CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2012; **24**:2302–2327
Published online 23 February 2012 in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/cpe.2812

Towards the optimal synchronization granularity for dynamic scheduling of pipelined computations on heterogeneous computing systems

I. Riakiotakis [1], F. M. Ciorba [2,*,†], T. Andronikos [3], G. Papakonstantinou [1] and A. T. Chronopoulos [4]

[1] *School of Electrical and Computer Engineering, National Technical University of Athens, 9, Heroon Polytechnioy, Zografou, 15773, Athens, Greece*
[2] *Center for Information Services and High Performance Computing, Technische Universität Dresden, Zellescher Weg 12/14, Dresden, 01062, Germany*
[3] *Department of Informatics, Ionian University, 7, Tsirigoti Square, 49100 Corfu, Greece*
[4] *Department of Computer Science, University of Texas at San Antonio, TX 78249, USA*

SUMMARY

Loops are the richest source of parallelism in scientific applications. A large number of loop scheduling schemes have therefore been devised for loops with and without data dependencies (modeled as dependence distance vectors) on heterogeneous clusters. The loops with data dependencies require synchronization via cross-node communication. Synchronization requires fine-tuning to overcome the communication overhead and to yield the best possible overall performance. In this paper, a theoretical model is presented to determine the granularity of synchronization that minimizes the parallel execution time of loops with data dependencies when these are parallelized on heterogeneous systems using dynamic self-scheduling algorithms. New formulas are proposed for estimating the total number of scheduling steps when a threshold for the minimum work assigned to a processor is assumed. The proposed model uses these formulas to determine the synchronization granularity that minimizes the estimated parallel execution time. The accuracy of the proposed model is verified and validated via extensive experiments on a heterogeneous computing system. The results show that the theoretically optimal synchronization granularity, as determined by the proposed model, is very close to the experimentally observed optimal synchronization granularity, with no deviation in the best case, and within 38.4% in the worst case. Copyright © 2012 John Wiley & Sons, Ltd.

Received 1 August 2008; Revised 10 December 2011; Accepted 11 December 2011

KEY WORDS: loops with data dependencies; pipelined computations; dynamic load balancing; communication model; performance prediction; performance evaluation; inter-processor communication; synchronization; heterogeneous systems

1. INTRODUCTION

A large number of algorithms has been devised for parallelizing and scheduling nested loops with and without data dependencies. In the case of loops with data dependencies, these dependencies are modeled as dependence distance vectors, and their existence incurs synchronization via cross-node communication during the execution of the parallelized loops. Ideally, the frequency of the cross-node communications should be set so that the synchronization overhead is kept to a minimum, while achieving the maximum degree of parallelism. In the coarse-grain decomposition approach, the problem domain, also called iteration space, is partitioned into chunks of iterations of equal

*Correspondence to: F. M. Ciorba, Center for Information Services and High Performance Computing, Technische Universität Dresden, Zellescher Weg 12/14, Dresden, 01062, Germany.
†E-mail: florina.ciorba@tu-dresden.de

or variable size, which are assigned to the existing processors. The class of self-scheduling algorithms is an important class of coarse-grain algorithms. Self-scheduling algorithms dynamically assign chunks of variable sizes to processors. They differ in the way they calculate the size of the chunk assigned to each processor. The self-scheduling algorithms were initially designed for loops *without* data dependencies, with a target on single-core processors and cluster systems [1–8]. Recent research results have been reported for designing loop self-scheduling methods for multicore, graphics processing unit, grid, and cloud systems [9–16]. The loop self-scheduling algorithms have recently been shown to efficiently be applicable for parallelizing loops *with* data dependencies, by inserting synchronization points at specific intervals throughout the execution of the work assigned to each processor [17, 18]. These new loop-scheduling algorithms target heterogeneous cluster systems, and have been implemented using an extended version of the master–worker model [18]. In the traditional master–worker model, the iteration space of an application containing loops with data dependence distance vectors is partitioned into chunks, and the chunks are assigned to workers by the master upon request. However, because of the data dependencies, the loop iterations in one chunk depend on loop iterations in other chunks. Hence, when the iterations need data for their local computations or when data are required in other workers' iterations for remote computations, in the extended master–worker model, the workers communicate with their neighboring co-workers at predefined synchronization intervals. The resulting synchronization mechanism involves partitioning of each chunk of iterations into subchunks. A large subchunk size signifies that inter-processor synchronization occurs less frequently at the cost of less parallelism and often poorer load balance [19]. It is therefore of interest to provide a model for determining the optimal granularity of synchronization, because inter-processor communication is one of the most important factors for performance degradation when parallelizing loops with data dependencies.

When scheduling applications on heterogeneous distributed systems, three major issues must be addressed: (i) *heterogeneity*, both machine and network related; (ii) *variable workload*, when the system is not dedicated and the workload varies unpredictably; and (iii) *communication overhead*, associated with distributing the work and/or exchanging data during the computation. To address heterogeneity in parallel and distributed systems, loop-scheduling schemes must take into account the delivered computational power of each computer in the system. The delivered computational power depends on CPU speed, memory, cache structure, and even the program type. Load balancing methods adapted to distributed environments take into account the computational powers of the machines [7, 17, 20–23]. These computational powers are used as relative weights that scale the size of the task assigned to each processor. The execution of loops with data dependencies on homogeneous or heterogeneous computing systems follows a pipeline execution (as explained later in Section 3). *Pipelining* is an approach that enables parallelism through explicit synchronization, although it involves a trade-off between synchronization granularity and degree of parallelism [19]. In this work, a theoretical value is derived for the synchronization granularity that minimizes the parallel execution time. Via extensive experimentation, the theoretically determined synchronization granularity is shown to be very close to the empirically observed synchronization granularity that gives the minimum parallel execution time.

The following notation is used in Table I:

- CSS: Chunk self-scheduling
- PSS: Pure self-scheduling
- GSS: Guided self-scheduling
- FSS: Factoring self-scheduling
- TSS: Trapezoid self-scheduling
- TFSS: Trapezoid factoring self-scheduling
- DTSS: Distributed TSS
- ArchConsc: Architecture conscious
- WF: Weighted factoring
- AWF: Adaptive weighted factoring
- AF: Adaptive factoring
- VOL: Volume of a polytope

DOI: 10.1002/cpe

Table I. Classification of several existing scheduling methods for loops with and without data dependencies on homogeneous and heterogeneous systems.

Loops without data dependencies	
Homogeneous systems	Heterogeneous systems
CSS [1], PSS [2], GSS [3], FSS [4, 39], TSS [5], Bitonic [37]	ArchConsc [37], WF [6], AF [7], TFSS [21], DTSS [40], AWF [8], VOL [20], HSS [22]
Loops with data dependencies	
Homogeneous systems	Heterogeneous systems
CB, IB, RSB, BP [29], OPS comm/comp ovrlp [24], 2-D OrthoTiling [25], RT-PPL [30], Loop tiling [26], RT w. & w/o caching [31], Time min tiling [27], Sparse tiling [28]	Loop tiling w. data alloc. [32], Var. size loop tiling [33], DMPS [17], SAS [23], $\mathcal{SW}$-CSS, FSS, GSS, TSS [18], h-**DCSS**, **DTSS**, **DFSS**, **DGSS**, DTFSS [34], h-CSS [35]

- HSS: History-aware self-scheduling
- CB: Coordinate bisection
- IB: Inertial bisection
- RSB: Recursive coordinate bisection
- BP: Block partitioning
- OPS: Optimal packet size
- RT-PPL: Runtime scheduling for partially parallel loops
- DMPS: Dynamic multi-phase scheduling
- SAS: Self-adapting scheduling
- $\mathcal{SW}$: synchronized & weighted
- h: synchronization granularity

Related work. In Table I, a classification of several existing approaches to scheduling loops with and without data dependencies for homogeneous and heterogeneous systems is given. Even though the focus of this work is on loops with data dependencies, Table I includes selected work on scheduling loops without data dependencies as well, because the dynamic loop-scheduling methods proposed therein have been used and modified to schedule loops with data dependencies in heterogeneous systems [17, 18]. The upper part of Table I presents the systems and applications that the earlier methods were originally devised for. This paragraph however elaborates on the past results referenced in the lower part of Table I as it is closely relevant to the work proposed here. A significant amount of work exists for determining the optimal partitioning (tile size, block size, and grain size) of loops with data dependencies for homogeneous systems ([24–28] and references therein). Ponnusamy *et al*. [29] used the inspector/executor model to predict the communication requirements of the code (assuming the data access patterns may change), and carry out communication optimization at runtime for irregular concurrent problems. Huang *et al*. [30] presented a scheduler that divides a partially parallel loop into independent wavefronts (called chunks), and then concurrently executes the iterations within a chunk. Lowenthal *et al*. [31] proposed a method for accurately selecting at runtime the block size in pipelined programs, in which the computation assigned to each processor is performed in blocks. The block size (the amount of computation before sending a message) is selected at runtime according to the workload variations that occur with highly irregular programs. However, the problem of finding the optimal partitioning of loops with data dependencies for heterogeneous systems has not been studied to the same extent. Boulet *et al*. [32] were the first to apply loop tiling on heterogeneous systems. They divide fully permutable loops into equally sized tiles

DOI: 10.1002/cpe

and allocate blocks with more tiles to faster processors. In order to achieve a good tile allocation, they use an algorithm that selects the best possible allocation based on a cost function. They avoid assigning variable size tiles in order to simplify code generation. Chen and Xue [33] assign tiles of variable size to processors based on a function that minimizes the parallel execution time according to a computation and a communication cost model. In their computation model, the computational power is modeled by CPU speed and memory size, whereas the communication cost model isolates the effects of send and receive operations, and quantifies network congestion. None of the aforementioned works for scheduling loop tasks for heterogeneous systems deal with the case of dynamically computing and allocating chunks of computations to processors. To the best of our knowledge, the work in [17] and [18] is the first to apply self-scheduling schemes to loops *with* data dependencies by inserting synchronization points throughout the parallel execution. The scheduling schemes used therein employ heuristic methods to determine the subchunk size. Hence, a method to determine the optimal subchunk size based on a theoretical model offers significant advantages, especially considering that a poor choice of the subchunk size leads to severe performance degradation, whereas the cost of finding the optimal value through exhaustive search is clearly prohibitive. In this work, the optimal synchronization granularity is studied for the scheduling schemes highlighted in bold font in Table I, namely *h*-**DCSS**, **DTSS**, **DFSS**, and **DGSS**.

Contributions. A theoretical model is developed in this paper for calculating the optimal granularity of synchronization, which minimizes the parallel execution time of loops with data dependencies that are parallelized using dynamic self-scheduling algorithms on heterogeneous systems. The case of limited and unlimited number of processors in a heterogeneous system is investigated, and the proposed model is applied for the CSS, FSS, GSS, and TSS self-scheduling algorithms on a heterogeneous distributed system. Previous efforts to address the same problem were presented [34, 35]. The work in this paper improves on both as follows. The communication model used herein is more sophisticated than the one in [34]. The current communication model parameters are application-independent and are estimated with the help of the `mpptest` from the `perftest` suite [36]. In contrast to the communication model used in [34], the current communication model takes into account the effects of message size and network contention on the communication cost, as the `mpptest` benchmark was specifically designed for this purpose at Argonne National Laboratory. The `mpptest` benchmark also exposes scalability issues. Compared with the work in [34], the current work also studies the case where the number of (virtual) workers is equal to the total number of chunks produced by a specific loop-scheduling scheme. Furthermore, new and significantly more extensive experimental tests were conducted for three computational kernels (Floyd–Steinberg, Needleman–Wunsch, and heat diffusion) on a larger heterogeneous system with different CPU characteristics and interconnection network. In [35], the total parallel execution time in the general case, where the total number of chunks exceeds the number of available workers, is based on a detailed analysis of the special case where the total number of chunks is equal to the number of available workers. This approach indicates that the parallel execution times of all pipelines are approximately the same. This assumption holds when CSS and DCSS are the algorithms of choice, as the chunk sizes remain constant throughout the parallel execution. However, when the chunk sizes change, as is the case with decreasing chunk size algorithms (TSS, FSS, GSS, and their distributed versions), the model in [35] is no longer suitable.

The model presented in this work is designed to account for variable chunk sizes by directly calculating and using the corresponding (variable) chunk size for each scheduling step to estimate the total parallel execution time. Moreover, in the present work, the worker's heterogeneity is addressed via the use of the *virtualization* concept. Herein, virtualization is defined as the transformation of an array of *physical* heterogeneous processors into an array of *virtual* homogeneous processors using the following simple principle: the *slowest* physical worker is considered as a single virtual worker, and each worker that is k times faster than the slowest worker corresponds to k virtual workers. The number of the original physical heterogeneous workers is m and the number of the resulting virtual homogeneous workers is A. Using this concept, the case of m heterogeneous workers can be addressed as a case of A homogeneous workers. As stated later in the article, the difference between the physical heterogeneous workers and the virtual homogeneous workers is that communications occur *only* between physical workers, a fact taken into account by the model proposed herein.

DOI: 10.1002/cpe

In addition to the work in [34] and [35], new formulas are given in this work for estimating the total number of scheduling steps for the case where there is a *threshold* on the minimum chunk size assigned to a processor (cf. Table II). The total number of scheduling steps influences the total parallel execution time, as a large number of steps can incur a large scheduling and/or communication overhead. Accurately calculating the total number of chunks plays a *significant* role in efficiently estimating the performance of the self-scheduling schemes, and having a threshold on the minimum chunk size assigned to a processor yields a smaller number of scheduling steps. In this approach, the chunks computed by a self-scheduling algorithm (taking into account the threshold on the minimum chunk size) are also *weighted* according to each worker's computational power, as described in detail in Section 2. Finally, this work includes a review and a classification of existing approaches to scheduling loops with and without data dependencies on heterogeneous systems (cf. Table I).

Organization. Section 2 contains an overview of the class of self-scheduling algorithms. The pipelining paradigm for scheduling loops with data dependencies is described in Section 3. The computation and communication models used to estimate the parallel execution time on heterogeneous systems are presented in Section 4.1, followed by the proposed theoretical model to determine the optimal synchronization granularity for heterogeneous systems in Section 4.2. The experimental analysis and validation of the proposed model is discussed in Section 5. The conclusions and directions for future work are summarized in Section 6.

2. OVERVIEW OF THE SELF-SCHEDULING ALGORITHMS FOR LOOPS WITHOUT DATA DEPENDENCIES

Self-scheduling algorithms partition the problem domain, hereinafter referred to as iteration space, modeled as a Cartesian coordinates space, into *chunks* of consecutive rows along one specific dimension. This dimension is called the *scheduling dimension*, and its length is denoted by U_c (see Figure 2). The processors of the heterogeneous system are called *workers*. Every worker has a run queue, which contains all the jobs that run on that particular worker, including the application of interest. For dedicated runs, the run queue contains only one job, that is, the application of interest. The computational power of a worker is estimated using the normalized execution time of a small size part (e.g., 5–10%) of the application [37]. Every job running on a worker is assumed to take an equal share of its computing resources (i.e., space-sharing). The *available* computational power of a worker is determined using the number of jobs in the run queue by dividing its actual computational power with the number of jobs in its run queue. As shown in [38], the number of jobs in the run queue is the best choice for workload descriptor. The idle workers request work from a *master* processor. Timing the work assignment and the amount of assigned work play a significant role in the performance of a self-scheduling algorithm. This is crucial in loops with data dependencies, where it is imperative to preserve the execution order that satisfies the data dependencies. Following is an overview of the self-scheduling schemes CSS, TSS, FSS, GSS, and DTSS as they were initially proposed to handle loops *without* data dependencies (see the references in the upper part of Table I). The simple versions of these schemes are the versions suitable for homogeneous systems with single-user jobs (dedicated) execution mode. The *distributed* versions are suitable for heterogeneous systems (see the references in the lower part of Table I). A master–worker model with m workers is employed, where at the i-th *scheduling step* the master allocates a chunk of size V_i to a worker. The following notation is used in this section:

- m: number of physical heterogeneous workers ($P_1, \ldots, P_m$)
- A: number of virtual homogeneous workers ($P_1, \ldots, P_A$)
- N: number of scheduling steps
- U_c: the length of the scheduling dimension of the iteration space
- F: first chunk size of the problem
- L: last chunk size of the problem
- C: threshold for the minimum chunk size of a processor
- V_i: size of chunk i

DOI: 10.1002/cpe

- V_j: size of a chunk in batch j (used by FSS)
- R_i: number of remaining iterations (used by GSS)
- R_j: number of remaining iterations in batch j (used by FSS)

- *Chunk self-scheduling* (CSS) [1, 2] assigns constant size chunks to each worker: $1 \leqslant V_i = constant \leqslant \frac{U_c}{m}$, where m is the number of workers. The chunk size is chosen by the user. If $V_i = 1$ then CSS is the so-called *pure* self-scheduling. A large chunk size reduces scheduling overhead but also increases the chance of load imbalance because it is difficult to predict the 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 variable chunk size algorithms, and their difference lies in the choice of the first chunk and of the amount of variation between chunk sizes.
- In *guided self-scheduling* (GSS) [3], each worker is assigned a chunk given by the number of remaining iterations divided by the number of workers m: $V_i = R_i/m$, where R_i is the number of remaining iterations. According to GSS, R_0 is the total number of iterations, that is, U_c, and $V_i = \lceil R_i/m \rceil$, where $R_{i+1} = R_i - V_i$. Thus, $V_i = \lceil \frac{U_c}{m}(1 - \frac{1}{m})^i \rceil$. Using the chunk size formula, it can be easily proved that the number of scheduling steps is $N \simeq \frac{1}{\ln \lceil \frac{m}{m-1} \rceil} \ln \lceil \frac{U_c}{m} \rceil$. If there is a threshold C for the minimum allowed chunk size, it can be easily proved that $N \simeq \frac{1}{\ln \lceil \frac{m}{m-1} \rceil}(\ln \lceil \frac{U_c}{m} \rceil - \ln C)$. The initial chunk sizes are large in order to reduce the communication/scheduling overhead in the beginning. In the last steps, very small chunks are assigned to improve the load balancing at the expense of increased communication/scheduling overhead.
- The *factoring self-scheduling* (FSS) [4] scheme schedules iterations in batches of m equal chunks. In each batch j, a worker is assigned a chunk size given by a subset of the remaining iterations (usually half) divided by the number of workers. The chunk size for the batch j is $V_j = \lceil \frac{R_j}{\alpha m} \rceil$ (thus $V_i = \lceil \frac{U_c}{m}(\frac{1}{\alpha})^{j+1} \rceil$) and $R_{j+1} = R_j - (mV_j)$, where the parameter α is computed (by a probability distribution) or is sub-optimally chosen $\alpha = 2$. The total number of steps is approximately equal to $N \simeq 1.44m \ln \lceil \frac{U_c}{m} \rceil$ [39]. If there is a threshold C for the minimum allowed chunk size, it can be easily proved that $N \simeq m \left(1.44 \ln \lceil \frac{U_c}{m} \rceil - \ln\ C\right)$. The weakness of this scheme is the difficulty to determine the optimal parameters. However, tests show improvement on previous adaptive schemes (possibly) caused by fewer adaptations of the chunk size.
- The *trapezoid self-scheduling* (TSS) [5] scheme linearly decreases the chunk size V_i. In TSS, the first and last chunk size pair (F, L) may be set by the programmer. In a conservative selection, the (F, L) pair is determined as $F = \frac{U_c}{2m}$ and $L = 1$, where m is the number of workers. This ensures that the workload of the first chunk is less than $1/m$ of the total load in most loop distributions and reduces the chance of imbalance caused by a large first chunk. One may improve this by choosing $L > 1$. The proposed number of steps needed for the scheduling process is $N = \frac{2U_c}{(F+L)}$. If there is a threshold C for the minimum allowed chunk size, then the last chunk is set to C. Consequently, the decrement between consecutive chunks is $D = (F - L)/(N - 1)$, and the chunk sizes are $V_1 = F, V_2 = F - D, V_3 = F - 2D, \ldots$. TSS improves on GSS by decreasing the chunk size linearly.
- The *distributed trapezoid self-scheduling* (DTSS) [40] (and references therein) improves on TSS by selecting the chunk sizes according to the available computational power of the workers. The programmer may determine the pair (F, L) according to TSS, and the following formula may be used in the conservative selection approach: $F = \frac{U_c}{2A}$, where $A = \sum_{i=1}^{m} A_i$, A_i being the available computational power of each worker, and $L = 1$. The total number of steps is $N = \frac{2U_c}{F+L}$ and the chunk decrement is $D = (F - L)/(N - 1)$. If there is a threshold C for the minimum allowed chunk size, then $L = C$. The size of a chunk in DTSS is $V_i = A_k(F - D(S_{k-1} + \frac{A_k - 1}{2}))$, where $S_{k-1} = A_1 + \ldots + A_{k-1}$, for $k \geqslant 2$, and $S_1 = A_1$. In a dedicated homogeneous system, the chunks assigned by DTSS are equal to those assigned by TSS. The important difference between DTSS and TSS is that in DTSS the next chunk is

DOI: 10.1002/cpe

allocated according to the worker's available computational power. Hence, faster workers get more iterations than slower ones. In contrast, TSS simply treats all workers in the same way.

3. PIPELINE SCHEDULING OF LOOPS WITH DATA DEPENDENCIES

The following notation is used in this section:

- m: number of stages of a pipeline (equal to the number of physical workers)
- M: number of instances of a pipeline (equal to the number of synchronization points)
- subchunk: a subset of a chunk between two consecutive synchronization points
- $\mathbb{J} \subseteq \mathbb{Z}^n$: the n-dimensional iteration space
- U_s: the length of the synchronization dimension of the iteration space
- $x = (x_1, \ldots, x_n) \in \mathbb{J}$: a typical iteration point
- DS: the set of data dependence vectors
- $\mathbf{d}_j \in DS$: data dependence vector j, $1 \leqslant j \leqslant r$, $r \geqslant 2$
- p: number of pipelines (equal to the number of chunk assignment rounds)

The iteration space of a nested loop is typically modeled as a finite subset $\mathbb{J}$ of the n-dimensional space $\mathbb{Z}^n$, where $\mathbb{Z}$ is the set of integers and n is the depth of the loop nest. Each point of this n-dimensional space corresponds to a single iteration of the loop body. Without loss of generality, it is assumed that the loops have index points $(x_1, \ldots, x_n)$, where $1 \leqslant x_i \leqslant U_i$, $1 \leqslant i \leqslant n$. The data dependencies are modeled by n-dimensional vectors of $\mathbb{Z}^n$, called dependence distance vectors [41], and their set is denoted by $DS = \{\mathbf{d}_1, \ldots, \mathbf{d}_r\}$, where r is the number of dependence distance vectors. The data dependence distance vectors are assumed to be constant, that is, the data dependencies are uniform. These constant dependence distance vectors give rise to cross-node data dependencies when the computation of a data element in one worker in a certain node requires some data element(s) from another worker in a different node. The existence of such data dependencies can slow down and (in extreme cases) even serialize the parallel execution. These dependencies are, nonetheless, common in scientific computations such as, for example, signal processing, bioinformatics, partial differential equations, and computer graphics, to name a few.

Pipelining the computations of such applications can provide an efficient solution to addressing the cross-node data dependencies in loops. In fine-grain pipelining, the pipeline parallelism can be increased at the cost of increased communication overhead. Coarse-grain pipelining attempts to balance the parallelism with the communication overhead. In this work, the focus is on the coarse-grain pipelining approach. Applying self-scheduling algorithms to nested loops with data dependencies leads to a pipelined execution because of the data dependencies [34, 42]. Specifically, the iteration space of an application containing a loop nest of depth n is modeled as an n-dimensional Cartesian coordinates space, where one axis is called the *scheduling* dimension, its size being denoted by U_c, whereas another axis (usually the one with the largest length) is called the *synchronization* dimension, and its size is denoted by U_s, as illustrated in Figure 2. The scheduling dimension of the application is partitioned into chunks of iterations according to the rules of a self-scheduling algorithm, and synchronization points are inserted along the synchronization dimension. Thus, it is assumed that the application has at least two dimensions, that is, $n \geqslant 2$, such that the synchronization and scheduling dimensions can be distinguished. If the application has more than two loops, that is, $n > 2$, one can consider the two outer loops as the scheduling and synchronization dimensions, respectively. In this work, the depth of a loop nest is assumed to be $n \geqslant 2$.

The projections of chunks on the scheduling dimension are denoted by V_i, $i = 1, \ldots, N$, where N is the number of chunks given by the self-scheduling algorithm of choice. For a loop nest of depth n, the chunks resulting from the partitioning along the scheduling dimension are n-dimensional parallelepipeds. Each chunk is assigned to a processor for execution upon a request for work made by each worker to the master processor. The pipelined execution follows the extended master–worker model, in which synchronization between workers is achieved by inserting equally spaced *synchronization points* along the synchronization dimension. Because of the synchronization points, each n-dimensional chunk is partitioned along the synchronization dimension into *subchunks* of

DOI: 10.1002/cpe

equal size. The length of the subchunk along the synchronization dimension is the same for every chunk and determines the *granularity* of synchronization between workers, denoted by h.

For a fixed number of processors, the number of chunks, N, is likely to be larger than the number of physical processors, m. This means that each processor will execute more than one chunk in a cyclic fashion in successive assignment rounds. Each chunk assignment round corresponds to a pipeline with as many stages as the number of workers m (see Figure 1). The total number of assignment rounds yields the number of pipelines, denoted by p. In a pipeline organization, each worker synchronizes with its neighbor, and synchronization is performed at predetermined points, that is, the synchronization points. Assuming that M synchronization points are inserted along the synchronization dimension (see Figure 2), the total number of steps required for the completion of a single pipeline is $m+(M-1)$, which also corresponds to the *latency* of a pipeline with m stages. In every pipeline, data produced at the end of one stage are fed to the next stage. The synchronization granularity determines the amount of computation to be performed before sending a message and plays a crucial role in the total parallel execution time. Smaller granularity implies more communication, whereas larger granularity may limit the inherent parallelism. It is therefore important to establish the optimal synchronization granularity that minimizes the parallel execution time.

4. DETERMINING THE OPTIMAL SYNCHRONIZATION GRANULARITY

In this section, the process of virtualization is described first, that is, transforming the set of physical heterogeneous processors into a set of virtual homogeneous workers to facilitate the derivation of a general theoretical model for determining the optimal synchronization granularity. This is followed by a description of the models used to estimate the computation and communication costs. To simplify the presentation of the proposed model, a two-dimensional loop, that is, $n = 2$, is presented. The proposed theoretical model for estimating the parallel execution time and determining the optimal synchronization granularity is however not limited to two-dimensional loops. The following notation is used in this section:

- A_k: available computational power of physical worker k
- t_p: computation cost for a subchunk
- c_p: computation time/iteration using the slowest physical worker
- t_c: communication cost for a subchunk
- c_d: start-up cost for transferring a message
- c_c: network throughput for transferring a message
- t_{ms}: transmission time needed for the work request to reach the master and for the master's reply to reach the worker
- c_{sch}: scheduling overhead (master side)
- h_{opt}: optimal synchronization granularity
- T_{wa}: time for the master to assign work to workers
- T_{tr}: time for transferring the necessary data from one pipeline to the next (case when $A < N$)
- T_{comp}: time spent in computation

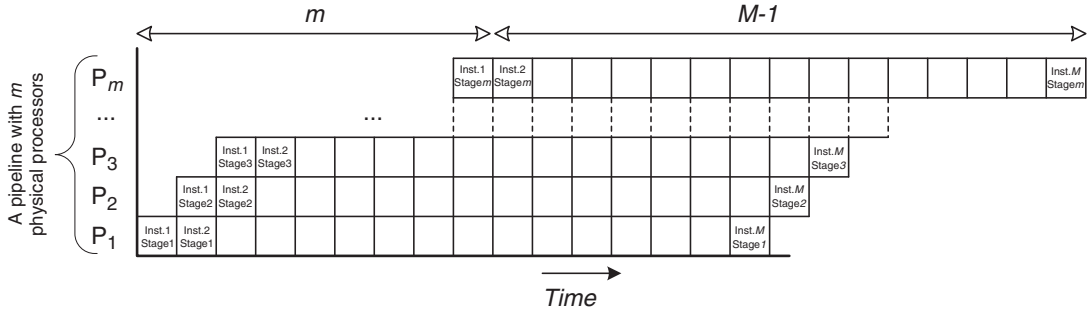


Figure 1. Space-time mapping of a coarse-grain pipeline with m stages, M instances, and $m+(M-1)$ steps.

DOI: 10.1002/cpe

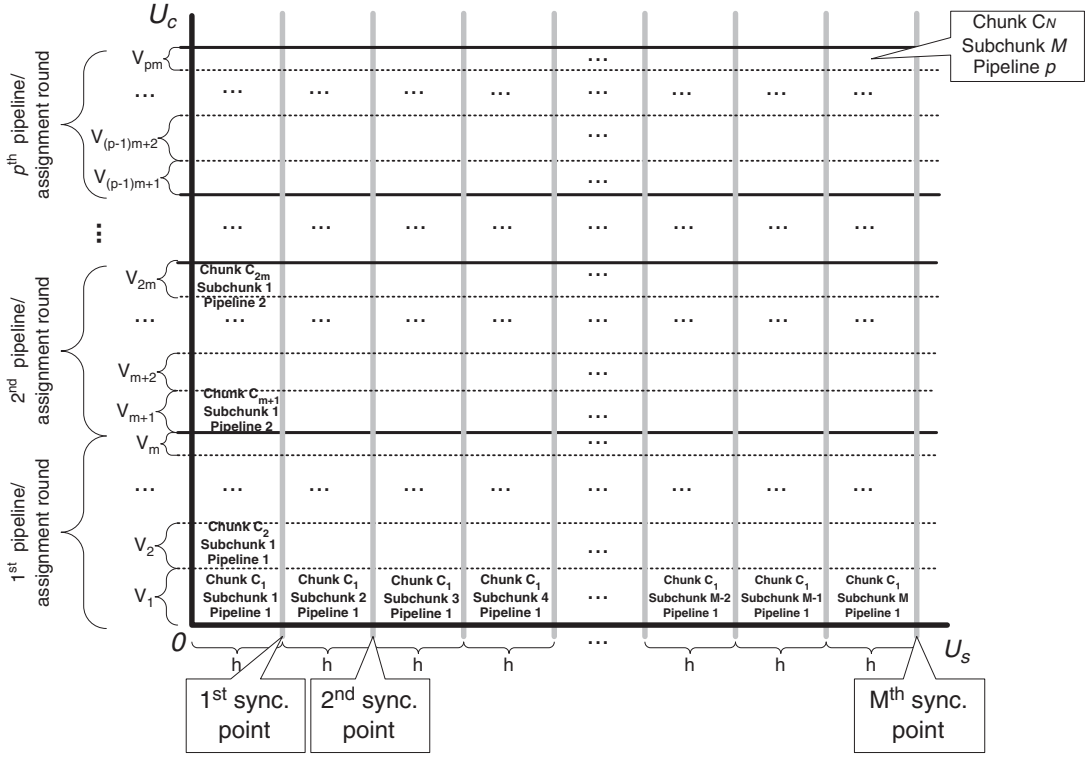


Figure 2. The iteration space of a two-dimensional loop with data dependencies is partitioned into chunks along the scheduling dimension, and synchronization points are inserted along the synchronization dimension.

- T_{comm}: time spent in communication
- T_{par}: total parallel execution time

4.1. The computation and communication cost models

Virtualization. Distributed versions of the self-scheduling algorithms (CSS, TSS, FSS, and GSS) have been studied in the past ([8, 18, 21], and references therein). The approach adopted by Distributed TSS (DTSS) of using the available computational power of the workers can be applied to all other self-scheduling algorithms so that they can be efficiently used on non-dedicated heterogeneous systems (see [21] and [18]). In this work, the distributed approach is accomplished via *virtualization* of the physical processing elements. The m physical heterogeneous processors are transformed into $A > m$ virtual homogeneous processors $(P_1, \ldots, P_A)$, each with computational power 1. A heterogeneous worker k is modeled as a set of A_k virtual homogeneous workers, where A_k represents its available computational power. The total number of virtual processors is equal to the total available power of the heterogeneous system, that is, $A = \sum_{k=1}^{m} A_k$. Following the virtualization of the physical workers, each pipeline consists of A stages (instead of m stages before virtualization) and M instances. In contrast to the approach in [21] and [18], in this work the virtualization method is directly used in the implementation of the algorithms. In the remaining of the paper, the distributed algorithms applied to arrays of A virtual homogeneous processors are referred to as DCSS, DTSS, DFSS, and DGSS.

A model is needed to determine the synchronization granularity that minimizes the parallel execution time of loops with data dependencies in a heterogeneous system. The following computation and communication cost models are introduced towards this goal.

DOI: 10.1002/cpe

Computation cost model. The computation cost is defined as a linear function of the computation cost per iteration multiplied with the number of iterations. The computational power of each virtual processor is assumed to be equal to the computational speed of the *slowest* physical processor. Therefore, the computation cost of a subchunk of size hV_i (for a two-dimensional loop) is:

$$t_p = hV_i c_p, \tag{1}$$

where h is the synchronization granularity, V_i is the projection of chunk i along the scheduling dimension, and c_p is the computation time per iteration of the *slowest* worker (see Figures 4 and 5).

Communication cost model. The analysis of the communication is based on the two-sided MPI communication model [36]. The cost of *sending* a message is assumed to be equal to the cost of *receiving* a message. This is a simple yet realistic assumption, based on the fact that, in most cases, the send and the receive operations are executed in pairs between the communicating workers. The message size is an important factor in the performance of an MPI application. In many situations, increasing the message size leads to better performance caused by increased bandwidth. To quantify this behavior, a series of experiments have been conducted using `mpptest` of the `perftest` suite [36]. The results of these experiments are shown in Figure 3.

The cost t_c of communicating a message of size h between two workers is a piecewise linear function of h. The allocation of bandwidth increases until the message size reaches a certain threshold value. This threshold value is determined by a system tunable parameter usually called the *eager limit*, which signifies the transition from the eager to the rendezvous MPI exchange protocol [36]. This transition induces a performance penalty that is clearly illustrated in Figure 3, indicating that for messages of 4000 bytes, the bandwidth is approximately 1.5×10^6 bytes/s. Therefore, the threshold value for the test system considered is reached at 4000 bytes, and t_c is determined as follows.

$$t_c = \begin{cases} c_d + hc_c & \text{for} \quad \text{msg.size} < 4000 \text{ bytes} \\ c'_d + hc'_c & \text{for} \quad \text{msg.size} \geqslant 4000 \text{ bytes} \end{cases} \tag{2}$$

where c_d and c'_d capture the start-up cost (the time to send a zero-length message including the hardware/software overhead of sending the message), whereas c_c and c'_c correspond to the network throughput defined as $\frac{1}{sustained\ bandwidth}$, where $sustained\ bandwidth$ is the ratio of the amount of data sent over the actual time measured at the application level.

The following observations are made regarding the communication: (i) In a heterogeneous system, processors may communicate at different speeds, even if the interconnection network is homogeneous. When several processors send a message at the same time, network congestion occurs. Both of these aspects are taken into account in the proposed model via the communication parameters determined by the `mpptest` experiments; and (ii) The communication occurs only between the m physical processors.

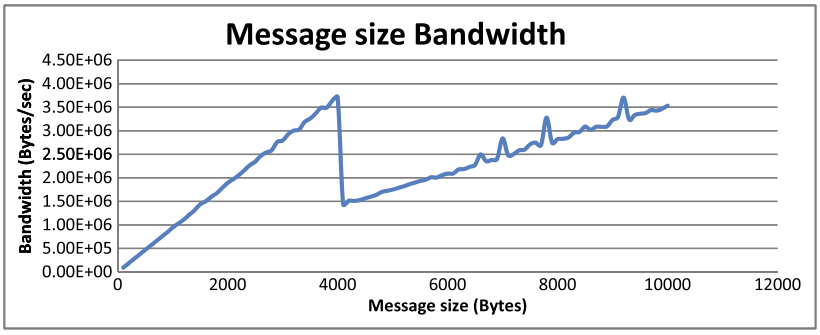


Figure 3. Bandwidth of MPI communications as a function of message size, illustrating the performance penalty associated with the transition from the eager to the rendezvous protocol.

DOI: 10.1002/cpe

4.2. The theoretical parallel execution model

When self-scheduling algorithms are utilized with an array of A virtual processors, they provide better load balancing and, hence, better performance than if they were utilized with an array of m physical processors. This is because more work is assigned to faster processors. The cases investigated in this work are when $A = N$ and $A < N$, meaning the number of virtual processors is equal to or larger than the number of chunks. The first case ($A = N$) is a novel contribution to the work in [34]. The special case when $A > N$ can be reduced to the case when A = N by dropping the extra processors.

4.2.1. Case $A = N$. In this case, the number of available virtual processors is equal to the number of application chunks, and each processor is assigned exactly *one* chunk. This is the case of coarse-grain parallelism, which is different from the case where there are as many virtual processors available as tasks (fine-grain parallelism). For simplicity, let us assume that DCSS is used to partition the scheduling dimension into N equal chunks of size $V_i = \frac{U_c}{A}$, and that M synchronization points are inserted in each chunk along the synchronization dimension at equal intervals h. Figure 4 illustrates such an iteration space partitioned into N equal chunks (horizontal segments), which are assigned to the A virtual processors.

The total parallel execution time in this case is the time needed to compute the $A + (M - 1)$ number of steps of the pipeline. Except from the first and last step, all other steps involve a *receive*, a *compute*, and a *send* operation. The first virtual processor P_1 requires only a *compute* and a *send* operation and the last virtual processor P_A only a *receive* and a *compute* operation. Recall that h is the synchronization granularity, U_c and U_s are the lengths of the scheduling and synchronization dimensions, respectively, and $M = \frac{U_s}{h}$ is the number of synchronization points. Also recall that c_p is the computation time per iteration of the slowest physical worker, N is the number of scheduling steps, and $V_i = \frac{U_c}{A}$ is the projection of the i-th chunk size on the scheduling dimension, as given by the DCSS self-scheduling algorithm. Then the total computation time is:

$$T_{comp}^{A=N} = \frac{m}{A} hU_c c_p + (hV_i c_p)(M - 1) \tag{3}$$

The first term of Equation (3) is the computation time of all the last subchunks of the pipeline. This is illustrated by the vertically shaded rectangle in Figure 4. Because c_p is used to denote the computation time per iteration of the slowest worker, the first term must be scaled by $\frac{m}{A}$. The second term of Equation (3) represents the computation time of the $(M-1)$ instances of the pipeline. This is shown in Figure 4 by the horizontally shaded rectangles. It is obvious that the total computation time is the sum of all vertically and horizontally shaded rectangles, corresponding to the total number of steps of the pipeline (see Figures 1 and 4).

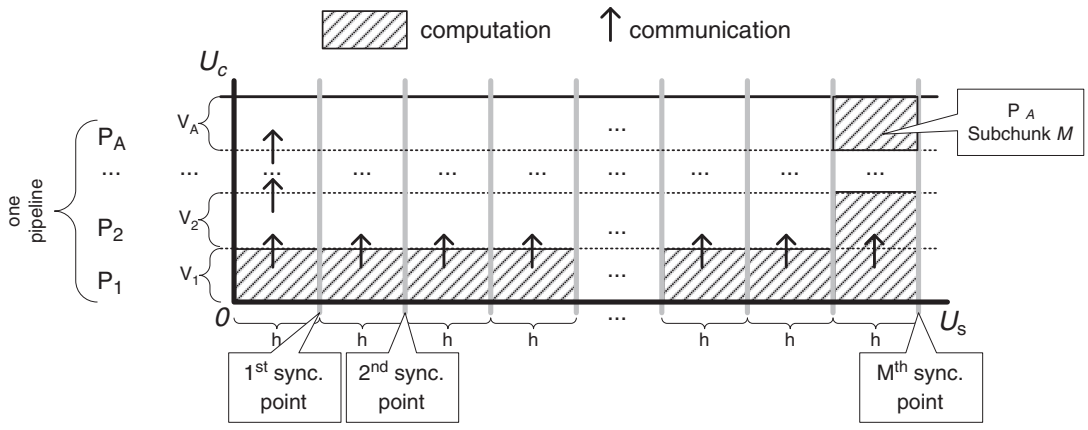


Figure 4. Case $A = N$. Space-time mapping of a two-dimensional loop and communication and computation pattern for a *single* pipeline having A stages, M instances, and $A + (M - 1)$ number of steps.

DOI: 10.1002/cpe

Data exchanges occur only between the m physical processors. The time to send or receive a message , denoted by t_c, is taken twice for a complete communication operation as given by Equation (2). Thus, the time to perform all receive and send operations in this case is:

$$T_{comm}^{A=N} = (m-2)(2t_c) + (M-1)(2t_c) \tag{4}$$

For a given problem size, the communication time is a linearly increasing function of m. The first term of Equation (4) represents the communication time between the m physical processors associated with the computation of the last subchunks (the first term of Equation (3)). The second term of Equation (4) is the communication time associated with the computation of the $M-1$ instances of the pipeline (the second term of Equation (3)).

Because of the use of the master–worker model, the work assignment time is $T_{wa} = 2t_{ms} + c_{sch}$, where $t_{ms} = c_d + Vc_c$ is the transmission time needed for the work request to reach the master and for the master's reply to reach the worker, and c_{sch} is the time needed by the master to compute the next executable chunk size, called *scheduling overhead*. Therefore, the total parallel execution time in this case is:

$$T_{par}^{A=N} = T_{comp}^{A=N} + T_{comm}^{A=N} + T_{wa} \tag{5}$$

T_{wa} is taken only once because the work assignment time for every other chunk is overlapped with the worker's computation or communication operations, except for the first chunk of the problem.

The minimum parallel execution time $T_{par}^{A=N}$ is determined as a function of h. For the values of N and V_i given by the DCSS self-scheduling algorithm (see Table II; for DCSS V_i is a constant $1 \leqslant V_i \leqslant \frac{U_c}{A}$) the minimum parallel execution time can be found by differentiating $T_{par}^{A=N}$ and verifying that the second-order derivative is positive. The values of N and V_i are computed as described in Section 2 by applying DCSS to an array of A processors.

Proposition 1
The parallel execution time $T_{par}^{A=N}$ as a function of h achieves its minimum value at:

$$h_{opt}^{A=N} = \sqrt{\frac{2U_s A c_d}{c_p(V_i A - U_c m) - 2c_c A(m-2)}} \tag{6}$$

Proof
The interest is to find the synchronization interval h that minimizes $T_{par}^{A=N}$. This, $T_{par}^{A=N}$ is differentiated with respect to h, which yields:

$$\frac{d}{dh} T_{par}^{A=N} = 2c_c(m-3) + c_p\left(U_c \frac{m}{A} - V_i\right) - \frac{2U_s c_d}{h^2} \tag{7}$$

and the solution of $\frac{d}{dh} T_{par}^{A=N} = 0$ is given by Equation (6). Mathcad [43] was used to obtain the second derivative of $T_{par}^{A=N}$ at $h_{opt}^{A=N}$. This verifies that $\frac{d^2}{d(h_{opt}^{A=N})^2} T_{par}^{A=N}$ is indeed positive, and implies that the solution of Equation (6) is a local minimum for $T_{par}^{A=N}$. □

Table II. Scheduling steps and chunk sizes with threshold C (= L).

	N	V_i
DCSS	$\frac{U_c}{V_i}$	V_i
DGSS	$\frac{1}{\ln\lceil \frac{A}{A-1} \rceil}(\ln\lceil \frac{U_c}{A} \rceil - \ln C)$	$\frac{U_c}{A}(1-\frac{1}{A})^i$
DTSS	$\frac{2U_c}{F+L}$	$F-(i-1)D$
DFSS	$A(1.44\ln\lceil \frac{U_c}{A} \rceil - \ln C)$	$\frac{U_c}{A}(\frac{1}{\alpha})^{i+1}$

DOI: 10.1002/cpe

Remark
Note that for each of the distributed self-scheduling algorithms, the value of V_i in Equation (6) is calculated using the formulas from Table II, which take into account the threshold C for the minimum chunk size of a processor.

4.2.2. Case $A < N$. This is the general case when there are more application chunks than the total available computational power, and each virtual processor is assigned *more than one* chunk in successive assignment rounds. Each assignment round corresponds to one pipeline execution, and p pipelines are considered. In practice, it may happen that not all virtual processors are used in the last assignment round. The output of one pipeline is the input of the next pipeline. Figure 5 represents a collection of p replications of Figure 1. The theoretical parallel execution time in this case is the sum of the completion time of all p pipelines plus the time required to transfer data between the p pipelines. The total parallel execution time corresponds to the completion time of the last subchunk, illustrated in Figure 5 as "P_A, subchunk M, pipeline p". Each pipeline can be completed in $A + (M - 1)$ steps (see Figures 1 and 4). Each step, except from the first and last, consists of one *receive*, one *compute*, and one *send* operation. As in the previous case (when $A = N$), the steps performed in the first virtual processor of every pipeline consist only of one compute and one send step, and the steps performed in the last virtual processor of every pipeline consist only of one receive and one compute step.

The total computation time for all pipelines is p times the computation time of a single pipeline. Assuming that V_{jA+1} (> 0) is the size of the last chunk of each pipeline, as given by one of the self-scheduling algorithms, the total computation time for all pipelines is derived as follows:

$$T_{comp}^{A<N} = \frac{m}{A} c_p h U_c + c_p h (M-1) \sum_{j=0}^{p-1} V_{jA+1}. \tag{8}$$

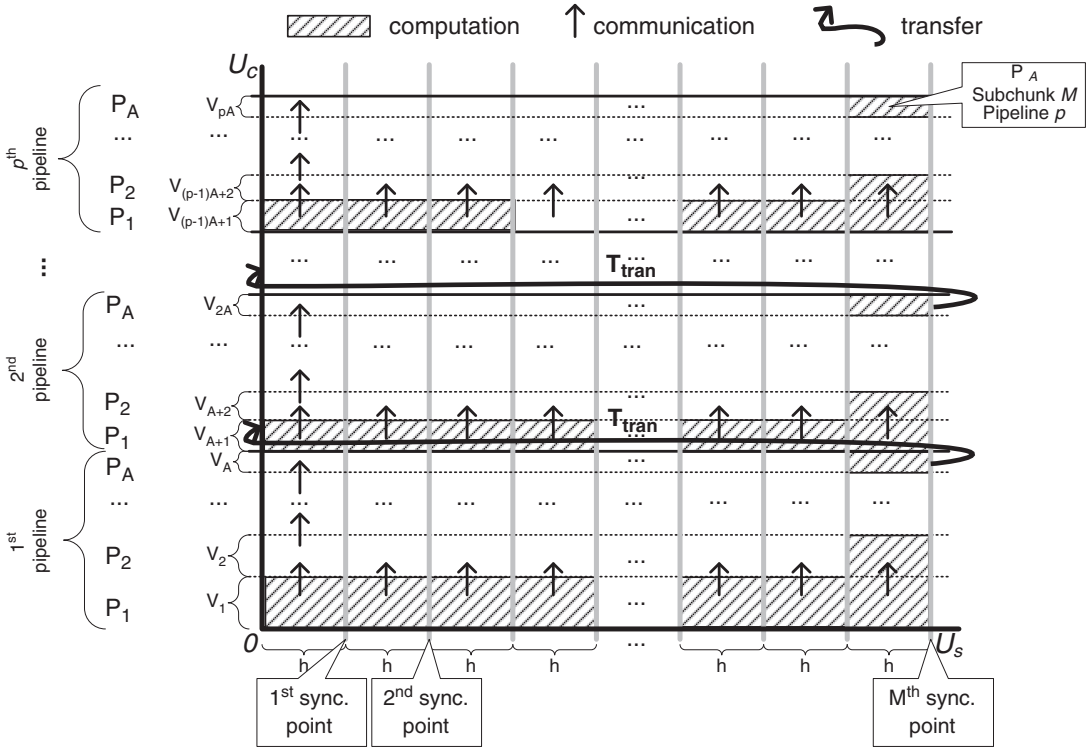


Figure 5. Case $A < N$. Space-time mapping of a two-dimensional loop and communication and computation pattern for p pipelines, each with A stages, M instances, and $A + (M - 1)$ number of steps.

DOI: 10.1002/cpe

The first term of Equation (8) is the computation time of all the last subchunks. This is depicted by the vertically shaded rectangles in Figure 5. Given that c_p is the computation time per iteration of the *slowest* workers, the first term must be scaled by $\frac{m}{A}$. The second term of Equation (8) represents the computation time of the $M-1$ instances of every pipeline, corresponding to the $p(M-1)$ compute steps. This is also illustrated in Figure 5 by the horizontally shaded rectangles. It is obvious that the total computation time is the sum of all vertically and horizontally shaded rectangles (see Figures 1 and 5).

The total number of send and receive steps for all pipelines is: $p((m-2)+(M-1))+(p-1)$. By letting T_{tr} be the time taken to transfer the necessary data from one pipeline to the next, the time required to perform all the receive and send operations, that is, the communication time for all pipelines and for transferring the data between two adjacent pipelines is:

$$T_{comm}^{A<N} = p(m-2)(2t_c) + p(M-1)(2t_c) + (p-1)T_{tr} \tag{9}$$

As expected, the communication time is an increasing function of m and p. The first term of Equation (9) represents the communication time associated with all the last subchunks (analogous to the first term of Equation (8)) during the $p(m-2)$ communication steps. Similarly, the second term of Equation (9) is the communication time associated with the computation of all the $M-1$ instances of each pipeline (analogous to the second term of Equation (8)). T_{tr} is the time required to transfer the necessary data from one pipeline to the next given by $2(c_d + U_s c_c)$ and taken $p-1$ times (see Figure 5).

As in the previous case, the work assignment time is $T_{wa} = 2t_{ms} + c_{sch}$, where $t_{ms} = c_d + V_1 c_c$. Hence, the total parallel execution time when $A < N$ is:

$$T_{par}^{A<N} = T_{comp}^{A<N} + T_{comm}^{A<N} + T_{wa} \tag{10}$$

Proposition 2 gives the value of h that minimizes the parallel time $T_{par}^{A<N}$. Table II summarizes the number of scheduling steps and the chunk sizes for the various self-scheduling schemes. In the case of DCSS, V_i is a constant $1 \leqslant V_i \leqslant \frac{U_c}{A}$. For the various N and V_i given by the different self-scheduling algorithms (see Table II), the minimum parallel execution time can be found by differentiating $T_{par}^{A<N}$ and verifying that the second-order derivative is positive. The values of N and V_i are computed as described in Section 2 by applying the self-scheduling algorithms to an array of A processors.

Proposition 2
The parallel execution time $T_{par}^{A<N}$ considered as a function of h assumes its minimum at:

$$h_{opt}^{A<N} = \sqrt{\frac{2\,(U_s\ p\ A\ c_d)}{(2m-6)Ac_c p + U_c c_p m - c_p A \sum_{j=0}^{p} V_{jA+1}}} \tag{11}$$

Proof
To find the synchronization interval h that minimizes the total parallel execution time, $T_{par}^{A<N}$ is differentiated with respect to h. This yields:

$$\frac{d}{dh}T_{par}^{A<N} = U_c c_p \frac{m}{A} - c_p \sum_{j=0}^{p-1} V_{jA+1} + (2m-6)c_c p - 2\frac{U_s c_d}{h^2} \tag{12}$$

and the solution of $\frac{d}{dh}T_{par}^{A<N} = 0$ is given by Equation (11). Mathcad [43] was used in order to obtain the second derivative of $T_{par}^{A<N}$ at $h_{opt}^{A<N}$ and verify that $\frac{d^2}{d(h_{opt}^{A<N})^2}T_{par}^{A<N}$ is positive. This means that $h_{opt}^{A<N}$, as given by Equation (11), is indeed a local minimum for $T_{par}^{A<N}$. □

Remark
Note that for each of the distributed self-scheduling algorithms, the value of V_{jA+1} in Equation (11) is calculated using the formulas from Table II, which take into account the threshold C for the minimum chunk size of a processor.

DOI: 10.1002/cpe

5. EXPERIMENTAL VALIDATION

Experimental system setup. With the advent of the multi-core technology computer, clusters nowadays are made of multi-processor nodes (SMP clusters). The experiments were performed on a heterogeneous system that consisted of nodes taken from two different Linux SMP clusters (kernel 2.6.24.2). The first cluster has 16 nodes (called clones). Each "clone" node has two quad-core Intel® Xeon® chips based on the Intel Core 2 micro-architecture E5405 (12-M Cache, 2.00-GHz, 1333-MHz FSB), yielding a total of 128 processing cores. The second cluster also has 16 nodes (called twins). Each "twin" node, however, has a dual-core Intel® Core^{TM}2 Duo Processor E8200 (6-M Cache, 2.66-GHz, 1333-MHz FSB).

A series of experiments was conducted using 16 cores (from two nodes) of the first cluster and 16 cores (from eight nodes) of the second cluster, interconnected by a Gigabit ethernet network (see Section 5.1). To further evaluate the effectiveness of the proposed theoretical model, a second experiment was conducted on the first cluster (clones) using a 10-Gbit Myrinet interconnection network (see Section 5.2). The results presented next are the average of 10 runs for each of the two experiments. In both experiments, a threshold of C=20 iterations on the minimum chunk size for all self-scheduling schemes was used. The applications test cases and the loop-scheduling algorithms have been implemented in C/C++, and the communication was implemented using the OpenMPI 1.4.2 distribution of MPI. The codes were compiled using the GNU C/C++ compiler (version 4.4.5 20100909) with the -O3 option.

For both experiments, three well-known computational kernels were used as application test cases:

1. **Floyd–Steinberg** (F-S) is an image-processing algorithm that is used for the error-diffusion dithering of a $width$ by $height$ gray-scale image. The boundary conditions are ignored. The pseudocode, given next, is a two-dimensional nested loop with four data dependencies [44].

Listing 1: The Floyd-Steinberg computational kernel.

```
for (i=0; i<width; i++){ /* synchronization dimension */
   for (j=0; j<height; j++){ /* scheduling 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);
   }
}
```

2. **Needleman–Wunsch** (DNA) is an algorithm used for the global sequence alignment of two DNA sequences [45]. The two sequences to be compared are placed along the left (X) and top (Y) margins of the scoring matrix. The similarity matrix is initialized with decreasing values (0, -1, -2, -3, . . .) along the first row and the first column to penalize for consecutive gaps. The elements of the similarity matrix $SM[n1, n2]$ are calculated by the following recurrence equation:

$$SM[i, j] = \begin{cases} SM[i, j-1] + gp \\ SM[i-1, j-1] + ss \\ SM[i-1, j] + gp \end{cases}$$

where gp is the gap penalty and ss is the substitution score. In this work, gp is -2 and ss is 1 if the elements match, and 0 otherwise.

DOI: 10.1002/cpe

3. **Heat diffusion** (HEAT) is one of the most widely used scientific computational kernels in the literature. Its loop body is similar to the majority of the numerical methods used for solving partial differential equations. It computes the temperature in each point of its two-dimensional domain based on two values of the current time step $A[i-1][j]$, $A[i][j-1]$ and two values from the previous time step $B[i+1][j]$, $B[i][j+1]$. The pseudocode is given next:

Listing 2: Heat diffusion computational kernel.

```
for (i=1; i<width; i++){
   for (j=1; j<height; j++){
      A[i][j] = 1/4*(A[i-1][j] + A[i][j-1] +
                B[i+1][j] + B[i][j+1]);
   }
}
```

Estimation of communication parameters. The `mpptest` from the `perftest` suite [36] developed at Argonne National Laboratory was used to quantify the *communication parameters* of the experimental system. `Mpptest` can be used with any MPI implementation and is designed to measure the performance of the point-to-point MPI message passing routines. `mpptest` was very useful for the experiments because of its ability to measure communication performance with many participating processes, thus, exposing network contention and scalability problems. The communication parameters are independent of the application kernel or scheduling algorithm and therefore are the same for all test cases. In the experimental setup, the `-roundtrip` and `-sync` options of `mpptest` were used to measure the round trip time of blocking send and receive calls of messages of sizes from 100 to 10,000 bytes. The results for the Gigabit ethernet interconnection network are summarized in Figure 6.

As can be seen in Figure 6, the communication cost is almost constant in both regions, that is, when the message size is less than 4000 bytes and when it is greater than or equal to 4000 bytes. This implies that the effect of c_c (network throughput) in Equation (2) is negligible and that only c_d (transfer start-up cost) contributes to the actual communication cost.

Estimation of computation parameters. To quantify the *computation parameter(s)*, each computational kernel was executed for a representative problem size of $10,000 \times 10,000$ iterations, which amounts to $\sim$8.5% of the actual problem size for each kernel. During these runs, the

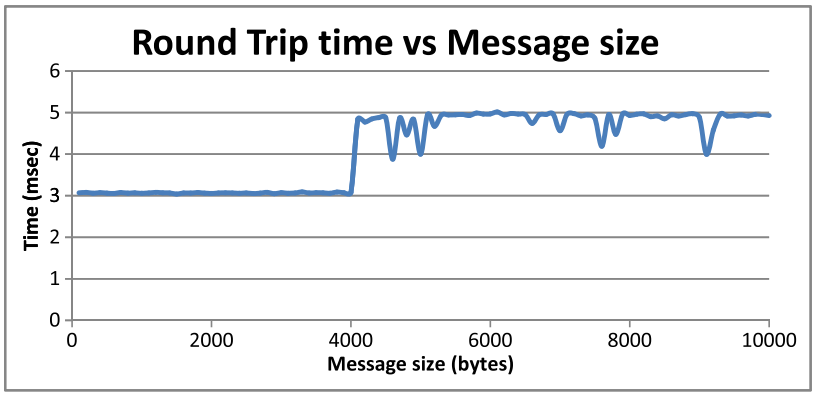


Figure 6. Round trip time of the experimental system with the Gigabit ethernet network as a function of the message size.

DOI: 10.1002/cpe

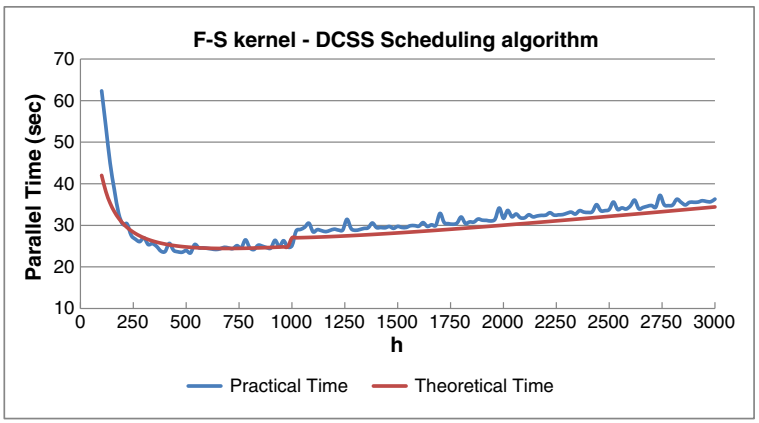
performance of each worker was measured and it was concluded that the twin workers are 1.3 times faster that the clone workers. Thus, each worker from the clones cluster was assigned a virtual power of 1, and each worker from the twins cluster was assigned a virtual power of 1.3. The total virtual power of the heterogeneous cluster was $A = 16 \times 1 + 16 \times 1.3 = 36.8$.

The scheduling overhead, c_{sch}, was also measured during the same test runs as earlier, for each scheduling algorithm (DCSS, DTSS, DFSS, and DGSS), with the following results: 3.75×10^{-5} s (DCSS), 4×10^{-5} s (DTSS), 8×10^{-5} s (DFSS), and 8.5×10^{-5} s (DGSS). Table III summarizes the estimated computation, communication, and scheduling parameters.

Table III. Estimated parameters for the scheduling, and the communication and computation models used to estimate the theoretical parallel execution time for all computational kernels.

	c_d	c_c	c_p^{slow}	VP^{slow}	VP^{fast}
F-S	$0.003 - 0.005$ s	2×10^{-8} s	4.3×10^{-8} s	1	1.3
DNA	$0.003 - 0.005$ s	2×10^{-8} s	6.4×10^{-8} s	1	1.3
HEAT	$0.003 - 0.005$ s	2×10^{-8} s	1.6×10^{-8} s	1	1.3

	DCSS	DTSS	DFSS	DGSS
c_{sch}	3.75×10^{-5} s	4×10^{-5} s	8×10^{-5} s	8.5×10^{-5} s



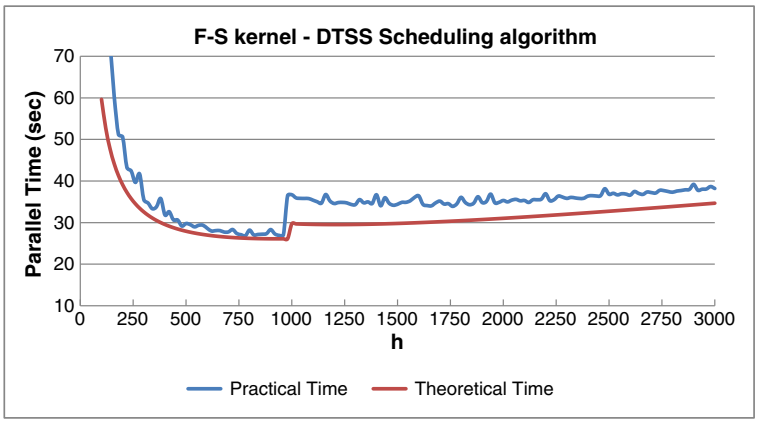


Figure 7. F-S: Theoretical versus actual (practical) parallel execution times (s) and synchronization granularity h for the DCSS and DTSS scheduling schemes.

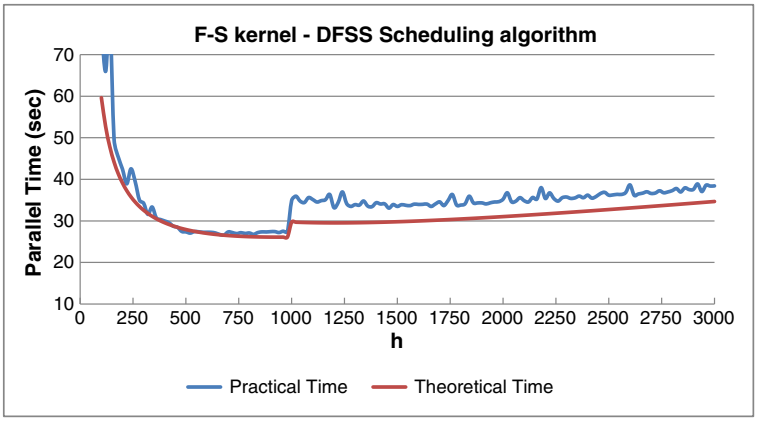
DOI: 10.1002/cpe

5.1. *Experimental results using the Gigabit ethernet interconnection network*

The following notation is used in this section:

- h^t_{opt}: the theoretical *optimal* synchronization granularity
- h^p_{opt}: the practical *optimal* synchronization granularity
- h^p: the practical synchronization granularity
- T^t_{min}: the theoretical *minimum* parallel execution time using the h^t_{opt} synchronization granularity
- T^p_{min}: the practical *minimum* parallel execution time using the h^p_{opt} synchronization granularity

In this section, the actual performance of the self-scheduling algorithms for the earlier mentioned computational kernels is compared with their theoretical performance as estimated by the model proposed in this paper. To assess the actual performance, the F-S, DNA, and HEAT computational kernels were executed for a problem size of $U_s \times U_c = 120,000 \times 100,000$ iterations using the DCSS, DTSS, DFSS, and DGSS scheduling algorithms. The synchronization granularity, h, for the experiments was in the range of $100, 200, \ldots, 3000$ iterations. The actual parallel execution times of these runs were measured. To produce more accurate results, the experiments were repeated 10 times for each computational kernel, and the average of the actual parallel execution times is reported. This was followed by the estimation of the theoretical parallel execution time for all kernels and for all self-scheduling algorithms using Equation (10), using the communication and computation parameters as they were determined previously (cf. Table III), for the same synchronization granularity range considered in the actual runs, that is, $h = 100, 200, \ldots, 3000$ iterations. The values for the



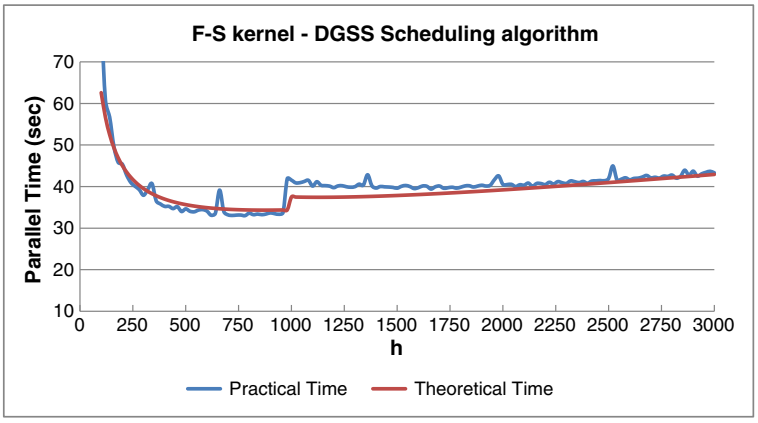
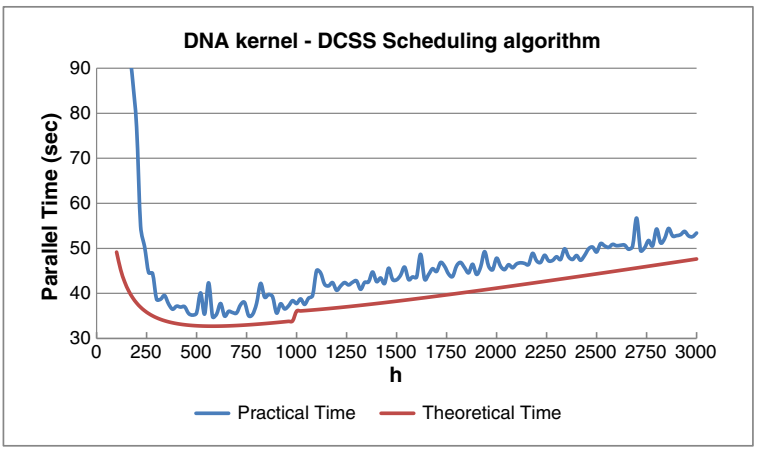


Figure 8. F-S: Theoretical versus actual (practical) parallel execution times (s) and synchronization granularity h for the DFSS and DGSS scheduling schemes.

DOI: 10.1002/cpe

number of scheduling steps N and the chunk sizes V_i for each self-scheduling algorithm were calculated using the formulas in Table II. In both the actual test runs and in the theoretical estimations, the total virtual power of the cluster of virtual homogeneous processors was assumed to be $A = 36.8$, and that the threshold C for the smallest chunk size assigned to a worker was set to 20 iterations.

The results from both the actual runs and the theoretical estimations are presented in the following plots (cf. Figures 7). One can see that the proposed model captures the system behavior accurately because the theoretical curve follows closely the shape of the actual curve in all cases. To assess the accuracy of the proposed theoretical model, the values of the synchronization granularity giving the minimum practical parallel execution time for each kernel and every self-scheduling algorithm were extracted from the plots. These values are denoted by h^p_{opt}. Next, the corresponding theoretical optimal synchronization granularity values were estimated using Equation (11), and these values were denoted by h^t_{opt}. The theoretical and practical optimal synchronization granularity values are summarized in Table IV. From Table IV, one can see that the estimated values of h^t_{opt} are very close to the actual ones, h^p_{opt}, with an average percentage difference of 13.92%. It must be noted that even in the worse case, that is, for the F-S kernel, where the largest percentage difference of h^t_{opt} and h^p_{opt} was 38.38% for the FSS scheduling algorithm, the corresponding performance degradation of the practical parallel execution time versus the theoretical parallel execution time is only 1.86% (or +0.495 s). In addition, the best synchronization granularity estimates were obtained for the HEAT computational kernel, where the percentage difference between the theoretical and practical optimal synchronization granularity values is zero for DTSS and DGSS. Thus, for the same theoretical and practical optimal synchronization granularity, the performance degradation of the practical parallel



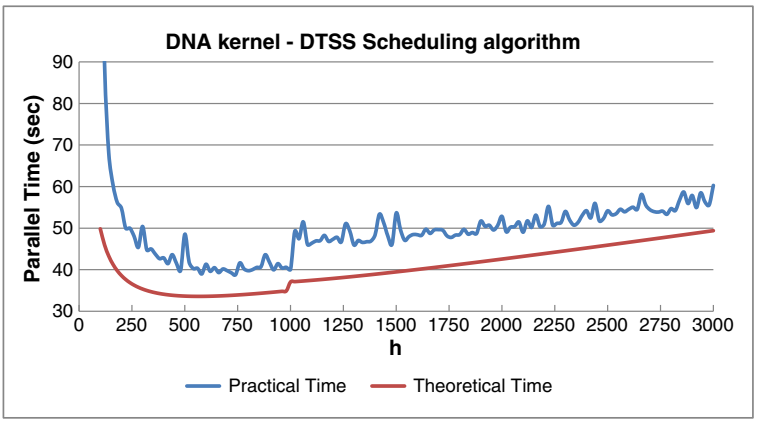


Figure 9. DNA: Theoretical versus actual (practical) parallel execution times (s) and synchronization granularity h for the DCSS and DTSS scheduling schemes.

DOI: 10.1002/cpe

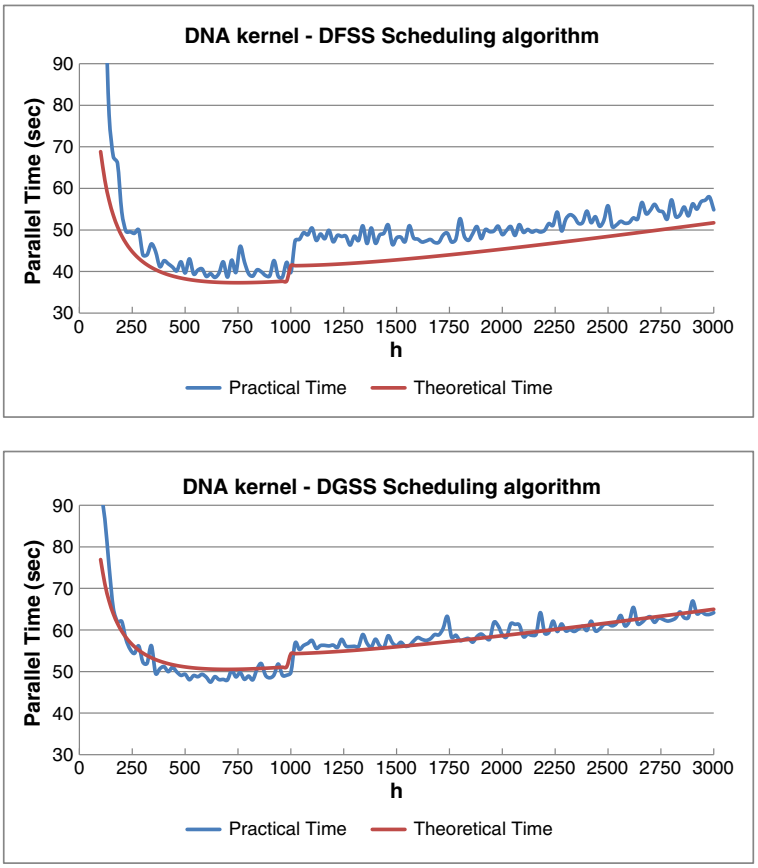


Figure 10. DNA: Theoretical versus actual (practical) parallel execution times (s) and synchronization granularity h for the DFSS and DGSS scheduling schemes.

execution time versus the theoretical parallel execution time using DTSS is 19.27% (or +2.69 s), whereas using DGSS is 0.99% (or +0.16 s). The values in Table IV indicate that one can efficiently execute a parallel application using the theoretically determined optimal synchronization granularity h^t_{opt}, which, in practice, gives a parallel execution time very close to the smallest practical parallel execution time.

In each plot in the Figures 7–12, the effects of the nonlinear communications, as indicated earlier in the description of the communication model (cf. Figure 6), can be clearly identified. Specifically, there is a sudden increase in both theoretical and practical parallel execution time when $h^p = 1000$ iterations. This increase is correlated to the communication penalty as explained in Figures 3 and 6. The value of $h^p = 1000$ iterations corresponds to the transition point of t_c for message sizes of 4000 bytes in Equation (2). In the implementation of each kernel, the iteration point values are expressed in floating point numbers, which are 4 bytes long each. Therefore, a synchronization granularity value of $h^p = 1000$ iterations signifies that workers exchange messages of 1000 floating point numbers or 4000 bytes, during every synchronization event. This increase is more noticeable in certain kernels, and specifically in the HEAT computational kernel. This is explained by the fact that this kernel has the smallest computation cost value, c_p, among all kernels (cf. Table II), which indicates that the parallel execution time of this kernel is largely dominated by the communication cost. It can be further concluded that as the ratio of communication time to the total parallel execution time increases, the value of h^p_{opt} that gives the minimum parallel execution time increases as well. This conclusion is in agreement with the intuitive assumption that as the communication cost increases, the synchronization granularity should increase, that is, synchronizations should occur less frequently. For the HEAT computational kernel, the value of h^p_{opt} that gives the minimum actual

DOI: 10.1002/cpe

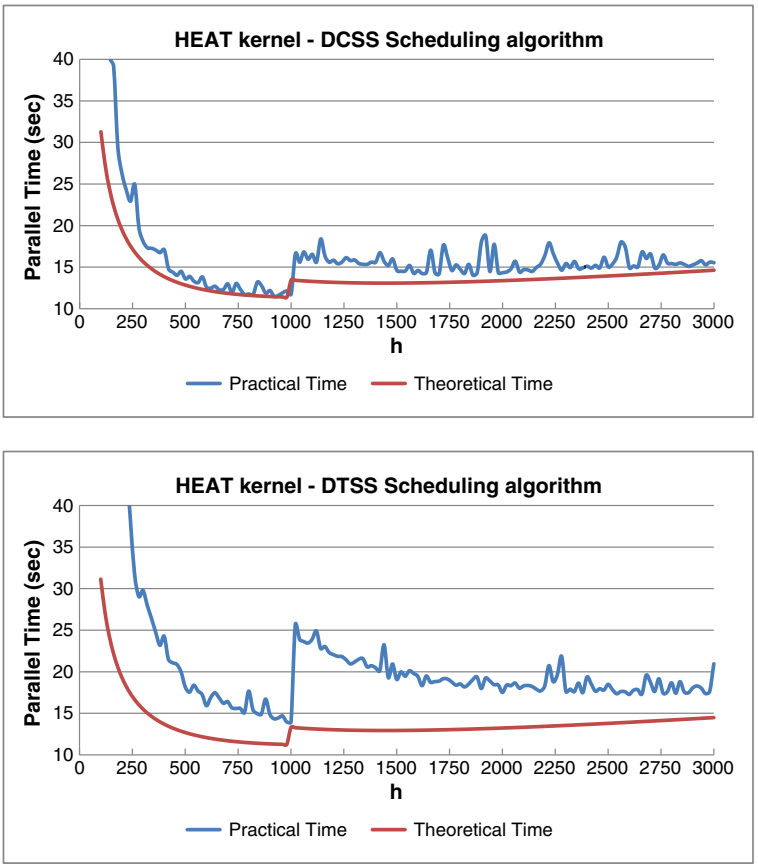


Figure 11. HEAT: Theoretical versus actual (practical) parallel execution times (s) and synchronization granularity h for the DCSS and DTSS scheduling schemes.

parallel execution time is found to be, in all cases, very close to the transition value of $h^p = 1000$ iterations. Finally, for the F-S and DNA kernels, the minimum actual parallel execution time is achieved for synchronization granularity values below the transition value, that is, $h^p < 1000$.

5.2. *Experimental results using the Myrinet interconnection network*

The second experiment was conducted on the first cluster using a 10-Gbit Myrinet interconnection network. The following configuration switches: `-mca btl mx,sm,self` have been used to enable the use of the Myrinet interconnection network for all OpenMPI communications. The total number of workers in this experiment was 24, coming from three nodes of the first cluster ($3 \times 8 = 24$ cores). The test case used was the Floyd–Steinberg computational kernel described earlier. As in the first set of experiments, `mpptest` was used to estimate the communication parameters for the Myrinet interconnection network. The results of the `mpptest` are depicted in Figure 13, in which the two linear regions, before and after 4000 bytes, can be identified. The estimated communication parameters for this case are summarized in Table V.

The computation model parameter was found to be the same as in the first experiment: for the workers of the first cluster, $c_p = 4.34 \times 10^{-8}$ s. In this experiment, a homogeneous system with 24 physical workers was used, each worker having a virtual computational power of 1; thus, the total virtual power of the cluster is 24. The problem size was $U_s \times U_c = 64,000 \times 100,000$ iterations, and the scheduling algorithm of choice was CSS. The threshold on the minimum chunk size was again set to 20 iterations.

DOI: 10.1002/cpe

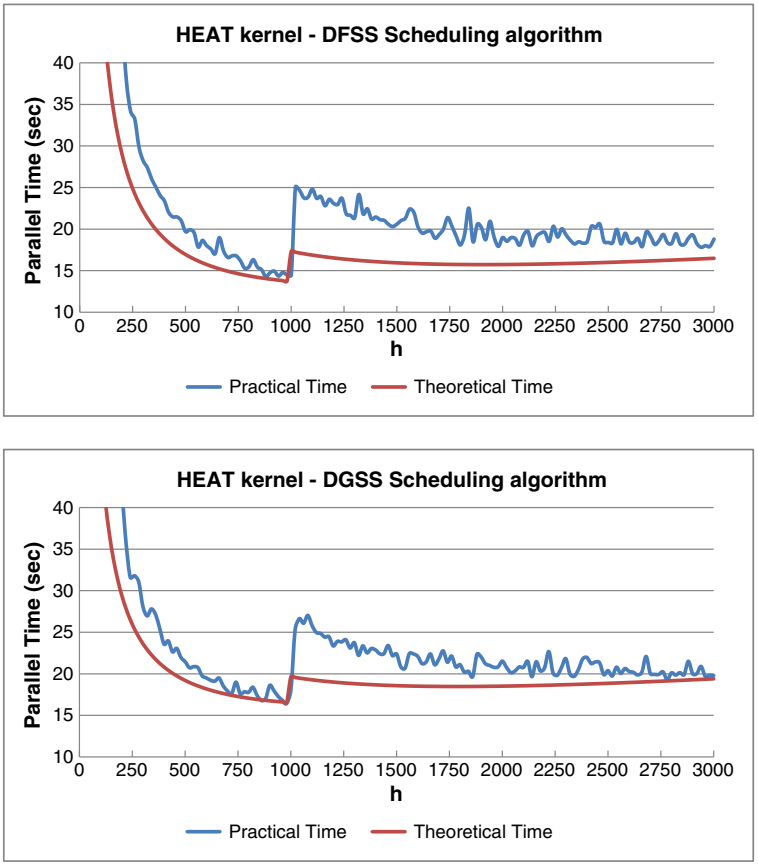


Figure 12. HEAT: Theoretical versus actual (practical) parallel execution times (s) and synchronization granularity h for the DFSS and DGSS scheduling schemes.

Table IV. Theoretical optimal and actual synchronization granularities, theoretical minimum and actual minimum parallel execution times and their percentage differences.

Sch. alg.	h^t_{opt} (iter.)	h^p_{opt} (iter.)	$\frac{\lvert h^t_{opt}-h^p_{opt}\rvert}{h^p_{opt}}\cdot 100$	T^t_{min} (s)	T^p_{min} (s)	$\frac{\lvert T^t_{min}-T^p_{min}\rvert}{T^p_{min}}\cdot 100$
F-S computational kernel, A=36.8 virtual homogeneous processors						
DCSS	697	520	34.04%	24.46	23.38	4.65%
DTSS	698	780	10.51%	26.50	26.80	1.12%
DFSS	**941**	**680**	**38.38%**	**26.12**	**26.62**	**1.86%**
DGSS	872	780	11.79%	34.35	33.03	4.00%
DNA computational kernel, A=36.8 virtual homogeneous processors						
DCSS	572	580	1.38%	32.73	34.841	5.98%
DTSS	572	740	22.70%	33.58	38.83	13.51%
DFSS	754	640	17.81%	37.33	38.60	3.28%
DGSS	698	620	12.58%	50.51	47.47	6.41%
HEAT computational kernel, A=36.8 virtual homogeneous processors						
DCSS	980	920	6.52%	11.41	11.46	0.39%
DTSS	**980**	**980**	**0.00%**	**11.27**	**13.96**	**19.27%**
DFSS	980	880	11.36%	13.753	14.27	3.62%
DGSS	**980**	**980**	**0.00%**	**16.58**	**16.42**	**0.99%**

The measured parallel execution time for values of h ranging from 0 to 1100 iterations, as well as the theoretically calculated parallel execution time given by Equation (10), are shown in Figure 14. One can note that the graph of the theoretically calculated parallel execution time closely follows

DOI: 10.1002/cpe

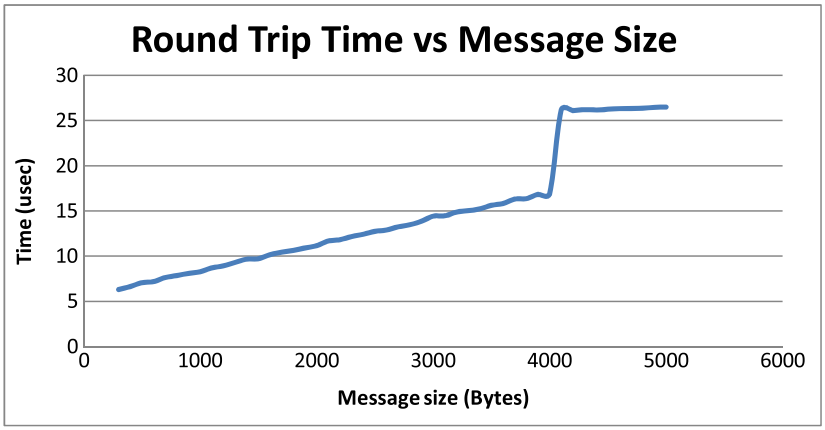


Figure 13. Round trip time versus message size for Myrinet interconnection network.

Table V. Communication and computation model parameters for the Myrinet case.

Message size (bytes)	c_c	c_d	c_p
$\leqslant 4000$	$2.87 \times 10^{-9}\ s$	$5.54 \times 10^{-6}\ s$	$4.34 \times 10^{-8}\ s$
> 4000	$6.14 \times 10^{-10}\ s$	$2.35 \times 10^{-5}\ s$	

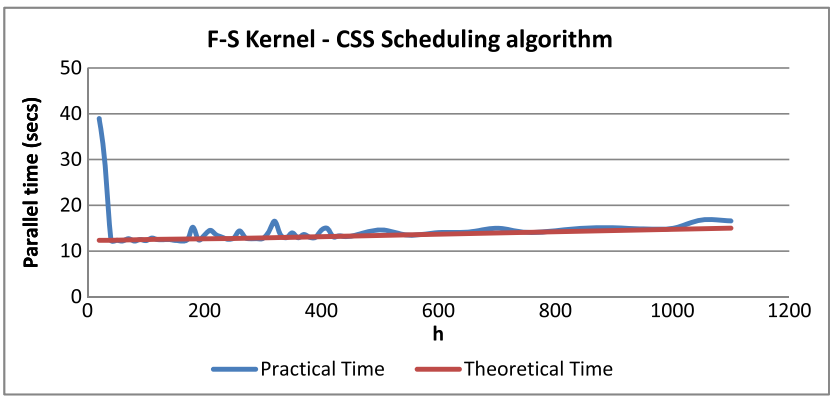


Figure 14. F-S: Theoretical versus actual (empirical) parallel execution times (s) and synchronization granularity h for the CSS scheduling scheme on a system with Myrinet interconnection network.

that of the empirically measured parallel execution time. Specifically, the slope of the practical parallel execution time begins with a steep "fall" followed by a slight linear increase. The theoretical parallel execution time captures the linear increase portion of the practical parallel execution time curve, without capturing the steep fall portion. The minimum theoretically calculated parallel execution time was obtained for $h^t_{opt} = 30$ iterations, whereas the minimum empirical parallel execution time is very close to that, with a value of $h^p_{opt} = 80$ iterations. The theoretical and actual parallel execution times plotted in Figure 14 indicate that there is no significant difference in the parallel execution time before and after the $h = 1000$ iterations transition point. This is in contrast to the results presented in Section 5.1, in Figures 7–12, and is caused by the fact that the communication cost for messages sent over the Myrinet interconnection network is much smaller and practically negligible compared with the cost of performing the F-S kernel computations, which remains the same regardless of the interconnection network. Thus, in the Myrinet interconnection case, the total (actual and theoretical) parallel execution times are dominated by the applications' computation time.

DOI: 10.1002/cpe

6. CONCLUSION AND FUTURE WORK

A theoretical model has been presented for determining the optimal synchronization granularity when parallelizing applications containing loops with data dependence distance vectors on heterogeneous systems. New formulas were given for estimating the total number of scheduling steps, under the assumption of a threshold on the minimum allowed chunk size assigned to a worker by a self-scheduling scheme. Estimating the total number of scheduling steps has a significant impact on the parallel execution performance. Hence, good estimates play a very important role in the efficient estimation of the performance of the self-scheduling schemes when applied to scientific applications on heterogeneous systems. The proposed model is general and therefore applicable to every self-scheduling algorithm and to any application containing constant data dependencies.

The significance and usefulness of the proposed model stem from mitigating the effects of a poor choice of synchronization granularity, which leads to poor load balance and significant performance degradation, while the cost of determining the best performance through exhaustive search among all possible granularity values is clearly prohibitive. The theoretical optimal granularity of synchronization determined by the proposed model has been shown to be very close to the actual (empirical) optimal synchronization granularity. The accuracy of the proposed methodology is, in all cases, confirmed by extensive experimental results on a heterogeneous system with two types of interconnection network.

The plans for future work include: (i) study of the impact of intra-node communication (for multicore and SMP systems) on the performance of the proposed theoretical model; and (ii) augmentation of the self-scheduling algorithms with the (offline) knowledge of the proposed theoretical model, so that the optimal synchronization granularity can be determined on-the-fly and the algorithms can automatically adapt it to any changes that may occur unexpectedly in the system.

ACKNOWLEDGEMENTS

The authors wish to express their sincere gratitude to the editor and the anonymous referees for their patience and constructive suggestions that considerably improved the quality of this manuscript.

This research of A. T. Chronopoulos was partly supported by an NSF grant (HRD-0932339) to the University of Texas at San Antonio.

REFERENCES

1. Kruskal CP, Weiss A. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering* 1985; **11**(10):1001–1016.
2. Tang P, Yew PC. Processor self-scheduling for multiple-nested parallel loops. *Proceedings of the International Conference on Parallel Processing*, St. Charles, IL, USA, 1986; 528–535.
3. Polychronopoulos CD, Kuck DJ. Guided self-scheduling: a practical self-scheduling scheme for parallel supercomputers. *IEEE Transactions on Computers* 1987; **C-36**(12):1425–1439.
4. Hummel SF, Schonberg E, Flynn LE. Factoring: a method for scheduling parallel loops. *Communications of the ACM* 1992; **35**(8):90–101.
5. Tzen TH, Ni LM. Trapezoid self-scheduling: a practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems* 1993; **4**(1):87–98.
6. Hummel SF, Schmidt J, Uma RN, Wein J. Load-sharing in heterogeneous systems via weighted factoring. *Proceedings of 8th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM New York, NY, USA, 1996.
7. Banicescu I, Liu Z. Adaptive factoring: a dynamic scheduling method tuned to the rate of weight changes. *Proceedings of the High Performance Computing Symposium 2000*, Washington, USA, 2000; 122–129.
8. Banicescu I, Velusamy V, Devaprasad J. On the scalability of dynamic scheduling scientific applications with adaptive weighted factoring. *Cluster Computing* 2003; **6**:215–226.
9. Herrera J, Huedo E, Montero RS, Llorente IM. Loosely-coupled loop scheduling in computational grids. *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, 3rd High-Performance Grid Computing Workshop (HPGC 2006)*, Rhodes Island, Greece, 25–29 April 2006.
10. Penmatsa S, Chronopoulos AT, Karonis NT, Toonen B. Implementation of distributed loop scheduling schemes on the TeraGrid. *Proceedings of the 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007), 4th High-Performance Grid Computing Workshop*, Long Beach, California, USA, March 2007; 1–8.

DOI: 10.1002/cpe

11. Fujita S. Semi-dynamic multiprocessor scheduling with an asymptotically optimal performance ratio. *IEICE Transactions on Fundamentals of Electronics. Communications and Computer Sciences* 2009; **E92.A**(8):1764–1770.
12. Díaz J, Reyes S, Nio A, Muñoz-Caro C. Derivation of self-scheduling algorithms for heterogeneous distributed computer systems: application to Internet-based grids of computers. *Future Generation Computer Systems, Elsevier Publishers* 2009; **25**(6):617–626.
13. Shih W-C, Tseng S-S, Yang C-T. Performance study of parallel programming on cloud computing environments using MapReduce. *Information Science and Applications (ICISA)*, Seul, Korea, 2010; 1–8.
14. Yang C-T, Wu C-C, Chang J-H. Performance-based parallel loop self-scheduling using hybrid OpenMP and MPI programming on multicore SMP clusters. *Concurrency and Computation-Practice and Experience* 2011; **23**(8):721–744.
15. Li P, Zhu Q, Ji Q, Zhu X. An approach of chunk-based task runtime prediction for self-scheduling on multi-core desk grid. *Journal of Computers* 2011; **6**(7):1339–1345.
16. Wu C-C, Yang C-T, Lai K-C, Chiu P-H. Designing parallel loop self-scheduling schemes using the hybrid MPI and OpenMP programming model for multi-core grid systems. *The Journal of Supercomputing* 2012; **59**:42–60.
17. Ciorba FM, Andronikos T, Riakiotakis I, Chronopoulos AT, Papakonstantinou G. Dynamic multi-phase scheduling for heterogeneous clusters. *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006)*, Rhodes, Greece, 2006.
18. Ciorba FM, Riakiotakis I, Andronikos T, Papakonstantinou G, Chronopoulos AT. Enhancing self-scheduling algorithms via synchronization and weighting. *Journal of Parallel Distributed Computing* 2008; **68**(2):246–264.
19. Allen R, Kennedy K. *Optimizing Compilers for Modern Architectures – A Dependence-based Approach*. Morgan Kaufmann Publishers Inc.: San Francisco, CA, USA, 2001.
20. Yang C-T, Cheng K-W, Li K-C. An efficient load balancing scheme for grid-based high performance scientific computing. *Proceedings of the 19th International Conference on Advanced Information Networking and Applications (AINA'05)*, Tamkang University, Taiwan, 2005.
21. Chronopoulos AT, Penmatsa S, Xu J, Ali S. Distributed loop-scheduling schemes for heterogeneous computer systems. *Concurrency and Computation: Practice and Experience* 2006; **18**(7).
22. Kejariwal A, Nicolau A, Polychronopoulos CD. History-aware self-scheduling. *Proceedings of the 2006 International Conference on Parallel Processing (ICPP 2006)*, Columbus, Ohio USA, 2006; 185–192.
23. Riakiotakis I, Ciorba FM, Andronikos T, Papakonstantinou G. Self-adapting scheduling for tasks with data dependencies in stochastic environments. *Proceedings of the 5th International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar 06) of the CLUSTER 2006*, Barcelona, Spain, 2006.
24. Desprez F, Ramet P, Roman J. Optimal grain size computation for pipelined algorithms. *Proceeding Euro-Par '96 Proceedings of the Second International Euro-Par Conference on Parallel Processing* 1996; **1**:165–172.
25. Andonov R, Rajopadhye S. Optimal orthogonal tiling of 2-D iterations. *Journal of Parallel and Distributed Computing* 1997; **45**(2):159–165.
26. Xue J. *Loop Tiling for Parallelism*. Kluwer Academic Publishers Norwell: MA, USA, August 2000.
27. Xue J, Cai W. Time-minimal tiling when rise is larger than zero. *Parallel Computing* 2002; **28**(6):915–939.
28. Strout MM, Carter L, Ferrante J, Kreaseck B. Sparse tiling for stationary iterative methods. *International Journal of High Performance Computing Applications* 2004; **18**(1):95–113.
29. Ponnusamy R, Saltz J, Choudhary A, Hwang Y-S, Fox G. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Transactions on Parallel and Distributed Systems* 1995; **6**(8):815–831.
30. Huang T-C, Hsu P-H, Sheng T-N. Efficient runtime scheduling for parallelizing partially parallel loop. *Proceedings of the 3rd International Conference on Algorithms and Architectures for Parallel Processing*, Melbourne, Victoria, Australia, 1997; 397–403.
31. Lowenthal DK. Accurately selecting block size at runtime in pipelined parallel programs. *International Journal of Parallel Programming* 2000; **28**(3):245–274.
32. Boulet P, Dongarra J, Rastello F, Robert Y, Vivien F. Algorithmic issues on heterogeneous computing platforms. *Parallel Processing Letters* 1998; **9**(2):197–213.
33. Chen S, Xue J. Partitioning and scheduling loops on NOWs. *Computer Communications Journal* 1999; **22**: 1017–1033.
34. Ciorba FM, Riakiotakis I, Andronikos T, Chronopoulos AT, Papakonstantinou G. Optimal synchronization frequency for dynamic pipelined computations on heterogeneous systems (poster). *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER 2007)*, Austin, TX USA, 2007.
35. Andronikos T, Ciorba FM, Riakiotakis I, Papakonstantinou G, Chronopoulos AT. Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems. *Journal of Performance Evaluation* 2010; **67**(12):1324–1339.
36. MPI Performance Topics. Available from:https://computing.llnl.gov/tutorials/mpi_performance/#MessageSize, Mpptest: http://www.mcs.anl.gov/research/projects/mpi/mpptest/, and Perftest package: http://wilbur.mcs.anl.gov/pub/mpi/tools/.
37. Cierniak M, Li W, Zaki MJ. Loop scheduling for heterogeneity. *Proceedings of the 4th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, 1995; 78–85.
38. Kunz T. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Transactions on Software Engineering* 1991; **17**(7):725–730.

DOI: 10.1002/cpe

39. Yue KK, Lijla DJ. Parallel loop scheduling for high-performance computers. *Technical Report No. HPPC-94-12*, Dept. of Computer Science, Univ. of Minnesota, 1994.
40. Chronopoulos AT, Penmatsa S, Yu N, Yu D. Scalable loop self-scheduling schemes for heterogeneous clusters. *International Journal of Computational Science and Engineering* 2005; **1**(2/3/4):110–117.
41. Wolfe M. Definition of dependence distance. *ACM Transactions on Programming Languages & Systems* 1994; **16**(4):1114–1116.
42. Wilkinson B, Allen M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, (2nd edn). Prentice Hall: Upper Saddle River, New Jersey 07458, 2005.
43. Mathcad. Available from: http://www.mathcad.com.
44. Floyd RW, Steinberg L. An adaptive algorithm for spatial grey scale. *Proceedings of the Society for Information Display* 1976; **17**:75–77.
45. Needleman SB, Wunsch CD. A general method applicable to the search for similarities in the amino acid sequence of two sequences. *Journal of Molecular Biology* 1970; **48**:443–453.

DOI: 10.1002/cpe