Optimal Synchronization Frequency for Dynamic Pipelined Computations on Heterogeneous Systems

F. M. Ciorba [#], I. Riakiotakis [#], T. Andronikos [*], A. T. Chronopoulos [§], G. Papakonstantinou [#]

[#] *School of Electrical and Computer Engineering, National Technical University of Athens*
9, Heroon Polytechnioy, Zografou, 15773, Athens, Greece
{cflorina,iriak,papakon}@cslab.ntua.gr

[*] *Department of Informatics, Ionian University, 7, Tsirigoti Square, 49100 Corfu - Greece*
andronikos@ionio.gr

[§] *Department of Computer Science, University of Texas at San Antonio, TX USA 78249*
atc@cs.utsa.edu

Abstract— **In this paper we give a theoretical model for determining the synchronization frequency that minimizes the parallel execution time of loops with uniform dependencies dynamically scheduled on heterogeneous systems. Using this model we determine the synchronization frequency that minimizes the estimated parallel time. The accuracy of our method is validated through experiments on a heterogeneous cluster. The results show that the synchronization frequency minimizing the parallel time determined by our method, is very close to the synchronization frequency found experimentally.**

I. Introduction

Many scheduling algorithms have been devised for nested loops with and without dependencies on general heterogeneous distributed systems. When scheduling tasks on heterogeneous distributed systems, there are three issues that must be addressed: machine and network *heterogeneity*), *variable workload* (non-dedicated system) and *communication overhead*. The existence of dependencies in nested loops incurs cross-node communication. In order to minimize the communication overhead, the frequency of the cross-node communications should be set such that the highest level of parallelism is achieved while the synchronization overhead is kept to a minimum. In a coarse grain approach to parallelizing applications, the problem domain is partitioned into chunks of equal or variable size, which are assigned to the existing processing nodes. An important class of coarse grain scheduling algorithms are the self-scheduling schemes, which dynamically assign chunks of variable sizes to processing nodes. The self-scheduling algorithms differ in the way they compute the amount of work assigned to each node. In [11] it was shown that these algorithms can be efficiently applied to loops with dependencies. The focus of this work is finding the best trade-off between parallelism and synchronization overhead of such loops in heterogeneous systems. We find the synchronization frequency that minimizes the parallel time and we show it is very close to the actual synchronization frequency that gives the least parallel time.

Related Work. A significant amount of work exists for determining the optimal partitioning of nested loops with dependencies for homogeneous distributed systems ([3], [5], [8], [9] and references therein). Ponnusamy et al [2] used the inspector/executor model to predict the communication requirements of the code and carry out communication optimization at runtime for irregular concurrent problems. Huang et al [4] presented a scheduler that divides a partially parallel loop into independent wavefronts and then executes the iterations in a wavefront parallelly. Lowenthal et al [7] proposed a method for accurately selecting at runtime the block size in pipelined programs. The block size was selected at run-time according to the workload variations that occur with highly irregular programs. The problem of finding the optimal partitioning of iteration spaces for heterogeneous systems has not been studied to the same extent. Boulet et al [14] were the first to apply loop tiling on heterogeneous systems. They divide fully permutable loops into equally sized tiles and allocate blocks with more tiles to faster processors. A good tile allocation is achieved through an algorithm that selects the best possible allocation based on a cost function. Variable size tiles are avoided in order to simplify code generation. Chen and Xue [6] assign tiles of variable size to processors based on a function that minimizes the parallel time according to a computation and a communication cost model. None of the above works addressed the case of dynamically computing and allocating chunks of computations to processors on heterogeneous systems.

Our Approach. In this paper we propose a theoretical model for finding the synchronization frequency which minimizes the parallel execution time of loops with dependencies when these are scheduled on heterogeneous systems using self-scheduling algorithms. Self-scheduling algorithms [11] are dynamic load balancing algorithms and they *dynamically* compute the size of a chunk and assign it to a node. In our approach, the chunks computed by a self-scheduling algorithm are also *weighted* according to each node's computational power. The computational power is determined according to the number of processes in the run queue of a node and its nominal computation power. The nominal computational power of each machine can be established as the normalized execution time of a small subset of the application on each machine. Using the number of processes in the run queue, as workload descriptor, was shown to be the best choice compared to other workload descriptors [1].

Contribution. This paper gives an experimentally verified theoretical model that determines the synchronization interval that minimizes the communication overhead, while yielding sufficient concurrency for reaching the best possible overall performance on non-dedicated heterogeneous systems.

Organization. The paper is organized as follows. Section II gives some background on loops with dependencies and pipelined computations. We present the model that gives the optimal synchronization frequency for heterogeneous systems in Section III. We experimentally validate our model in Section IV and we draw conclusions in Section V.

II. Background and notations

The iteration space of a parallel loop is modeled as an n-dimensional Cartesian space J ($J \subset Z^n$), where n is the depth of the loop nest and each point of this n-dimensional space is a distinct iteration of the loop body. Without loss of generality we assume that the loops have index points $(u_1, \ldots, u_n)$, where $1 \leq u_i \leq U_i$, $1 \leq i \leq n$. Typically self-scheduling algorithms divide the iterations into chunks of consecutive rows, resulting in the partition of a single dimension of the iteration space. This dimension, denoted U_c, is called the *chunk* dimension (see Fig. 2). An overview of the CSS, GSS, FSS, TSS and DTSS self-scheduling schemes (as they were devised for loops without dependencies) is given in [10] and references therein.

In nested loops, dependencies are modeled by dependence vectors (oriented vectors in the n-dimensional Cartesian space) whose set is denoted by $DS = \{\tilde{\mathrm{d}}_1, \ldots, \tilde{\mathrm{d}}_r\}$. We assume the dependence vectors are constant. Dependencies give rise to cross-node dependencies, which result when the computation of a data element on one node cannot be performed without the data element(s) on another node. The existence of such dependencies can slow down and (in extreme cases) even serialize the parallel execution. These dependencies are common in scientific codes, including, for example partial differential equations or computer graphics.

In self-scheduling schemes the idle processors (workers) usually request work from a master. In this work we assume a master-worker model with m workers, which are assigned the iterations to execute by the master. We denote by N the total number of scheduling steps. Applying self-scheduling algorithms to nested loops with dependencies, leads to a pipelined parallel execution due to the data dependencies [11][15]. The chunk sizes given by the self-scheduling algorithms *along* U_c are denoted V_i, for $i = 1, \ldots, N$. For a nest with n loops, the resulting chunks are n-dimensional parallelepipeds. The size of the *entire* n-dimensional chunk at scheduling step i is C_i. Pipelined execution is achieved by inserting equally spaced synchronization points along another (usually

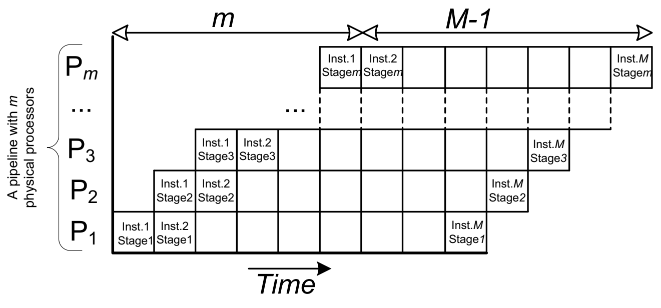

Fig. 1. Space diagram of a coarse-grain pipeline with m stages, M instances and $m + (M - 1)$ steps.

the largest) dimension of the iteration space, which we call U_s. It is obvious that the application must have at least two dimensions ($n \geq 2$). If the application has more than two loops, we take as U_c and U_s the two outer loops.

Generally, each chunk assignment round corresponds to a pipeline with as many stages as the number of workers m (see Fig. 1). The total number of assignment rounds, i.e., the number of pipelines is denoted by p. In a pipeline organization, each worker synchronizes with its neighbors (called local synchronization). Synchronization is performed throughout the execution of each pipeline at predetermined points, called synchronization points. A *synchronization interval*, denoted h, is the number of iterations along U_s upon the computation of which neighboring workers synchronize. We define the *synchronization frequency* as the number of synchronization points over the synchronization dimension. We insert M synchronization points along U_s (see Fig. 2). The pipeline requires $m+(M-1)$ steps, where m is the number of workers, which is also the *pipeline latency* of a pipeline with m stages. Data produced at the end of one stage are fed to the next stage. The length of the synchronization interval determines the amount of computation before sending a message. It becomes obvious that the synchronization frequency plays an important role in the total parallel time. Frequent synchronization implies excessive communication, whereas infrequent synchronization may limit the inherent parallelism.

III. Determining the synchronization frequency for heterogeneous systems

Distributed versions of the loop self-scheduling algorithms (CSS, TSS, FSS, GSS) have been studied before ([12], [10], [16] and references therein). The approach used in Distributed TSS can be applied to the other self-scheduling algorithms so that they can be efficiently run on non-dedicated heterogeneous systems [10] [16]. In this work, this is accomplished by transforming the array of m physical heterogeneous processors into an array of A virtual homogeneous processors, 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 and $k = 1, \ldots, m$. The available computational power is taken as the ratio of the virtual computational power (VP_k) of the worker

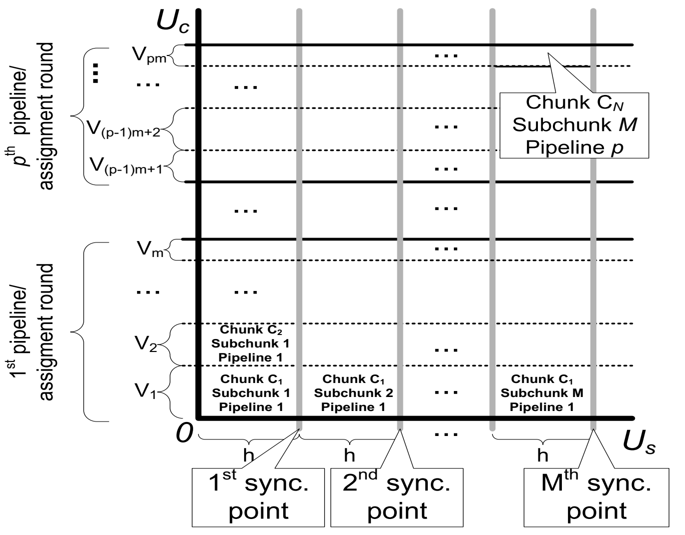
U_c
p^{th} pipeline/ assignment round
1st pipeline/ assigment round
V_{pm}
$V_{(p-1)m+2}$
$V_{(p-1)m+1}$
V_m
V_2
V_1
Chunk C_N Subchunk M Pipeline p
Chunk C_2 Subchunk 1 Pipeline 1
Chunk C_1 Subchunk 1 Pipeline 1
Chunk C_1 Subchunk 2 Pipeline 1
Chunk C_1 Subchunk M Pipeline 1
0
U_s
h
1st sync. point
2nd sync. point
Mth sync. point

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

over the number of processes in its run-queue (q_k) [1][10]. We consider that the slowest worker has $A_k = 1$. The number of virtual processors is equal to the total available power of the heterogeneous system, i.e., $A = \sum_{k=1}^{m} A_k$. The ratio $\frac{m}{A}$ is an indicator of the system heterogeneity. In this case, each pipeline consists of A stages instead of m, and M instances. Unlike the approaches in [10] and [16], here, we use the virtualization method directly in the implementation of the algorithms. In the rest of the paper we will refer to the distributed algorithms applied to arrays of A virtual processors as DCSS, DGSS, DTSS and DFSS.

In order to estimate the parallel time we describe the computation and communication cost models:

Computation cost model: We define the computation cost as a linear function of the computation cost per iteration times the number of iterations. Each virtual processor has a computational power equal to the computational power of the slowest physical processor. Hence, the computation cost of a subchunk is:

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

where V_i is the size of chunk C_i along the scheduling dimension U_c, and c_p is the computation time per iteration of the *slowest* worker (Fig. 3).

Communication cost model: We assume that the cost of sending a message is equal to the cost of receiving a message. This is a simplistic but realistic assumption, since in most cases the send and receive operations are executed in pairs between communicating workers. Therefore, the cost t_c of communicating a message of size h (Fig. 2 and 3) between two workers is:

$$t_c = c_d + hc_c, \tag{2}$$

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

Observation: In a heterogeneous system, processors may communicate at different speeds, even if the interconnection network is homogeneous. Also, when several processors send a message at the same time, network congestion occurs. We take both these facts into account by timing multiple ping-pong tests between multiple pairs of processors and taking the average c_c and c_d.

Fig. 3 represents p replications of Fig. 1. The total parallel execution time is the completion time of the last subchunk of the problem, illustrated in Fig. 3 as "P_A, subchunk M, pipeline p". The total parallel time can be estimated as the sum of the total *computation* time, the total *communication* time and the *work assignment* time. We assume that the iteration space is computed in p pipelines, where $p = \lceil \frac{N}{A} \rceil$. The output of one pipeline is the input of the next pipeline. Thus, the total parallel time is the sum of the completion times of all p pipelines plus the time to transfer date between pipelines. Each pipeline can be completed in $A+(M-1)$ steps (see Fig. 1 and 3). Each of these steps consists of a *receive*, *compute* and *send* operation.

Let V_{jA+1} denote the size of the first chunk for $j = 0, \ldots, p-1$ given by a self-scheduling algorithm. The total *computation* time for all pipelines is:

$$T_{comp} = hU_c c_p \frac{m}{A} + c_p(U_s - h)\sum_{j=0}^{p-1} V_{jA+1} \tag{3}$$

Remark 1. (1.a) Since we have p pipelines, the problem can be computed in $p[A+(M-1)]$ steps. The first term of eq. (1) is the computation time of all last subchunks of the problem. This is illustrated by the vertical shaded rectangle in Fig. 3. Since we use as c_p the computation time per iteration of the slowest workers, we must scale the first term by $\frac{m}{A}$.
(1.b) Similarly, the second term of eq. (1) represents the computation time of the $M-1$ instances of every pipeline. This is also illustrated in Fig. 3 by the horizontal shaded rectangles. For the self-scheduling algorithms considered herein we take into account the computation time of the first chunk of each pipeline because it is always greater than the computation time of all other chunks in the pipeline. It is obvious that the total computation time is the sum of all vertical and horizontal shaded rectangles, meaning the total number of steps of all pipelines (see Fig. 1 and 3).

We note that data exchange occurs only between the m physical processors and not between the A virtual processors. We denote by t_c the time to send or receive a message (taken twice for a complete communication operation) and defined by eq. (2), and by T_{tran} the time to transfer the necessary data from one pipeline to the next. Thus, the time to perform all receive and send operations, i.e., the communication time for all pipelines and for transferring the data between two adjacent pipelines is derived as:

$$\begin{aligned} T_{comm} &= p(m-2)(2t_c) + pt_c(\frac{U_s}{h} - 1) \\ &\quad + (p-1)T_{tran} \end{aligned} \tag{4}$$

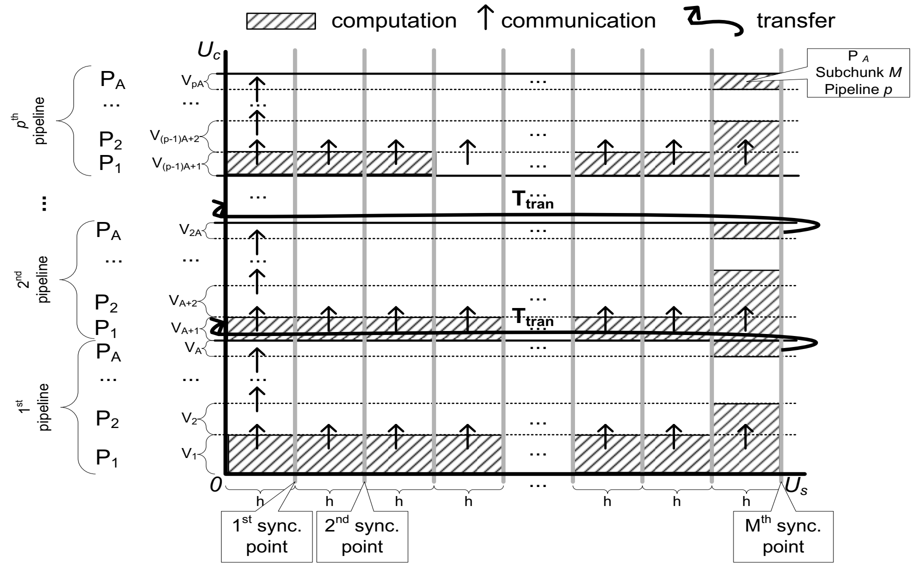

Fig. 3. Computation and communication pattern of an iteration space with p pipelines, each with A stages, M instances and $A + (M - 1)$ number of steps.

Remark 2. (2.a) For a given problem size, the communication time is a linearly increasing function of m and p. The first term of eq. (1) represents the communication time associated with the computation of all last subchunks of the problem (i.e. the first term of eq. (1)).
(2.b) Similarly, the second term of eq. (1) is the communication time associated with the computation of all $M - 1$ instances of each pipeline (i.e. the second term of eq. (1)). T_{tran} is the time to transfer the necessary data from one pipeline to the next and is given by 2 $(c_d + U_s c_c)$ (see Fig. 3).

Since we use the master-worker model, the work assignment time is $T_{wa} = 2t_c + c_{sch}$, where $2t_c$ is the transmission time needed for the work request to reach the master and for the master's reply to reach the worker; c_{sch} is the time needed for the master to compute the next executable chunk size, called *scheduling overhead*. Therefore, the total parallel time is:

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

Note that 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.

We are interested in finding the minimum of the parallel time defined above as a function of h. This is stated as Proposition 1. Table I summarizes the formulas giving the number of scheduling steps and the chunk sizes of the various schemes. F, L, D and α are parameters used by the algorithms and are described in [10]. For the various N and V_i given by the different self-scheduling algorithms, the minimum parallel time can be found by differentiating of T_{par} and verifying that the second order derivative is positive. We note that the values of N and V_i are computed as described in Section [10] by applying the self-scheduling algorithms to an array of A processors.

TABLE I
SCHEDULING STEPS AND CHUNK SIZES

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

Proposition 1. It can be proved that the parallel time T_{par} assumes a minimum as a function of h at:

$$h_{opt} = \sqrt{\frac{U_s\ p\ A\ c_d}{(2m-5)Ac_cp + U_cc_pm - c_pA\sum_{j=0}^{p-1}V_{jA+1}}} \tag{6}$$

IV. EXPERIMENTAL VALIDATION

Setup. The experiments were performed on a cluster of 17 (1 master and 16 workers) Intel Xeons CPUs at 2.8GHz, with 2GB RAM, interconnected with GigaBit Ethernet. In order to make the system heterogeneous, we put an artificial load in the background of m/2 of the workers. The results presented below are the average of 20 runs for each experiment. As a test case we used the Floyd-Steinberg (F-S) error dithering kernel from computer graphics, which is a doubly nested loop with 4 dependencies. We ran F-S for $m = 16$ workers.
Estimation of parameters. To quantify the *communication parameters* we developed a benchmark program, that simulates a small scale master-worker model with cross-node communication. It performs send and receive calls between all pairs of workers for different messages sizes, in order to simulate levels of network congestion similar to those of the actual application. We measured the average round-trip time for all data exchanges and then divided it by 2, assuming that the send and receive times are equal. This allowed us to

estimate the start-up time $c_d = 8 \times 10^{-5}s$ and throughput $c_c = 6.55 \times 10^{-7}s$. The scheduling overhead was measured for each algorithm and the values of c_{sch} are as follows: DCSS $c_{sch} = 3.2 \times 10^{-5}s$, DGSS $c_{sch} = 8.5 \times 10^{-5}s$, DTSS $c_{sch} = 3.4 \times 10^{-5}s$ and DFSS $c_{sch} = 7 \times 10^{-5}s$. We ran a small size version of the (F-S) application on both the overloaded (slow) and ordinary (fast) workers, in order to obtain their c_p. The c_p for the fast worker was $c_p^{fast} = 5.6 \times 10^{-8}s$ and for the slow $c_p^{slow} = 2c_p^{fast}$. And thus, $VP^{slow} = 1$ for the slow and $VP^{fast} = 2$ for the fast workers.

For the parallel tests, we overloaded $m/2$ of the processors. For $m = 16$ nodes, the total available power was $A = 24$. We ran F-S with DCSS, DTSS, DFSS, and DGSS on an image frame of $50K \times 150K$ pixels for a synchronization interval in the range of $20, 40, \ldots, 1000$ and measured the actual parallel time. We plotted this time against the theoretical parallel time obtained by eq. (5). We also determined theoretically the optimal synchronization interval h_{opt} for each algorithm, on 24 virtual processors. The results are shown in Fig. 4. One can see that our method describes accurately the behavior of the system because the theoretical curve follows closely the actual curve in all cases. The results show that the optimal h_{opt} given by eq. (6) is very close to the actual h_a that gives the minimum parallel time in practice.

V. Conclusion

We proposed a method for determining the optimal synchronization frequency for loops with dependencies on heterogeneous systems. The method is general and applicable to the class of self-scheduling algorithms. The method is useful because a poor choice of the synchronization frequency leads to severe performance degradation and the cost of determining the best performance through exhaustive searching of all possible values of h is clearly prohibitive. We showed that the optimal synchronization interval given by our method is very close to the actual synchronization interval. The accuracy of the proposed methodology, in all cases, is confirmed by experimental results on a heterogeneous cluster.

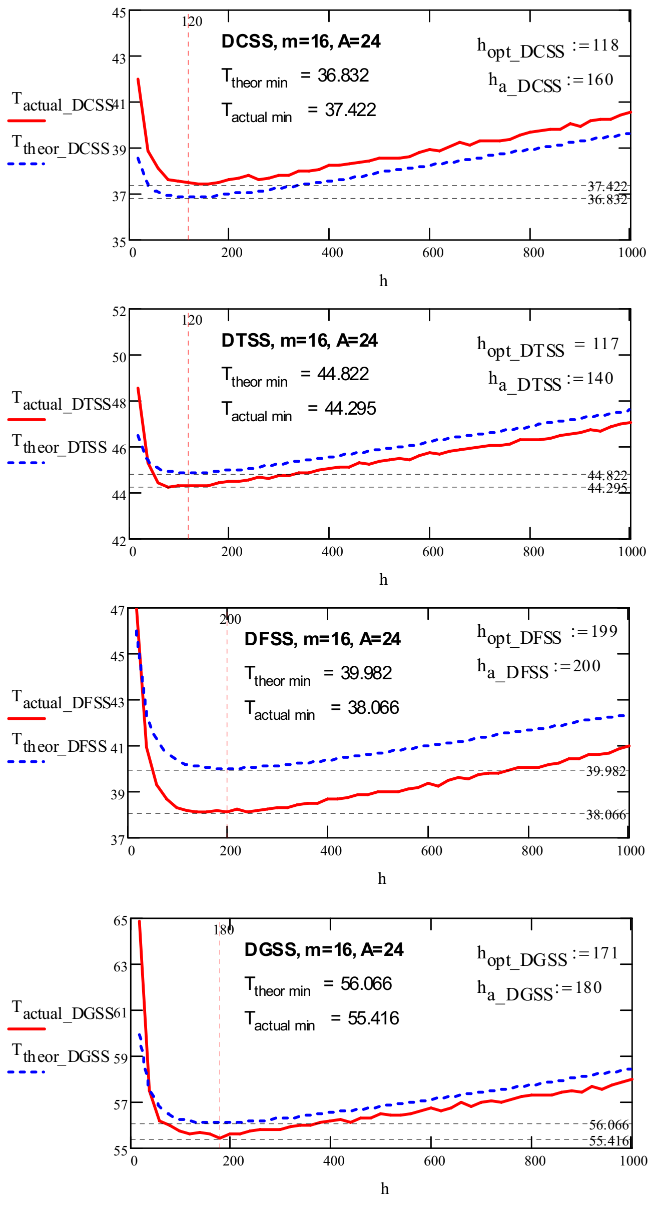

Fig. 4. Theoretical versus actual parallel times (sec) and theoretical versus actual h for DCSS, DTSS, DFSS, and DGSS, and for $h = 20, 40, \ldots, 1000$.

References

[1] T. Kunz. The Influence of Different Workload Descriptions on a Heuristic Load Balancing Scheme. *IEEE Trans. on Soft. Eng.* pp. 725–730, 1991.

[2] R. Ponnusamy, J. Saltz, A. Choudhary, Y.-S. Hwang, G. Fox. Runtime Support and Compilation Methods for User-Specified Irregular Data Distributions. *IEEE Trans. on Par. and Dist. Systems*, 6:8, pp. 815–831, 1995.

[3] F. Desprez, P. Ramet and J. Roman. Optimal Grain Size Computation for Pipelined Algorithms, *Euro-Par*, 1:165–172, 1996.

[4] T.-C. Huang, P.-H. Hsu, T.-N. Sheng. Efficient Run-time Scheduling for Parallelizing Partially Parallel Loop. *Proc. of the 3rd Int'l Conf. on Algorithms and Architectures for Parallel Processing*. pp. 397–403, 1997.

[5] R. Andonov and S. Rajopadhye. Optimal Orthogonal Tiling of 2-D Iterations, *Int. J. of Parallel and Distrib. Computing*, 45(2):159–165, 1997.

[6] S. Chen and J. Xue. Partitioning and scheduling loops on NOWs, *Comp. Comm. J.*, 22:1017–1033, 1999.

[7] D.K. Lowenthal. Accurately Selecting Block Size at Run Time in Pipelined Parallel Programs, *Int. J. of Parallel Programming*, 28(3):245-274, 2000.

[8] J. Xue and W. Cai. Time-minimal Tiling when Rise is Larger than Zero, *Par. Comp.*, 28(6):915–939, 2002.

[9] M. M. Strout, L. Carter, J. Ferrante and B. Kreaseck. Sparse Tiling for Stationary Iterative Methods, *Int'l J. of High Perf. Comp. Apps*, 18(1):95–113, 2004.

[10] A.T. Chronopoulos, S. Penmatsa , J. Xu, and S. Ali. Distributed loop-scheduling schemes for heterogeneous computer systems, it Concurrency and Computation: Practice and Experience, 18(7):771-785, 2006.

[11] F.M. Ciorba, T. Andronikos, I. Riakiotakis, A.T. Chronopoulos and G. Papakonstantinou. Dynamic Multi Phase Scheduling for Heterogeneous Clusters, *Proc. of the 20th IEEE Int. Parallel & Distributed Processing Symposium (IPDPS 2006)*, Rhodes, Greece, 2006.

[12] I. Banicescu, V. Velusamy and J. Devaprasad. On the Scalability of Dynamic Scheduling Scientific Applications with Adaptive Weighted Factoring, *Cluster Computing 6*, 2003, pp. 215–226.

[13] K. K. Yue, D. J. Lijla. Parallel Loop Scheduling for High-Performance Computers, *Tehcnical Report No. HPPC-94-12*, Dept. of Computer Science, Univ. of Minnessota, 1994.

[14] P. Boulet, J. Dongarra, F. Rastello, Y. Robert and F. Vivien. Algorithmic Issues on Heterogeneous Computing Platforms. *Parallel Processing Letters*, 9(2): 197–213, 1998.

[15] B. Wilkinson and M. Allen. Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers. Prentice Hall 2005.

[16] F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou and A. T. Chronopoulos. Enhancing self-scheduling algorithms via synchronization and weighting. *Journal of Parallel and Distributed Computing*, To appear.