Performance Evaluation 67 (2010) 1324–1339

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems☆

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

[a] Department of Informatics, Ionian University, Corfu, Greece
[b] Computing Systems Laboratory, Department of Electrical & Computer Engineering, National Technical University of Athens, Greece
[c] Center for Advanced Vehicular Systems, Mississippi State University, MS, USA
[d] Department of Computer Science, University of Texas at San Antonio, 6900 N. Loop 1604 West, San Antonio, TX 78249, USA

ARTICLE INFO

Article history:
Received 6 August 2007
Received in revised form 4 August 2009
Accepted 16 August 2010
Available online 16 September 2010

Keywords:
Performance evaluation
Dynamic load balancing
Self-scheduling algorithms
Loops with dependencies
Synchronization frequency
Inter-processor communication
Heterogeneous systems

ABSTRACT

In this work, we develop and evaluate a theoretical model, which we then use to study the impact of the synchronization frequency on the performance of dynamic self-scheduling algorithms. These algorithms are used to parallelize loops with data dependencies on heterogeneous systems. The proposed model uses a formula to estimate the parallel time as a function of the synchronization frequency. Inter-node communication has been proven to be the dominant factor for the performance degradation of applications containing loops with data dependencies. The synchronization mechanism therefore requires careful fine-tuning in order to give the best possible performance. The proposed model determines the optimal synchronization frequency that results in the minimum parallel time. We use this model to study the impact of the synchronization frequency on the parallel execution of a computational kernel from image processing. For this kernel, the synchronization frequency giving the minimum parallel time predicted by our theoretical model was very close to the synchronization frequency giving the least parallel time in practice. We validate our model by extensive comparisons of the theoretically predicted parallel time and synchronization frequency against those obtained from practical experiments. The comparisons show that the proposed model is highly accurate, its predictions for the optimal synchronization frequency being within **0.0250**% of the experimentally optimal synchronization frequency in the best case, and within **0.1750**% of the experimentally optimal synchronization frequency in the worst case. Finally, the comparisons show that the proposed model improves on a previously existing model in heterogeneous systems, whereas it gives similar results in homogeneous systems.

1. Introduction

In this paper, we propose and evaluate a linear, yet very accurate, mathematical model for determining the optimal synchronization frequency of dynamic self-scheduling algorithms, designed to parallelize nested loops with data

☆ 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. The work of the A.T. Chronopoulos was partly supported by NSF grant (HRD-0932339).

* Corresponding author at: Computing Systems Laboratory, Department of Electrical & Computer Engineering, National Technical University of Athens, Greece.
E-mail addresses: andronikos@ionio.gr (T. Andronikos), florina@cavs.msstate.edu (F.M. Ciorba), iriak@cslab.ece.ntua.gr (I. Riakiotakis), papakon@cslab.ece.ntua.gr (G. Papakonstantinou), atc@cs.utsa.edu (A.T. Chronopoulos).

0166-5316/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2010.08.020

dependencies on heterogeneous systems. Many scheduling algorithms have been proposed in the past for nested loops with and without data dependencies on heterogeneous distributed systems. An important class of dynamic scheduling algorithms is that of the self-scheduling schemes, such as chunk self-scheduling (CSS) [1], guided self-scheduling (GSS) [2], trapezoid self-scheduling (TSS) [3] and factoring self-scheduling (FSS) [4] (see also [5–9] and references therein). These algorithms dynamically assign work to processing nodes in chunks of variable sizes. The simpler versions of these algorithms are suitable for homogeneous systems working in single-user-job (dedicated) execution mode. The distributed versions of these algorithms target heterogeneous computing systems [10]. In such systems, loop-scheduling schemes must take into account the differences in the computational power of each processing node. The computational power of a processing node depends on the CPU speed, memory, cache structure and even the application type. Therefore, load-balancing methods adapted to distributed computing environments take into account the relative computing powers of the processing resources [6–8,11,9, 12–14]. The relative computing powers are used as weights that scale the size of the subproblem assigned to each processing node.

A class of dynamic self-scheduling algorithms was developed in [13,15] specifically to address applications containing loops with data dependencies. The algorithms in this class were inspired from the chunk self-scheduling, trapezoid self-scheduling and distributed trapezoid self-scheduling (DTSS) schemes, which were initially devised for tasks without data dependencies. Self-scheduling algorithms follow the master–worker model, and their modus operandi is to partition the iteration space of the loop into chunks, which are then assigned to worker processors by the master processor upon request. However, due to the existence of data dependencies, certain iterations in one chunk depend on certain iterations in other chunks. Hence, every worker communicates with neighboring workers at predefined synchronization points, since it requires data for its local computations and/or other workers require data for remote computations. Inter-processor communication is, therefore, one of the most important factors for performance degradation when parallelizing applications containing loops with data dependencies. In [13,15], the synchronization frequency was chosen in an ad hoc manner. In this work, we develop a rigorous mathematical model for the performance evaluation of the dynamic self-scheduling schemes as a function of the synchronization frequency. This model allows us to theoretically determine the optimal synchronization frequency, i.e., the one leading to the minimum parallel time.

Related work. A significant amount of work exists for determining the optimal partitioning (tile size, block size, grain size) of the iteration space of nested loops in homogeneous systems ([16–21] and references therein). In [16], Desprez et al. presented a method for overlapping communications on homogeneous systems for pipelined algorithms. They provided a general theoretical model to find the optimal packet size. In [17], Andonov and Rajopadhye addressed the problem of finding the tile size that minimizes the total execution time, on homogeneous systems. Xue studied the problem of time minimal tiling in [19]. Tiles are statically assigned in a block or block cyclic fashion to homogeneous processors. The optimal tile size is determined based on the critical path of the last processor. In [20], Xue and Cai presented a solution to the problem of finding the optimal tile size on homogeneous systems, when the rise is larger than zero. In [18], Lowenthal et al. proposed a method for selecting the block size at run time in pipelined programs; they target problems with irregular workloads on homogeneous systems. In [21], Strout et al. proposed a run-time reordering transformation, i.e., full sparse tiling, that improves the data locality for stationary iterative methods. However, the problem of finding the optimal partitioning of iteration spaces for *heterogeneous systems* has not been given enough attention so far. In [22], Chen and Xue proposed a method for obtaining the optimal tile size on heterogeneous networks of workstations, in which the shape and sizes of tiles are statically determined and scheduled using a block distribution approach. A first effort to determine the optimal synchronization frequency was presented in [23].

Our approach. This work differs from the aforementioned methods in that it targets dynamic scheduling algorithms for application tasks with data dependencies on heterogeneous systems. This work is an extension of [23], which only outlined the central idea behind a theoretical model, such as the one elaborated and extended herein. In the quest to find the optimal synchronization frequency, the first step is to construct a theoretical model that will predict the minimum parallel time in the less complicated case of *homogeneous dedicated* computing systems. For this first step we select a simple self-scheduling algorithm, CSS, as enhanced in [13], to handle data dependencies. We show that the minimum *actual* parallel time for homogeneous systems is achieved with a synchronization frequency that is within **0.0250**% of the theoretically optimal synchronization frequency in the *best case*, and within **0.0750**% of the theoretically optimal one in the *worst case*. The next step is to build a theoretical model for predicting the parallel performance in *heterogeneous dedicated* systems, in which worker processors have different computational speeds. Again, we select CSS as the self-scheduling algorithm, and we show that the minimum actual parallel time for a particular heterogeneous system is obtained with a synchronization frequency within **0.0750**% of the theoretically optimal synchronization frequency in the *best case*, and within **0.1750**% of the theoretically optimal one in the *worst case*. For the experiments on the heterogeneous clusters we had to modify CSS, such that the chunk assigned to a worker was *weighted* according to its computational power. As an application test case we used the Floyd–Steinberg [24] error dithering algorithm, an image-processing computational kernel described in Section 5.

Contributions. In this work, we do not propose a new scheduling scheme; we consider an existing scheme, first presented in [13,15], and propose a model to evaluate its performance. The contribution of this paper is twofold: (1) proposing a verifiable model for fine-tuning the synchronization mechanism between worker processors that minimizes the communication overhead and yields sufficient concurrency for reaching the best possible overall performance; (2) analyzing the case of both an unlimited and a limited number of processors in heterogeneous systems, and improving on a previously existing model [22] on heterogeneous systems that gives similar results on homogeneous systems. Extensive experimental

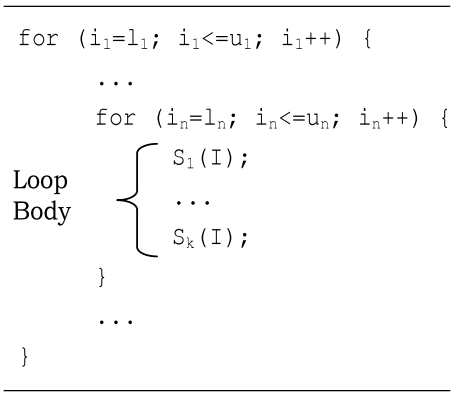

Fig. 1. Algorithmic model.

tests show that our model, despite its simplicity – a deliberate choice based on linearity in order to preserve a low time complexity – works well because it determines the optimal synchronization frequency with a high accuracy. Nevertheless, in future work it would be interesting to study other notable, yet more complicated, scheduling schemes, such as those based on factoring and its variants (see [6–8]).

Organization. The paper is organized as follows. In Section 2, the program model is explained and the necessary notation is defined. The theoretical parallel time estimation models for the homogeneous and heterogeneous systems are presented in Sections 3 and 4, respectively. The experimental validation and comparison of our model with another existing model are described in Section 5. Conclusions and future work are presented in Section 6.

2. Program model and notation

An n-nested loop is a program structure typically modeled as a discrete subset J of the n-dimensional Euclidean space $\mathbb{R}^n$ ($J \subset \mathbb{R}^n$), called the *iteration space* of the loop. Each point of this n-dimensional iteration space corresponds to a distinct iteration of the loop body. $\mathbf{L} = (l_1, \ldots, l_n)$ and $\mathbf{U} = (u_1, \ldots, u_n)$ are the *initial* and *terminal* points of the discrete iteration space (Fig. 1). There are two categories of nested loops: parallel loops and dependence loops. Parallel loops have no data dependencies among their iterations and, thus, these can be executed in any order or even simultaneously. In dependence loops, the iterations depend on each other, which imposes a certain execution order in order to satisfy the existing data dependencies. In this work we assume that an n-nested loop has r uniform dependencies which are modeled by n-dimensional dependence vectors. A dependence vector is written as $\vec{\mathbf{d}}_i \in \mathbb{R}^n$, $1 \leq i \leq r$.

Consider the following toy scale example, representing a two-dimensional nested loop with data dependencies:

```
for (i=1; i<=24; i++)
    for (j=1; j<=8; j++) {
        A[i,j] = A[i,j] * A[i-2,j-2];
        B[i,j] = 2*B[i-3,j-1] + A[i-2,j-2] - 1;
    }
```

The iteration space of this example contains $24 \times 8 = 192$ points; the initial and terminal points are $\mathbf{L} = (1, 1)$ and $\mathbf{U} = (24, 8)$ respectively, and the two dependence vectors are $\vec{\mathbf{d}}_1 = (2, 2)$ and $\vec{\mathbf{d}}_2 = (3, 1)$.

Self-scheduling algorithms partition the iteration space into chunks along one of its dimensions, called the *chunk* dimension, and denoted by u_c. This partitioning yields a pool of tasks, which are then dynamically assigned to the available worker processors upon request. The simplest self-scheduling algorithm is *pure self-scheduling* (PSS). PSS assigns one iteration per request to each worker processor. To reduce the synchronization overhead of PSS, the *chunk self-scheduling* (CSS) algorithm assigns to each worker a chunk of iterations of fixed size. For a detailed exposition of self-scheduling schemes, the interested reader is referred to [25]. The CSS algorithm was extended in [13,15] to handle loops with data dependencies in addition to parallel loops. The extension involved the insertion of synchronization points along another dimension of the iteration space, called the *synchronization* dimension, and denoted by u_s.

The following notation will be used in the rest of this work.

- NP is the number of worker processors in the system.
- $P1, \ldots, P_{NP}$ are the worker processors.
- N is the number of steps of the scheduling algorithm.
- u_c is the *chunk* dimension, which is divided into chunks according to a self-scheduling algorithm.
- A few consecutive iterations of the loop are called a *chunk*; C_i is the chunk size at the i-th scheduling step.

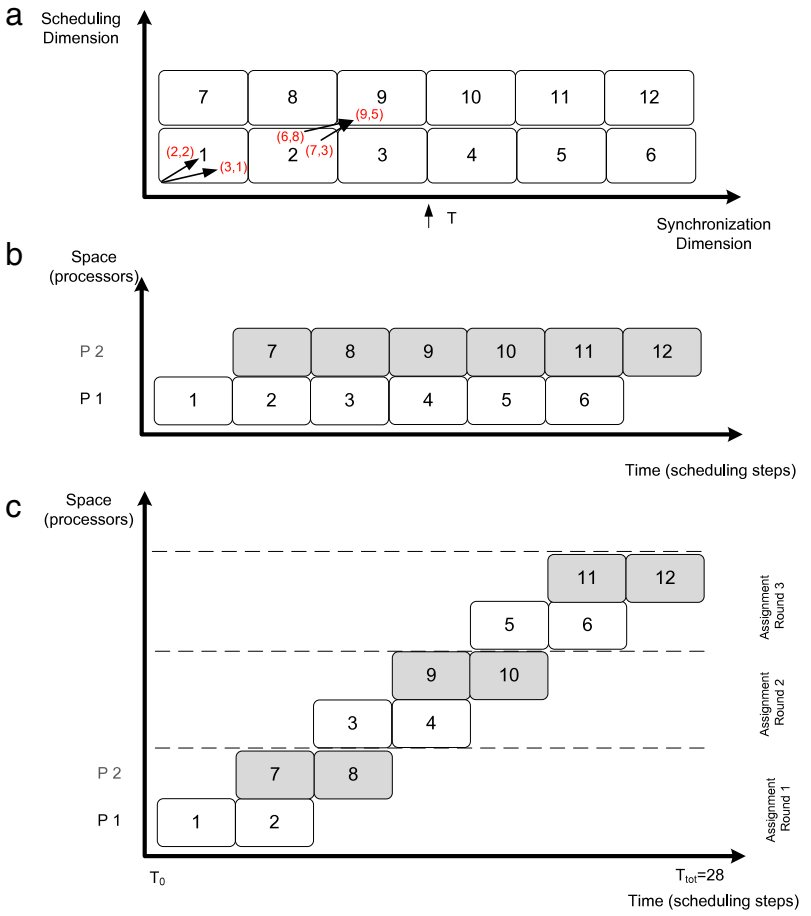

Fig. 2. (a) The toy scale iteration space is partitioned into two chunks, and each chunk is partitioned into six subchunks. (b) A space–time mapping of chunks to a processor and a naive time schedule following the execution flow. (c) The flow of execution in space and time for a balanced workload on processors of equal speed.

- V_i is the length of the projection of chunk i in the u_c dimension.
- u_s is the *synchronization* dimension, along which synchronization points are inserted. The synchronization points are uniformly distributed along u_s.
- h is the length of the synchronization interval, i.e., the distance between successive synchronization points. Note that h is the same for every chunk.
- The set of iterations of a chunk between successive synchronization points is called a *subchunk*.

Let us consider again the toy scale example described earlier. The iteration space is depicted in Fig. 2 with the two uniform dependence vectors (2, 2) and (3, 1). If only workers with equal computational power are available, then the resulting schedule would be the one in Fig. 2(a), where the iteration space is partitioned into two chunks of equal size. Worker P1 is assigned chunk 1 comprising of subchunks 1–6, all of equal size, and worker P2 is assigned chunk 2 comprising of subchunks 7–12, also of equal size. Partitioning a chunk into subchunks is a necessary step imposed by the data dependencies. Workers 1 and 2 cannot compute simultaneously subchunks 1 and 7 respectively because subchunk 7 requires data from subchunk 1. For all workers, the execution flow follows the "receive, compute, send" order. After P1 finishes subchunk 1, it sends the necessary data to P2, which in turn receives the data and proceeds to compute subchunk 7. In effect the subchunks establish a communication and synchronization mechanism between workers. If P1 were to compute the entire chunk without any data exchange with P2, the result would be the sequential execution of chunks 1 and 2. The presence of dependencies necessitates the partitioning of chunks into subchunks and at the same time imposes a precedence order on their execution. Certain subchunks, e.g., 1 and 7, 2 and 8, etc., cannot be computed simultaneously, in order to respect the data dependencies. Certain other subchunks can be processed in parallel as long as no data dependencies are violated. For instance, subchunks 4 and 9 can be computed simultaneously; a typical iteration point of chunk 9 requires data from points in subchunks 2 and 3. For instance, (9, 5) requires data from points (7, 3) and (6, 4). These have already been computed by P1. Assuming that P1 communicates these data to P2 upon completion of subchunk 3 and before beginning the computation of subchunk

4, then subchunks 4 and 9 can be computed in parallel. For this type of synchronization between workers P1 and P2, the chunk–subchunk assignment is depicted in Fig. 2(b), whereas Fig. 2(c) shows the execution flow in time.

Next, we describe the communication and computation cost models we used in order to estimate the parallel time in a distributed computing system.

2.1. Communication cost model

In this work, we adopt the one-port model as the communication model. In this model, a processor at each time step can either send or receive. Of course, distinct processor pairs communicate simultaneously. This assumption is made to simplify the proposed model and our analysis to determine the optimal synchronization frequency. More elaborate communication models can be considered in the future. The cost of communicating a message between two workers is assumed to be the sum of two parts: the *start-up* cost c_d, representing the time to send a zero-length message, including the hardware/software overhead of sending the message, and the *transmission cost* of the message. The transmission cost varies according to the size and type (e.g., float, double or any other user-defined type) of data. The transmission cost per unit and type of data is denoted by c_c.

To quantify the *communication* parameters, we developed a benchmark program (which can be applied in any network architecture), that simulates a small-scale model with inter-node communication. It performs send and receive calls between all pairs of nodes for messages of increasing size. We measured the average round-trip time for all data exchanges and we observed that this yields a linear model. We chose to perform data exchanges between all pairs of nodes in order to simulate levels of network contention similar to those of the actual application. We then divided this average time by 2, assuming that the send (t_s) and receive (t_r) times are equal. This is a realistic assumption, since the sent and received data concern equal number of elements of the same array that are of the same size. Also, the send and receive operations always occur in pairs. Thus we do not measure send and receive operations separately. The specific c_d and c_c values for our test case are determined in Section 5.2. Therefore, the cost t_r or t_s of communicating a message consisting of h data elements is given by

$$t_r = t_s = c_d + hc_c. \tag{1}$$

2.2. Computation cost model

We define the computation cost as a linear function of the computation cost per iteration, c_p, times the number of iterations. The computation cost per iteration is application and processor dependent. In this work, we consider loops with regular loop bodies. We assume that, for a single application, c_p is the same for every iteration. To accurately quantify c_p, we ran a small subset of the selected application (in our case the Floyd–Steinberg kernel) on each processor type. We divided the total computation time by the total number of iterations in the subset to obtain the c_p for this application. The computation time per iteration, c_p, differs from processor to processor in heterogeneous systems, as shown later in the paper (see Section 5.2). Hence the computation cost of a subchunk (i.e., the number of iterations between two successive synchronization points) is given by

$$t_p = hV_i c_p. \tag{2}$$

3. Model for parallel time prediction in homogeneous systems

In this section, we investigate the impact of the synchronization interval h on the parallel time. The synchronization interval determines the synchronization frequency; we construct a mathematical model that estimates the parallel time for both homogeneous and heterogeneous systems (the latter are assumed to be dedicated to our application) as a function of the synchronization interval. The proposed model is valid if the following two premises are satisfied.

- (**PR**$_1$) The time to compute a subchunk is the same in all cases and for all worker processor types. This is a reasonable assumption, because the main advantage of the underlying scheduling algorithm is its ability to adjust the subchunk size according to the computational power of each worker processor.
- (**PR**$_2$) The communication time (to send and/or to receive data) for every subchunk is the same.

The special case when $NP > N$ is not investigated here, since it can be reduced to the case when $NP = N$ by discarding the extra processors. Thus in this work we study two cases: $NP = N$ and $NP < N$. The homogeneous systems case is presented with the sole purpose of facilitating the understanding of the heterogeneous systems case, which is analyzed in Section 4.

Case $NP = N$. This is the special case where the number of available processors equals the number of *chunks* of tasks, so that each processor is assigned exactly *one* chunk.

Fig. 3 illustrates a small-scale iteration space partitioned into four chunks (horizontal segments) assigned to four workers. Ten synchronization points are inserted in each chunk in such a way that the distance between successive synchronization points is h. The parallelization strategy for this case is given in Fig. 4. All four worker processors start by requesting work

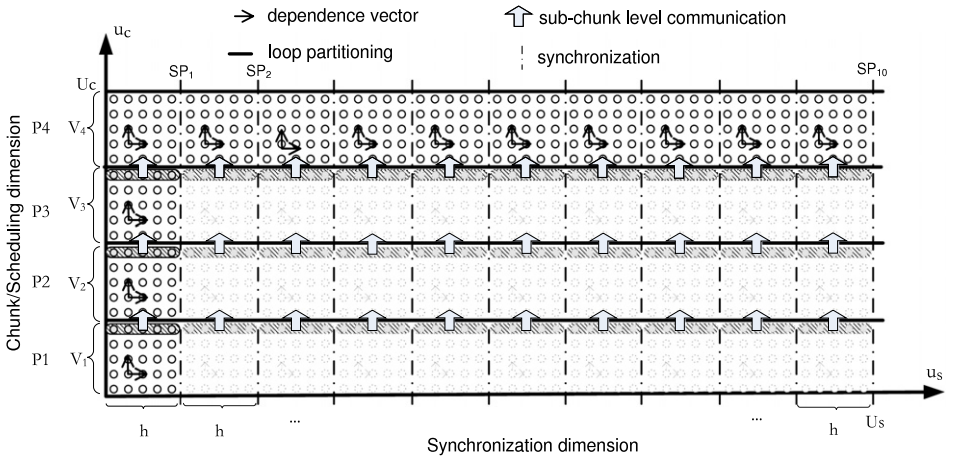

Fig. 3. A two-dimensional loop is partitioned into four chunks and ten synchronization points are placed in each chunk ($NP = N$).

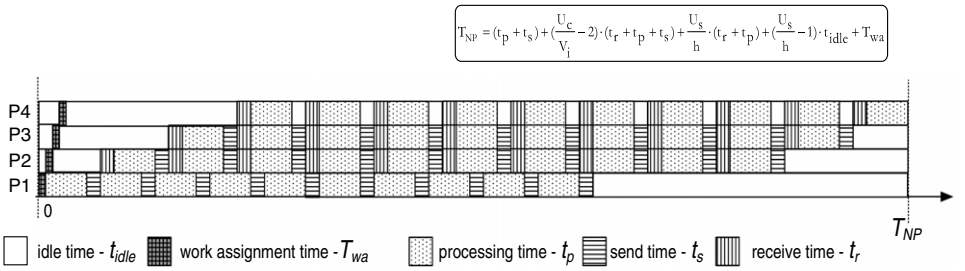

Fig. 4. Parallel execution flow on a homogeneous system with $NP = N$.

from the master processor. The master receives the first request from P1, calculates the size of the first executable chunk and then assigns it to P1. The master continues to serve the other incoming work requests in the same fashion. On the worker processors side, P1 begins computing its assigned chunk. Due to the data dependencies, P2 can start receiving and computing its assigned chunk only after P1 has computed its first subchunk and sent the necessary data to P2. This occurs at the first synchronization point. Similarly, P3 can begin computing only after P2 has sent the required data, again at the first synchronization point but at a later time. This is the idle time t_{idle} shown in Fig. 4 by the horizontal white strip. The same holds for P4, except for the sending part. P4 does not need to send any data since it is the last worker processor. However, it must wait for P3 to send the necessary data, thus introducing an expected idle time, as depicted in Fig. 4 by the white small boxes between computing and receiving events. In a homogeneous system, V_i and c_p are the same for all processors. Therefore, the time needed to compute the iterations in a subchunk, $t_p = hV_ic_p$, is the same for all processors. The communication between two workers at the subchunk level consists of two parts: the sending part and the receiving part. The send operation takes $t_s = c_d + hc_c$ as does the receive operation, according to our communication model.

When $NP = N$, the total number of chunks is NP and the size of each chunk is $V_i = U_c/NP$. The theoretical parallel time for this situation, denoted T_{NP}, is the completion time of the last subchunk of the problem (in our example this is the last subchunk of P4). T_{NP} can be estimated as the completion time of the highlighted subchunks in Fig. 3. For every chunk, except for the first and the last, a worker has to *receive*, *compute* and *send* data. For the first chunk, a worker needs only to *compute* and *send* data. Therefore, the time to *compute* each subchunk in the first chunk and to *send* the necessary data to the next worker is $t_p + t_s$. Likewise, for the last chunk, a worker needs only to *receive* and *compute* data, i.e., the time required to *receive* the necessary data and to *compute* every subchunk in the last chunk is $t_r + t_p$. The time needed to complete the first subchunk of chunks 2, 3, ..., $NP - 1$ is $(NP - 2)(t_r + t_p + t_s)$, where NP can be written as $NP = \frac{U_c}{V_i}$, since $NP = N$ (see Fig. 3).

Claim 3.1. *The time required to compute the last chunk is the product of the time $t_r + t_p$ required to compute any of its subchunks and the number $\frac{U_s}{h}$ of subchunks, including the expected total idle time, $\left(\frac{U_s}{h} - 1\right) t_{idle}$, spent waiting for the required data to be sent from the previous worker. We assume that t_{idle} is approximately equal to t_s, as illustrated in Fig. 4.*

The claim holds because, due to the data dependencies, workers cannot compute their assigned chunks in parallel without exchanging data throughout the execution of a chunk. A worker starts computing only after it receives the necessary data from the previous worker. Therefore, the total parallel time is the completion time of the worker that computes the last chunk of the problem. Since we consider the master–worker model, the work assignment time is $T_{wa} = t_r + c_{sch} + t_s$, where

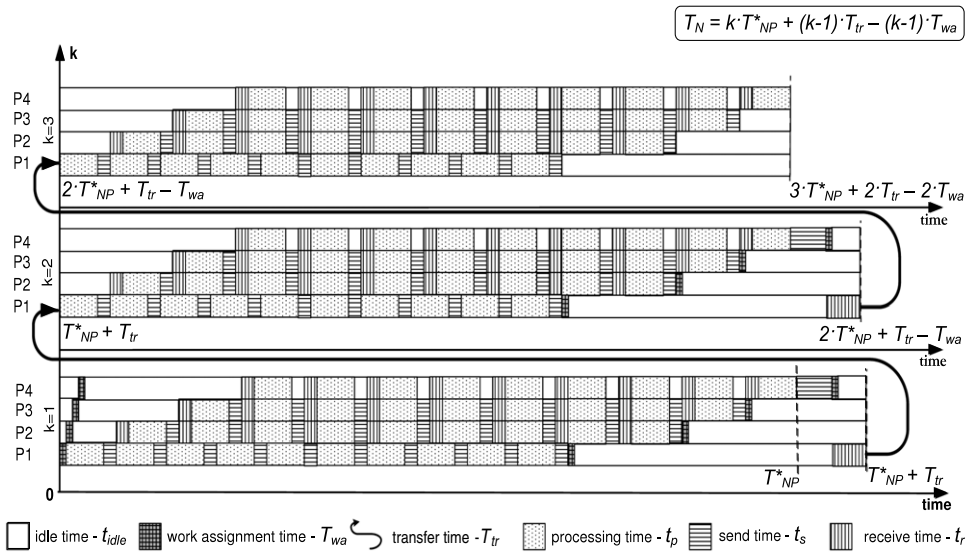

Fig. 5. Parallel execution flow on a homogeneous system with $NP < N$ and $k = 3$.

t_r is the transmission time needed for the work request to reach the master, t_s is the time needed for the master's reply to reach the worker, and c_{sch} is the time needed for the master to compute the next executable chunk size. Hence the total parallel time in this case is

$$T_{NP} = \left(t_p + t_s\right) + \left(\frac{U_c}{V_i} - 2\right)\left(t_r + t_p + t_s\right) + \frac{U_s}{h}\left(t_r + t_p\right) + \left(\frac{U_s}{h} - 1\right) t_{idle} + T_{wa}. \tag{3}$$

Note that in Eq. (3) T_{wa} is taken only once, because the work assignment time for every chunk is overlapped with the worker's computation or communication operations, except for the first chunk of the problem. Fig. 4 gives a timing diagram which clarifies formula (3). Using the above formula in conjunction with formulas (1) and (2), one can determine the theoretically optimal value of h for which T_{NP} is minimized. This is done by differentiating T_{NP} with respect to h, which yields

$$h_{NP} = \sqrt{\frac{2c_d U_s}{(U_c - V_i)c_p + \left(2\frac{U_c}{V_i} - 4\right)c_c}}. \tag{4}$$

Case $NP < N$. In this case, the number of chunks is greater than the number of available processors, so each processor is assigned *more than one* chunk.

Fig. 5 depicts such a situation, in which 12 chunks are assigned to four workers, each worker being assigned three chunks. In our analysis, we assume that all workers are assigned the *same number* of chunks. We address this case by assuming that a problem of size $U_c \times U_s$ with $NP < N$ processors can be decomposed into $k = N/NP$ subproblems of size $U^\star_c \times U_s$, where $U^\star_c = \frac{U_c}{k}$ and $V_i = \frac{U^\star_c}{NP}$. These subproblems are interdependent, in the sense that part of the data produced by one subproblem is consumed by the next subproblem. Therefore the computation of a subproblem cannot start until the previous subproblem is completed. Upon completion of a subproblem, the processor assigned the last chunk of the previous subproblem transmits in a single message all necessary data to the processor assigned the first chunk of the next subproblem. The time to complete this data transfer is called T_{tr}, and is given by $T_{tr} = 2(c_d + U_s c_c)$. T_{tr} designates the time required to send and receive a data packet of size equal to the size of the scheduling dimension. Fig. 5 illustrates such a problem where $NP < N$. The entire problem is divided into three subproblems, since $N/NP = 3$ and the same set of four processors is used for each subproblem.

In the general case, assuming that the initial problem is divided into $k = N/NP$ subproblems, the parallel time for the completion of the *first* subproblem is $T^\star_{NP}$, which is the time required to compute a subproblem of size $U^\star_c \times U_s$. The parallel time for the completion of every subsequent subproblem is $T^\star_{NP} - T_{wa}$, because the work assignment time for every chunk is overlapped with other computation or communication operations, except for the chunks of the first subproblem. Finally, the time to transfer the necessary data between all the successive subproblems is $(k-1)T_{tr}$.

Claim 3.2. *When* $NP < N$, *the total parallel time, denoted by* T_N, *is*

$$T_N = kT^\star_{NP} + (k-1)T_{tr} - (k-1)T_{wa}. \tag{5}$$

To see that the above claim is true, one must take into account the existence of data dependencies. These dependencies are responsible for the fact that the execution of each subproblem can start only *after* the first worker of a subproblem receives

the necessary data from the last worker of the previous subproblem. Therefore, the total parallel time is the completion time of the last subproblem, i.e., the time when the last chunk of the initial problem is computed, plus the time required for transferring the partial results from one subproblem to the next. As in the previous case, one can determine the theoretical optimal value of h for which T_N is minimized, by differentiating T_N with respect to h:

$$(T_N)' = k(T_{NP})' + ((k-1)T_{tr})' + ((k-1)T_{wa})', \tag{6}$$

where $T'_{tr} = 0$ and $T'_{wa} = 0$. This yields

$$h_N = \sqrt{\frac{2c_d U_s}{(U_c^\star - V_i)c_p + \left(2\frac{U_c^\star}{V_i} - 4\right)c_c}}. \tag{7}$$

4. Model for parallel time prediction in heterogeneous systems

To describe our theoretical model for dedicated heterogeneous systems we use the following additional notation.

- α_j, $j = 1, \ldots, \xi$—the worker types of the heterogeneous system.
- $NP^{\alpha_j} \geq 1$—the number of processors of type α_j, $j = 1, \ldots, \xi$.
- VP^{α_j}—the virtual computing power of a worker of type α_j. The virtual computing power for each machine type is established as the normalized execution time of the same test program on each machine type.
- $V_i^{\alpha_j}$—the length of the projection of chunk i on u_c, assigned to a worker of type α_j and weighted according to its virtual power.
- $c_p^{\alpha_j}$—the computation time per iteration on a worker of type α_j.
- $t_p^{\alpha_j}$—the computation time of $hV_i^{\alpha_j}$ iterations on a worker of type α_j.

In the heterogeneous case, the processor loads are taken into account in the evaluation of their virtual computing power in the scheduling scheme. These virtual powers are used in the proposed performance evaluation model. The validity of our model is, again, based on the two premises outlined in the beginning of Section 3. In order to derive a formula for the theoretical parallel time on a heterogeneous system, we first devise a formula for the case when $NP = N$, which we then use to infer the formula for the case $NP < N$. We assume that workers are assigned work in *decreasing* order of their virtual computing power. This implies that the last chunk is assigned to the slowest worker. In order to satisfy the premise ($\mathbf{PR}_1$), which asserts that the time to compute a subchunk is the same in all cases and for all worker types, the size of the chunk must be *weighted* according to the virtual computing power of each worker type. We assume that workers are grouped according to their type and that the group of workers having the greatest computational power per worker gets assigned work first.

Case $NP = N$. An illustration of this case, for a small-scale iteration space partitioned into four chunks assigned to four heterogeneous workers, would be similar to the one in Fig. 3, with the difference that the chunk sizes V_i would vary according to the virtual computing powers of the four workers. As in the $NP = N$ case for homogeneous systems, the parallel time is given by the completion time of the last subchunk of the problem. The completion time of the first subchunk assigned to the *last* worker is the total time required for the completion of the first subchunk of all previous chunks. In this scheme, workers are grouped according to their type, and each worker is assigned exactly one chunk. For all workers of type α_j, the total time for the completion of all corresponding *first* subchunks is $NP^{\alpha_j}(t_r + t_p^{\alpha_j} + t_s)$. Therefore, the total time for all worker types is given by $\sum_{j=1}^{\xi}(NP^{\alpha_j})(t_r + t_p^{\alpha_j} + t_s)$. From this sum we subtract t_r and t_s, since there is no receive operation for the first subchunk of the problem, and no send operation for the last subchunk of the problem. The time required for the computation of the remaining subchunks of the last chunk is $\left(\frac{U_s}{h} - 1\right)(t_r + t_p^{\alpha_\xi} + t_{\text{idle}})$, and this corresponds to a worker of type α_ξ, which is the slowest type of worker in the heterogeneous system.

Claim 4.1. *When $NP = N$, the theoretical parallel time of a heterogeneous system is*

$$T_{NP} = \sum_{j=1}^{\xi}(NP^{\alpha_j})\left(t_r + t_p^{\alpha_j} + t_s\right) - t_r - t_s + \left(\frac{U_s}{h} - 1\right)\left(t_r + t_p^{\alpha_\xi} + t_{idle}\right) + T_{wa}. \tag{8}$$

The proof of this claim is the same as the one for Claim 3.1. To determine the optimal synchronization interval, denoted by h_{NP}, we differentiate T_{NP} with respect to h, and obtain

$$h_{NP} = \sqrt{\frac{2c_d U_s}{\sum_{j=1}^{\xi}(NP^{\alpha_j})(V_i^{\alpha_j}c_p^{\alpha_j} + 2c_c) - V_i^{\alpha_\xi}c_p^{\alpha_\xi} - 4c_c}}. \tag{9}$$

Case $NP < N$. In this case, the number of chunks is greater than the number of available processors, so each processor is assigned *more than one* chunk. Chunks are weighted according to the virtual computing powers of the workers.

Claim 4.2. *The theoretical parallel time of a heterogeneous system when NP < N is given by*

$$T_N = kT^{\star}_{NP} + (k-1)T_{tr} - (k-1)T_{wa}, \tag{10}$$

where

- $T^{\star}_{NP}$ *is the parallel time according to the heterogeneous case* $NP = N$ *for a subproblem of size* $\frac{U_c}{k} \times U_s$,
- $T_{tr} = 2(c_d + U_s c_c)$, *and*
- $T_{wa} = t_r + c_{sch} + t_s$.

The proof of this claim is similar to that of Claim 3.2. The theoretical optimal synchronization interval h_N that minimizes the parallel time is determined by differentiating T_N with respect to h:

$$(T_N)' = k(T_{NP})' + ((k-1)T_{tr})' + ((k-1)T_{wa})'. \tag{11}$$

Taking into account that $T'_{tr} = 0$ and $T'_{wa} = 0$, we conclude that

$$h_N = \sqrt{\frac{2c_d U_s}{\sum_{j=1}^{\xi}(NP^{\alpha_j})(V_i^{\alpha_j} c_p^{\alpha_j} + 2c_c) - V_i^{\alpha_\xi} c_p^{\alpha_\xi} - 4c_c}}. \tag{12}$$

Formula (12) is similar to formula (9), with the difference that the values of $V_i^{\alpha_j}$ are not equal, since each worker is assigned more than one chunk.

5. Experimental validation

The accuracy of the proposed theoretical model for parallel time prediction in homogeneous and heterogeneous systems can be established only through comparison with experimental tests. We computed a set of theoretical results analytically. The parallel times were plotted as a function of the synchronization interval using formulas (3), (5), (8) and (10). For the experimental results, we ran a series of tests for different values of the synchronization interval h, measured the actual execution time, and plotted it in order to find the actual optimal synchronization interval. In every case, the comparison between the theoretical and the experimental curve shows that, even though the two curves are not identical, the size of the optimal synchronization interval (which in turn defines the synchronization frequency) is predicted by the theoretical curve with a high accuracy. As the experiments demonstrate, using the theoretical optimal synchronization interval given by formulas (4), (7), (9) and (12) results in a parallel time that deviates 0.02616 s from the actual optimal value in the *best case*, and 0.1040 s in the *worst case*. This shows that, despite the simplifying assumptions we made in order to arrive at a linear model, the model itself is both accurate and robust. The experimental measurements were taken using caching. This means that our model is not sensitive to caching. Taking caching into account would simply complicate the model without significant practical improvements.

5.1. Experimental environment

The computational kernel and the scheduling algorithm are both implemented in C, using MPI for inter-processor communication. The experiments were performed on two homogeneous clusters and one heterogeneous cluster.

Experiment #1: a homogeneous cluster of 11 Intel Pentium III machines called 'kids' with 500 MHz, 512 MB RAM, with virtual power $VP^{kid} = 0.63$.

Experiment #2: a homogeneous cluster of 7 Intel Pentium III machines called 'twins' with 800 MHz, 256 MB RAM, with virtual power $VP^{twin} = 1$.

Experiment #3: a heterogeneous cluster of 5 'kids' and 6 'twins' (one 'twin' was used as master).

We measured the virtual computing power of each worker by running a small test problem (which involved nested loops with floating point operations) 10 times, serially, on each computer and averaging the measured execution times. The machines are interconnected by a 100 Mbits/s fast Ethernet network. The results given in the following subsections are the average of 10 runs for each experiment. As the application test case we used the Floyd–Steinberg computational kernel [24], an image-processing algorithm used for error-diffusion dithering of a *width* by *height* grayscale image. The pseudocode is given below, and it has four unitary data dependencies. Note that although this kernel has unitary dependencies, the scheduling method and the theoretical model are not limited to kernels with unitary data dependence vectors. Our basic premise is that the data dependence vectors are such that they allow the parallelization of the computation, meaning that the computational kernel is not inherently sequential. For this to hold, the data dependencies must be strictly greater than $\vec{\mathbf{0}} = (0, \ldots, 0)$ when ordered lexicographically (see [26]). These data dependence vectors are known in advance.

Table 1
Estimated model parameters.

Exp.	c_d (μs)	c_c (μs)	c_p^{kids} (μs)	c_p^{twins} (μs)	c_{sched} (μs)	t_{m-s} (μs)
#1	99	0.69	0.526	–	500	1.253
#2	99	0.65	–	0.319	500	1.25
#3	99	0.69	0.526	0.319	650	1.25

```
/* 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);
    }
}
```

We compare our model for the case $NP = N$ with the model of Chen and Xue [22]. We chose to compare with their work because, to the best of our knowledge, it is the only one to address this problem for heterogeneous systems. They consider only block assignment of tiles, on both homogeneous and heterogeneous systems, which corresponds only to our $NP = N$ case for homogeneous and heterogeneous systems.

5.2. Estimation of parameters

To quantify the *communication* parameters of our model, we developed a benchmark program in C, using MPI, that simulates a small scale master–worker model with inter-node communication. It performs send and receive calls between all pairs of worker processors for messages of different sizes. We chose to perform data exchanges between all pairs of workers in order to simulate levels of network contention similar to those of the actual application. We measured the average round-trip time for all data exchanges. We then halved this average time, assuming that the send (t_s) and receive (t_r) times are equal. This is a simple but realistic assumption, since in most cases the send and receive operations are executed in pairs between communicating workers. This allows us to estimate the start-up time c_d and the transmission cost per unit and type of data c_c. Their values are given in Table 1. The computation cost per iteration is application and processor dependent. In our model, we assume that it is the same for every iteration of the same application. To accurately quantify the *computation per iteration* cost, we ran a small subset of the selected application, i.e., the Floyd–Steinberg kernel, on each processor type. We divided the total computation time by the total number of iterations to obtain the c_p for this application. The values for each machine type (*kids* and *twins*) are also given in Table 1.

5.3. Description of the experiments

In the following series of experiments, we determine the actual optimal synchronization interval $h_{m,a}$, which gives the minimum actual parallel time. We then compare $h_{m,a}$ with the theoretical optimal value $h_{m,t}$, which is obtained from formulas (4), (7), (9) and (12), depending on the case under examination. The maximum possible value of h is actually the size of the synchronization dimension u_s. This scenario is never realized in practice because it would serialize the computation. The difference between $h_{m,t}$ and $h_{m,a}$ is measured as a percentage of u_s.

Experiment #1. This experiment was performed on the homogeneous cluster of 10 + 1 kids. We ran the Floyd–Steinberg kernel for an image size of 10K × 20K pixels (K = 1000).

(a) Initially we examined the case where $NP = N$ and compared our results with the results of Chen and Xue [22]. They compute the vertical tile size using formula (4) of [22]: $n_1^{opt} = N_1/P$. This can be written in our terms as $V_i^{kids} = U_c/NP$. Similarly, they compute the horizontal tile size as $n_2^{opt} = \sqrt{\frac{P(\alpha_s+\alpha_r+\gamma(P-1))N_2}{(P-1)(N_1 tc+\beta_s+\beta_r)}}$, which in our notation becomes $h_{NP} = \sqrt{\frac{NP(c_d+c_d)U_s}{(NP-1)(U_c c_p^{kids}+2c_c)}}$. For the Floyd–Steinberg kernel on an image size of 10K×20K pixels, their formulas give a vertical tile size equal to $V_i^{kids} = 1000$, a horizontal tile size equal to $h_{NP} = 29$, and a parallel time of $T_{NP} = 10.8359$ s. The values obtained by our model for the same case are chunk size equal to $V_i^{kid} = 1000$, $h_{NP} = 30$, and parallel time of $T_{NP} = 10.8361$ s. It is obvious that both Chen and Xue's model and ours predict that the minimum execution time is obtained for approximately the same synchronization interval.

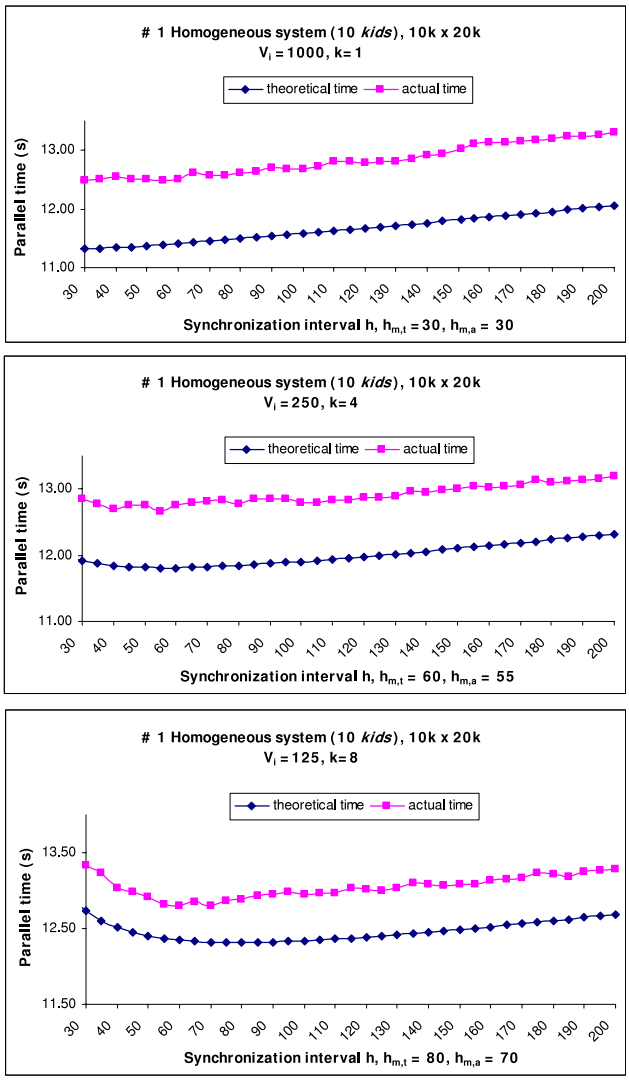

Fig. 6. Theoretical versus actual parallel time on a homogeneous system with ten workers and $k = 1, 4$ and 8 subproblems.

(b) Next we studied the case where $NP < N$. Each worker was assigned k chunks, where k assumed the values of 4 and 8. V_i was computed according to following formula:

$$V_i = \frac{U_c}{kNP}. \tag{13}$$

In Fig. 6, we have plotted the theoretical versus the actual parallel times on the *kids* homogeneous cluster. The theoretical time was obtained using the constants from Table 1 (Exp. #1). The curve representing the theoretical parallel time has a global minimum (i.e., the parallel time is minimized for $h_{m,t}$), and for every value of h greater than $h_{m,t}$ the parallel time increases approximately linearly with a (very) small slope. Respectively, for every value of h less than $h_{m,t}$, excessive synchronization occurs which results in significant performance degradation. As can be seen in Fig. 6, the actual parallel time fits the theoretical pattern. The actual parallel time is always greater than the corresponding theoretical time. In Fig. 6, the global minimum of the actual parallel time is positioned to the right of the theoretical global minimum. In all cases the difference between these two values, $h_{m,a}$ and $h_{m,t}$, and the actual time for each of these intervals can be seen in Table 2. The different values in Table 2 show that it is possible to execute the parallel application using the theoretical optimal synchronization interval $h_{m,t}$, which in practice gives a parallel time very close to the actual minimum parallel time. Using the theoretical optimal h_N given by Eq. (7), one can expect to deviate from the optimal actual parallel time by less than $+0.0069\%$, as shown in Table 2, column 7.

Experiment #2. To confirm the validity of our theoretical model, we tested it in a second homogeneous cluster with other characteristics (i.e., processor speed, memory size) than the first homogeneous system. We repeated the measurements of

Table 2
Theoretical and actual optimal synchronization interval, actual parallel time for both intervals, the difference in the actual parallel time, and the deviation from the minimum actual parallel time.

k	$h_{m,t}$	$h_{m,a}$	$T_a(h_{m,t})$ (s)	$T_a(h_{m,a})$ (s)	$\lvert T_a(h_{m,t}) - T_a(h_{m,a})\rvert$ (s)	$\frac{\lvert T_a(h_{m,t}) - T_a(h_{m,a})\rvert}{T_a(h_{m,a})}$ (%)	$\lvert h_{m,t} - h_{m,a}\rvert$	$\frac{\lvert h_{m,t} - h_{m,a}\rvert}{U_s}100$ (%)
Exp. #1: Homogeneous cluster with 10 + 1 kids								
1	30	30	12.4787	12.4787	0.00000	0.0000	0	0.0000
4	60	55	12.7500	12.6627	0.08730	0.0069	5	0.0250
8	80	70	12.8900	12.7982	0.09180	0.0072	10	0.0500
Exp. #2: Homogeneous cluster with 6 + 1 twins								
1	40	55	7.04723	7.03397	0.01326	0.0019	15	0.0750
4	75	70	7.18130	7.15514	0.02616	0.0037	5	0.0250
8	110	100	7.29295	7.28574	0.00721	0.0010	10	0.0500
Exp. #3: Heterogeneous cluster with 5 kids and 5 + 1 twins								
1	35	35	10.2356	10.2356	0.0000	0.0000	0	0.0000
4	65	100	10.6341	10.5301	0.1040	0.0099	35	0.1750
8	90	105	10.8348	10.7911	0.0437	0.0040	15	0.0750

the first experiment, but on the 6 + 1 twins homogeneous cluster, running the Floyd–Steinberg kernel for an image size of 8K × 16K pixels (K = 1000).

(a) Initially, we examined the case where $NP = N$ and compared our results with the results of Chen and Xue [22]. They use the formulas $n_1^{opt} = N_1/P$ and $n_2^{opt} = \sqrt{\frac{P(\alpha_s+\alpha_r+\gamma(P-1))N_2}{(P-1)(N_1tc+\beta_s+\beta_r)}}$ to compute the vertical and horizontal tile size, respectively. Using our notation, the above formulas can be written as $V_i^{twin} = U_c/NP$ and $h_{NP} = \sqrt{\frac{NP(c_d+c_d)U_s}{(NP-1)(U_cc_p^{twin}+2c_c)}}$, respectively. For the Floyd–Steinberg kernel on 8K × 16K pixels, their formulas give a vertical tile size equal to $V_i^{twins} = 1333$, a horizontal tile size equal to $h_{NP} = 39$, and the parallel time of $T_{NP} = 6.9987$ s. Our values in this case were chunk size $V_i^{twin} = 1334$, $h_{NP} = 40$, and parallel time $T_{NP} = 6.9988$ s. Again the results obtained by Chen and Xue are very similar to ours.

(b) For the case where $NP < N$, each worker was assigned k chunks, where k assumed again the values of 4 and 8, and V_i was computed using formula (13). Fig. 7 shows the theoretical versus the actual parallel time. As in the previous case, the theoretical time was obtained using the parameters from Table 1 (Exp. #2). The curve of the theoretical parallel time follows the curve of the actual parallel time. The differences between $h_{m,t}$ and $h_{m,a}$ and the differences between their corresponding actual parallel times are given in Table 2. Using the theoretical optimal h_N given by Eq. (7), one can expect to deviate from the minimum actual parallel time by less than 0.0063%, as shown in Table 2, column 7.

Experiment #3. In this experiment, we tested our model in a heterogeneous cluster of 5 kids and 5 + 1 twins. We ran the Floyd–Steinberg kernel on an image size of 10K × 20K pixels (K = 1000).

(a) For the heterogeneous case when $NP = N$ (and $k = 1$), the vertical tile size given by formula (8) from [22] is $n_1^i = N_1\frac{C_i}{\sum_P^{i=1}C_i}$. In our notation, this becomes $V_i^{\alpha_j} = U_c\frac{V_iP^{\alpha_j}}{\sum_\xi^{j=1}(VP^{\alpha_j}NP^{\alpha_j})}$. Similarly, the horizontal tile size is $n_2^i = n_2^{opt} = \sqrt{\frac{P(\overline{\alpha_s}+\overline{\alpha_r}+\overline{\gamma}(P-1))N_2}{(P-1)(N_1\overline{tc}+\overline{\beta_s}+\overline{\beta_r})}}$, which in our notation becomes $h_{NP=N_{opt}}^{ht} = \sqrt{\frac{\sum_\xi^{j=1}NP^{\alpha_j}(c_d+c_d)U_s}{\left(\sum_\xi^{j=1}NP^{\alpha_j}-1\right)\left(U_c\overline{c_p^{\alpha_j}}+2c_c\right)}}$. For the Floyd–Steinberg kernel and an image size of 10K × 20K, the vertical tile size for a twin worker is $V_i^{\text{twin}} = 1223$, and for a kid worker it is $V_i^{kid} = 776$; the horizontal tile size is $h_{NP=N_{opt}}^{ht} = 162$ and the parallel time $T_{NP}^{ht} = 8.8273$ s. For the V_i^{twin}, V_i^{kid} and h values obtained with Chen and Xue's model, our model gives a theoretical parallel time of $T_{NP}^{ht} = 8.8316$ s. The optimal theoretical parallel time predicted by our model is smaller; specifically, it is $T_{NP}^{ht} = 8.4501$ s, and is obtained for $h_{NP=N_{opt}}^{ht} = 35$. In this case, as can be seen in Table 2, our model gives better estimates of $h_{a,t}$, which is equal to 35.

(b) For the case where $NP < N$, each worker was assigned k chunks, where k assumed again the values of 4 and 8. The value of V_i was different for each node type. Because all chunks were *weighted* according to the virtual computing power of the worker, twin workers received larger chunks than kid workers. The chunks were weighted as follows:

$$V_i^{kid} = \frac{U_c}{kNP}VP^{kid} \tag{14}$$

$$V_i^{twin} = \frac{U_c}{kNP}VP^{twin}. \tag{15}$$

The theoretical and actual parallel times for this experiment are plotted in Fig. 8. The theoretical time was obtained using the parameters from Table 1 (Exp. #3). Also, the curve of the theoretical parallel time follows the curve of the actual parallel time, and the differences between $h_{m,t}$ and $h_{m,a}$, and between their corresponding parallel times, are given in Table 2. The

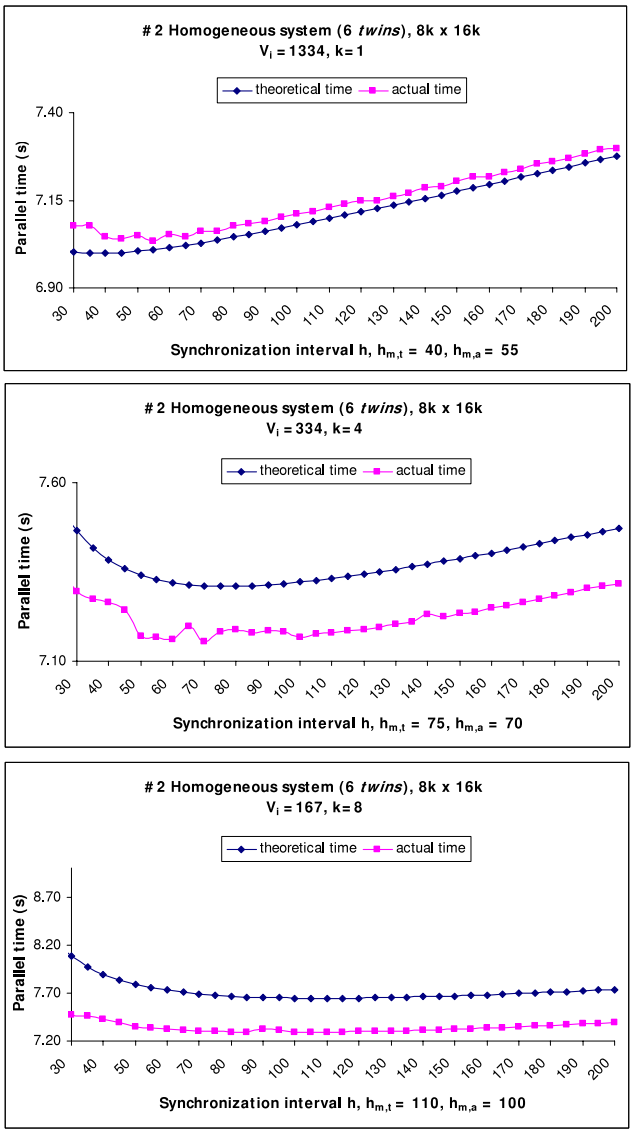

Fig. 7. Theoretical versus actual parallel time on a homogeneous system with six workers and $k = 1$, 4 and 8 subproblems.

values in Table 2, column 7, signify that, using the theoretical optimal h_N given by Eq. (12), one can expect to deviate from the optimal parallel time by less than 0.0062%.

6. Conclusions and future work

In this paper, we have proposed and evaluated a model that determines the optimal synchronization frequency for a typical dynamic self-scheduling algorithm designed to parallelize loops with data dependencies on homogeneous and heterogeneous clusters. The accuracy of the proposed model is confirmed in all cases by experimental results. The main contribution of this work is the fact that formulas (4), (7), (9) and (12) provide the means for approximating the optimal synchronization frequency. For every experiment, the theoretically predicted optimal synchronization interval is very close to the actual optimal synchronization interval obtained from practical measurements. The performance loss corresponding to the difference between the actual and the theoretically predicted optimal h is very small, being in the range 5–10% for all experiments. Moreover, the cost of determining the optimal synchronization interval through extensive testing is clearly prohibitive, and a poor choice of the synchronization interval value leads to increased performance loss. Finally, the proposed theoretical model improves on a previously existing model [22] for heterogeneous systems, while obtaining similar results on homogeneous systems.

Future work. The processors in real-life systems are not always dedicated to the applications they execute. This means that the available computing power fluctuates during the execution of an application. Ongoing efforts exist for extending

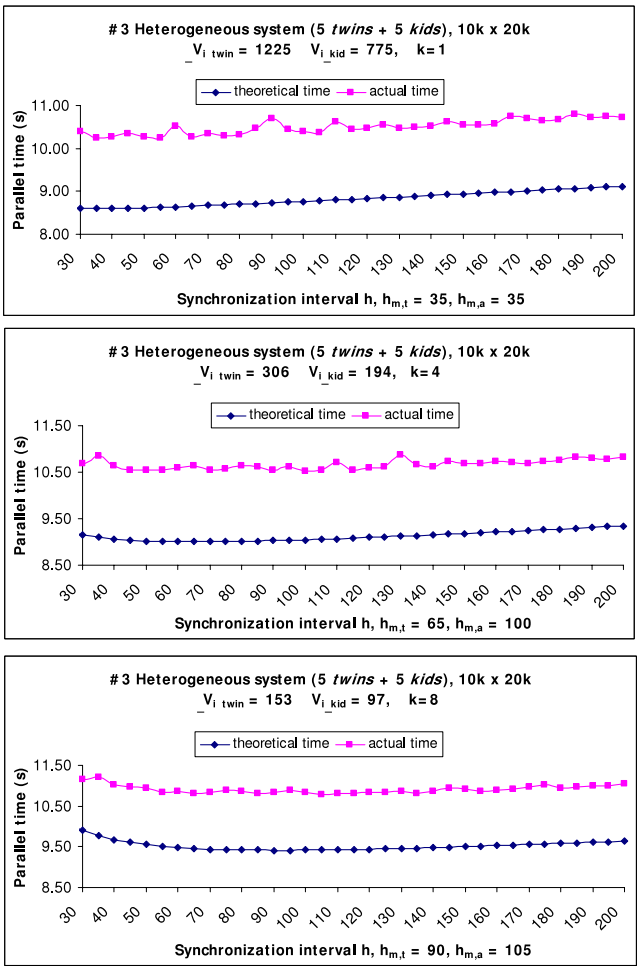

Fig. 8. Theoretical versus actual parallel time on a heterogeneous system with ten workers and $k = 1$, 4 and 8 subproblems.

the proposed model to estimate the optimal synchronization frequency in cases where the computational speed of a processor varies unpredictably. In the future, we plan to modify our model to include a threshold for the computing power of each worker, below which the master would not assign work to the worker or the worker would not make a request for work to the master. This is expected to increase the overall performance of the application by eliminating some of the communication costs that cannot be compensated by the high computation costs associated with slow processors. We also intend to investigate the possibility of extending our model to deal with more sophisticated communication schemes than the one-port model, such as the bounded multi-port model.

Acknowledgements

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.

References

[1] C.P. Kruskal, A. Weiss, Allocating independent subtasks on parallel processors, IEEE Trans. Softw. Eng. 11 (10) (1985) 1001–1016.
[2] 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.
[3] T.H. Tzen, L.M. Ni, Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers, IEEE Trans. Parallel Distrib. Syst. 4 (1) (1993) 87–98.
[4] S.F. Hummel, E. Schonberg, L.E. Flynn, Factoring: a method for scheduling parallel loops, Commun. ACM 35 (8) (1992) 90–101.
[5] I. Banicescu, Susan Flynn Hummel, Balancing processor loads and exploiting data locality in *N*-body simulations, SC, 1995.
[6] I. Banicescu, Z. Liu, Adaptive factoring: a dynamic scheduling method tuned to the rate of weight changes, in: Proc. of the High Perf. Comp. Symp. 2000, Washington, USA, 2000, pp. 122–129.
[7] I. Banicescu, V. Velusamy, Load balancing highly irregular computations with the adaptive factoring, in: Proceedings of the 16th International Parallel and Distributed Processing Symposium, 2002.
[8] I. Banicescu, V. Velusamy, J. Devaprasad, On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring, Cluster Comput. 6 (2003) 215–226.
[9] A.T. Chronopoulos, S. Penmatsa, J. Xu, S. Ali, Distributed loop-scheduling schemes for heterogeneous computer systems, Concurr. Comput.: Pract. Exp. 18 (7) (2006).

[10] A.T. Chronopoulos, R. Andonie, M. Benche, D. Grosu, A class of distributed self-scheduling schemes for heterogeneous clusters, in: Proc. of the 3rd IEEE Int. Conf. on Cluster Computing, CLUSTER'01, Newport Beach, CA, USA, 2001.
[11] C.-T. Yang, K.-W. Cheng, K.-C. Li, An efficient load balancing scheme for grid-based high performance scientific computing, in: Proc. of the 19th Int. Conf. on Advanced Info. Netw. and Apps, AINA'05, Tamkang University, Taiwan, 2005.
[12] A. Kejariwal, A. Nicolau, C.D. Polychronopoulos, History-aware self-scheduling, in: Proc. of the 2006 Int'l Conf. on Par. Proc., ICPP'06, Columbus, Ohio, USA, 2006, pp. 185–192.
[13] F.M. Ciorba, T. Andronikos, I. Riakiotakis, A.T. Chronopoulos, G. Papakonstantinou, Dynamic multi phase scheduling for heterogeneous clusters, in: Proc. of the 20th IEEE Int'l Par. & Dist. Proc. Symp., IPDPS'06, Rhodes, Greece, 2006.
[14] I. Riakiotakis, F.M. Ciorba, T. Andronikos, G. Papakonstantinou, Self-adapting scheduling for tasks with dependencies in stochastic environments, in: Proc. of the 5th Int'l HeteroPar'06 Workshop of CLUSTER'06, Barcelona, Spain, 2006.
[15] F.M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, A.T. Chronopoulos, Enhancing self-scheduling algorithms via synchronization and weighting, J. Parallel Distrib. Comput. 68 (2) (2008) 246–264.
[16] F. Desprez, P. Ramet, J. Roman, Optimal grain size computation for pipelined algorithms, Euro-Par 1 (1996) 165–172.
[17] R. Andonov, S. Rajopadhye, Optimal orthogonal tiling of 2-D iterations, J. Parallel Distrib. Comput. 45 (2) (1997) 159–165.
[18] D.K. Lowenthal, Accurately selecting block size at run time in pipelined parallel programs, Int. J. Parallel Program. 28 (3) (2000) 245–274.
[19] J. Xue, Loop Tiling for Parallelism, Kluwer Academic Publishers, 2000.
[20] J. Xue, W. Cai, Time-minimal tiling when rise is larger than zero, Parallel Comput. 28 (6) (2002) 915–939.
[21] M.M. Strout, L. Carter, J. Ferrante, B. Kreaseck, Sparse tiling for stationary iterative methods, Int. J. High Perform. Comput. Appl. 18 (1) (2004) 95–113.
[22] S. Chen, J. Xue, Partitioning and scheduling loops on NOWs, Comput. Commun. J. 22 (1999) 1017–1033.
[23] F.M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, A.T. Chronopoulos, Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems. Poster Presented at the 16th Int. Conf. on Parallel Architectures and Compilation Techniques, PACT, Brasov, Romania, September 15–19, 2007.
[24] R.W. Floyd, L. Steinberg, An adaptive algorithm for spatial grey scale, Proc. Soc. Inf. Display 17 (1976) 75–77.
[25] E.P. Markatos, T.J. LeBlanc, Using processor affinity in loop scheduling on shared-memory multiprocessors, IEEE Trans. Parallel Distrib. Syst. 5 (4) (1994) 379–400.
[26] D.I. Moldovan, Parallel Processing: From Applications to Systems, M. Kaufmann, 1993.

Theodore Andronikos received his Diploma and his Ph.D. in Computer Engineering from the School of Electrical and Computer Engineering, National Technical University of Athens. His research interests are primarily in designing dynamic algorithms for parallel and distributed systems, emphasizing in large-scale heterogeneous systems and GRIDS. He is also interested in the use of temporal logics for the automatic verification and synthesis of reactive systems. He currently holds the position of Lecturer at the Department of Informatics, Ionian University.

Florina M. Ciorba received her B.Sc. Degree in Computer Engineering from the University of Oradea, Romania, in 2001 and her Ph.D. in Computer Science from the National Technical University of Athens, Greece, in 2008. She is currently a Postdoctoral Research Associate at Mississippi State University. She has published 6 journals and 13 peer-reviewed conference proceedings publications in the area of Parallel and Distributed Computing, High Performance Computing and Scientific Computing. Her research interests are in the field of scientific computing and include, among others: high-performance computing, autonomic computing systems, dynamic load balancing, parallel algorithms, robust algorithms, energy-efficient algorithms, performance modeling and prediction, scheduling algorithms for emerging parallel architectures (multicore, manycore, SMPs), performance evaluation & optimization, large-scale multiscale scientific simulations. She is a member of IEEE since 2004 and ACM since 2007.

Ioannis Riakiotakis received his B.Sc. in Electronics Engineering in 1999, his M.Sc. in Communication Engineering and Signal Processing in 2000 both from the University of Plymouth (UK), and his Ph.D. in Computer Science from the National Technical University of Athens (Greece), in 2008. His research interests include distributed computing, dynamic load balancing algorithms.

George Papakonstantinou received his Diploma in Electrical Engineering from the National Technical University of Athens in 1964, the P.I.I. Diploma in Electronic Engineering from Philips Int. Inst in 1966, and his M.Sc. in Electronic Engineering from N.U.F.F.I.C. 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) and as Director of the Computer Division at the Greek Atomic Energy Commission (1981–1984). From 1984 till 2009 he served as a Professor of Computer Engineering at the National Technical University of Athens. Currently he is Professor Emeritus at the same University.

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 300 papers in international referred journals and in proceedings of international conferences (more than 1000 citations, h-index: 16). He has also published 9 Greek textbooks and contributed 7 chapters in English textbooks, on computer engineering. He has taught several undergraduate and graduate courses at NTUA, supervised many diploma and Ph.D. theses, and has been reviewer in International Journals, Conferences, and research project proposals. He has participated (as project leader or member) in about 25 Greek and European R&D projects.

Anthony T. Chronopoulos received his Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 1987. He is currently a Professor in Computer Science at the University of Texas at San Antonio. He has published 45 journals and 59 peer-reviewed conference proceedings publications in the areas of Distributed and Parallel Computing, High Performance Computing, Scientific Computing. He has been awarded 15 federal/state government research grants. His work is cited in over 400 non-coauthors' research articles. He is a senior member of IEEE (since 1997).