order imposed by the dependencies of the heat equation. Whenever a synchronization point is reached, data is exchanged between the processors executing neighboring chunks.

Table 2 shows comparative results we obtained for the heat equation, for the three series of experiments: DMPS(CSS), DMPS(TSS) and DMPS(DTSS), on a dedicated and a non-dedicated heterogeneous cluster. The values represent the parallel times (in seconds) for different number of slaves. Three synchronization intervals were chosen, and the total ACP ranged according to the number of slaves from 3.5–7.5.

Fig. 6 presents the speedups for the heat equation on an index space of \( 5000 \times 10000 \) points, for one time step (i.e. \( \text{loop} = 1 \)), for chunks sizes computed with CSS, TSS and DTSS and synchronization interval 150, on a dedicated cluster and a non-dedicated cluster.

4.3. Floyd-Steinberg

The Floyd-Steinberg computation [17] is an image processing algorithm used for the error-diffusion dither-
Figure 5. The dependence patterns for heat equation and Floyd-Steinberg.

Table 2. Parallel execution times (sec) for heat equation

<table>
<thead>
<tr>
<th>Sync. interval</th>
<th>Dedicated</th>
<th>Number of slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DMPS(CS)</td>
<td>3 4 5 6 7 8 9</td>
</tr>
<tr>
<td>100</td>
<td>2.52</td>
<td>1.75 1.75 1.75 1.75 1.75 1.75 1.75 1.75 1.75</td>
</tr>
<tr>
<td>150</td>
<td>2.25</td>
<td>1.75 1.75 1.75 1.75 1.75 1.75 1.75 1.75 1.75</td>
</tr>
<tr>
<td>200</td>
<td>2.21</td>
<td>1.74 1.74 1.74 1.74 1.74 1.74 1.74 1.74 1.74</td>
</tr>
</tbody>
</table>

Figure 6. Speedups for the heat equation

An illustration of the dependence patterns is given.
in Fig. 5. The iterations in a chunk are executed in the order imposed by the dependencies of the Floyd-Steinberg algorithm. Whenever a synchronization point is reached, data is exchanged between the processors executing neighboring chunks.

Comparative results for the Floyd-Steinberg case study on a dedicated and a non-dedicated heterogeneous cluster are given in Table 3. The values represent the parallel times (in seconds) for different number of slaves. Three synchronization intervals were chosen, and the total ACP ranged according to the number of slaves from 3.5–7.5.

Table 3. Parallel execution times (sec) for Floyd-Steinberg

<table>
<thead>
<tr>
<th>Sync. interval</th>
<th>Dedicated</th>
<th>Number of slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DMPS(CS)</td>
<td>3</td>
</tr>
<tr>
<td>DMPS(TSS)</td>
<td>22.32</td>
<td>19.77</td>
</tr>
<tr>
<td>DMPS(DTSS)</td>
<td>19.63</td>
<td>14.87</td>
</tr>
<tr>
<td>100</td>
<td>DMPS(CS)</td>
<td>27.85</td>
</tr>
<tr>
<td>DMPS(TSS)</td>
<td>20.22</td>
<td>15.99</td>
</tr>
<tr>
<td>DMPS(DTSS)</td>
<td>19.62</td>
<td>14.82</td>
</tr>
<tr>
<td>150</td>
<td>DMPS(CS)</td>
<td>27.88</td>
</tr>
<tr>
<td>DMPS(TSS)</td>
<td>20.22</td>
<td>15.99</td>
</tr>
<tr>
<td>DMPS(DTSS)</td>
<td>19.62</td>
<td>14.82</td>
</tr>
</tbody>
</table>

Table 3. Parallel execution times (sec) for Floyd-Steinberg

<table>
<thead>
<tr>
<th>Sync. interval</th>
<th>Non-dedicated</th>
<th>Number of slaves</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>DMPS(CS)</td>
<td>3</td>
</tr>
<tr>
<td>50</td>
<td>27.72</td>
<td>22.43</td>
</tr>
<tr>
<td>DMPS(TSS)</td>
<td>25.18</td>
<td>19.72</td>
</tr>
<tr>
<td>DMPS(DTSS)</td>
<td>21.88</td>
<td>15.96</td>
</tr>
<tr>
<td>100</td>
<td>DMPS(CS)</td>
<td>27.49</td>
</tr>
<tr>
<td>DMPS(TSS)</td>
<td>25.18</td>
<td>19.66</td>
</tr>
<tr>
<td>DMPS(DTSS)</td>
<td>21.55</td>
<td>15.96</td>
</tr>
<tr>
<td>150</td>
<td>DMPS(CS)</td>
<td>27.57</td>
</tr>
<tr>
<td>DMPS(TSS)</td>
<td>25.18</td>
<td>19.65</td>
</tr>
<tr>
<td>DMPS(DTSS)</td>
<td>21.55</td>
<td>15.96</td>
</tr>
</tbody>
</table>

Fig. 7 presents the speedup results of the Floyd-Steinberg algorithm, for the three variations. The size of the index space was 10000×20000. Chunks sizes were computed with CSS, TSS and DTSS and synchronization interval chosen to be 100, on a dedicated cluster and a non-dedicated cluster.

4.4. Interpretation of the results

As expected, the results for the dedicated cluster are much better for both case studies. In particular, DMPS(TSS) seems to perform slightly better than DMPS(CS). This was expected since TSS provides better load balancing than CSS for simple parallel loops without dependencies. In addition, DMPS(DTSS) outperforms both algorithms. This is because it explicitly accounts for the heterogeneity of the slaves. For the non-dedicated case, one can see that DMPS(CSS) and DMPS(TSS) cannot handle workload variations as effectively as DMPS(DTSS). This is shown in Fig. 6. The speedup for DMPS(CSS) and DMPS(TSS) decreases as loaded slaves are added, whereas for DMPS(DTSS) it increases even when slaves are loaded. In the non-dedicated approach, our choice was to load the slow processors, so as to incur large differences between the processing power of the two machine types. Even in this case, DMPS(DTSS) achieved good results.

The ratio of computation to communication along with the selection of the synchronization interval play a key role in the overall performance of our scheme. A rule of thumb is to maintain this ratio ≥ 1 at all times. The choice of the (fixed) synchronization interval has a crucial impact on the performance, and it is dependent on the concrete problem. H must be chosen so as to ensure the ratio of computation to communication is maintained above 1, even when V decreases at every scheduling step. Assuming that for a certain H, the constraint H ≥ 1 is satisfied, small changes in the value of H do not significantly alter the overall performance. Notice that the best synchronization interval for the heat equation was H = 150, whereas for the Floyd-Steinberg better results were obtained for H = 100. The performance differences for
interval sizes close to the ones depicted in Fig. 6 and 7 are small.

5. Conclusion

In this paper we presented a novel dynamic scheduling scheme for dependence loops on heterogeneous clusters. We tested three variations of our method on a heterogeneous cluster, both in dedicated and non-dedicated mode. The main contribution of our work is extending three previous schemes by taking into account the existing iteration dependencies of the problem, and hence providing a scheme for inter-slave communication. We tested our method on two real-life applications: heat equation and Floyd-Steinberg algorithm. The results demonstrate that our new scheme is effective for distributed applications with dependence loops.

Future work will focus on establishing a model for predicting the optimal synchronization interval ($H$) such that communication is minimized for every problem. Also we intend to extend other well known dynamic algorithms to be applied to dependence loops, and incorporated in an automatic parallel code generation tool for heterogeneous systems.

Acknowledgments

The project is partially co-funded by the European Social Fund (75%) and National Resources (25%) - Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS. This work of F. Ciorba was supported by the Greek State Scholarships Foundation. This work of Dr. Chronopoulos was supported in part by National Science Foundation under grant CCR-0312323.

References


