Available online at www.sciencedirect.com
ScienceDirect
J. Parallel Distrib. Comput. 68 (2008) 246–264
Journal of Parallel and Distributed Computing
www.elsevier.com/locate/jpdc

Enhancing self-scheduling algorithms via synchronization and weighting

Florina M. Ciorba[a,*], Ioannis Riakiotakis[a], Theodore Andronikos[b], George Papakonstantinou[a], Anthony T. Chronopoulos[c]

[a]*Computing Systems Laboratory, Department of Electrical and Computer Engineering, National Technical University of Athens, Zografou Campus, 15773 Athens, Greece*
[b]*Department of Informatics, Ionian University, 7, Tsirigoti Square, 49100 Corfu, Greece*
[c]*Department of Computer Science, University of Texas at San Antonio, 6900 N. Loop 1604 West, San Antonio, TX 78249, USA*

Received 2 June 2006; received in revised form 9 January 2007; accepted 12 July 2007
Available online 8 August 2007

Abstract

Existing dynamic self-scheduling algorithms, used to schedule independent tasks on heterogeneous clusters, cannot handle tasks with dependencies because they lack the support for internode communication. To compensate for this deficiency we introduce a synchronization mechanism that provides inter-processor communication, thus, enabling self-scheduling algorithms to handle efficiently nested loops with dependencies. We also present a weighting mechanism that significantly improves the performance of dynamic self-scheduling algorithms. These algorithms divide the total number of tasks into chunks and assign them to processors. The weighting mechanism adapts the chunk sizes to the computing power and current run-queue state of the processors. The synchronization and weighting mechanisms are orthogonal, in the sense that they can simultaneously be applied to loops with dependencies. Thus, they broaden the application spectrum of dynamic self-scheduling algorithms and improve their performance. Extensive testing confirms the efficiency of the synchronization and weighting mechanisms and the significant improvement of the synchronized–weighted versions of the algorithms over the synchronized-only versions.
© 2007 Elsevier Inc. All rights reserved.

Keywords: Dynamic load balancing algorithms; Loops with dependencies; Synchronization; Weighting; Non-dedicated heterogeneous systems

1. Introduction

Many scheduling algorithms were devised in the past few years for general distributed systems, composed of non-identical processing nodes, called heterogeneous distributed systems. Load balancing is one of the most challenging issues in attaining high performance in heterogenous distributed systems. In this article we deal with two important cases of the load balancing problem when running loops with and without dependencies. We consider: (1) loops with tasks of uneven size and (2) loops executed on non-dedicated distributed systems. Load balancing is usually achieved by relocating application tasks from busy nodes to lightly loaded or idle nodes [8]. Some load balancing algorithms for homogeneous systems were presented in [2,16], and for heterogeneous systems in [11,24].

Another categorization of scheduling algorithms for distributed systems is static versus dynamic. A review of classic static algorithms for task graphs is given in [15]. Static scheduling algorithms [1,22] are not effective for non-dedicated distributed systems. One reason is that the workload distribution of many applications cannot be predicted. Also the available computation power of the processing node may not be known in advance or may not remain constant throughout the application's execution. Dynamic algorithms [10] strive to achieve load balancing under load variation. The performance of dynamic load balancing algorithms in practice is determined by their ability to adapt to a dynamic and, in most cases, unpredictable environment. They use runtime state information of the system in order to make informative decisions on balancing the workload. This makes them applicable to a much larger spectrum of applications. In [8] different dynamic load balancing algorithms with different complexities were compared.

Many important applications involve loops with or without dependencies among their iterations. An important class of

[*] Corresponding author. Fax: +30 2107721533.
E-mail address: cflorina@cslab.ece.ntua.gr (F.M. Ciorba).

doi:10.1016/j.jpdc.2007.07.003

dynamic scheduling algorithms are the self-scheduling schemes: chunk self-scheduling (CSS) [14], guided self-scheduling (GSS) [21], trapezoid self-scheduling (TSS) [23], factoring self-scheduling (FSS) [13]. These algorithms were devised for nested loops without dependencies executed on homogenous systems. Self-scheduling algorithms divide the total number of tasks into chunks, which are then assigned to processors (slaves). In their original form, these algorithms do not perform satisfactorily on non-dedicated heterogeneous systems and cannot handle loops with dependencies. A first attempt to make self-scheduling algorithms suitable for heterogeneous systems was weighted factoring (WF) that was proposed in [12]. WF differs from FSS in that the chunks sizes are weighted according to the processing powers of the slaves. However, in WF the processor weights remain constant throughout the parallel execution. Banicescu et al. proposed in [4] a method called adaptive weighted factoring (AWF) that adjusts the processor weights according to timing information reflecting variations of slaves computation power. This was designed for time-stepping scientific applications. Chronopoulos et al. extended in [5] the TSS algorithm, proposing the distributed TSS (DTSS) algorithm suitable for distributed systems. In DTSS the chunks sizes are weighted by the slaves relative power and the number of processes in their run-queue.

However, in spite of these developments, self-scheduling algorithms were still not applicable to loops with dependencies. To the best of our knowledge, the first work that applied self-scheduling algorithms to dependence loops was [7]. Therein, CSS, TSS and DTSS were enhanced via synchronization points (SPs) so as to enable inter-slave communication and to satisfy the loop dependencies. Loop unrolling for enhancing parallelization of loops with dependencies has been studied in [20] and references therein. However, loop unrolling is not practical for a large number of iterations because the unrolling factor depends on the size of the available registers. Moreover, it is not easy to determine the unrolling factor in advance.

In this paper we extend and generalize the work in [7] by constructing a general *synchronization* mechanism $\mathcal{S}$, which applies to all loop self-scheduling schemes. When this mechanism is applied to a self-scheduling algorithm, it enables it to handle efficiently loops with dependencies. The synchronization mechanism $\mathcal{S}$ inserts SPs in the execution flow so that slaves perform the appropriate data exchanges. This mechanism is not incorporated within the self-scheduling algorithm, but it is an additional stand-alone component, applicable without further modifications. Given a self-scheduling algorithm $\mathcal{A}$, its synchronized version is denoted $\mathcal{S}$-$\mathcal{A}$. By enabling self-scheduling algorithms to be applicable to loops with dependencies, which was not the case before, the spectrum of applications is extended.

In addition, motivated by the results in [3,5,12], we define a *weighting* mechanism $\mathcal{W}$, aimed at improving the load balancing and, thus, the performance of all non-adaptive self-scheduling algorithms on non-dedicated heterogeneous systems. This mechanism is inspired from the approach used in [5], i.e., it uses the relative powers of the slaves combined with information regarding their run-queues to compute chunks. However, in contrast to previous approaches to chunk weighting, this mechanism is not embedded within the self-scheduling algorithm, but it is an external stand-alone component applicable to any dynamic algorithm without modifications. Given any self-scheduling algorithm $\mathcal{A}$, its weighted version will be called $\mathcal{W}$-$\mathcal{A}$.

Self-scheduling algorithms were usually implemented on distributed memory systems using the master–slave model. In this model, communication takes place only between the master and the slaves. The existence of dependencies necessitates communication among slaves, making the standard master-slave model inadequate. Therefore, an extended version of the master–slave model is required in the synchronized version of the self-scheduling algorithms. In the extended model communication among slaves is performed by direct data exchanges from slave to slave, and not through the master. In this approach, the master has a global view of the system's load and decides upon allocating the tasks to each slave.

One of the goals of this work was to develop algorithm independent mechanisms that can be simultaneously applied to any self-scheduling algorithm, for the dynamic load balancing of loops with dependencies. The synchronization and weighting mechanisms can be regarded individually as stand-alone components. The two mechanisms can be combined and applied in a single step to any self-scheduling scheme. This is illustrated in Fig. 1, which shows that starting with the classic master–slave model (see Fig. 1(a)), and applying the synchronization mechanism, the model is augmented with inter-slave communication links and the synchronization component $\mathcal{S}$ (see Fig. 1(c)). The communication is handled by the $\mathcal{S}$ component. Similarly, by applying the weighting mechanism, the model is augmented with a stand alone weighting component $\mathcal{W}$ (see Fig. 1(d)). Fig. 1(b) depicts the case where the weighting capability is embedded within the self-scheduling algorithm. Finally, by applying the combination of the two mechanisms, the master–slave model is augmented by the two mechanism-related stand alone components $\mathcal{S}$ and $\mathcal{W}$, and with inter-slave communication links, as illustrated in Fig. 1(e).

The contributions of this paper are summarized in Table 1. In order to confirm that the synchronization mechanism is general and efficient, it was applied to all significant self-scheduling algorithms: CSS, GSS, FSS and TSS, hence obtaining $\mathcal{S}$-CSS, $\mathcal{S}$-GSS, $\mathcal{S}$-FSS and $\mathcal{S}$-TSS. Furthermore, each of these algorithms was compared with the synchronized and weighted $\mathcal{SW}$-TSS (originally presented in [7] under the name of DMPS(DTSS)), verifying once more the superiority of $\mathcal{SW}$-TSS over the synchronized-only schemes. Subsequently, the weighting mechanism was applied to CSS, GSS and FSS, yielding $\mathcal{W}$-CSS, $\mathcal{W}$-GSS, $\mathcal{W}$-FSS and $\mathcal{W}$-TSS and their performance was compared. In all cases, the weighted algorithm significantly outperformed its corresponding non-weighted algorithm. Finally, both mechanisms were combined and the resulting $\mathcal{SW}$-CSS, $\mathcal{SW}$-GSS, $\mathcal{SW}$-FSS, $\mathcal{SW}$-TSS algorithms were compared against their synchronized-only version, i.e., $\mathcal{S}$-CSS, $\mathcal{S}$-GSS, $\mathcal{S}$-FSS, $\mathcal{S}$-TSS. Again, every weighted

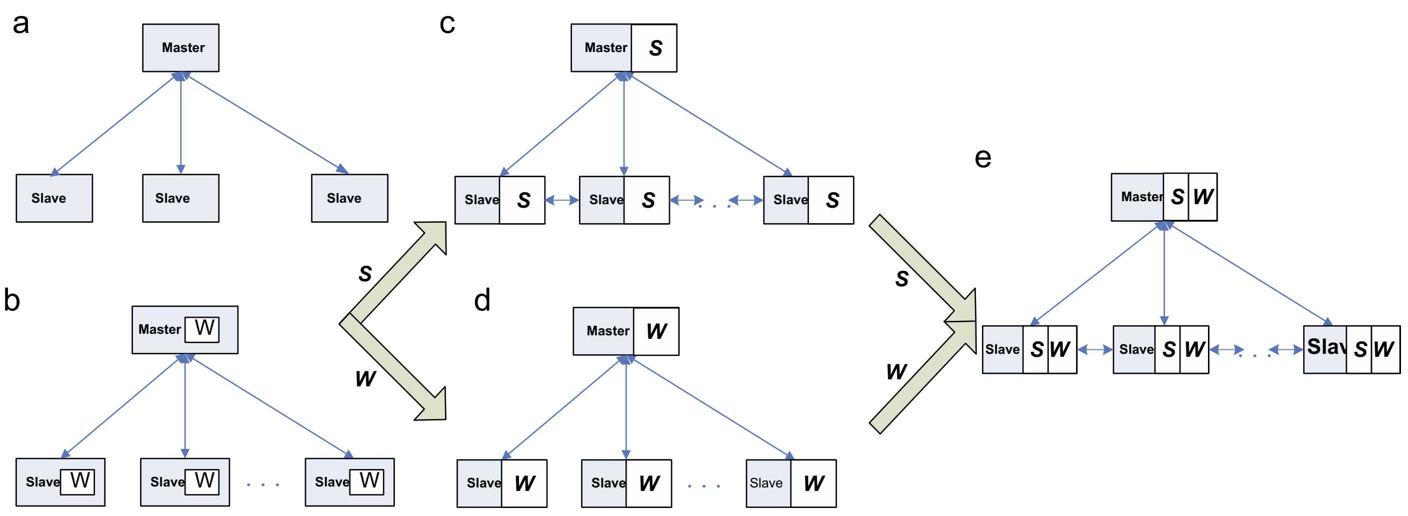


Fig. 1. The master–slave model with synchronization and/or weighting component(s).

Table 1
The contributions of this paper

Self-scheduling algorithm & reference number	Synchronized algorithm	Weighted algorithm	Synchronized & weighted algorithm
CSS [14]	$\mathcal{S}$-CSS presented in [7]	$\mathcal{W}$-CSS	$\mathcal{SW}$-CSS
FSS [13]	$\mathcal{S}$-FSS	$\mathcal{W}$-FSS other weighted approaches exist in [3,12]	$\mathcal{SW}$-FSS
GSS [21]	$\mathcal{S}$-GSS	$\mathcal{W}$-GSS	$\mathcal{SW}$-GSS
TSS [23]	$\mathcal{S}$-TSS new notation for DMPS(TSS) [7]	$\mathcal{W}$-TSS new notation for DTSS [5]	$\mathcal{SW}$-TSS new notation for DMPS(DTSS) [7]

algorithm outperformed its non-weighted version. From the tests and results obtained in this paper, one can conclude that loops with dependencies can be efficiently scheduled with any self-scheduling algorithm.

The paper is organized as follows. Section 2 gives a brief introduction of existing self-scheduling algorithms. The synchronization mechanism proposed in this paper is given in Section 3. The synchronization mechanism is presented along with its application to the existing self-scheduling algorithms. Moreover, this section includes a study of the impact of the number of SPs on the performance of the synchronized algorithms. The weighting mechanism is presented in Section 4, which shows how self-scheduling algorithms can be enhanced with this mechanism in order to achieve better load balancing. Next, the combination of the two aforementioned mechanisms for better performance is given in Section 5. Finally, the experiments and results are given in Section 6. Section 6.1 contains a short description of the three test cases we used in our experiments. The experimental setup and the three series of experiments are presented and interpreted in the same section. We conclude the paper and present future work in Section 7.

2. Overview of existing self-scheduling algorithms

In this section we give a brief overview of existing self-scheduling algorithms, as they were originally proposed for applications with independent tasks. We consider a distributed system consisting of a master (computer) and m slaves (computers). We use the following notations throughout this article:

- $P_1, \dots, P_m$ are slaves.
- VP_k is the virtual computing power of slave P_k.[1]
- $VP = \sum_{k=1}^{m} VP_k$ 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.
- $A_k = \left\lfloor \frac{VP_k}{Q_k} \right\rfloor$ is the available computing power (ACP) of P_k.
- $A = \sum_{k=1}^{m} A_k$ is the total ACP of the cluster. In contrast to VP, A varies over times depending on load fluctuations.

[1] The virtual power for each machine type can be established as the normalized execution time of the same test suite on each machine type.

- N is the number of scheduling steps, $i = 1, \ldots, N$, for a given algorithm.
- A few consecutive iterations of the loop are called a *chunk*; C_i is the chunk size at the ith scheduling step.

A nested loop is modeled as an n-dimensional Cartesian space J ($J \subset Z^n$), called *index space*, where n is the depth of the loop nest. Each point of this n-dimensional index space is a distinct iteration of the loop body. Without loss of generality we assume that for every index point $(u_1, \ldots, u_n)$ it holds that $1 \leqslant u_i \leqslant U_i$, $1 \leqslant i \leqslant n$.

We can now give an overview of the following self-scheduling schemes: CSS, GSS, FSS, TSS and DTSS, where we assume that the master–slave model is used and the slaves are assigned the iterations to be executed by the master.

CSS [14] assigns constant size chunks to each slave, i.e., $C_i = constant$. The chunk size is chosen by the user. If $C_i = 1$ then CSS is the so-called (pure) self-scheduling. A large chunk size reduces scheduling overhead, but also increases the chance of load imbalance, due to the difficulty to predict an optimal chunk size. As a compromise between load imbalance and scheduling overhead, other schemes start with large chunk sizes in order to reduce the scheduling overhead and reduce the chunk sizes throughout the execution to improve load balancing. These schemes are known as reducing chunk size algorithms and their difference lies in the choice of the first chunk and the computation of the decrement.

In the GSS [21] scheme, each slave is assigned a chunk given by the number of remaining iterations divided by the number of slaves, i.e., $C_i = R_i/m$, where R_i is the number of remaining iterations. Assuming that the loop which is scheduled with GSS is the r-th loop, where $1 \leqslant r \leqslant n$, then R_0 is the total number of iterations, i.e., $|U_r|$, and $R_{i+1} = R_i - C_i$, where $C_i = \lceil R_i/m \rceil = \left\lceil \left(1 - \frac{1}{m}\right)^i \cdot \frac{|U_r|}{m} \right\rceil$. This scheme initially assigns large chunks, which implies reduced communication/scheduling overheads in the beginning. At the last steps small chunks are assigned to improve the load balancing, at the expense of increased communication/scheduling overhead.

The TSS [23] scheme linearly decreases the chunk size C_i. The first and last (assigned) chunk size pair (F, L) may be set by the programmer. A conservative selection for the (F, L) pair is: $F = \frac{|U_r|}{2*m}$ and $L = 1$, where m is the number of slaves. 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 a large first chunk. Still many synchronizations may occur. One can improve this by choosing $L > 1$. The proposed number of steps needed for the scheduling process is $N = \frac{2 \times |U_r|}{(F+L)}$. Thus, the decrement between consecutive chunks is $D = (F - L)/(N - 1)$, and the chunk sizes are $C_1 = F, C_2 = F - D, C_3 = F - 2 \times D, \ldots, C_N = F - (N - 1) \times D$. TSS improves GSS for loops with varying tasks sizes, as it is explained in detail in [18].

The FSS [3] scheme schedules iterations in batches of m equal chunks. In each batch, a slave is assigned a chunk size given by a subset of the remaining iterations (usually half) divided by the number of slaves. The chunk size in this case is $C_i = \left\lceil \frac{R_i}{\alpha * m} \right\rceil$ and $R_{i+1} = R_i - (m \times C_i)$, where the parameter α is computed (by a probability distribution) or is sub-optimally chosen $\alpha = 2$. The weakness of this scheme is the difficulty to determine the optimal parameters. However, tests show [3] improvement on previous adaptive schemes (possibly) due to fewer adaptations of the chunk-size.

DTSS [5,6] 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 slave. Every process running on a slave is assumed to take an equal share of its computing resources. The application programmer may determine the pair (F, L) according to TSS, or the following formula may be used in the conservative selection approach: $F = \frac{|U_r|}{2 \times A}$ and $L = 1$ (assuming that the loop which is scheduled by DTSS is the r-th loop). The total number of steps is $N = \frac{2 \times |U_r|}{(F+L)}$ 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: $S_{k-1} = A_1 + \cdots + A_{k-1}$. When all slaves are dedicated to a single user job then $A_k = VP_k$. Also, when all slaves have the same speed, then $VP_k = 1$ 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 the slave's ACP. Hence, faster slaves get more loop iterations than slower ones. In contrast, TSS simply treats all slaves in the same way.

3. The synchronization mechanism

The self-scheduling schemes described in the previous section are applicable to loops without dependencies. Nested loops, however, fall in two categories: parallel and dependence loops. Parallel loops have no dependencies among iterations and, thus, their iterations can be executed in any order or even simultaneously. In dependence loops the iterations depend on each other, hence, imposing a certain execution order, according to the existing dependencies. The loop body may contain general program statements that include **if** blocks and other **for** or **while** loops. We assume that the nested loop has p uniform dependencies. These are modeled by dependence vectors and their set is denoted $DS = \{\vec{\mathbf{d}}_1, \ldots, \vec{\mathbf{d}}_p\}$.

The purpose of the synchronization mechanism is to enable self-scheduling algorithms to handle loops with dependencies. Recall that self-scheduling algorithms are devised for parallel loops and as such do not provide inter-slave communication. In [7] it was shown that CSS, TSS and DTSS can be applied to dependence loops by *SPs* to compensate for this deficiency. In this paper we generalize this approach and define a synchronization mechanism applicable to all self-scheduling algorithms. In all cases the master assigns chunks to slaves which synchronize with each other at *SPs*. We must emphasize that the synchronization mechanism is completely independent of the self-scheduling algorithm and does not enhance the load balancing capability of the algorithm. Therefore, synchronized self-scheduling algorithms perform well on heterogeneous systems only if the self-scheduling algorithm itself explicitly takes into account the heterogeneity. The synchronization overhead

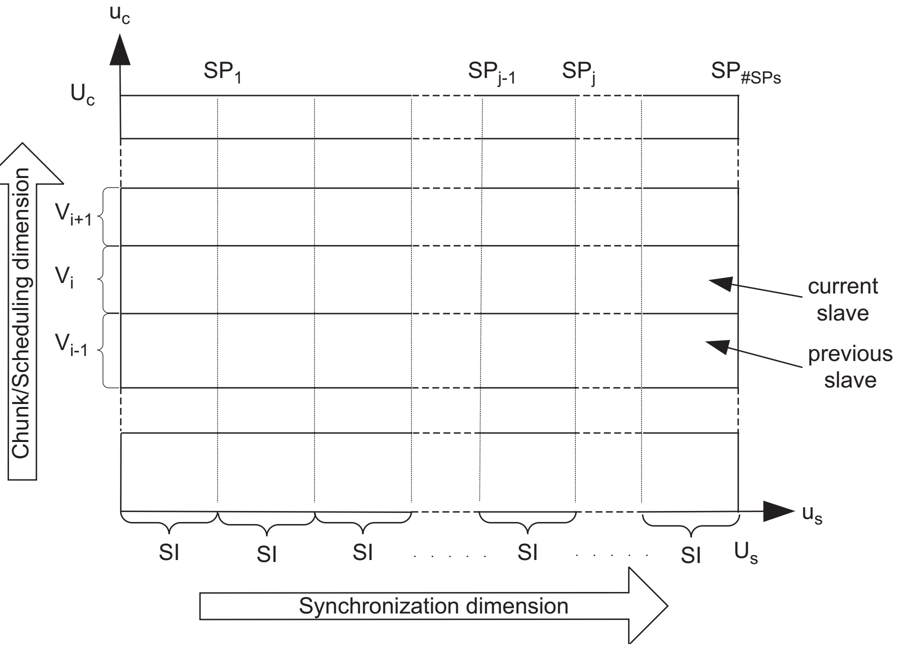


Fig. 2. Partitioning of a 2D loop into chunks, and placement of synchronization points.

is compensated by the increase of parallelism resulting in an overall performance improvement.

To describe formally the synchronization mechanism we use the following additional notations:

- $\mathcal{A}$ stands for any self-scheduling algorithm.
- #—denotes the 'number of'.
- $u_s = 1, \ldots, U_s$ designates the *synchronization* dimension, along which the synchronization mechanism is applied. Without loss of generality we choose as u_s the maximal dimension of the index space; it makes sense to map consecutive rows along the longest dimension to the same processor since it saves the communication overhead along that dimension.
- $u_c = 1, \ldots, U_c$ denotes the *chunk* dimension, which will be divided into chunks according to a self-scheduling algorithm (the weighting mechanism is applied on this dimension). In practice, u_c is taken as the second largest dimension of the problem; doing so enables more processors to be employed therefore enhancing the degree of parallelism.
- V_i is the length of the projection of the chunk i on the chunk dimension u_c. We note that $C_i = V_i \times \frac{\prod_{j=1}^{n} U_j}{U_c}$.
- SP_j stands for SP j, where $j = 1, 2, \ldots \#SPs$ is the number of SPs.
- The synchronization mechanism inserts $\#SPs$ SPs in each chunk, uniformly distributed along u_s.
- $SI = \left\lceil \frac{U_s}{\#SPs} \right\rceil$ is the interval between two SPs and it is the same for every chunk.
- The set of iterations of chunk i, between SP_{j-1} and SP_j is called subchunk $SC_{i,j}$.
- The *current* slave is the slave assigned the latest chunk i, whereas the *previous* slave is the slave assigned with the chunk $i-1$. This information is needed only by the master.
- The *send-to* is the slave id to which P_k must send computed data to. The *receive-from* is the slave id from which P_k must receive data, in order to begin its current computation. These ids are communicated to P_k by the master, based on the *current* and *previous* slave ids.

The synchronization mechanism $\mathcal{S}$ provides the synchronization interval along u_s

$$SI = \left\lceil \frac{U_s}{\#SPs} \right\rceil \tag{1}$$

and a framework for inter-slave communication. Fig. 2 illustrates V_i and SI. The horizontal strip sections are assigned to single slaves. Synchronization points are placed in the u_s dimension so that other slaves can start computing as soon as possible. Note that in this example, C_i is the number of loop iterations in the horizontal strip, i.e., $C_i = V_i \times U_s$.

3.1. Implementation details related to the mechanism $\mathcal{S}$

In Fig. 3 chunks $i-1, i, i+1$ are assigned to slaves P, P_k, P_{k+1}, respectively. The shaded areas denote sets of iterations that are computed concurrently on different slaves. When P_k reaches the SP SP_{j+1}, after computing $SC_{i,j+1}$, it sends to P_{k+1} the data that P_{k+1} needs to begin the execution of $SC_{i+1,j+1}$, called *communication set*. P_k sends to P_{k+1} data only for those iterations of $SC_{i,j+1}$ on which the iterations of $SC_{i+1,j+1}$ depend on. Communication is incurred only by those dependence vectors that cross chunks. Note that no dependence vector can span more than two chunks.

Slave P_k receives from P_{k-1} the data P_k requires in order to proceed with the execution of $SC_{i,j+2}$. Note that slaves do not reach the same SP 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 SPs leads to a *wavefront* execution, as shown in Figs. 3 and 5. The choice of SI plays a crucial role in the parallel performance. It should be chosen so as to minimize

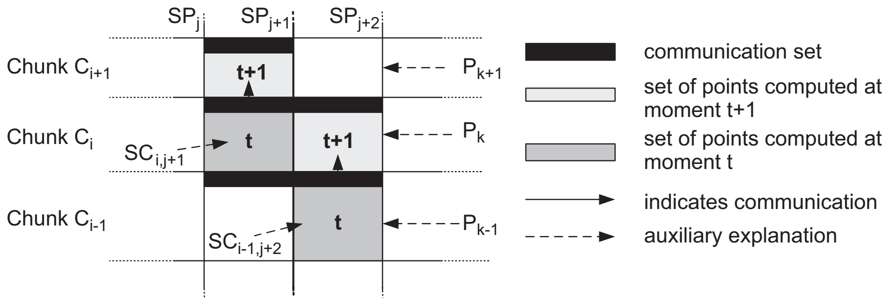
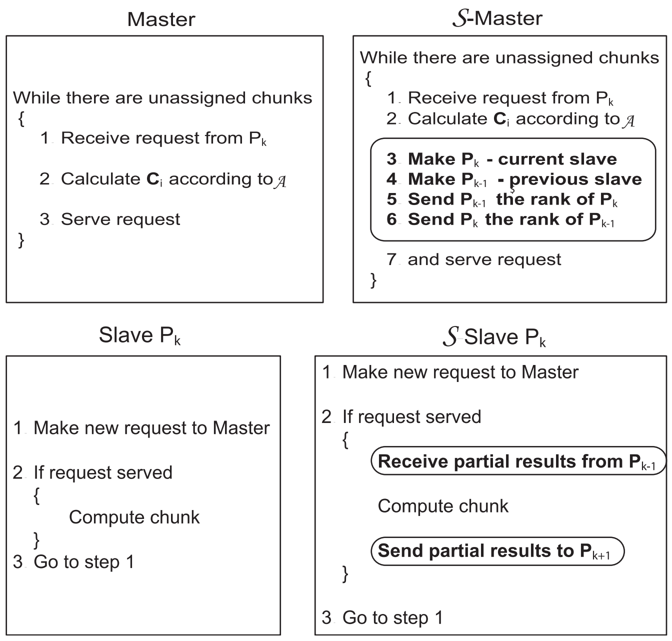


Fig. 3. The synchronization mechanism.


```
Master
While there are unassigned chunks
{
    1 Receive request from P_k
    2 Calculate C_i according to 𝒜
    3 Serve request
}

𝒮-Master
While there are unassigned chunks
{
    1 Receive request from P_k
    2 Calculate C_i according to 𝒜
    3 Make P_k - current slave
    4 Make P_{k-1} - previous slave
    5 Send P_{k-1} the rank of P_k
    6 Send P_k the rank of P_{k-1}
    7 and serve request
}

Slave P_k
1 Make new request to Master
2 If request served
    {
        Compute chunk
    }
3 Go to step 1

𝒮 Slave P_k
1 Make new request to Master
2 If request served
    {
        Receive partial results from P_{k-1}
        Compute chunk
        Send partial results to P_{k+1}
    }
3 Go to step 1
```

Fig. 4. The master–slave model with synchronization mechanism.

the total makespan and maintain the $\frac{communication}{computation}$ ratio well below one. The wait time in the synchronization step equals $\left[\frac{U_s}{\#SP} \times \text{Time of operations in subchunk}(V_i \times SI) + \text{time for}\right.$ data transmit to *send-to* slave].

3.2. *Enhancing self-scheduling algorithms via synchronization*

Mechanism $\mathcal{S}$ adds three components to the original algorithm $\mathcal{A}$ (see Fig. 4): (1) **transaction accounting** (master side)—according to slaves' requests, the master extracts information and decides upon the identity of the slaves participating in a data exchange (*previous* and *current* slaves); (2) **receive part** (slave side)—uses the information from (1) to receive the corresponding communication sets, and (3) **transmit part** (slave side)—uses the information from (1) to send the corresponding communication sets. Details regarding components (2) and (3) are given in the pseudocode Algorithm 1. The flexibility of the synchronization mechanism is twofold:

(1) P_k checks for the identity of the *send-to* slave at every SP. Suppose that P_k first learns the identity of the *send-to* slave P_{k+1} at SP SP_j. In this case, P_k sends to P_{k+1} all the locally stored data that P_{k+1} requires in one step because it is more efficient to send one large message than sending the same amount of data in smaller consecutive messages. Hence, P_{k+1} is not delayed or stalled at any SP up to SP_j. Moreover, the fact that P_k is j SPs ahead of P_{k+1}, where $2 \leqslant j \leqslant \#SPs$, means that P_{k+1} receives from P_k the data it needs because they have already been computed.

(2) In the extreme case where no *send-to* is designated by the time P_k reaches the last SP, then P_k stores all the data it should have sent to the *send-to* slave in a local buffer. When the master designates the *send-to* slave P_{k+1}, then P_k sends to P_{k+1} all stored data. Note that in this case P_k could also be the *send-to* slave.

We discuss next the implementation of the synchronization mechanism. The issues to be addressed are: (a) the placement of SPs along the dimension u_s and (b) the transmission of data to the adjacent slave. The SPs are computed as follows: $SP_1 = SI$ and $SP_{i+1} = SP_i + SI$. SI depends on $\#SP$, which is determined empirically, as demonstrated in the experiments of Section 6, or selected by the user. The pseudocode of Algorithm 1 provides the basis for two C-code blocks that handle the transmission part. The code blocks are inserted in the slave code, in the positions indicated in Fig. 4. Before "Compute chunk" we insert the block responsible for handling the reception part of the communication, and after "Compute chunk" we insert the block that handles the send part. We have assumed that the outer dimension of the loop is the synchronization dimension. This implies that the receive and send code blocks are inserted between the loop iterating over the synchronization dimension and before the loop iterating over the chunk dimension. Whenever the index of synchronization dimension is iterated SI times, these blocks are activated in order for the data exchange event to occur. On the master side, a code block is inserted so as whenever a new slave makes a request, it is registered as the *current slave*, and the last registered slave is renamed as the *previous slave*. This information is then transmitted to the last two registered slaves. A preprocessor could be implemented for automatic parsing, detecting and inserting the appropriate code blocks. In our experiments we did this manually. In either case the initial code does not have to be rewritten. This also applies to the implementation of the $\mathcal{W}$ and the $\mathcal{SW}$ mechanisms.

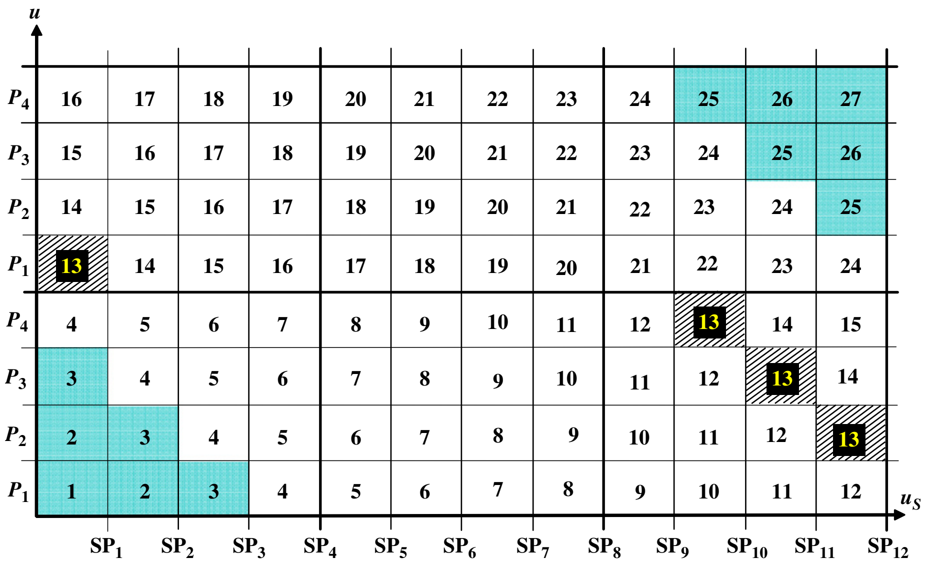


Fig. 5. Parallel execution in wavefront fashion.

3.3. Empirical determination of the appropriate #*SPs*

In order to understand the impact on performance of the choice of #*SPs*, we give in Fig. 5 the parallel execution on $m = 4$ slaves of a hypothetical example. For simplicity, assume that each slave is assigned only two chunks and that all chunks are of the same size. The slave request order, i.e., P_1, P_2, P_3, P_4, and the SPs, $SP_1, \ldots, SP_{12}$, inserted by the synchronization mechanism, are also shown. The numbers in each subchunk $SC_{i,j}$ indicate the time step in which it is executed. The flow of execution follows a wavefront fashion. During time steps 1–3 (*initial time steps*) and 25–27 (*final time steps*), denoted by the gray areas, one can see that not all slaves are active, whereas during steps 4 to 24 (*intermediate time steps*) all slaves are active. At the end of the 12th time step, slave P_1 has completed its first chunk and it is then assigned a second chunk by the master. It begins computing the second chunk at time step 13, since it has all the necessary data from slave P_4. In the same time step, slaves P_2, P_3, P_4 are still computing their first chunk. As soon as slave P_2 completes its first chunk (at the end of time step 13), it proceeds with its next chunk at time step 14. The transition to the second chunk for P_3 and P_4 takes place at time steps 14 and 15, respectively. In other words, except for the initial and the final time steps, the execution proceeds with no delays, apart from the synchronization between slaves, as imposed by the SPs. The *#initial steps* is $(m - 1)$ and is equal to the *#final steps*, while *#intermediate steps* = *#total steps* $- 2(m - 1)$, as illustrated in Fig. 5. The number of total steps depends on the number of *SPs* and the number of chunks produced by the scheduling algorithm. Since the number of chunks is algorithm dependent, the choice of the number of *SPs* should maximize the percentage of intermediate time steps over the total time steps.

The selection of the number of *SPs* is a tradeoff between synchronization overhead and parallelism. A choice of a large #*SPs* incurs too frequent data exchanges and a high synchronization overhead, whereas a small #*SPs* restricts the inherent parallelism. We believe that the optimal selection of the synchronization interval depends on many factors, such as: the dependencies of the loop, the characteristics of the underlying communication network and of the processors, and the self-scheduling algorithm used. Extensive experimental runs for various test cases and self-scheduling algorithms (contained in Section 6) show that a good, albeit arbitrary, choice is $\#SPs \geqslant 3 * m$, where m is the number of slaves. In our example (Fig. 5), $\#SPs = 12$, which yields that *#total steps* = 27, *#intermediate steps* = 21, and a percentage of 77% of execution without idle times.

4. The weighting mechanism

The weighting mechanism is used to enhance self-scheduling algorithms to account for load variations and cluster heterogeneity. A brief review of similar attempts and how the current approach differs from them was given in the Introduction. To achieve good load balancing in a heterogenous environment, chunks should be computed according to the current run-queue state of each slave. The run-queue state of a slave is not always the same for every chunk assignment. For this reason we define a weighting mechanism, applicable to any self-scheduling algorithms, e.g., CSS [14], GSS [21], TSS [23], FSS [13], such that the resulting chunk sizes are adjusted according to the current load and the available computational power of each slave.

Suppose a self-scheduling algorithm $\mathcal{A}$ is used to parallelize a nested loop on a heterogenous cluster of m slaves, each with $VP_1, \ldots, VP_m$ virtual computational power. Furthermore, assume that during the i-th scheduling step, slave P_k has Q_k processes in its run-queue. The weighting mechanism $\mathcal{W}$, calculates the chunk $\widehat{C_i}$ assigned to P_k, using the following formula:

$$\widehat{C_i} = C_i \times \frac{VP_k}{Q_k}. \tag{2}$$

```
procedure RECEIVE PART    ▷ Receive partial results from $P_{k-1}$
    in synchronization point $SP_i$ check for partial results from previous slave $P_{k-1}$
    if current and previous slaves are the same, that is, $P_k = P_{k-1}$ then
        all partial results exist in local memory
        proceed to the computation without blocking in any $SP$ for the completion of current chunk
    else
        receive partial results from $P_{k-1}$
        check the number of communication sets $b$ of partial results received
        if $b > 1$ then
            skip the next $b$ $SPs$
        end if
    end if
    proceed to computation
end procedure
```

```
procedure TRANSMIT PART    ▷ Send partial results to $P_{k+1}$
    in synchronization point $SP_i$
    if $SP$ reached is $SP_1$ then
        make a non-blocking request to master for the rank of the next slave $P_{k+1}$
        if the rank of slave $P_{k+1}$ is not yet known to master then
            store partial results in local memory
            proceed to receive part
        else
            get the rank of $P_{k+1}$
            send it partial results
        end if
    else
        if the rank of $P_{k+1}$ is already known by $P_k$ then
            send $P_{k+1}$ the partial results
        else
            if a reply has been received by the master for the rank of the next slave $P_{k+1}$ then
                send $P_{k+1}$ all previous partial results in a single packet
                proceed to receive part
            else
                store partial results in local memory
                proceed to receive part
            end if
        end if
    end if
end procedure
```

Algorithm 1. Pseudocode of the communication scheme implementation on the slave side.

In the above formula VP_k and Q_k are the virtual power and number of processes in the run-queue of slave P_k and C_i is the chunk size given by the original self-scheduling algorithm $\mathcal{A}$. Hence, $\widehat{C_i}$ is the "weighted" chunk size, given the current load conditions of P_k. In most cases the addition of the $\mathcal{W}$ mechanism improves the performance. However, when the loop is run on a dedicated homogeneous cluster, the $\mathcal{W}$ mechanism does not improve the performance and could be omitted.

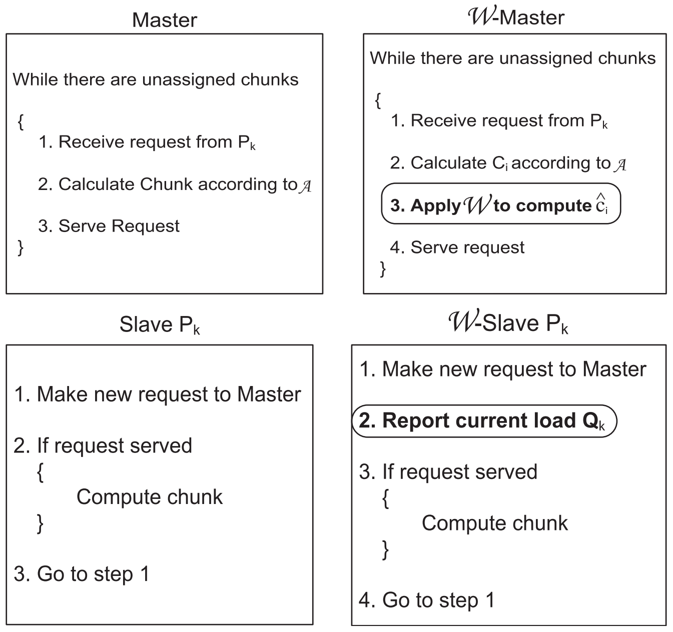

Master
While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate Chunk according to 𝒜
3. Serve Request
}
𝒲-Master
While there are unassigned chunks
{
1. Receive request from P_k
2. Calculate C_i according to 𝒜
3. Apply 𝒲 to compute Ĉ_i
4. Serve request
}
Slave P_k
1. Make new request to Master
2. If request served
{
Compute chunk
}
3. Go to step 1
𝒲-Slave P_k
1. Make new request to Master
2. Report current load Q_k
3. If request served
{
Compute chunk
}
4. Go to step 1

Fig. 6. The master–slave model with weighting mechanism.

4.1. Enhancing self-scheduling algorithms via weighting

Fig. 6 shows the effect of the weighting mechanism on a self-scheduling algorithm $\mathcal{A}$. The mechanism adds two components to the original algorithm: (1) **chunk weighting** (master side)—the master adjusts the chunk size based on the slave's load information and computational power, and (2) **run-queue monitor** (slave side)—it keeps track of the number of processes that require CPU time, updates Q_k and informs the master of its current load.

Table 2 shows the chunk sizes given by the original and weighted self-scheduling algorithms. These chunks were obtained for a parallel loop (Mandelbrot computation)[2] with an index space of $10\,000 \times 10\,000$ points. Four slaves were used, having virtual computing powers $VP_1 = 1, VP_2 = 0.8, VP_3 = 1$ and $VP_4 = 0.8$. The two slowest slaves were loaded with an extra process, i.e., $Q_2 = 2, Q_4 = 2$, and their ACP halved: $A_2 = 0.4$ and $A_4 = 0.4$.

Table 2 also shows the order in which slaves requested work from the master, which differs from algorithm to algorithm. Self-scheduling algorithms were devised for homogenous systems and they tend to assign large initial chunks to all slaves. They make the assumption that all slaves compute their assigned chunk in roughly the same time and advance to the next chunk simultaneously, as it is explained in [18]. This assumption is not valid for heterogenous systems. Slower slaves may fall behind faster ones because they need more time to compute chunks of equal size. In most cases, slower

[2] The Mandelbrot test case is described in Section 6.1.

Table 2
Chunk sizes given by the original and weighted algorithms for the Mandelbrot set, index space size $|J| = 10\,000 \times 10\,000$ points and $m = 4$

$\mathcal{A}$	Chunk sizes with $\mathcal{A}$ with respect to the processors' request order	Chunk sizes with $\mathcal{W}$-$\mathcal{A}$ with respect to the processors' request order	Parallel time for $\mathcal{A}$ (s)	Parallel time for $\mathcal{W}$-$\mathcal{A}$ (s)
CSS	1250(P_1) 1250(P_2) 1250(P_3) 1250(P_4) 1250(P_3) 1250(P_1) 1250(P_3) 1250(P_1)	1250(P_1) 1250(P_3) 500(P_4) 500(P_2) 1250(P_3) 500(P_2) 500(P_4) 1250(P_1) 1250(P_3) 500(P_4) 1250(P_1)	120.775	66.077
FSS	1250(P_1) 1250(P_3) 1250(P_2) 1250(P_4) 625(P_3) 625(P_3) 625(P_1) 625(P_3) 390(P_1) 390(P_1) 390(P_3) 390(P_1) 244(P_3) 244(P_4) 244(P_1) 208(P_3)	1250(P_1) 1250(P_3) 500(P_2) 500(P_4) 812(P_3) 324(P_2) 324(P_4) 324(P_1) 324(P_3) 812(P_3) 630(P_1) 630(P_1) 630(P_4) 252(P_3) 176(P_1) 441(P_4) 441(P_2) 176(P_3) 123(P_1) 308(P_2) 308(P_4) 113(P_1)	120.849	56.461
GSS	2500(P_1) 1875(P_2) 1406(P_3) 1054(P_4) 791(P_3) 593(P_3) 445(P_3) 334(P_1) 250(P_3) 188(P_1) 141(P_3) 105(P_1) 80(P_3) 80(P_1) 80(P_3) 78(P_1)	2500(P_1) 1875(P_3) 562(P_2) 506(P_4) 455(P_4) 410(P_2) 923(P_3) 692(P_3) 519(P_1) 155(P_4) 140(P_2) 315(P_3) 94(P_4) 213(P_1) 160(P_3) 120(P_1) 90(P_3) 80(P_2) 80(P_1) 80(P_3) 31(P_1)	145.943	58.391
TSS	1250(P_1) 1172(P_3) 1094(P_2) 1016(P_4) 938(P_3) 860(P_1) 782(P_3) 704(P_1) 626(P_3) 548(P_4) 470(P_2) 392(P_1) 148(P_3)	1250(P_1) 1172(P_3) 446(P_2) 433(P_4) 1027(P_3) 388(P_4) 375(P_2) 882(P_1) 804(P_3) 299(P_4) 286(P_2) 660(P_1) 582(P_3) 504(P_1) 179(P_4) 392(P_3) 134(P_2) 187(P_1)	89.189	63.974

slaves requested work only once throughout the whole execution. The weighting mechanism compensates for this deficiency as shown by the request orders in Table 2. The gain of the weighted algorithms over the non-weighted ones is also demonstrated by the times of the parallel execution in the same table.

The code blocks that implement the weighting mechanism are much shorter than those of the synchronization mechanism, as it can be seen from Fig. 6. In particular, we insert a block in the slave's code that monitors the current load of the slave at the time it makes a new request for work to the master. This load is then reported to the master along with the new request. On the master side, the block code implementing $\mathcal{W}$ performs a multiplication of the chunk size produced by the original self-scheduling algorithm according to the slave's reported virtual power and current load, using formula (2).

5. The combined $\mathcal{SW}$ mechanisms

In Section 3 we applied the synchronization mechanism $\mathcal{S}$ to dependence loops whereas in Section 4 we applied the weighting mechanism $\mathcal{W}$ to parallel loops. In this section we combine the two mechanisms and demonstrate their effectiveness. The synchronization mechanism, while necessary to parallelize dependence loops, does not provide any load balancing. This makes advantageous the simultaneous use of the weighting mechanism in order to improve the overall performance.

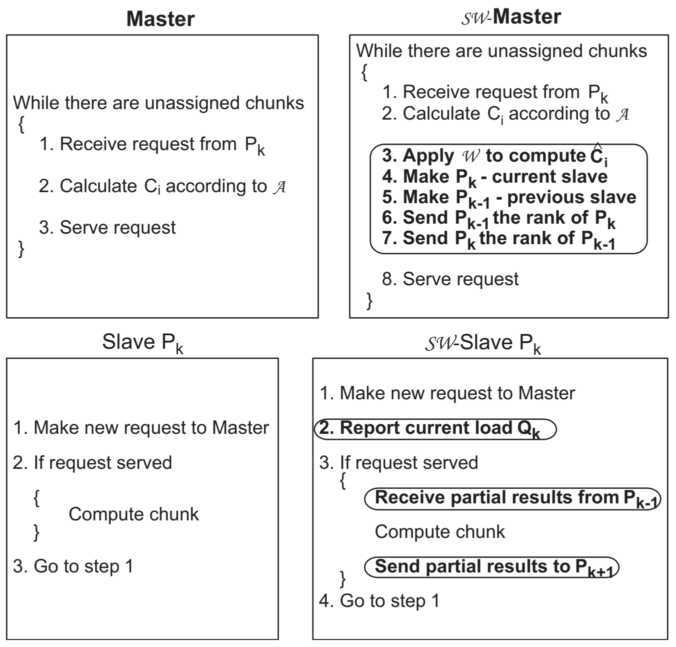

Master
While there are unassigned chunks {
1. Receive request from P_k
2. Calculate C_i according to $\mathcal{A}$
3. Serve request }
$\mathcal{SW}$-Master
While there are unassigned chunks {
1. Receive request from P_k
2. Calculate C_i according to $\mathcal{A}$
3. Apply $\mathcal{W}$ to compute $\hat{C}_i$
4. Make P_k - current slave
5. Make P_{k-1} - previous slave
6. Send P_{k-1} the rank of P_k
7. Send P_k the rank of P_{k-1}
8. Serve request }
Slave P_k
1. Make new request to Master
2. If request served { Compute chunk }
3. Go to step 1
$\mathcal{SW}$-Slave P_k
1. Make new request to Master
2. Report current load Q_k
3. If request served {
Receive partial results from P_{k-1}
Compute chunk
Send partial results to P_{k+1} }
4. Go to step 1

Fig. 7. The master–slave model with the combination of synchronization and weighting mechanisms.

Recall that the synchronization interval is the same for every chunk, meaning that the size of the communication sets is

Table 3
Chunk sizes given by the synchronized-only and synchronized–weighted algorithms for the Floyd–Steinberg loop, index space size $|J| = 10\,000 \times 10\,000$ points and $m = 4$

$\mathcal{A}$	Chunk sizes with $\mathcal{S}$-$\mathcal{A}$ with respect to the processors' request order	Chunk sizes with $\mathcal{SW}$-$\mathcal{A}$ with respect to the processors' request order	Parallel time for $\mathcal{S}$-$\mathcal{A}$ (s)	Parallel time for $\mathcal{SW}$-$\mathcal{A}$ (s)
CSS	1250(P_1) 1250(P_3) 1250(P_2) 1250(P_4) 1250(P_1) 1250(P_3) 1250(P_2) 1250(P_4)	1250(P_1) 1250(P_3) 500(P_2) 500(P_4) 1250(P_1) 1250(P_3) 500(P_2) 500(P_4) 1250(P_1) 1250(P_3) 500(P_2)	27.335	16.582
FSS	1250(P_1) 1250(P_3) 1250(P_2) 1250(P_4) 625(P_1) 625(P_3) 625(P_2) 625(P_4) 390(P_1) 390(P_3) 390(P_2) 390(P_4) 244(P_1) 244(P_3) 244(P_2) 208(P_4)	1250(P_1) 1250(P_3) 500(P_2) 500(P_4) 812(P_1) 812(P_3) 324(P_2) 324(P_4) 630(P_1) 630(P_3) 252(P_2) 252(P_4) 488(P_1) 488(P_3) 195(P_2) 195(P_4) 378(P_1) 378(P_3) 151(P_2) 151(P_4) 40(P_1)	27.667	16.556
GSS	2500(P_1) 1875(P_3) 1406(P_2) 1054(P_4) 791(P_1) 593(P_3) 445(P_2) 334(P_4) 250(P_1) 188(P_3) 141(P_2) 105(P_4) 80(P_1) 80(P_3) 80(P_2) 78(P_4)	2500(P_1) 1875(P_3) 562(P_2) 506(P_4) 1139(P_1) 854(P_3) 256(P_2) 230(P_4) 519(P_1) 389(P_3) 116(P_2) 105(P_4) 237(P_1) 178(P_3) 80(P_2) 80(P_4) 108(P_1) 81(P_3) 80(P_2) 80(P_4) 25(P_1)	28.526	18.569
TSS	1250(P_1) 1172(P_2) 1094(P_3) 1016(P_4) 938(P_1) 860(P_2) 782(P_3) 704(P_4) 626(P_1) 548(P_2) 470(P_3) 392(P_4) 148(P_1)	509(P_2) 1217(P_1) 464(P_4) 1105(P_3) 420(P_2) 995(P_1) 376(P_4) 885(P_3) 332(P_2) 775(P_1) 288(P_4) 665(P_3) 244(P_2) 555(P_1) 200(P_4) 445(P_3) 156(P_2) 335(P_1) 34(P_4)	25.587	14.309

Table 4
Problem sizes for Floyd–Steinberg and Hydro test cases

Problem size	Small	Medium	Large
Floyd–Steinberg	$5000 \times 15\,000$	$10\,000 \times 15\,000$	$15\,000 \times 15\,000$
Upper/lower threshold	500/10	750/10	1000/10
Hydro	$5000 \times 5 \times 10\,000$	$7500 \times 5 \times 10\,000$	$10\,000 \times 5 \times 10\,000$
Upper/lower threshold	500/10	750/10	1000/10

constant which in turn implies that the communication time at any two SPs is approximately the same for every processor. Similarly, since the size of the chunk is weighted according to the requesting processor's run-queue state, it is expected that the computation time of a subchunk is approximately the same for every processor. This yields a constant $\frac{communication}{computation}$ ratio, which results in a good load balancing. It is therefore advantageous to apply the weighting mechanism *in addition* to the synchronization mechanism. It is obvious that a self-scheduling algorithm with both synchronization and weighting will outperform the same self-scheduling algorithm without weighting in a heterogeneous system.

Fig. 7 shows the effects of both synchronization and weighting mechanisms for the scheduling of dependence loops on heterogenous systems. As with the previous cases, the combined $\mathcal{SW}$ mechanisms add two components to the master: (1) **chunk weighting** and (2) **transaction accounting**, and three components to the slave: (3) **run-queue monitor**, (4) **receive part** and (5) **transmit part**. Component (1) (master-side) along with component (3) (slave-side) are related to the weighting mechanism, whereas all other components (both from master and slave) belong to the synchronization mechanism.

The chunk sizes of the synchronized-only and synchronized–weighted algorithms for a dependence loop (Floyd–Steinberg computation)[3] with an index space of $10\,000 \times 10\,000$ points are shown in Table 3. Notice that due to the existing de-

[3] The Floyd–Steinberg test case is described in Section 6.1.

pendencies and synchronization, the slaves request order stays the same for a particular algorithm. The chunk sizes differ from the ones in Table 2, where no synchronization was used and the slaves request order was random. Again, from the parallel times in Table 3 one can see that the synchronized–weighted algorithms perform better than the synchronized-only ones.

The implementation of the combination of the two mechanisms is actually the insertion of all code blocks associated with each of the mechanisms, both in the code of the slave and the code of the master, as it can be seen in Fig. 7.

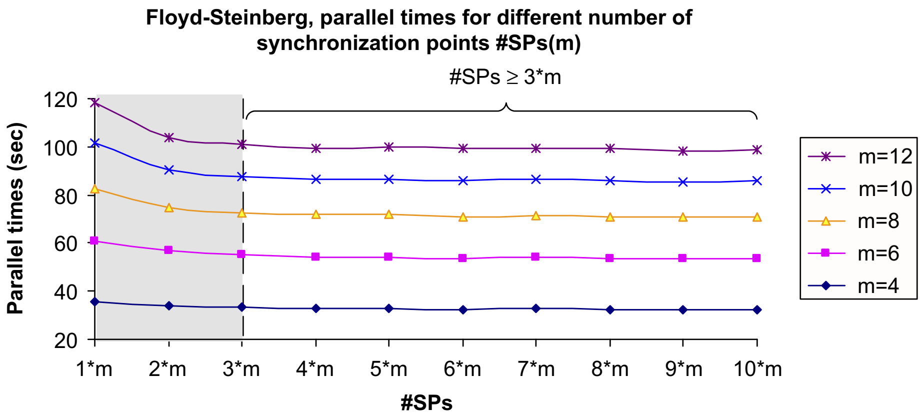
6. Experiments and results

6.1. Test cases

The first test case we use in this paper is a Mandelbrot set generator [17], a type of fractal model generator. Fractal models are used in many supercomputing applications. They are useful for predicting systems that demonstrate chaotic behavior. The Mandelbrot set is obtained from the quadratic recurrence equation $z_{n+1} = z_n^2 + C$, with $z_0 = C$, where points C in the complex plane for which the orbit of z_n does not tend to infinity are in the set. Setting z_0 equal to any point in the set gives the same result.
The Mandelbrot set was originally called a μ molecule by Mandelbrot [17]. This application is a real life example that has no loop iteration dependencies but it is considered herein because the loop iterations tasks are highly irregular in size. The pseudocode of the Mandelbrot fractal computation is given below:

```
/* Mandelbrot */
   for (hy=1; hy<=hyres; hy++)  { /* chunk dimension */
      for (hx=1; hx<=hxres; hx++)  {
         cx = (((float)hx)/((float)hxres)-0.5)/
         magnify*3.0-0.7;
        cy = (((float)hy)/((float)hyres)-0.5)/magnify*3.0;
         x = 0.0; y = 0.0;
       for (iteration=1; iteration<itermax; iteration++)
{
            xx = x*x-y*y+cx;
            y = 2.0*x*y+cy;
            x = xx;
            if (x*x+y*y>100.0)  iteration = 999999;
         }
         if (iteration<99999)  color(0,255,255);
         else color(180,0,0);
      }
   }
```

The second test case is the Floyd–Steinberg computation [9], an image processing algorithm used for the error-diffusion dithering of a *width* by *height* grayscale image. The pseudocode is



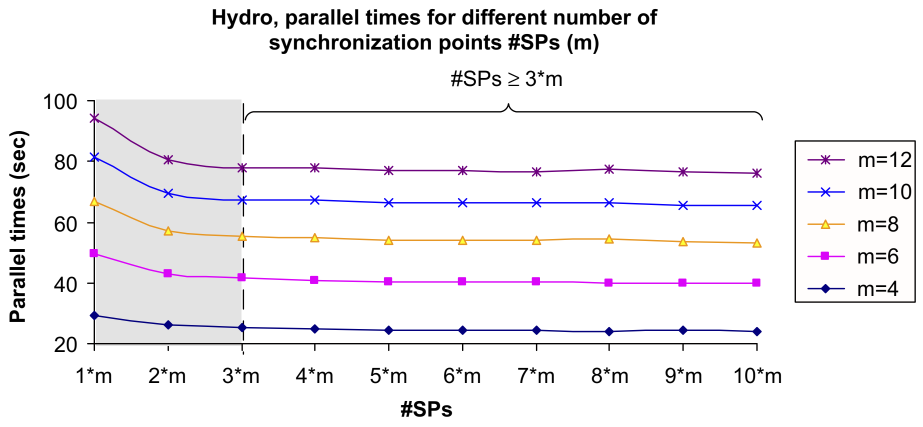


Fig. 8. Impact of #SPs on the parallel execution times for Floyd–Steinberg and Hydro case studies.

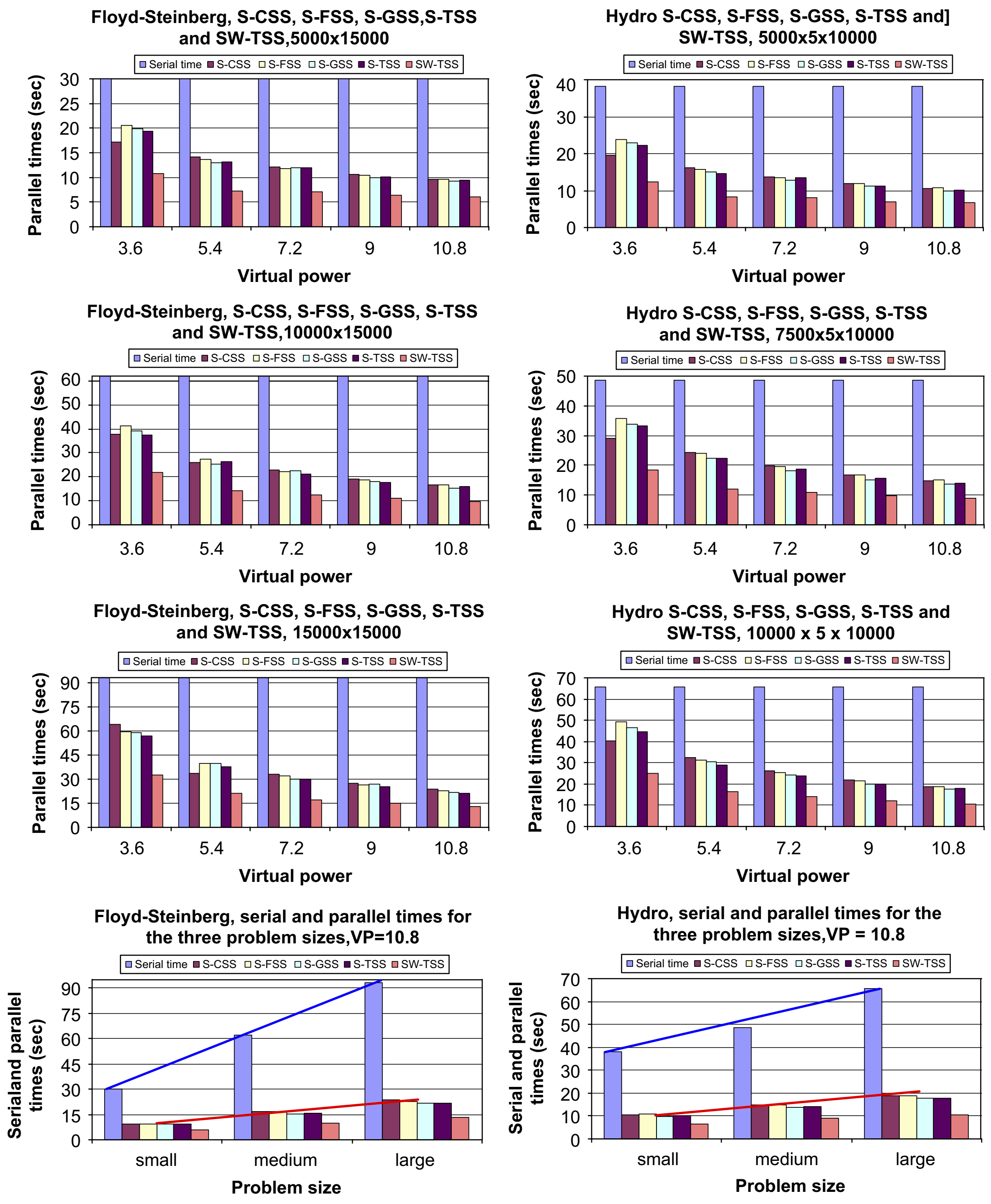


Fig. 9. Parallel times of the synchronized-only algorithms for Floyd–Steinberg and Hydro case studies.

given below:

```
/* Floyd--Steinberg */
for (i=0; i<width; i++){ /* synchronization dimension */
   for (j=0; j<height; j++){ /* chunk dimension */
         I[i][j] = trunc(J[i][j]) + 0.5;
         err = J[i][j] - I[i][j]*255;
         J[i][j+1] += err*(7/16);
         J[i+1][j-1] += err*(3/16);
         J[i+1][j] += err*(5/16);
         J[i+1][j+1] += err*(1/16);
   }
}
```

The third test case is a modified version of the Livermore kernel 23-Hydro (Implicit Hydrodynamics fragment) [19], widely used in hydrodynamics. It is a 3-dimensional

Table 5
Speedups for Floyd–Steinberg and Hydro test cases

Test case	VP	$\mathcal{S}$-CSS	$\mathcal{S}$-FSS	$\mathcal{S}$-GSS	$\mathcal{S}$-TSS	$\mathcal{SW}$-TSS
Floyd–Steinberg	3.6	1.45	1.57	1.59	1.63	2.86
	5.4	2.76	2.35	2.33	2.47	4.35
	7.2	2.81	2.92	3.09	3.10	5.39
	9	3.41	3.50	3.49	3.70	6.27
	10.8	3.95	4.07	4.27	4.34	7.09
Hydro	3.6	1.64	1.33	1.42	1.47	2.61
	5.4	2.02	2.10	2.16	2.28	4.04
	7.2	2.53	2.57	2.72	2.75	4.73
	9	3.01	3.07	3.33	3.29	5.43
	10.8	3.49	3.49	3.72	3.69	6.16

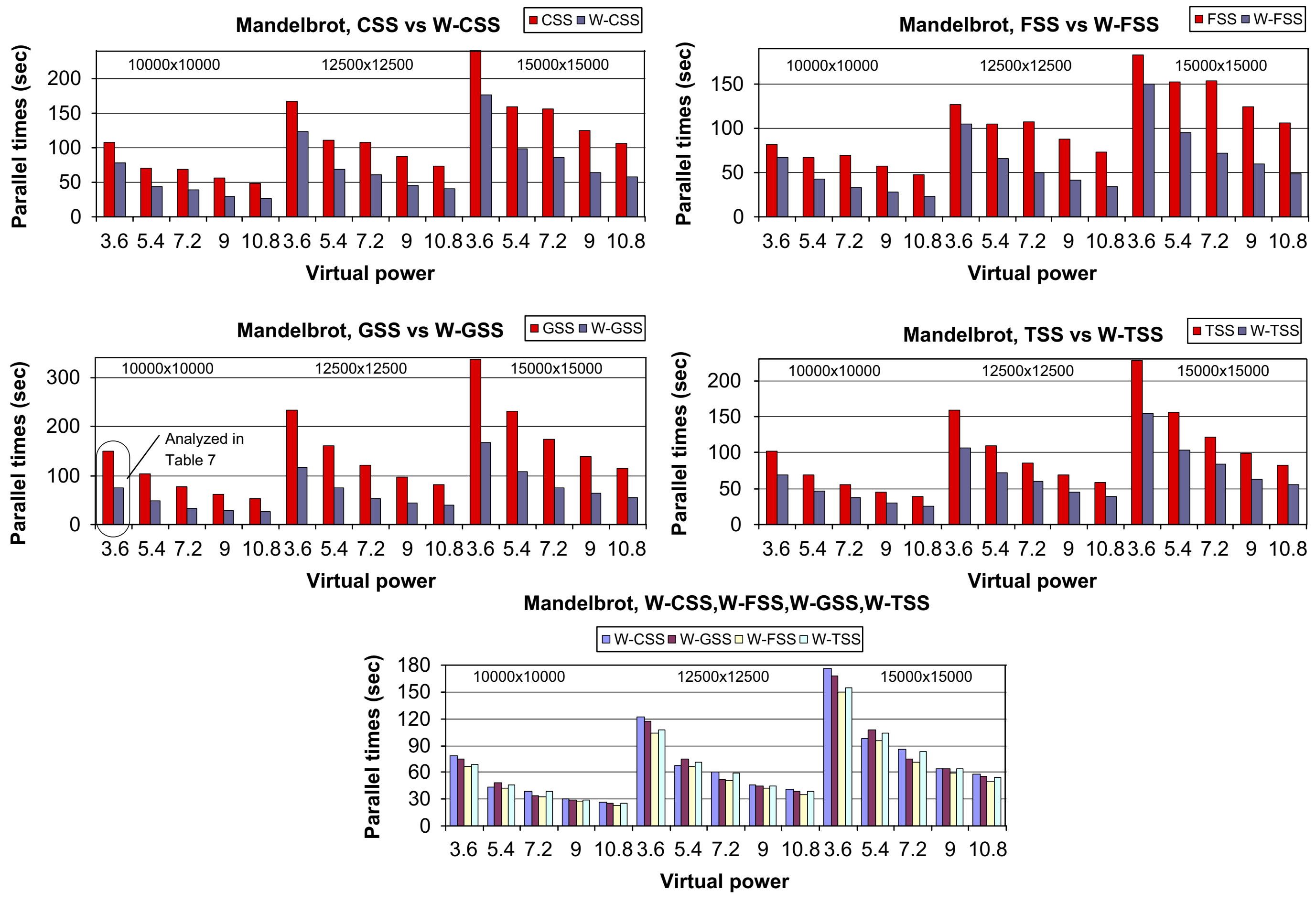


Fig. 10. Parallel times of the weighted and non-weighted algorithms for Mandelbrot case study.

dependence loop, which we modified in order to explicitly show the 3-dimensional dependencies among iterations for the array za. The pseudocode is given below:

```
/* implicit hydrodynamics fragment */
for (l=1; l<=loop; l++) { /* synchronization dimension */
   for (j=1; j<5; j++)  {
      for (k=1; k<n; k++){ /* chunk dimension */
     qa = za[l-1][j+1][k]*zr[j][k] + za[l][j-1][k]*zb[j]
          [k] +za[l-1][j][k+1]*zu[j][k] + za[l][j][k-1]
            *zv[j][k] +zz[j][k];
        za[l][j][k] += 0.175 * (qa - za[l][j][k] );
      }
   }
}
```

Table 6
Gain of the weighted over the non-weighted algorithms for the Mandelbrot test case

Test case	Problem size	VP	$\mathcal{S}$-CSS vs $\mathcal{SW}$-CSS (%)	$\mathcal{S}$-GSS vs $\mathcal{SW}$-GSS (%)	$\mathcal{S}$-FSS vs $\mathcal{SW}$-FSS (%)	$\mathcal{S}$-TSS vs $\mathcal{SW}$-TSS (%)
Mandelbrot	10 000 × 10 000	3.6	27	50	18	33
		5.4	38	54	37	34
		7.2	43	57	52	32
		9	48	53	52	35
		10.8	43	52	52	34
	12 500 × 12 500	3.6	27	50	18	33
		5.4	38	54	37	34
		7.2	44	57	53	30
		9	47	54	52	35
		10.8	44	52	53	34
	15 000 × 15 000	3.6	27	50	18	33
		5.4	38	54	37	34
		7.2	45	57	53	31
		9	49	54	52	35
		10.8	46	52	54	33
Confidence interval (95%)	Overall 42 ± 3%		40 ± 6	53 ± 6	42 ± 8	33 ± 4

Table 7
Load balancing in terms of total number of iterations per slave and computation times per slave, GSS vs $\mathcal{W}$-GSS (Mandelbrot test case)

Slave	GSS		$\mathcal{W}$-GSS	
	# Iterations (10^6)	Comp. time (s)	# Iterations (10^6)	Comp. time (s)
twin2	56.434	34.63	55.494	62.54
kid1	18.738	138.40	15.528	62.12
twin3	10.528	39.37	15.178	74.63
kid2	14.048	150.23	13.448	61.92

6.2. Experimental environment setup

The implementation of the master–slave scheme was made using the MPI message-passing interface. The experiments were performed on a heterogeneous Linux cluster of 13 nodes (1 master and 12 slaves). The cluster consists of two machine types: (a) 7 Intel Pentiums III 800 MHz with 256 MB RAM (called *twins*), with virtual power $VP_k = 1$; and (b) 6 Intel Pentiums III 500 MHz with 512 MB RAM (called *kids*), with virtual power $VP_k = 0.8$. In order to obtain the virtual power of each slave we ran 10 times a test problem (which involved nested loops with floating point operations) serially on each computer and averaged the measured execution times. Although this is a simple model, it is appropriate for the type of applications we study, namely nested loops with floating point operations. The machines are interconnected by a 100 Mbits/s Fast Ethernet network.

We experiment on the *non-dedicated* cluster. In particular, at the beginning of execution, we start a resource expensive process on some of the slaves, which halves their ACP. We ran three series of experiments: (1) for the synchronization mechanism ($\mathcal{S}$), (2) for the weighting mechanism ($\mathcal{W}$) and (3) for the combined case ($\mathcal{SW}$). We ran the above series for $m = 4, 6, 8, 10, 12$ slaves. The results given in the following subsections are the average of 10 runs for each experiment.

We used the following machines: twin1(master), twin2, **kid1**, twin3, **kid2**, twin4, **kid3**, twin5, **kid4**, twin6, **kid5**, twin7, **kid6**. In all cases, the overloaded machines were the *kids* (written in boldface). When $m = 4$, the first four machines are used from the above list, when $m = 8$ the first eight machines are used, and so on.

6.3. Experiment 1

For the first series, we experimented on two real-life applications: Floyd–Steinberg and Hydro. For each application we used three problem sizes, given in Table 4, in order to perform a scalability analysis with respect to the problem size. In each

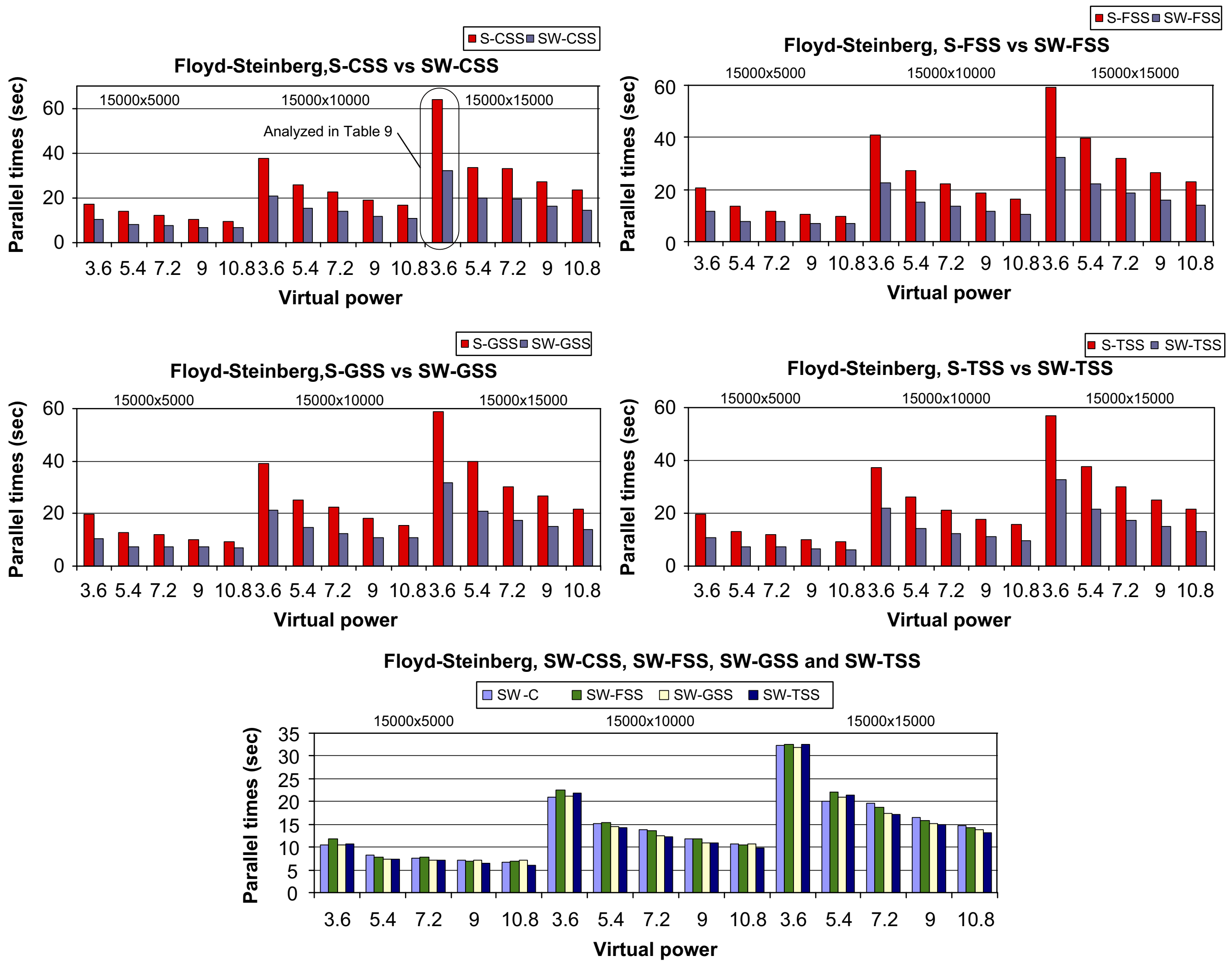


Fig. 11. Parallel times of the synchronized–weighted and synchronized-only algorithms for Floyd–Steinberg case study, for three different problem sizes: 15 000 × 5000, 15 000 × 10 000 and 15 000 × 15 000.

case, U_s was taken as the maximal dimension, whereas U_c was taken as the second largest dimension. For each of these applications we compared the parallel execution times against the serial execution times.

We scheduled each test case using the five self-scheduling algorithms, i.e., CSS, FSS, GSS, TSS and W-TSS, to which we applied the synchronization mechanism $\mathcal{S}$. The algorithms were implemented as described in Section 2. In the case of CSS we used the chunk size $\frac{U_c}{2*m}$ in order to ensure that slaves receive work twice. To avoid excessive synchronization overheads and large idle times we used lower and upper bounds (see Table 4) on the size of the chunks given by the self-scheduling algorithms.

The SI for both test cases is given by formula (1), where $\#SPs = 3 * m$ was used. In order to show that the #SPs must be at least $3 * m$ we ran multiple tests with different values for the $\#SPs$, ranging from $1 * m$ to $10 * m$ in order to assess the impact of choice of #SPs on the parallel execution times. The results for both applications with dependencies are given in Fig. 8. One can see that a good performance can be obtained if $\#SPs \geqslant 3 * m$.

Both the Floyd–Steinberg and Hydro applications have unitary dependence vectors, i.e., unitary projection lengths along the chunk dimension u_c. This yields a relatively small volume of communication, making it easy to maintain the $\frac{communication}{computation}$ ratio below 1. The results for all problem sizes are shown in Fig. 9.

We plotted the parallel and serial times vs the virtual power of the cluster. With four slaves VP is 3.6, with six slaves VP is 5.4, with eight slaves is 7.2, with 10 slaves VP is 9 and with 12 slaves is 10.8. The serial time was measured on the fastest slave type, i.e., *twin*.

The results show that the synchronization mechanism can be applied to all existing self-scheduling algorithms, leading to their synchronized versions, which can efficiently parallelize real-life dependence loops. One can notice that $\mathcal{S}$-CSS, $\mathcal{S}$-FSS, $\mathcal{S}$-GSS, and $\mathcal{S}$-TSS give significant speedup over the serial execution (see Table 5), proving the efficiency of the transformed

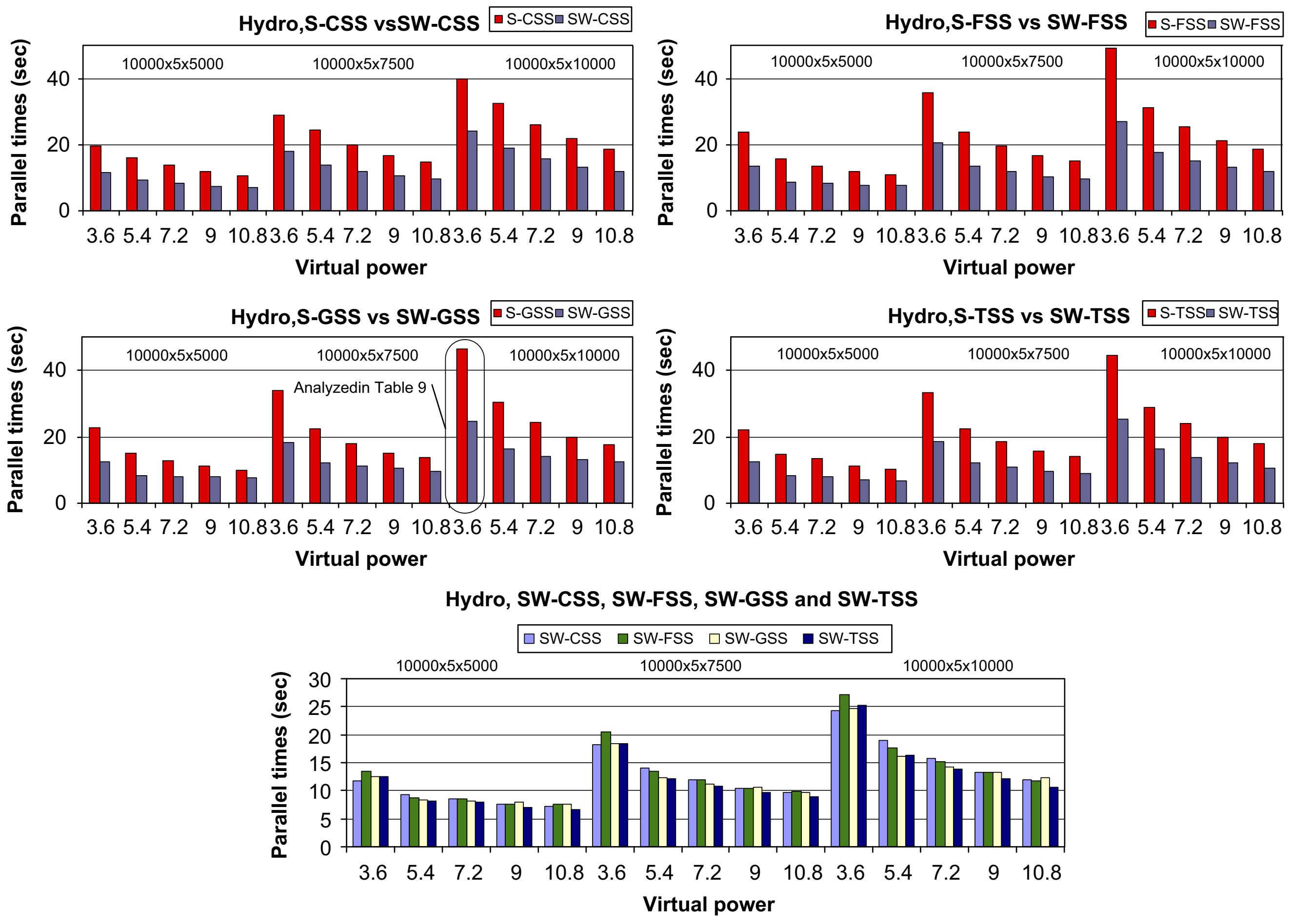


Fig. 12. Parallel times of the synchronized–weighted and synchronized-only algorithms for Hydro case study, for three different problem sizes $10\,000 \times 5 \times 5000$, $10\,000 \times 5 \times 7500$ and $10\,000 \times 5 \times 10\,000$.

algorithms. In all cases, $\mathcal{SW}$-TSS, which explicitly accounts for system's heterogeneity, shows an even greater speedup over all synchronized-only algorithms. The two charts from the bottom of Fig. 9 illustrate the serial and parallel times on 12 processors for both applications. The serial times increase faster than the parallel time as the problem size increases in both cases. This shows that the larger the problem size, the more processors can be effectively employed in the parallel execution. This yields larger speedups for larger problem sizes. This was anticipated since with larger index spaces and for the same task granularity, a greater degree of parallelism becomes available.

6.4. Experiment 2

For the second series of experiments, we used a well known parallel loop, the Mandelbrot fractal computation algorithm, on the domain $[-2.0, 1.25] \times [-1.25, 1.25]$, for different window sizes: $7500 \times 10\,000$, $10\,000 \times 10\,000$ and $12\,500 \times 12\,500$. The computation involves unpredictably irregular loop tasks. We scheduled the Mandelbrot set with CSS, FSS, GSS, TSS and compared their performance against their weighted versions $\mathcal{W}$-CSS, $\mathcal{W}$-FSS, $\mathcal{W}$-GSS, $\mathcal{W}$-TSS. In this series, the weighting mechanism is expected to facilitate the scheduling algorithms to distribute the work to slaves more evenly and to improve the overall performance. This is confirmed by the experimental results, depicted in Fig. 10. One can see that in all cases the weighted algorithm clearly outperforms the non-weighted algorithm.

The **gain** of the weighted over non-weighted algorithms is also given in Table 6, computed as $\frac{T_{\mathcal{A}}-T_{\mathcal{W}\text{-}\mathcal{A}}}{T_{\mathcal{A}}}$, where $T_{\mathcal{A}}$ is the parallel time of the non-weighted algorithm $\mathcal{A}$ and $T_{\mathcal{W}\text{-}\mathcal{A}}$ is the parallel time of the weighted algorithm $\mathcal{W}$-$\mathcal{A}$. We are also interested in establishing an adequate confidence interval for the performance gain in each case. As it is common in practice, we consider 95% confidence. The algorithm with the least parallel time is $\mathcal{SW}$-FSS for all problem sizes; the performance gain of $\mathcal{SW}$-FSS over FSS ranges from 18% to 53%. With 0.95 probability the performance gain lies in the interval $42 \pm 8\%$. However, the algorithm with the best overall performance gain is GSS, ranging from 50% to 57%. With 0.95 probability the gain lies in the interval $53 \pm 6\%$. As shown in Fig. 10, the difference in performance of the weighted algorithms is much smaller than the performance difference between their non-weighted versions.

Table 8
Gain of the synchronized–weighted over the synchronized-only algorithms for the Floyd–Steinberg and Hydro test cases

Test case	Problem size	VP	$\mathcal{S}$-CSS vs $\mathcal{SW}$-CSS (%)	$\mathcal{S}$-GSS vs $\mathcal{SW}$-GSS (%)	$\mathcal{S}$-FSS vs $\mathcal{SW}$-FSS (%)	$\mathcal{S}$-TSS vs $\mathcal{SW}$-TSS (%)
Floyd–Steinberg	15 000 × 5000	3.6	39	47	43	45
		5.4	42	43	44	44
		7.2	37	40	35	40
		9	34	27	34	36
		10.8	31	23	28	35
	15 000 × 7500	3.6	44	46	45	42
		5.4	41	42	44	45
		7.2	39	45	38	42
		9	37	40	37	38
		10.8	36	30	36	38
	15 000 × 10 000	3.6	50	46	45	43
		5.4	41	48	44	43
		7.2	41	42	41	42
		9	39	43	40	41
		10.8	38	36	38	39
Confidence interval (95%)	Overall 40 ± 1		39 ± 2	40 ± 3	40 ± 2	41 ± 2
Hydro	10 000 × 5 × 5000	3.6	40	46	43	44
		5.4	43	44	44	44
		7.2	39	37	37	41
		9	37	29	36	38
		10.8	32	23	29	34
	10 000 × 5 × 7500	3.6	38	46	43	44
		5.4	43	45	44	46
		7.2	40	37	39	42
		9	37	29	38	38
		10.8	35	30	35	36
	10 000 × 5 × 10 000	3.6	40	47	45	44
		5.4	42	47	44	43
		7.2	40	41	41	42
		9	39	33	38	39
		10.8	37	30	37	40
Confidence interval (95%)	Overall 39 ± 1		39 ± 2	38 ± 4	40 ± 2	41 ± 2

In order to examine the effect of the weighting mechanism on load balancing, Table 7 provides the total computation times of each slave together with the total number of iterations it was assigned, for the computation of the Mandelbrot test-case. The total computation time of each slave is the time it spends performing actual work. In the ideal case, all slaves should have exactly the same computation time, and a large divergence shows great load imbalance.

The data in Table 7 correspond to Fig. 10 (GSS vs $\mathcal{W}$-GSS, the parallel times obtained for four slaves with total $VP = 3.6$), and shows the difference between the non-weighted and the weighted algorithm. In particular, Table 7 analyzes the parallel times of GSS and $\mathcal{W}$-GSS on each of the four slaves and the number of iterations computed by each slave. Note that the parallel time plotted in Fig. 10 is close to the computation time of the slowest slave. It is clear that with the non-weighted algorithm the computation times of each slave vary in relation to the computation power and load of the slave, i.e., slow and overloaded slaves have larger total computation times. In the case of the weighted algorithm this variation is reduced significantly. Also, with GSS, even though kid2 and twin2 were assigned 14.048×10^6 and 56.434×10^6 loop iterations, respectively, twin2 required 34.63 s to compute these loop iterations, in contrast with 150.23 s required by kid2 (this is roughly the parallel time for this case). This led to a huge load imbalance, which deteriorated significantly the algorithm's performance.

Table 9
Load balancing in terms of total number of iterations per slave and computation times per slave, $\mathcal{S}$-FSS vs $\mathcal{SW}$-FSS

Test	Slave	# Iterations (10^6)	Comp. time (s)	# Iterations (10^6)	Comp. time (s)
		$\mathcal{S}$-CSS	$\mathcal{S}$-CSS	$\mathcal{SW}$-CSS	$\mathcal{SW}$-CSS
Floyd–Steinberg	twin2	59.93	19.25	89.90	28.88
	kid1	59.93	62.22	29.92	30.86
	twin3	59.93	19.24	74.92	24.06
	kid2	44.95	46.30	29.92	29.08
		$\mathcal{S}$-GSS	$\mathcal{S}$-GSS	$\mathcal{SW}$-GSS	$\mathcal{SW}$-GSS
Hydro	twin2	84.50	15.32	117.94	21.39
	kid1	78.38	42.60	38.03	22.49
	twin3	62.69	17.44	106.48	20.75
	kid2	73.58	33.72	36.41	19.46

Unlike GSS, $\mathcal{W}$-GSS execution times for all slaves are about the same which confirms that $\mathcal{W}$-GSS indeed achieves good load balancing.

6.5. Experiment 3

For the third series of experiments we repeat the first series, applying now the $\mathcal{W}$ mechanism to all synchronized-only algorithms. In particular, we schedule the Floyd–Steinberg and Hydro test cases with the following synchronized–weighted algorithms: $\mathcal{SW}$-CSS, $\mathcal{SW}$-FSS, $\mathcal{SW}$-GSS and $\mathcal{SW}$-TSS. The results in Figs. 11 and 12 show that in all cases the synchronized–weighted algorithms clearly outperform their synchronized-only counterparts. One can notice that all $\mathcal{SW}$ algorithms give comparable parallel times.

The above results are also illustrated in Table 8, which shows the gain of $\mathcal{SW}$-$\mathcal{A}$ over $\mathcal{S}$-$\mathcal{A}$, computed as $\frac{T_{\mathcal{S}\text{-}\mathcal{A}}-T_{\mathcal{SW}\text{-}\mathcal{A}}}{T_{\mathcal{S}\text{-}\mathcal{A}}}$, where $T_{\mathcal{S}\text{-}\mathcal{A}}$ is the parallel time of the synchronized-only algorithm $\mathcal{A}$ and $T_{\mathcal{SW}\text{-}\mathcal{A}}$ is the parallel time of the synchronized–weighted algorithm $\mathcal{SW}$-$\mathcal{A}$. Confidence intervals are also given in the same Table, both with respect to every algorithm and overall confidence intervals per test case.

In Table 8, we show the algorithm with the highest gain for each application. Subsequently, in Table 9 we analyze the gain of $\mathcal{SW}$-CSS over $\mathcal{S}$-CSS for Floyd–Steinberg and of $\mathcal{SW}$-GSS over $\mathcal{S}$-GSS for Hydro. This gain is attributed to better load balancing, which is expressed in terms of the total computation time per slave. In particular: $\mathcal{S}$-CSS and $\mathcal{S}$-GSS, respectively, assigned approximately the same number of iterations to all slaves; this led to larger computation times for the slower slaves (i.e., kids) in comparison to faster slaves (i.e., twins). With $\mathcal{SW}$-CSS and $\mathcal{SW}$-GSS, respectively, the difference in the slaves' computation time is reduced because the number of iterations assigned to each slave has been adjusted according to their available power.

7. Conclusions and future work

In this paper we deal with the problem of load balancing in scheduling loops with (or without) dependencies on heterogeneous non-dedicated distributed systems. We study existing self-scheduling schemes and propose two mechanisms to improve their performance using a master–slave model. Firstly, the synchronization mechanism, which enables the application of existing self-scheduling algorithms to loops with dependencies. Secondly, the weighting mechanism, which improves the load balancing capability of these algorithms. We ran experiments from practical applications involving loops with uniform dependencies and also a test with a parallel loop with uneven tasks. Our results show that the synchronization mechanism enables the scheduling algorithms to obtain significant speedups for the dependence loops. Furthermore, the weighting mechanism makes the existing algorithms most suitable for heterogenous non-dedicated systems because significant gains were obtained over the algorithms without weighting.

Although in this article we study the problem of scheduling loops with uniform dependencies we expect that our results can be extended to apply to loops with non-uniform dependencies. In the future we plan to study the scheduling problem for non-uniform dependence loops in distributed systems. We also plan to further investigate the synchronization mechanism in terms of the number of SPs required for achieving the best performance.

Acknowledgments

The authors express their thanks to the editor and the anonymous referees for their helpful and constructive suggestions, which considerably improved the quality of the paper. This work was 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.

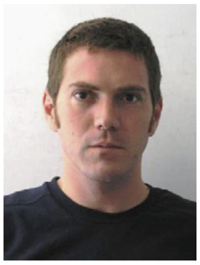
References

[1] M. Adler, P. Berenbrink, K. Schroder, Analyzing an infinite parallel job allocation process, in: Proceedings of the 6th European Symposium on Algorithms (ESA), 1998, pp. 417–428.

[2] I. Ahmad, A. Ghafoor, K. Mehrotra, Performance prediction of distributed load balancing on multicomputer systems, in: Proceedings of the Supercomputing '91, 1991, pp. 830–839.

[3] I. Banicescu, Z. Liu, Adaptive factoring: a dynamic scheduling method tuned to the rate of weight changes, in: Proceedings of the High Performance Computing Symposium 2000, Washington, USA, 2000, pp. 122–129.
[4] I. Banicescu, V. Velusamy, J. Devaprasad, On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring, Cluster Comput. J. Networks Software Tools Appl. 6 (3) (2003) 215–226.
[5] A.T. Chronopoulos, R. Andonie, M. Benche, D. Grosu, A class of distributed self-scheduling schemes for heterogeneous clusters, in: Proceedings of the 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001), Newport Beach, CA, USA, 2001.
[6] A.T. Chronopoulos, S. Penmatsa, J. Xu, S. Ali, Distributed loop scheduling schemes for heterogeneous computer systems, Concurrency Comput. Pract. Experience 18 (7) (2006) 771–785.
[7] F.M. Ciorba, T. Andronikos, I. Riakiotakis, A.T. Chronopoulos, G. Papakonstantinou, Dynamic multiphase scheduling for heterogeneous clusters, in: Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), Rhodes, Greece, 2006.
[8] D.L. Eager, E.D. Lazowska, Adaptive load sharing in homogeneous distributed systems, IEEE Trans. Software Eng. 12 (5) (1986).
[9] R.W. Floyd, L. Steinberg, An adaptive algorithm for spatial grey scale, Proc. Soc. Inf. Display 17 (1976) 75–77.
[10] M. Harchol-Balter, A.B. Downey, Exploiting process lifetime distributions for dynamic load balancing, ACM Trans. Comput. Systems 15 (3) (1997) 253–285.
[11] C.-J. Hou, K.G. Shin, Load sharing with consideration of future task arrivals in heterogeneous distributed real-time systems, IEEE Trans. Comput. 49 (9) (1994) 1076–1090.
[12] S.F. Hummel, J. Schmidt, R.N. Uma, J. Wein, Load-sharing in heterogeneous systems via weighted factoring, in: Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures, 1996.
[13] S.F. Hummel, E. Schonberg, L.E. Flynn, Factoring: a method for scheduling parallel loops, Commun. ACM 35 (8) (1992) 90–101.
[14] C.P. Kruskal, A. Weiss, Allocating independent subtasks on parallel processors, IEEE Trans. Software Eng. 11 (10) (1985) 1001–1016.
[15] Y.K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surveys 31 (4) (1999) 406–471.
[16] Q. Lu, S.-M. Lau, K.-S. Leung, Dynamic load distribution using antitasks and load state vectors, Concurrency: Pract. Experience 10 (14) (1998) 1251–1269.
[17] B.B. Mandelbrot, Fractal Geometry of Nature, W. H. Freeman & Co, New York, August 1988.
[18] E.P. Markatos, T.J. LeBlanc, Using processor affinity in loop scheduling on shared-memory multiprocessors, IEEE Trans. Parallel Distrib. Systems 5 (4) (1994) 379–400.
[19] F.H. McMahon, The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range, Lawrence Livermore National Laboratory, Livermore, CA, UCRL-53745, 1986.
[20] D. Petkov, R. Harr, S. Amarasinghe, Efficient pipelining of nested loops: unroll-and-squash, in: Proceedings of the 16th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2002), Ft. Lauderdale, FL, USA, 2002.
[21] C.D. Polychronopoulos, D.J. Kuck, Guided self-scheduling: a practical self-scheduling scheme for parallel supercomputers, IEEE Trans. Comput. C-36 (12) (1987) 1425–1439.
[22] E. Silva, M. Gerla, Queueing network models for load balancing in distributed systems, J. Parallel Distrib. Comput. 12 (1991) 24–38.
[23] T.H. Tzen, L.M. Ni, Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers, IEEE Trans. Parallel Distrib. Systems 4 (1) (1993) 87–98.
[24] J.-L. Wang, L.-T. Lee, Y.-J. Huang, Load balancing policies in heterogeneous distributed systems, in: Proceedings of the 26th Southeastern Symposium System Theory, 1994.

Florina M. Ciorba received her Diploma in Computer Engineering from the University of Oradea, Romania, in 2001. Since 2002 she is a Ph.D. candidate at the National Technical University of Athens, Greece. Her research interests include parallel and distributed high performance computing, resource management, scheduling and load balancing. She is an IEEE student member since 2004.



Ioannis Riakiotakis received his B.Sc. in Electronics Engineering in 1999 and his M.Sc. in Communication Engineering and Signal Processing in 2000 both from the University of Plymouth (UK). Since 2002 he is a Ph.D. candidate at the National Technical University of Athens (Greece). His research interests include distributed computing, dynamic load balancing algorithms.

Theodore Andronikos received his Degree in Electrical and Computer Engineering and his Ph.D. from the Computer Science Division, School of Electrical and Computer Engineering, National Technical University of Athens. He is currently a lecturer at the Department of Informatics of the Ionian University. His research interests include parallel and distributed computing, internet programming, algorithmic complexity and formal verification of reactive systems.

George Papakonstantinou received his Diploma in Electrical Engineering from the National Technical University of Athens in 1964, the PII Diploma in Electronic Engineering from Philips International Institute in 1966, and his M.Sc. in Electronic Engineering from NUFFIC, the Netherlands in 1967. In 1971, he received his Ph.D. in Computer Engineering from the National Technical University of Athens. He has worked as a Research Scientist at the Greek Atomic Energy Commission/Computer Division (1969–1984), as Director of the Computer Division at the Greek Atomic Energy Commission (1981–1984). From 1984 he serves as a Professor of Computer Engineering at the National Technical University of Athens. His current research interests include knowledge engineering, syntactic pattern recognition, logic design, embedded systems, as well as parallel architectures and languages. He has published more than 270 papers in international referred journals and in proceedings of international conferences. Dr. Papakonstantinou has over 200 citations.

Anthony T. Chronopoulos received his Ph.D. at the University of Illinois in Urbana-Champaign in 1987. He is a senior member of the IEEE and the ACM. He has published 40 journals and 45 refereed conference proceedings publications in the areas of distributed systems, game theory, networks and security, parallel processing. He has been awarded 15 federal/state government research grants. His work is cited in more than 260 nonco-authors' research articles.