Self-Adapting Scheduling for Tasks with Dependencies in Stochastic Environments

I. Riakotakis, F. M. Ciorba, T. Andronikos and G. Papakonstantinou
Computing Systems Laboratory,
Department of Electrical and Computer Engineering,
National Technical University of Athens,
Zografou Campus, 15773, Athens, Greece
Phone: +302107722494, Fax: +302107722496
Email: {iriak,cflorina,tedandro,papakon}@cslab.ntua.gr

Abstract

This paper addresses dynamic load balancing algorithms for non-dedicated heterogeneous clusters of workstations. We propose an algorithm called Self-Adapting Scheduling (SAS), targeted at nested loops with dependencies in a stochastic environment. This means that the load entering the system, not belonging to the parallel application under execution, follows an unpredictable pattern which can be modeled by a stochastic process. SAS takes into account the history of previous timing results and the load patterns in order to make accurate load balancing predictions. We study the performance of SAS in comparison with DTSS. We established in previous work that DTSS is the most efficient self-scheduling algorithm for loops with dependencies on heterogeneous clusters. We test our algorithm under the assumption that the interarrival times and lifetimes of incoming jobs are exponentially distributed. The experimental results show that SAS significantly outperforms DTSS especially with rapidly varying loads.

Keywords: dynamic algorithms, load balancing, loops with dependencies, stochastic environments.

1 Introduction

Many scheduling algorithms have been devised for general distributed systems composed of non-identical processing nodes, called heterogeneous distributed systems. Load balancing is one of the most important issues both for homogeneous and heterogeneous networks, as it has a direct impact over the system’s performance. Load imbalance can be attributed to: (1) application tasks not having a uniform distribution and (2) the distributed systems being non-dedicated to the application in question. If one of these factors is not accounted for, load imbalance may occur, causing poor performance. Load balancing is usually achieved by relocating application tasks from busy nodes to lightly loaded or idle nodes [1]. A few load balancing algorithms for homogeneous systems were presented in [2] [3], and for heterogeneous systems in [4] [5].

Apart from load balancing algorithms targeted at homogeneous or heterogeneous distributed systems, there are also static and dynamic scheduling algorithms. A review of classic static algorithms is given in [14]. Static algorithms [6] [7] are inappropriate for non-dedicated distributed systems for two reasons: (1) the workload distribution of many applications cannot be predicted, and (2) the available computation power of the processing node may not be known in advance or may not remain constant throughout the application’s execution. In contrast, dynamic algorithms [8] attempt to use the runtime state information of the system in order to make informative decisions for balancing the workload. This makes them applicable to a much larger spectrum of applications. In [11] different dynamic load balancing algorithms with different complexities were compared.

An important class of dynamic scheduling algorithms are the self-scheduling schemes: Chunk Self-Scheduling (CSS) [15], Guided Self-Scheduling (GSS) [17], Trapezoid Self-Scheduling (TSS) [11], Factoring Self-Scheduling (FSS) [12]. These algorithms were devised for nested loops without dependencies and for homogeneous systems. Self-scheduling algorithms divide the total number of tasks into chunks, which are then assigned to processors (slaves). In their original form they could not handle loops with dependencies nor perform satisfactorily on non-dedicated heterogeneous systems.

* Non-dedicated systems are those systems in which all computa-
tems. A first attempt to make self-scheduling algorithms suitable for heterogeneous systems was Weighted Factoring (WF) proposed in [13]. WF differs from FSS in that the chunks sizes are weighted according to the processing powers of the slaves. However, with WF processor weights remain constant throughout the parallel execution. Banicescu & Liu proposed in [16] a method called Adaptive Factoring (AF), that adjusts the processor weights according to timing information reflecting variations in the computation power of the slaves. This was designed for time-stepping scientific applications. Chronopoulos et al. extended the TSS algorithm proposing the Distributed TSS (DTSS) in [9], an adaptive algorithm suitable for distributed systems. With DTSS the chunks sizes are weighted by the slaves relative power and the number of processes in their run-queue, at the moment of requesting work from the master.

In this paper we propose a dynamic self-scheduling algorithm capable of performing efficiently in a non-dedicated heterogeneous system, in the presence of stochastic exogenous load. By exogenous load we refer to the load that enters the system and does not belong to the parallel application under execution. The exogenous load in general follows an unpredictable pattern and should therefore be modeled by a stochastic process. In previous work [10] we have extended the self-scheduling algorithm DTSS, proposing DTSS+SP to handle loops with dependencies on heterogeneous clusters via synchronization points (SPs). In the rest of the paper we simply write DTSS instead of DTSS+SP, with the understanding that when DTSS is applied to loops with dependencies it is augmented with SPs. DTSS does not make explicit assumptions about exogenous load in the system. Herein, we combine the strengths of DTSS (i.e., run-queue information) and of AF (i.e., timing information) and propose SAS (Self-Adapting Scheduling), a dynamic self-scheduling algorithm that attempts to dynamically balance the application workload in the presence of fluctuating exogenous load. SAS assigns chunks to processors according to: (1) the history of all previous timing results; (2) the history of incoming jobs; and (3) the current number of processes in the run-queue. Therefore, by considering the system’s history and the current system load, the workload is assigned in a balanced fashion. SAS is a dynamic self-scheduling algorithm, that employs the master-slave model and it is targeted at compute-intensive applications with dependent tasks, but in the same time, it can handle independent tasks as well. These applications are in the form of nested loops with (or without) dependencies.

In order to study the performance of SAS and DTSS in a stochastic environment, we assume that incoming jobs’ interarrival times and lifetimes follow an exponential distribution. We compare SAS with DTSS [10] and we show that SAS outperforms DTSS in all cases, especially with rapidly varying loads.

2 Background

This paper addresses the problem of dynamic scheduling for dependence loops. In dependence loops the iterations depend on each other, imposing a certain execution order, given by the existing (uniform in our case) dependencies. The general form of a dependence loop is illustrated in Fig. 1. A uniform dependence loop is a dependence loop with uniform dependencies. The index space \( J \) of an \( n \)-dimensional uniform dependence loop is a \( n \)-dimensional subspace of \( \mathbb{N}^n \). The loop body contains general programs statements that include assignment statements, conditional if statements and repetitions such as for or while. The lower and upper bounds for the loop indices are \( l_i \) and \( u_i \in \mathbb{Z} \), respectively. The depth of the loop nest, \( n \), determines the dimension of the index space \( J = \{ j \in \mathbb{N}^n \mid l_r \leq i_r \leq u_r, 1 \leq r \leq n \} \). Each point of the \( n \)-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. \( DS = \{d_1, \ldots, d_p\}, p \geq n \), is the set of the \( m \) dependence vectors, which are uniform, i.e., constant throughout the index space.

The following notation is used throughout the paper:

- \( P_1, \ldots, P_m \) are the slaves.
- \( V P_k \) is the virtual computing power of slave \( P_k \), and \( \sum_{k=1}^{m} V P_k \) 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 \( P_k \).

\( ^{1}\)The virtual power for each machine type can be established as the normalized execution time of the same test program on each machine type.
$A_k = \frac{V \cdot P_k}{q_k}$ is the available computing power of $P_k$, and $\sum_{k=1}^{m} A_k$ is the total available computing power of the cluster. Although using $q_k$ as above it is assumed that the processing slowdown is linearly proportional to the number of jobs present in the run queue, this works quite well in practice. In contrast to the total virtual power, the total available power of the cluster varies over time depending on load fluctuations.

- One of the $n$-dimensions is noted as the synchronization dimension $u_s$, along which synchronization points are introduced, and another one as the scheduling dimension $u_c$, which is divided into chunks according to a self-scheduling algorithm (see Fig. 2).

- A few consecutive iterations of the loop are called a chunk. The size of the $i$-th chunk is denoted $C_i$, and is directly proportional to the projection $V_i$ of the chunk $i$ on the scheduling dimension. Throughout this paper, only $V_i$ is used to indicate the size of the $i$-th chunk (see Fig. 2).

- $SP$: In each chunk we introduce $M$ synchronization points ($SP$) uniformly distributed along the synchronization dimension. $H$ is the synchronization interval between two $SP$s ($H$ is the same for every chunk). $SC_{i,j}$ is the set of iterations of chunk $i$, between $SP_{i-1}$ and $SP_j$.

- The current slave is the slave assigned with the latest chunk $i$, whereas the previous slave is the slave assigned with the chunk $i - 1$. This information concerns only the master.

- The send-to is the slave id to which a slave must send computed data to. The receive-from is the slave id from which a slave must receive data, so as to begin its current computation. This information is communicated to the particular slave by the master, based on the current and previous slave ids.

For the sake of simplicity, in Fig. 2 we consider a 2D dependence loop with $U = (u_1, u_2)$. The index space of this loop is divided into chunks along $u_2$. Along $u_1$ synchronization points are introduced at equal intervals. The synchronization interval $(H)$, chosen by the programmer, determines the number of synchronization points.

### 3 Overview of DTSS

This section gives a brief description of Distributed TSS (DTSS) [9]. DTSS employs the master-slave paradigm and improves on other self-scheduling algorithms by selecting the chunk sizes according to the computational power of the slaves. DTSS is an adaptive algorithm because it uses a model that includes the number of processes in the run-queue of each slave. Every process running on a slave is assumed to take an equal share of its computing resources. Recall that $|J|$ is the total number of iterations of the loop, then the first and last (assigned) chunk size pair $(F, L)$ may be set by the programmer as in TSS; or the following formula may be used in the conservative selection approach (by default): $F = \frac{|J|}{2 \times A}$ and $L = 1$. The total number of chunks is $N = \frac{2 \times |J|}{2(F + L)}$ 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 slaves are dedicated to a single process, then $A_k = V \cdot P_k$, and when all slaves have the same power, then $V \cdot P_k = 1$. The important difference between DTSS and earlier self-scheduling algorithms is that in DTSS the next chunk is allocated according to the available computing power of the slave, whereas in earlier schemes all slaves are simply treated in the same way. Thus, faster slaves get more iterations than slower ones in DTSS. DTSS has been evaluated for parallel loops (i.e., loops without dependencies) and it has been established in [9] that it improves on other schemes, when a constant or slowly varying load is assumed to exist on some slaves of the cluster.

### 4 Synchronization Points

In [10] it was shown that dynamic algorithms can be applied to dependence loops by introducing a synchronization scheme that enables inter-slave communication at specific intervals. In all cases, after the master assigns
chunks to slaves, the slaves synchronize via synchronization points. This is depicted in Fig. 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 slaves. When $P_k$ reaches the synchronization point $SP_{j+1}$, having computed $SC_{i,j+1}$, it sends to $P_{k+1}$ only the data $P_{k+1}$ requires to begin executing $SC_{i+1,j+1}$ (i.e. the communication set). The data sent to $P_{k+1}$ is only those iterations of $SC_{i,j+1}$ 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 $SC_{i,j+2}$. The size of the communication set is determined by the existing dependence vectors. The longer the length of the dependence vectors, the more points need to be communicated between slaves executing neighboring chunks. Usually, real-life applications have unitary dependence vectors.

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 a wavefront execution, as shown in Fig. 3 by the shaded areas. The synchronization interval $H$ is user defined and must be chosen so as to maintain the computation-to-communication ratio $\geq 1$, for every application. Small variations of the value of $H$ do not significantly affect the overall performance.

The flexibility of our communication scheme, based on synchronization points, is twofold:

1. In the case $P_k$ reaches a specific synchronization point, e.g., $SP_{i,j+1}$, and no $P_{k+1}$ slave has been assigned, then $P_k$ keeps the data required by $P_{k+1}$, until the latter is assigned (or known). This implies that when $P_{k+1}$ is assigned, it receives from $P_k$ all necessary data in one receiving. Hence, $P_{k+1}$ is not stalled by any synchronization point up to $SP_{i,j}$. Moreover, since $P_k$ is the synchronization point assigned to $P_{k+1}$, this means that $P_{k+1}$ always has the data it requires from $P_k$ available before reaching the corresponding synchronization point.

2. In case there is no $P_{k+1}$ assigned by the time $P_k$ reaches the last synchronization point, it stores the data it should have sent to $P_{k+1}$ in its local buffers, makes a new request for work to the master, and if it is assigned to be $P_{k+1}$, it proceeds with the new chunk execution; otherwise, it sends all stored data to the $P_{k+1}$. In both cases, $P_{k+1}$ does not stall at any synchronization point, since it already has the necessary data.

### Figure 3. Synchronization points.

![Diagram showing synchronization points](image)

5 **Self Adaptive Scheduling for Tasks with Dependencies**

This section describes Self-Adapting Scheduling, an attempt at dynamic load balancing for dependence loops in the presence of stochastic exogenous loads. SAS is a self-scheduling algorithm that assigns chunks to slaves by combining for each slave the history of computation times of previous chunks, the history of processes in its run-queue and the current number of jobs in its run-queue. It starts by sorting the slaves in decreasing order of their virtual power. Therefore, without loss of generality we assume that $VP_1 \geq VP_2 \geq \ldots \geq VP_m$ and that $VP_1 = 1$. It computes and assigns the first chunks to slaves according to their virtual computing power. The size of the first chunk assigned to $P_k$ is computed by the following formula:

$$\left\lfloor \frac{u_c \times VP_k}{2 \times \sum_{i=1}^{m} VP_i} \right\rfloor$$

Good load balancing is achieved if all the slaves take approximately equal time to compute their assigned chunks. For this reason SAS employs the concept of reference time, denoted $t_{Ref}$, in the sense that all slaves are expected to complete their chunk within $t_{Ref}$. We take as reference time the computation time of the first chunk. SAS distributes the workload to every slave so as each of them completes its chunk within the considered reference time. This is accomplished by using estimates for the completion time of a chunk based on previous timing results and run-queue history.

### Terminology

- $V_j^k, \ldots, V_j^k$ are the sizes of the first $j$ chunks assigned to $P_k$ and $t_j^1, \ldots, t_j^k$ are their computation times.
\* \( q_1^k, \ldots, q_j^k \) are the number of jobs in the run-queue of \( P_k \) when assigned the first \( j \) chunks.

\* The average time per iteration for the first \( j \) chunks of slave \( P_k \) denoted \( \bar{\mu}_j^k \) and is calculated by the following formula:

\[
\bar{\mu}_j^k = \frac{\sum_{i=1}^{j} t_i^k \times V_i^k}{j}
\]  

(2)

This formula takes into account the number of jobs in the run-queue of \( P_k \) because we have assumed that the processing slowdown is linearly proportional to the number of jobs present.

\* \( t_{j+1}^k = \bar{\mu}_j^k \times q_{j+1}^k \times V_j^k \) is the estimated time of \( P_k \) for the computation of chunk \( j+1 \) given that there are \( q_{j+1}^k \) jobs in its run-queue and that the size of chunk \( j+1 \) is equal to \( V_j \).

Therefore, SAS computes the size of the chunk it assigns to \( P_k \) based on the estimated time and the size and execution time of the previous chunk of \( P_k \) as follows:

\[
V_{j+1}^k = \frac{t_{Ref}}{t_{j+1}^k} \times V_j^k
\]  

(3)

The Self-Adapting Scheduling algorithm is described in the following pseudocode:

---

**Master:**

*Initialization:*  
(a) Register slaves. Slaves report their \( VP_k \) and \( q_1^k \).
(b) Sort slaves decreasingly according to their \( VP_k \) and assign the first chunk of each slave.

While there are unassigned iterations do:
(a) Receive request from slave \( P_k \) and record its \( q_{j+1}^k \) and \( t_j^k \).
(b) Compute \( V_{j+1}^k \) according to equation (3) and assign it to \( P_k \).

**Slave \( P_k \):**

*Initialization:* (a) Register with the master and report \( VP_k \) and \( q_1^k \).

1. Send request to the master, reporting \( q_{j+1}^k \) and the time \( t_j^k \) spent on completing the previous chunk.
2. Wait for reply.
   - If no more chunks, terminate.
   - Else receive the next chunk (of size \( V_{j+1}^k \)) and compute it.
3. Exchange data at \( SPs \) as described in \$4 \] and \$10\).

---

4. Measure the computation time \( t_{j+1}^k \) for chunk \( V_{j+1}^k \).
5. Go to step 1.

---

When slaves make a request for work to the master, they also piggyback the feedback information, so that the master can include it in the scheduling decision. This does not affect systems performance since the feedback information is very light.

---

6 **Stochastic Environment Modeling - Exogenous Load Distribution**

Non-adaptive self-scheduling schemes are characterized by fixed chunk sizes or by predictably decreasing chunk sizes. They perform well on dedicated systems, but poorly on non-dedicated ones. Adaptive self-scheduling schemes compensate this drawback either by giving weights to processing nodes or using timing information to distribute the load. These schemes should perform well on real non-dedicated heterogeneous systems. Existing non-dedicated systems have fluctuating exogenous load. Specifically, since the exogenous load is nondeterministic, such systems can be modeled with stochastic processes. We therefore advocate testing adaptive schemes in stochastic environments.

In order to model a stochastic environment, we assumed that the interarrival time of incoming jobs is exponentially distributed, i.e., the arrival process follows a Poisson distribution. In particular, if the arrival rate is \( \lambda \), then the probability that the next exogenous job arrives after time \( x \) is given by:

\[
1 - e^{-\lambda x}
\]  

(4)

where \( x \geq 0 \) [19]. We also assumed that the lifetime of each exogenous job follows the same distribution, with service rate \( \mu \). By giving different values to the arrival rate \( \lambda \) and the service rate \( \mu \), we model an exogenous load with a fast, medium or slow fluctuation rate. Generally, the number of exogenous jobs in the run-queue of a slave at a given moment can be estimated by the jobs’ arrival rate and their lifetime. That is, fast arriving jobs with long lifetimes lead to numerous jobs in the run-queue, whereas slowly arriving jobs with short lifetimes yield few jobs in the run-queue.

---

7 **Performance Results**

The Floyd-Steinberg computation [21] is an image processing algorithm used for the error-diffusion dithering of a width by height grayscale image. This problem is in the form of a nested loop with dependencies, and these dependencies are: \( d_1 = (1, 0), d_2 = (1, 1), \)
\[ \tilde{d}_x = (0, 1) \text{ and } \tilde{d}_y = (1, -1). \] The tested problem size was \(25000 \times 5000\) iterations.

The pseudocode is given below:

```c
/* Floyd-Steinberg */
for (i=1; i<width; i++) {
    for (j=1; j<height; j++) {
        I[i][j] = trunc(J[i][j]) + 0.5;
        err = J[i][j] - I[i][j]*255;
        J[i-1][j] += err*(7/16);
        J[i][j-1] += err*(3/16);
        J[i][j-1] += err*(5/16);
        J[i-j][j+1] += err*(1/16);
    }
}
```

### Experimental setup

The implementation of SAS and DTSS relies on the distributed programming framework offered by the mpich 1.2.6 implementation of the Message Passing Interface (MPI) [22], and the 1.2.6 version of the gcc compiler.

We used a heterogeneous distributed system that consists of 7 dual-node computers, one of them being the master. More precisely we used: (a) 4 Intel Pentiums III 800MHz (Coppermine) with 256MB RAM (called twins), assumed to have \(V P_k = 0.5\) (one of these was chosen to be the master); and (b) 3 Intel Pentiums III 1266MHz with 1GB RAM (called zealots), assumed to have \(V P_k = 1\). 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 take measurements with \(m = 4, 8, \) and 12 slaves.

As we have already mentioned above, we assumed that the system’s exogenous load (arrival times and lifetimes) follows the exponential distribution. We chose different arrival and lifetime rates, so as to simulate a fast, medium and slow fluctuating exogenous load. These rates are given in Table 1.

### SAS vs DTSS

We compared the performance of SAS vs DTSS for the Floyd-Steinberg test case, for no load (i.e., the system was dedicated to solving the problem and no other incoming jobs were allowed) and then for slow, medium and fast fluctuating exogenous load, with the rates given in Table 1. The parallel times shown in Fig. 4 were taken for a synchronization interval \(H = 100\). In fact, Fig. 4 shows the average parallel times of a 1000 runs. The parallel times of the loaded system are plotted with those of the unloaded system, in order to give an upper bound of the best performance of the algorithms on the specific system. The improvement percentage of SAS over DTSS for this case is given in Table 2.

![Floyd-Steinberg, Exponential distribution (fast)](image)

![Floyd-Steinberg, Exponential distribution (medium)](image)

![Floyd-Steinberg, Exponential distribution (slow)](image)

**Figure 4. SAS vs DTSS parallel times in the Floyd-Steinberg test case.**

### 8 Interpretation of the results

We chose to compare our proposed scheme to DTSS due to the fact that DTSS is the most efficient adaptive algorithm for dependence loops [10]. For the above arrival and lifetime rates of exogenous loads, the experiments show that SAS always outperforms DTSS. SAS as presented here, is a first attempt to adapt to stochastic loads. The results gathered so far, do not reveal the exact interaction between the arrival/lifetime rates and
Table 2. SAS improvement over DTSS for Floyd-Steinberg

<table>
<thead>
<tr>
<th>m</th>
<th>4</th>
<th>8</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>fast</td>
<td>26.60%</td>
<td>20.35%</td>
<td>10.23%</td>
</tr>
<tr>
<td>medium</td>
<td>12.24%</td>
<td>24.54%</td>
<td>15.38%</td>
</tr>
<tr>
<td>slow</td>
<td>25.80%</td>
<td>30.17%</td>
<td>14.00%</td>
</tr>
</tbody>
</table>

the performance gain of the algorithm. However it is clear that, by taking into account a history of previous
 timing results, the probability for more accurate predictions about the system’s load increases, hence yielding
 very good load balancing. SAS outperforms DTSS due to the fact that dependence loops require synchronization
 between slaves. This makes their performance more sensitive to load imbalance. Again, SAS outperforms
 DTSS even in an unloaded system, because the virtual power of the slaves used by DTSS may not be a very
 accurate estimate of the real computational power of the slave, whereas SAS uses timing information to remedy
 any such inaccuracy.

The algorithms were tested in a rather small scale environment. Nonetheless, this limited experiment shows
 that SAS is a promising algorithm. The fact that the scalability saturates as the number of slaves increases may
 be attributed to the limitation of the single-master–slave model employed. However, work is underway with testing
 the SAS and DTSS algorithms in much larger clusters and using a multi-master–slave model.

9 Conclusion

In this paper we proposed a novel dynamic scheduling scheme for dependence loops on non-dedicated hetero-
gerogeneous clusters. We tested our method against an earlier dynamic algorithm on a heterogeneous cluster
with stochastic loads. We applied the exponential distribution to model the interarrival times and lifetimes of
these loads. The main contribution of our work is taking into account exogenous load variations and providing
an algorithm that adapts better than existing algorithms to such conditions. We tested our approach on a real-life
application, i.e., the Floyd-Steinberg computation. The results demonstrate that our new scheme is more effective
than existing ones for applications with dependence loops.

Future work will focus on modeling the exogenous load with other well known distributions and to elaborate
on the relationship between distribution type and exogenous load fluctuation, as well as their impact on the
performance of SAS.

References


Schemes for Heterogeneous Clusters, Proc. of the 3rd IEEE International Conference on Cluster
Computing (CLUSTER 2001), Newport Beach, CA USA, 2001.

Multi Phase Scheduling for Heterogeneous Clusters, Proc. of the 20th IEEE International Parallel
& Distributed Processing Symposium (IPDPS 2006), Rhodes, Greece, 2006.


