Static Tiling for Heterogeneous Computing Platforms*

Pierre Boulet[1], Jack Dongarra[2,3], Yves Robert[4] and Frédéric Vivien[5]

[1] LIFL, Université de Lille, 59655 Villeneuve d'Ascq Cedex, France
[2] Department of Computer Science, University of Tennessee, Knoxville, TN 37996-1301, USA
[3] Mathematical Sciences Section, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[4] LIP, Ecole normale supérieure de Lyon, 69364 Lyon Cedex 07, France
[5] ICPS, Université de Strasbourg, Pôle Api, 67400 Illkirch, France
`Contact e-mail: Yves.Robert@ens-lyon.fr`

February 1999

Abstract

In the framework of fully permutable loops, tiling has been extensively studied as a source-to-source program transformation. However, little work has been devoted to the mapping and scheduling of the tiles on physical processors. Moreover, targeting heterogeneous computing platforms has, to the best of our knowledge, never been considered. In this paper we extend static tiling techniques to the context of limited computational resources with different-speed processors. In particular, we present efficient scheduling and mapping strategies that are asymptotically optimal. The practical usefulness of these strategies is fully demonstrated by MPI experiments on a heterogeneous network of workstations.

Key words: tiling, communication-computation overlap, mapping, limited resources, different-speed processors, heterogeneous networks

Corresponding author: Yves Robert
LIP, Ecole Normale Supérieure de Lyon, 69364 Lyon Cedex 07, France
Phone: + 33 4 72 72 80 37, Fax: + 33 4 72 72 80 80
E-mail: Yves.Robert@ens-lyon.fr

*This work was supported in part by the National Science Foundation Grant No. ASC-9005933; by the Defense Advanced Research Projects Agency under contract DAAH04-95-1-0077, administered by the Army Research Office; by the Office of Scientific Computing, U.S. Department of Energy, under Contract DE-AC05-84OR21400; by the National Science Foundation Science and Technology Center Cooperative Agreement No. CCR-8809615; by the CNRS–ENS Lyon–INRIA project *ReMaP*; and by the Eureka Project *EuroTOPS*. Yves Robert's work was conducted at the University of Tennessee, while he was on leave from École normale supérieure de Lyon and partly supported by DRET/DGA under contract ERE 96-1104/A000/DRET/DS/SR.

1 Introduction

Tiling is a widely used technique to increase the granularity of computations and the locality of data references. This technique applies to sets of fully permutable loops [23, 18, 13]. The basic idea is to group elemental computation points into tiles that will be viewed as computational units (the loop nest must be permutable so that such a transformation is valid). The larger the tiles, the more efficient are the computations performed using state-of-the-art processors with pipelined arithmetic units and a multilevel memory hierarchy (this feature is illustrated by recasting numerical linear algebra algorithms in terms of blocked Level 3 BLAS kernels [14, 10]). Another advantage of tiling is the decrease in communication time (which is proportional to the surface of the tile) relative to the computation time (which is proportional to the volume of the tile). A disadvantage of tiling may be an increased latency; for example, if there are lots of data dependences, the first processor must complete the whole execution of the first tile before another processor can start the execution of the second one. Tiling also presents load-imbalance problems: the larger the tile, the more difficult it is to distribute computations equally among the processors.

Tiling has been studied by several authors and in different contexts (see, for example, [17, 22, 21, 6, 19, 1, 9]). Rather than providing a detailed motivation for tiling, we refer the reader to the papers by Calland, Dongarra, and Robert [8] and by Högsted, Carter, and Ferrante [16], which provide a review of the existing literature. Briefly, most of the work amounts to partitioning the iteration space of a uniform loop nest into tiles whose shape and size are optimized according to some criterion (such as the communication-to-computation ratio). Once the tile shape and size are defined, the tiles must be distributed to physical processors and the final scheduling must be computed.

A natural way to allocate tiles to physical processors is to use a cyclic allocation of tiles to processors. Several authors [19, 16, 4] suggest allocating columns of tiles to processors in a purely scattered fashion (in HPF words, this is a `CYCLIC(1)` distribution of tile columns to processors). The intuitive motivation is that a cyclic distribution of tiles is quite natural for load-balancing computations. Specifying a columnwise execution may lead to the simplest code generation. When all processors have equal speed, it turns out that a pure cyclic columnwise allocation provides the best solution among all possible distributions of tiles to processors [8]—provided that the communication cost for a tile is not greater than the computation cost. Since the communication cost for a tile is proportional to its surface, while the computation cost is proportional to its volume,[1] this hypothesis will be satisfied if the tile is large enough.[2]

However, the recent development of heterogeneous computing platforms poses a new challenge: that of incorporating processor speed as a new parameter of the tiles allocation problem. Intuitively, if the user wants to use a heterogeneous network of computers where, say, some processors are twice as fast as some other processors, we may want to assign twice as many tiles to the faster processors. A cyclic distribution is not likely to lead to an efficient implementation. Rather, we should use strategies that aim at load-balancing the work while not introducing idle time. The design of such strategies is the goal of this paper.

The motivation to using heterogeneous networks of workstations is clear: such networks are ubiquitous in university departments and companies. They represent the typical poor man's parallel computer: running a large PVM or MPI experiment (possibly all night long) is a cheap alternative

[1]For example, for two-dimensional tiles, the communication cost grows linearly with the tile size while the computation cost grows quadratically.

[2]Of course, we can imagine a theoretical situation in which the communication cost is so large that a sequential execution would lead to the best result.

to buying supercomputer hours. The idea is to make use of *all* available resources, namely slower machines *in addition to* more recent ones.

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speed. Distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. At first sight, we may think that dynamic strategies like a greedy algorithm are likely to perform better, because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation (see the survey paper of Berman [5] and the more specialized references [2, 12] for further details). However, data dependences may lead to slow the whole process down to the pace of the slowest processor, as we demonstrate in Section 4.

The rest of the paper is organized as follows. In Section 2 we formally state the problem of tiles allocation and scheduling for heterogeneous computing platforms. All our hypotheses are listed and discussed, and we give a theoretical way to solve the problem by casting it in terms of an integer linear programming (ILP) problem. The cost of solving the linear problem turns out to be prohibitive in practice, so we restrict ourselves to columnwise allocations. Fortunately, there exist asymptotically optimal columnwise allocations, as shown in Section 3, where several heuristics are introduced and proved. In Section 4 we provide MPI experiments that demonstrate the practical usefulness of our columnwise heuristics on a network of workstations. Finally, we state some conclusions in Section 5.

2 Problem Statement

In this section, we formally state the scheduling and allocation problem that we want to solve. We provide a complete list of all our hypotheses and discuss each in turn.

2.1 Hypotheses

(H1) The computation domain (or iteration space) is a two-dimensional rectangle[3] of size $N_1 \times N_2$. Tiles are rectangular, and their edges are parallel to the axes (see Figure 1). All tiles have the same fixed size. Tiles are indexed as $T_{i,j}$, $0 \leq i < N_1$, $0 \leq j < N_2$.

(H2) Dependences between tiles are summarized by the vector pair

$$\left\{ \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix} \right\}.$$

In other words, the computation of a tile cannot be started before both its left and lower neighbor tiles have been executed. Given a tile $T_{i,j}$, we call both tiles $T_{i+1,j}$ and $T_{i,j+1}$ its successors, whenever the indices make sense.

(H3) There are P available processors interconnected as a (virtual) ring.[4] Processors are numbered from 0 to $P-1$. Processors may have different speeds: let t_q be the time needed by processor P_q to execute a tile, for $0 \leq q < P$. While we assume the computing resources are heterogeneous, we assume the communication network is homogeneous: if two adjacent tiles T and T' are

[3]In fact, the dimension of the tiles may be greater than 2. Most of our heuristics use a columnwise allocation, which means that we partition a single dimension of the iteration space into chunks to be allocated to processors. The number of remaining dimensions is not important.

[4]The actual underlying physical communication network is not important.

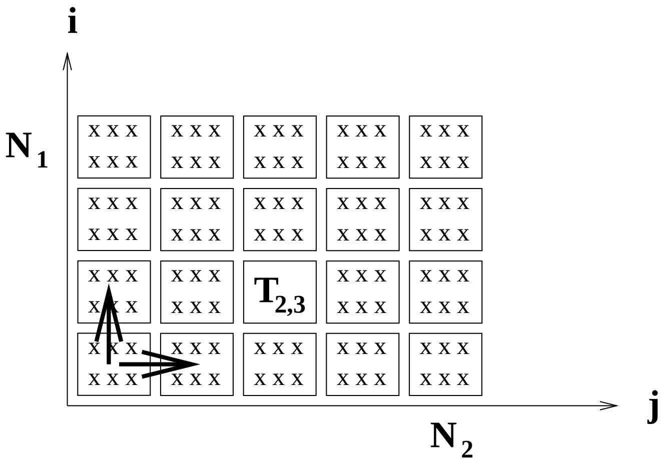


Figure 1: A tiled iteration space with horizontal and vertical dependences.

not assigned to the same processor, we pay the same communication overhead T_{com}, whatever the processors that execute T and T'.

(H4) Tiles are assigned to processors by using a scheduling σ and an allocation function *proc* (both to be determined). Tile T is allocated to processor proc(T), and its execution begins at time-step $\sigma(T)$. The constraints[5] induced by the dependences are the following: for each tile T and each of its successors T', we have

$$\begin{cases} \sigma(T) + t_{\text{proc}(T)} \leq \sigma(T') & \text{if proc}(T) = \text{proc}(T') \\ \sigma(T) + t_{\text{proc}(T)} + T_{\text{com}} \leq \sigma(T') & \text{otherwise} \end{cases}$$

The makespan $MS(\sigma, \text{proc})$ of a schedule-allocation pair (σ, proc) is the total execution time required to execute all tiles. If execution of the first tile $T_{0,0}$ starts at time-step $t = 0$, the makespan is equal to the date at which the execution of the last tile is executed:

$$MS(\sigma, \text{proc}) = \sigma(T_{N_1,N_2}) + t_{\text{proc}(T_{N_1,N_2})}.$$

A schedule-allocation pair is said to be optimal if its makespan is the smallest possible over all (valid) solutions. Let T_{opt} denote the optimal execution time over all possible solutions.

2.2 Discussion

We survey our hypotheses and assess their motivations, as well as the limitations that they may induce.

Rectangular iteration space and tiles. We note that the tiled iteration space is the outcome of previous program transformations, as explained in [22, 21, 6]. The first step in tiling amounts to determining the best shape and size of the tiles, assuming an infinite grid of virtual processors. Because this step will lead to tiles whose edges are parallel to extremal dependence vectors, we can perform a unimodular transformation and rewrite the original loop nest along the edge axes. The resulting domain may not be rectangular, but we can

[5]There are other constraints to express (e.g., any processor can execute at most one tile at each time-step). See Section 2.3 for a complete formalization.

approximate it using the smallest bounding box (however, this approximation may impact the accuracy of our results).

Dependence vectors. We assume that dependences are summarized by the vector pair $\mathcal{V} = \{(1,0)^t, (0,1)^t\}$. Note that these are dependences between tiles, not between elementary computations. Hence, having such dependences is a very general situation if the tiles are large enough. Technically, since we deal with a set of fully permutable loops, all dependence vectors have nonnegative components only, so that $\mathcal{V}$ permits all other dependence vectors to be generated by transitivity. Note that having a dependence vector $(0,a)^t$ with $a \geq 2$ between tiles, *instead of* having vector $(0,1)^t$, would mean unusually long dependences in the original loop nest, while having $(0,a)^t$ *in addition to* $(0,1)^t$ as a dependence vector between tiles is simply redundant. In practical situations, we might have an additional diagonal dependence vector $(1,1)^t$ between tiles, but the diagonal communication may be routed horizontally and then vertically, or the other way round, and even may be combined with any of the other two messages (because of vectors $(0,1)^t$ and $(1,0)^t$).

Computation-communication overlap. Note that in our model, communications can be overlapped with the computations of other (independent) tiles. Assuming communication-computation overlap seems a reasonable hypothesis for current machines that have communication coprocessors and allow for asynchronous communications (posting instructions ahead, or using active messages). We can think of independent computations going along a thread while communication is initiated and performed by another thread [20]. An interesting approach has been proposed by Andonov and Rajopadhye [4]: they introduce the *tile period* P_t as the time elapsed between corresponding instructions of two successive tiles that are mapped to the *same* processor, while they define the *tile latency* L_t to be the time between corresponding instructions of two successive tiles that are mapped to *different* processors. The power of this approach is that the expressions for L_t and P_t can be modified to take into account several architectural models. A detailed architectural model is presented in [4], and several other models are explored in [3]. With our notation, $P_t = t_i$ and $L_t = t_i + T_{\text{com}}$ for processor P_i.

Homogeneous communication network. We assume that the communication time T_{com} for a tile is independent of the two processors exchanging the message. This is a crude simplification because the network interfaces of heterogeneous systems are likely to exhibit very different latency characteristics. However, because communications can be overlapped with independent computations, they eventually have little impact on the performance, as soon as the granularity (the tile size) is chosen large enough. This theoretical observation has been verified during our MPI experiments (see Section 4.3).

Finally, we briefly mention another possibility for introducing heterogeneity into the tiling model. We chose to have all tiles of same size and to allocate more tiles to the faster processors. Another possibility is to evenly distribute tiles to processors, but to let their size vary according to the speed of the processor they are allocated to. However, this strategy would severely complicate code generation. Also, allocating several neighboring fixed-size tiles to the same processor will have similar effects as allocating variable-size tiles, so our approach will cause no loss of generality.

2.3 ILP Formulation

We can describe the tiled iteration space as a task graph $G = (V, E)$, where vertices represent the tiles and edges represent dependences between tiles. Computing an optimal schedule-allocation pair is a well-known task graph scheduling problem, which is NP-complete in the general case [11].

If we want to solve the problem as stated (hypotheses (H1) to (H4)), we can use an integer linear programming formulation. Several constraints must be satisfied by any valid schedule-allocation pair. In the following, T_{max} denotes an upper bound on the total execution time. For example, T_{max} can be the execution time when all the tiles are given to the fastest processor: $T_{max} = N_1 \times N_2 \times \min_{0 \le i < P} t_i$ (here, the t_i's are integral multiples of the unit time step).

We now translate these constraints into equations. In the following, let $i \in \{1, \ldots, N_1\}$ denote a row number, $j \in \{1, \ldots, N_2\}$ a column number, $q \in \{0, \ldots, P-1\}$ a processor number, and $t \in \{0, \ldots, T_{max}\}$ a time-step.

- **Number of executions**. Let $B_{i,j,q,t}$ be an integer variable indicating whether the execution of tile $T_{i,j}$ begins at time-step t on processor q: if this is the case, then $B_{i,j,q,t} = 1$, and $B_{i,j,q,t} = 0$ otherwise. Each tile must be executed once, and thus starts at one and only one time-step. Therefore, the constraints are

$$\forall i, j, q, t, \quad B_{i,j,q,t} \ge 0 \quad \text{and} \quad \forall i, j, \quad \sum_{q=0}^{P-1} \sum_{t=0}^{T_{max}} B_{i,j,q,t} = 1.$$

- **Execution place and date**. Using $B_{i,j,q,t}$, we can compute the date $D_{i,j}$ at which tile (i,j) starts execution. We can also check which processor q processes tile (i,j). The 0/1 result is stored in $P_{i,j,q}$:

$$\forall i, j, \quad D_{i,j} = \sum_{q=0}^{P-1} \sum_{t=0}^{T_{max}} t \times B_{i,j,q,t} \quad \text{and} \quad \forall i, j, q, \quad P_{i,j,q} = \sum_{t=0}^{T_{max}} B_{i,j,q,t}.$$

- **Communications**. There must be a communication delay between the end of execution of tile $(i-1, j)$ (resp. $(i, j-1)$) and the beginning of execution of tile (i,j) if and only if the two tiles are not executed by the same processor, that is, if and only if there exists q such that $P_{i,j,q} \neq P_{i-1,j,q}$ (resp. $P_{i,j,q} \neq P_{i,j-1,q}$). The boolean result is stored in $v_{i,j}$ (resp. $h_{i,j}$): $v_{i,j} = 1$ if tiles $(i-1, j)$ and (i,j) are not executed by the same processor, and $v_{i,j} = 0$ otherwise. We have a similar definition for $h_{i,j}$ using tiles $(i, j-1)$ and (i,j). The equations are:

$$\forall i \ge 2, j, q, \quad v_{i,j} \ge P_{i,j,q} - P_{i-1,j,q}, \quad v_{i,j} \ge P_{i-1,j,q} - P_{i,j,q}$$
$$\forall i, j \ge 2, q, \quad h_{i,j} \ge P_{i,j,q} - P_{i,j-1,q}, \quad v_{i,j} \ge P_{i,j-1,q} - P_{i,j,q}$$

  Note that if a communication delay is needed between the execution of tile $(i-1, j)$ and that of tile (i,j), then $v_{i,j}$ will impose one. If none is needed, $v_{i,j}$ may still be equal to 1, as long as this does not increase the total execution time.

- **Precedence constraints**. The execution of tile $(i-1, j)$ (resp. $(i, j-1)$) must be finished, and the data transferred, before the beginning of execution of tile (i,j):

$$\forall i \ge 2, j, \quad D_{i,j} \ge D_{i-1,j} + v_{i,j} T_{com} + \sum_{q=0}^{P-1} P_{i-1,j,q}\, t_q$$

$$\forall i, j \ge 2, \quad D_{i,j} \ge D_{i,j-1} + h_{i,j} T_{com} + \sum_{q=0}^{P-1} P_{i,j-1,q}\, t_q$$

$$
\begin{cases}
\min\left(D_{N_1,N_2} + \sum_q P_{N_1,N_2,q}\ t_q\right) & \\
 & \\
\sum_{t'=t-t_q+1}^{t} \sum_{i,j} B_{i,j,q,t'} \leq 1 & 0 \leq q \leq P-1, t_q - 1 \leq t \leq T_{max} \\
D_{i,j} \geq D_{i-1,j} + v_{i,j} T_{com} + \sum_q P_{i-1,j,q} t_q & 2 \leq i \leq N_1, 1 \leq j \leq N_2 \\
D_{i,j} \geq D_{i,j-1} + h_{i,j} T_{com} + \sum_q P_{i,j-1,q} t_q & 1 \leq i \leq N_1, 2 \leq j \leq N_2 \\
v_{i,j} \geq P_{i,j,q} - P_{i-1,j,q} & 2 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1 \\
v_{i,j} \geq P_{i-1,j,q} - P_{i,j,q} & 2 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1 \\
h_{i,j} \geq P_{i,j,q} - P_{i,j-1,q} & 1 \leq i \leq N_1, 2 \leq j \leq N_2, 0 \leq q \leq P-1 \\
h_{i,j} \geq P_{i,j-1,q} - P_{i,j,q} & 1 \leq i \leq N_1, 2 \leq j \leq N_2, 0 \leq q \leq P-1 \\
P_{i,j,q} = \sum_t B_{i,j,q,t} & 1 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1 \\
D_{i,j} = \sum_q \sum_t t B_{i,j,q,t} & 1 \leq i \leq N_1, 1 \leq j \leq N_2 \\
\sum_q \sum_t B_{i,j,q,t} = 1 & 1 \leq i \leq N_1, 1 \leq j \leq N_2 \\
B_{i,j,q,t} \geq 0 & 1 \leq i \leq N_1, 1 \leq j \leq N_2, 0 \leq q \leq P-1, 0 \leq t \leq T_{max}
\end{cases}
$$

Figure 2: Integer linear program that optimally solves the schedule-allocation problem.

- **Number of tiles executed at any time-step.** A processor executes (at most) one tile at a time. Therefore processor q can start executing at most one tile in any interval of time t_q (as t_q is the time to execute a tile by processor q):

$$\forall q, \quad t_q - 1 \leq t \leq T_{max}, \qquad \sum_{t'=t-t_q+1}^{t} \sum_{i=1}^{N_1} \sum_{j=1}^{N_2} B_{i,j,q,t'} \leq 1$$

Now that we have expressed all our constraints in a linear way, we can write the whole linear programming system. We need only to add the objective function: the minimization of the time-step at which the execution of the last tile T_{N_1,N_2} is terminated. The final linear program is presented in Figure 2. Since an optimal rational solution of this problem is not always an integer solution, this program must be solved as an *integer* linear program.

The main drawback of the linear programming approach is its huge cost. The program shown on Figure 2 contains more than $PN_1N_2T_{max}$ variables and inequalities. The cost of solving such a problem would be prohibitive for any practical application. Furthermore, even if we could solve the linear problem, we might not be pleased with the solution. We probably would prefer non-optimal but "regular" allocations of tiles to processors, such as columnwise or rowwise allocations. Fortunately, such allocations can lead to asymptotically optimal solutions, as shown in the next section.

3 Columnwise Allocation

In this section we present theoretical results on columnwise allocations. In the next section we will use these results to derive practical heuristics. Before introducing an asymptotically optimal columnwise (or rowwise) allocation, we give a small example to show that columnwise allocations (or equivalently rowwise allocations) are not optimal.

3.1 Optimality and Columnwise Allocations

Consider a tiled iteration space with $N_2 = 2$ columns, and suppose we have $P = 2$ processors such that $t_1 = 5 \times t_0$: the first processor is five times faster than the second one. Suppose for the sake of simplicity that $T_{\text{com}} = 0$. If we use a columnwise allocation,

- either we allocate both columns to processor 0, and the makespan is $MS = 2N_1t_0$,
- or we allocate one column to each processor, and the makespan is greater than N_1t_1 (a lower bound for the slow processor to process its column).

The best solution is then to have the fast processor execute all tiles. But if N_1 is large enough, we can do better by allocating a small fraction of the first column (the last tiles) to the slow processor, which will process them while the first processor is active executing the first tiles of the second column. For instance, if $N_1 = 6n$ and if we allocate the last n tiles of the first column to the slow processor (see Figure 3), the execution time becomes $MS = 11nt_0 = \frac{11}{6}N_1t_0$, which is better than the best columnwise allocation.[6]

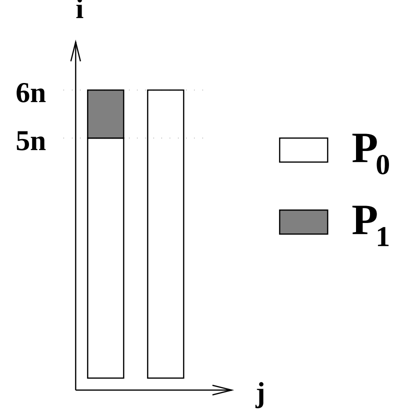


Figure 3: Allocating tiles for a two-column iteration space.

This small example shows that our target problem is intrinsically more complex than the instance with same-speed processors: as shown in [8], a columnwise allocation would be optimal for our two-column iteration space with two processors of equal speed.

3.2 Heuristic Allocation by Block of Columns

Throughout the rest of the paper we make the following additional hypothesis:

(H5) We impose the allocation to be columnwise:[7] for a given value of j, all tiles $T_{i,j}$, $1 \le i \le N_1$, are allocated to the same processor.

We start with an easy lemma to bound the optimal execution time T_{opt}:

Lemma 1

$$T_{opt} \ge \frac{N_1 \times N_2}{\sum_{i=0}^{P-1} \frac{1}{t_i}}.$$

[6] This is not the best possible allocation, but it is superior to any columnwise allocation.

[7] Note that the problem is symmetric in rows and columns. We could study rowwise allocations as well.

Proof Let x_i be the number of tiles allocated to processor i, $0 \leq i < P$. Obviously, $\sum_{i=0}^{P-1} x_i = N_1N_2$. Even if we take into account neither the communication delays nor the dependence constraints, the execution time T is greater than the computation time of each processor: $T \geq x_i t_i$ for all $0 \leq i < P$. Rewriting this as $x_i \leq T/t_i$ and summing over i, we get $N_1N_2 = \sum_{i=0}^{P-1} x_i \leq (\sum_{i=0}^{P-1} \frac{1}{t_i})T$, hence the result. ■

The proof of Lemma 1 leads to the (intuitive) idea that tiles should be allocated to processors *in proportion to* their relative speeds, so as to balance the workload. Specifically, let $L = \text{lcm}(t_0, t_1, \dots, t_{P-1})$, and consider an iteration space with L columns: if we allocate $\frac{L}{t_i}$ tile columns to processor i, all processors need the same number of time-steps to compute all their tiles: the workload is perfectly balanced. Of course, we must find a good schedule so that processors do not remain idle, waiting for other processors because of dependence constraints.

We introduce below a heuristic that allocates the tiles to processors by blocks of columns whose size is computed according to the previous discussion. This heuristic produces an asymptotically optimal allocation: the ratio of its makespan over the optimal execution time tends to 1 as the number of tiles (the domain size) increases.

In a columnwise allocation, all the tiles of a given column of the iteration space are allocated to the same processor. When contiguous columns are allocated to the same processor, they form a block. When a processor is assigned several blocks, the scheduling is the following:

1. Blocks are computed one after the other, in the order defined by the dependences. The computation of the current block must be completed before the next block is started.

2. The tiles inside a given block are computed in a rowwise order: if, say, 3 consecutive columns are assigned to a processor, it will execute the three tiles in the first row, then the three tiles in the second row, and so on. Note that (given 1.) this strategy is the best to minimize the latency (for another processor to start next block as soon as possible).

The following lemma shows that dependence constraints do not slow down the execution of two consecutive blocks (of adequate size) by two different-speed processors:

Lemma 2 *Let P_1 and P_2 be two processors that execute a tile in time t_1 and t_2, respectively. Assume that P_1 was allocated a block B_1 of c_1 contiguous columns and that P_2 was allocated the block B_2 consisting of the following c_2 columns. Let c_1 and c_2 satisfy the equality $c_1t_1 = c_2t_2$.*

Assume that P_1, starting at time-step s_1, is able to process B_1 without having to wait for any tile to be computed by some other processor. Then P_2 will be able to process B_2 without having to wait for any tile computed by P_1, if it starts at time $s_2 \geq s_1 + c_1t_1 + T_{com}$.

Proof P_1 (resp. P_2) executes its block row by row. The execution time of a row is c_1t_1 (resp. c_2t_2). By hypothesis, it takes the same amount of time for P_1 to compute a row of B_1 than for P_2 to compute a row of B_2. Since P_1 is able to process B_1 without having to wait for any tile to be computed by some other processor, it finishes computing the ith row of B_1 at time $s_1 + ic_1t_1$.

P_2 cannot start processing the first tile of the ith row of B_2 before P_1 has computed the last tile of the ith row of B_1 and has sent that data to P_2, that is, at time-step $s_1 + ic_1t_1 + T_{\text{com}}$. Since P_2 starts processing the first row of B_2 at time s_2, where $s_2 \geq s_1 + c_1t_1 + T_{\text{com}}$, it is not delayed by P_1. Later on, P_2 will process the first tile of the ith row of B_2 at time $s_2 + (i-1)c_2t_2 = s_2 + (i-1)c_1t_1 \geq s_1 + c_1t_1 + T_{\text{com}} + (i-1)c_1t_1 = s_1 + ic_1t_1 + T_{\text{com}}$; hence P_2 will not be delayed by P_1. ■

We are ready to introduce our heuristic.

Heuristic

Let $P_0, \ldots, P_{P-1}$ be P processors that respectively execute a tile in time $t_0, \ldots, t_{P-1}$. We allocate column blocks to processors by chunks of $C = L \times \sum_{i=0}^{P-1} \frac{1}{t_i}$, where $L = \mathrm{lcm}(t_0, t_1, \ldots, t_{P-1})$ columns. For the first chunk, we assign the block B_0 of the first L/t_0 columns to P_0, the block B_1 of the next L/t_1 columns to P_1, and so on until P_{p-1} receives the last L/t_p columns of the chunk. We repeat the same scheme with the second chunk (columns $C+1$ to $2C$) first, and so on until all columns are allocated (note that the last chunk may be incomplete). As already said, processors will execute blocks one after the other, row by row within each block.

Lemma 3 *The difference between the execution time of the heuristic allocation by columns and the optimal execution time is bounded by*

$$T - T_{opt} \leq (P-1)T_{com} + (N_1 + P - 1)lcm(t_0, t_1, \ldots, t_{P-1}).$$

Proof Let $L = \mathrm{lcm}(t_0, t_1, \ldots, t_{P-1})$. Lemma 2 ensures that, if processor P_i starts working at time-step $s_i = i(L + T_{\mathrm{com}})$, it will not be delayed by other processors. By definition, each processor executes one block in time LN_1. The maximal number of blocks allocated to a processor is

$$n = \left\lceil \frac{N_2}{L \times \sum_{i=0}^{P-1} \frac{1}{t_i}} \right\rceil.$$

The total execution time, T, is equal to the date the last processor terminates execution. T can be bounded as follows:[8]

$$T \leq s_P + n \times LN_1.$$

On the other hand, T_{opt} is lower bounded by Lemma 1. We derive

$$T - T_{opt} \leq (P-1)(L + T_{\mathrm{com}}) + LN_1 \left\lceil \frac{N_2}{L \times \sum_{i=0}^{P-1} \frac{1}{t_i}} \right\rceil - \frac{N_1 \times N_2}{\sum_{i=0}^{P-1} \frac{1}{t_i}}.$$

Since $\lceil x \rceil \leq x + 1$ for any rational number x, we obtain the desired formula. ■

Proposition 1 *Our heuristic is asymptotically optimal: letting T be its makespan, and T_{opt} be the optimal execution time, we have*

$$\lim_{N_2 \to +\infty} \frac{T}{T_{opt}} = 1.$$

The two main advantages of our heuristic are (i) its regularity, which leads to an easy implementation; and (ii) its guarantee: it is theoretically proved to be close to the optimal. However, we will need to adapt it to deal with practical cases, because the number $C = L \times \sum_{i=0}^{P-1} \frac{1}{t_i}$ of columns in a chunk may be too large.

[8]Processor P_{P-1} is not necessarily the last one, because the last chunk may be incomplete.

4 Practical Heuristics

In the preceding section, we described a heuristic that allocates blocks of columns to processors in a cyclic fashion. The size of the blocks is related to the relative speed of the processors and can be huge in practice. Therefore, a straightforward application of our heuristic would lead to serious difficulties, as shown next in Section 4.1. Furthermore, the execution time variables t_i are not known accurately in practice. We explain how to modify the heuristic (computing different block sizes) in Section 4.2.

4.1 Processor Speed

To expose the potential difficulties of the heuristic, we conducted experiments on a heterogeneous network of eight Sun workstations. To compute the relative speed of each workstation, we used a program that runs the same piece of computation that will be used later in the tiling program. Results are reported in Table 1.

Name	nala	bluegrass	dancer	donner	vixen	rudolph	zazu	simba
Description	Ultra 2	SS 20	SS 5	SS 5	SS 5	SS 10	SS1 4/60	SS1 4/60
Execution time t_i	11	26	33	33	38	40	528	530

Table 1: Measured computation times showing relative processor speeds.

To use our heuristic, we must allocate chunks of size $C = L \sum_{i=0}^{7} \frac{1}{t_i}$ columns, where $L = \mathrm{lcm}(t_0, t_1, \ldots, t_7) = 34{,}560{,}240$. We compute that $C = 8{,}469{,}789$ columns, which would require a very large problem size indeed. Needless to say, such a large chunk is not feasible in practice. Also, our measurements for the processor speeds may not be accurate,[9] and a slight change may dramatically impact the value of C. Hence, we must devise another method to compute the sizes of the blocks allocated to each processor (see Section 4.2). In Section 4.3, we present simulation results and discuss the practical validity of our modified heuristics.

4.2 Modified Heuristic

Our goal is to choose the "best" block sizes allocated to each processor while bounding the total size of a chunk. We first define the cost of a block allocation and then describe an algorithm to compute the best possible allocation, given an upper bound for the chunk size.

4.2.1 Cost Function

As before, we consider heuristics that allocate tiles to processors by blocks of columns, repeating the chunk in a cyclic fashion. Consider a heuristic defined by $\mathcal{C} = (c_0, \ldots, c_{P-1})$, where c_i is the number of columns in each block allocated to processor P_i:

Definition 1 *The cost of a block size allocation $\mathcal{C}$ is the maximum of the block computation times ($c_i t_i$) divided by the total number of columns computed in each chunk:*

$$cost(\mathcal{C}) = \frac{\max_{0 \le i \le P-1} c_i t_i}{\sum_{0 \le i \le P-1} c_i}$$

[9]The 8 workstations were not dedicated to our experiments. Even though we were running these experiments during the night, some other users' processes might have been running. Also, we have averaged and rounded the results, so the error margin roughly lies between 5% and 10%.

Considering the steady state of the computation, all processors work in parallel inside their blocks, so that the computation time of a whole chunk is the maximum of the computation times of the processors. During this time, $s = \sum_{0 \leq i \leq P-1} c_i$ columns are computed. Hence, the average time to compute a single column is given by our cost function. When the number of columns is much larger than the size of the chunk, the total computation time can well be approximated by $cost(\mathcal{C}) \times N_2$, the product of the average time to compute a column by the total number of columns.

4.2.2 Optimal Block Size Allocations

As noted before, our cost function correctly models reality when the number of columns in each chunk is much smaller than the total number of columns of the domain. We now describe an algorithm that returns the best (with respect to the cost function) block size allocation given a bound s on the number of columns in a chunk.

We build a function that, given a best allocation with a chunk size equal to $n-1$, computes a best allocation with a chunk size equal to n. Once we have this function, we start with an initial chunk size $n = 0$, compute a best allocation for each increasing value of n up to $n = s$, and select the best allocation encountered so far.

First we characterize the best allocations for a given chunk size s:

Lemma 4 *Let $\mathcal{C} = (c_0, \ldots, c_{P-1})$ be an allocation, and let $s = \sum_{0 \leq i \leq P-1} c_i$ be the chunk size. Let $m = \max_{0 \leq i \leq P-1} c_i t_i$ denote the maximum computation time inside a chunk.*

If $\mathcal{C}$ verifies

$$\forall i, 0 \leq i \leq P-1,\ t_i c_i \leq m \leq t_i(c_i + 1), \tag{1}$$

then it is optimal for the chunk size s.

Proof Take an allocation verifying the above Condition 1. Suppose that it is not optimal. Then there exists a better allocation $\mathcal{C}' = (c'_0, \ldots, c'_{P-1})$ with $\sum_{0 \leq i \leq P-1} c'_i = s$, such that

$$m' = \max_{0 \leq i \leq P-1} c'_i t_i < m.$$

By definition of m, there exists i_0 such that $m = c_{i_0} t_{i_0}$. We can then successively derive

$$\begin{aligned}
c_{i_0} t_{i_0} &= m > m' \geq c'_{i_0} t_{i_0} \\
c_{i_0} &> c'_{i_0} \\
\exists i_1, c_{i_1} &< c'_{i_1} \quad \left(\text{because} \sum_{0 \leq i \leq P-1} c_i = s = \sum_{0 \leq i \leq P-1} c'_i\right) \\
c_{i_1} + 1 &\leq c'_{i_1} \\
t_{i_1}(c_{i_1} + 1) &\leq t_{i_1} c'_{i_1} \\
m &\leq m' \quad (\text{by definition of } m \text{ and } m')
\end{aligned}$$

which contradicts the non-optimality of the original allocation. ■

There remains to build allocations satisfying Condition (1). The following algorithm gives the answer:

- For the chunk size $s = 0$, take the optimal allocation $(0, 0, \ldots, 0)$.

- To derive an allocation $\mathcal{C}'$ verifying equation (1) with chunk size s from an allocation $\mathcal{C}$ verifying (1) with chunk size $s-1$, add 1 to a well-chosen c_j, one that verifies

$$t_j(c_j+1) = \min_{0\leq i\leq P-1} t_i(c_i+1). \tag{2}$$

  In other words, let $c'_i = c_i$ for $0 \leq i \leq P-1, i \neq j$, and $c'_j = c_j + 1$.

Lemma 5 *This algorithm is correct.*

Proof We have to prove that allocation $\mathcal{C}'$, given by the algorithm, verifies Equation (1).

Since allocation $\mathcal{C}$ verifies equation (1), we have $t_i c_i \leq m \leq t_j(c_j+1)$. By definition of j from Equation (2), we have

$$m' = \max_{0\leq i\leq P-1} t_i c'_i = \max\left(t_j(c_j+1), \max_{1\leq i\leq q, i\neq j} t_i c_i\right) = t_j c'_j.$$

We then have $t_j c'_j \leq m' \leq t_j(c'_j+1)$ and

$$\begin{aligned}&\forall i \neq j, 1 \leq i \leq q,\\ &\mathbf{t_i c'_i} = t_i c_i \leq m \leq \mathbf{m'} \leq t_j c'_j = \min_{0\leq i\leq P-1} t_i(c_i+1) \leq t_i(c_i+1) = \mathbf{t_i(c'_i+1)},\end{aligned}$$

so the resulting allocation does verify Equation (1). ■

To summarize, we have built an algorithm to compute "good" block sizes for the heuristic allocation by blocks of columns. Once an upper bound for the chunk size has been selected, our algorithm returns the best block sizes, according to our *cost* function, with respect to this bound.

The complexity of this algorithm is $O(Ps)$, where P is the number of processors and s, the upper bound on the chunk size. Indeed, the algorithm consists of s steps where one computes a minimum over the processors. This low complexity allows us to perform the computation of the best allocation at runtime.

A Small Example. To understand how the algorithm works, we present a small example with $P = 3$, $t_0 = 3$, $t_1 = 5$, and $t_2 = 8$. In Table 2, we report the best allocations found by the algorithm up to $s = 7$. The entry "Selected j" denotes the value of j that is chosen to build the next allocation. Note that the cost of the allocations is not a decreasing function of s. If we allow chunks of size not greater than 7, the best solution is obtained with the chunk $(3, 2, 1)$ of size 6.

Finally, we point out that our modified heuristic "converges" to the original asymptotically optimal heuristic. For a chunk of size $C = L \times \sum_{i=0}^{P-1} \frac{1}{t_i}$, where $L = \text{lcm}(t_0, t_1, \ldots, t_{P-1})$ columns, we obtain the optimal cost

$$cost_{opt} = \frac{L}{C} = \left(\sum_{0\leq i\leq P-1} \frac{1}{t_i}\right)^{-1},$$

which is the inverse of the harmonic mean of the execution times divided by the number of processors.

Chunk Size	c_0	c_1	c_2	Cost	Selected j
0	0	0	0		0
1	1	0	0	3	1
2	1	1	0	2.5	0
3	2	1	0	2	2
4	2	1	1	2	0
5	3	1	1	1.8	1
6	3	2	1	1.67	0
7	4	2	1	1.71	

Table 2: Running the algorithm with 3 processors: $t_0 = 3$, $t_1 = 5$, and $t_2 = 8$.

4.2.3 Choosing the chunk size

Choosing a chunk size s is not easy. A possible approach is to slice the total work into phases. We use small-scale experiments to compute a first estimation of the t_i, and we allocate the first chunk of s columns according to these values for the first phase. During the first phase we measure the actual performance of each machine. At the end of the phase we collect the new values of the t_i, and we use these values to allocate the second chunk during the second phase, and so on. Of course a phase must be long enough, say a couple of seconds, so that the overhead due to the communication at the end of each phase is negligible. Hence the size s of the chunk is chosen by the user as a trade-off: the larger s, the more even the predicted load, but the longer the delay to account for variations in processor speeds.

4.2.4 Remark on the Multidimensional Approximation Problem

Our algorithm is related to the multidimensional approximation problem where one wants to approximate some real numbers with rationals sharing the same denominator. Many algorithms exist to solve this problem (see [7], for example), but these algorithms focus on finding a "best approximation" with respect to the real numbers while we want "good" approximations made up with small numbers.

4.3 MPI Experiments

We report several experiments on the network of workstations presented in Section 4.1. After comments on the experiments, we focus on cyclic and block-cyclic allocations and then on our modified heuristics.

4.3.1 General Remarks

We study different columnwise allocations on the heterogeneous network of workstations presented in Section 4.1. Our simulation program is written in C using the MPI library for communication. It is not an actual tiling program, but it simulates such behavior: we have not inserted the code required to deal with the boundaries of the computation domain. Actually, our code only simulates the communications generated by a tiling, it does fake computations (hence, no data allocation). The tiling is assumed given. Our aim is not to find the "best" tiling. The tile domain has 100 rows and a number of columns varying from 200 to 1000 by steps of 100. An array of doubles of size the square root of the tile area is communicated for each communication (we assume here that the computation volume is proportional to the tile area while the communication volume is proportional to its square root).

The actual communication network is a coax type Ethernet network. It can be considered as a bus, not as a point-to-point connection ring; hence our model for communication is not fully correct. However, this configuration has little impact on the results, which correspond well to the theoretical conditions.

As already pointed out, the workstations we use are multiple-user workstations. Although our simulations were made at times when the workstations were not supposed to be used by anybody else, the load may vary. The timings reported in the figures are the average of several measures from which aberrant data have been suppressed.

In Figures 4 and 6, we show for reference the sequential time as measured on the fastest machine, namely, "nala".

4.3.2 Cyclic Allocations

We have experimented with cyclic allocations on the 6 fastest machines, on the 7 fastest machines, and on all 8 machines. Because cyclic allocation is optimal when all processors have the same speed, this will be a reference for other simulations. We have also tested a block cyclic allocation with block size equal to 10, in order to see whether the reduced amount of communication helps. Figure 4 presents the results[10] for these 6 allocations (3 purely cyclic allocations using 6, 7, and 8 machines, and 3 block-cyclic allocations).

We comment on the results of Figure 4 as follows:

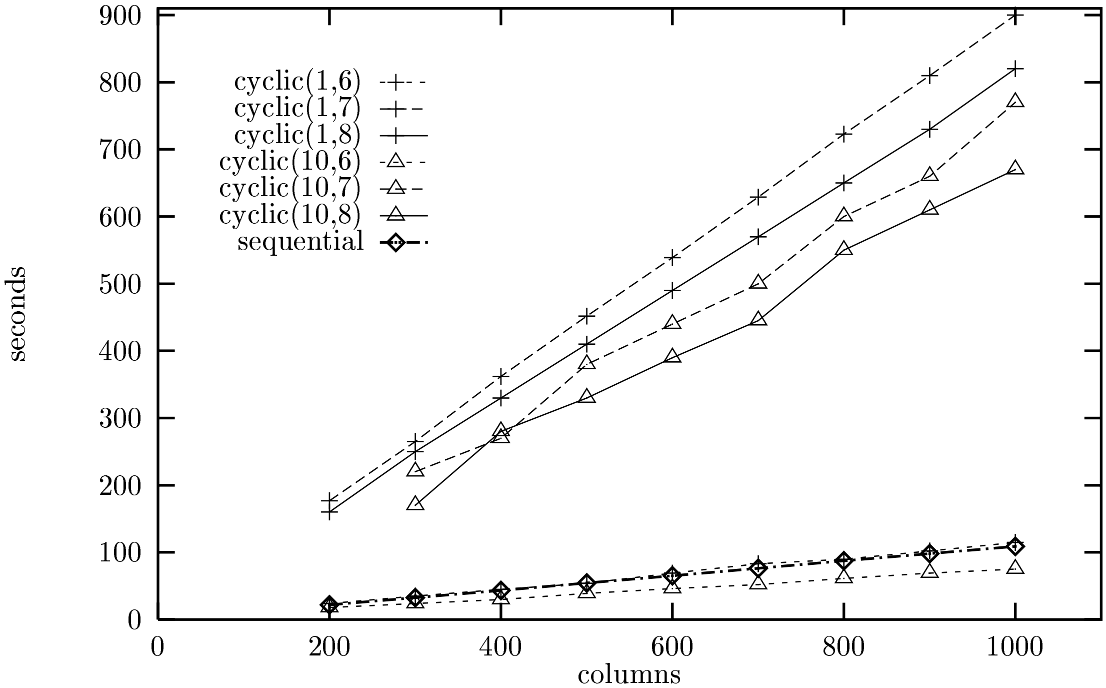
- With the same number of machines, a block size of 10 is better than a block size of 1 (pure cyclic).
- With the same block size, adding a single slow machine is disastrous, and adding the second one only slightly improve the disastrous performances.
- Overall, only the block cyclic allocation with block size 10 and using the 6 fastest machines gives some speedup over the sequential execution.

We conclude that cyclic allocations are not efficient when the computing speeds of the available machines are very different. For the sake of completeness, we show in Figure 5 the execution times obtained for the same domain (100 rows and 1000 columns) and the 6 fastest machines, for block cyclic allocations with different block sizes. We see that the block-size as a small impact on the performances, which corresponds well to the theory: all cyclic allocations have the same cost.

We point out that cyclic allocations would be the outcome of a greedy master-slave strategy. Indeed, processors will be allocated the first P columns in any order. Re-number processors according to this initial assignment. Then throughout the computation, P_j will return after P_{j-1} and just before P_{j+1} (take indices modulo p), because of the dependences. Hence computations would only progress at the speed of the slowest processor, with a cost $\frac{\max t_p}{P}$.

4.3.3 Using our modified heuristic

Let us now consider our heuristics. In Table 3, we show the block sizes computed by the algorithm described in Section 4.2 for different upper bounds of the chunk size. The best allocation computed with bound u is denoted as $\mathcal{C}_u$.



Remark cyclic(b,m) corresponds to a block cyclic allocation with block size b, using the m fastest machines of Table 1.

Figure 4: Experimenting with cyclic and block-cyclic allocations.

	nala	bluegrass	dancer	donner	vixen	rudolph	zazu	simba	*cost*	chunk
$\mathcal{C}_{25}$	7	3	2	2	2	2	0	0	4.44	18
$\mathcal{C}_{50}$	15	6	5	5	4	4	0	0	4.23	39
$\mathcal{C}_{100}$	33	14	11	11	9	9	0	0	4.18	87
$\mathcal{C}_{150}$	52	22	17	17	15	14	1	1	4.12	139

Table 3: Block sizes for different chunk size bounds.

The time needed to compute these allocations is completely negligible with respect to the computation times (a few milliseconds versus several seconds).

Figure 6 presents the results for these allocations. Here are some comments:

- Each of the allocations computed by our heuristic is superior to the best block-cyclic allocation.
- The more precise the allocation, the better the results.
- For 1000 columns and allocation $\mathcal{C}_{150}$, we obtain a speedup of 2.2 (and 2.1 for allocation $\mathcal{C}_{50}$), which is very satisfying (see below).

The optimal cost for our workstation network is $cost_{opt} = \frac{L}{C} = \frac{34,560,240}{8,469,789} = 4.08$. Note that $cost(\mathcal{C}_{150}) = 4.12$ is very close to the optimal cost. The peak theoretical speedup is equal to

[10]Some results are not available for 200 columns because the chunk size is too large.

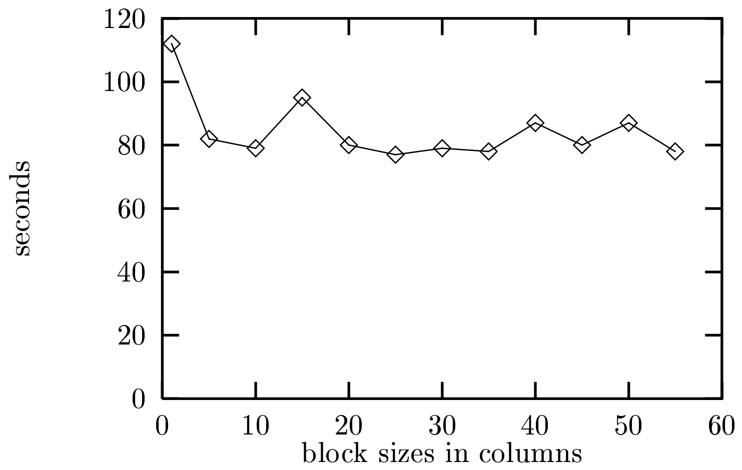


Figure 5: Cyclic allocations with different block sizes.

$\frac{\min_i t_i}{cost_{opt}} = 2.7$. For 1000 columns, we obtain a speedup equal to 2.2 for $\mathcal{C}_{150}$. This is satisfying considering that we have here only 7 chunks, so that side effects still play an important role. Note also that the peak theoretical speedup has been computed by neglecting all the dependences in the computation and all the communications overhead. Hence, obtaining a twofold speedup with 8 machines of very different speeds is not a bad result at all!

5 Conclusion

In this paper, we have extended tiling techniques to deal with heterogeneous computing platforms. Such platforms are likely to play an important role in the near future. We have introduced an asymptotically optimal columnwise allocation of tiles to processors. We have modified this heuristic to allocate column chunks of reasonable size, and we have reported successful experiments on a network of workstations. The practical significance of the modified heuristics should be emphasized: processor speeds may be inaccurately known, but allocating small but well-balanced chunks turns out to be quite successful: in practice we approach the peak theoretical speedup.

Heterogeneous platforms are ubiquitous in computer science departments and companies. The development of our new tiling techniques allows for the efficient use of older computational resources *in addition to* newer available systems.

The work presented in this paper is only a first step towards using heterogeneous systems. Heterogeneous networks of workstations or PCs represent the low end of the field of distributed and heterogeneous computing. At the high end of the field, linking the most powerful supercomputers of the largest supercomputing centers through dedicated high-speed networks will give rise to the most powerful computational science and engineering problem-solving environment ever assembled: the so-called *computational grid*. Providing desktop access to this "grid" will make computing routinely parallel, distributed, collaborative and immersive [15]. In the middle of the field, we can think of connecting medium-size parallel servers through fast but non-dedicated links. For instance, each institution could build its own specialized parallel machine equipped with application-specific databases and application-oriented software, thus creating a "meta-system". The user is then able to access all the machines of this meta-system remotely and transparently, without each institution duplicating the resources and the exploitation costs.

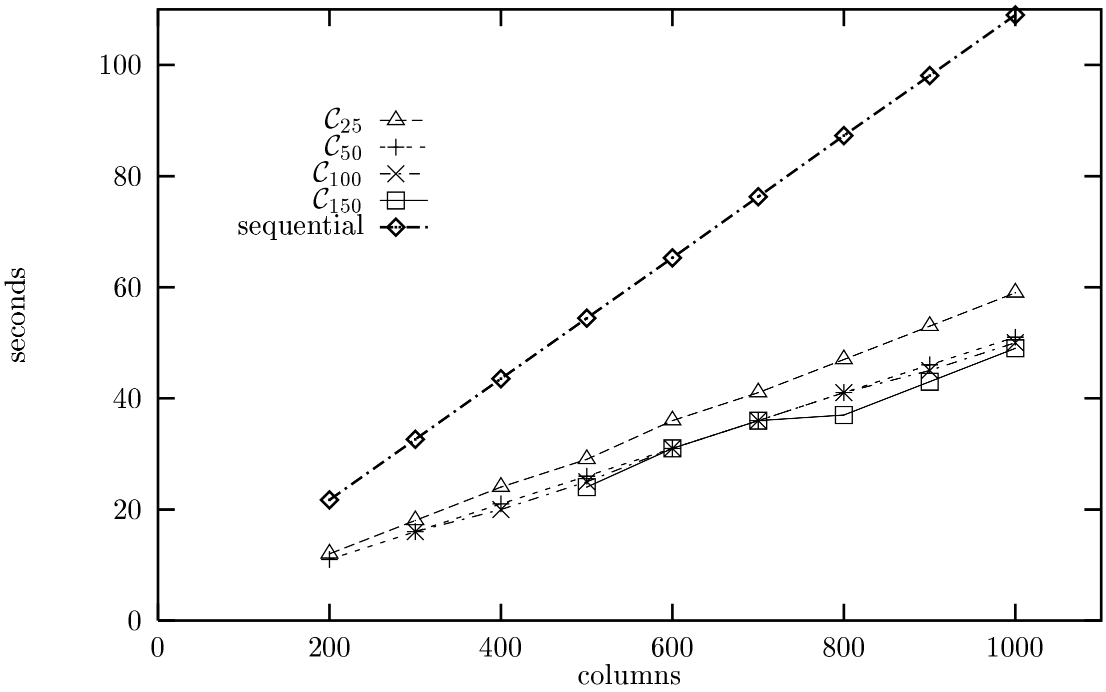


Figure 6: Experimenting with our modified heuristics.

Whereas the architectural vision is clear, the software developments are not so well understood. Lots of efforts in the area of building and operating meta-systems are targeted to infrastructure, services and applications. Not so many efforts are devoted to algorithm design and programming tools, while (we believe) they represent the major conceptual challenge to be tackled.

Acknowledgments

We thank the reviewers whose comments and suggestions have greatly improved the presentation of the paper.

References

[1] A. Agarwal, D.A. Kranz, and V. Natarajan. Automatic partitioning of parallel loops and data arrays for distributed shared-memory multiprocessors. *IEEE Trans. Parallel Distributed Systems*, 6(9):943–962, 1995.

[2] Stergios Anastasiadis and Kenneth C. Sevcik. Parallel application scheduling on networsk of workstations. *Journal of Parallel and Distributed Computing*, 43:109–124, 1997.

[3] Rumen Andonov, Hafid Bourzoufi, and Sanjay Rajopadhye. Two-dimensional orthogonal tiling: from theory to practice. In *International Conference on High Performance Computing (HiPC)*, pages 225–231, Trivandrum, India, 1996. IEEE Computer Society Press.

[4] Rumen Andonov and Sanjay Rajopadhye. Optimal orthogonal tiling of two-dimensional iterations. *Journal of Parallel and Distributed Computing*, 45(2):159–165, 1997.

[5] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1998.

[6] Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? *Integration, the VLSI Journal*, 17:33–51, 1994.

[7] A. J. Brentjes. *Multi-dimensional continued fraction algorithms.* Mathematisch Centrum, Amsterdam, 1981.

[8] P.Y. Calland, J. Dongarra, and Y. Robert. Tiling with limited resources. In L. Thiele, J. Fortes, K. Vissers, V. Taylor, T. Noll, and J. Teich, editors, *Application Specific Systems, Achitectures, and Processors, ASAP'97*, pages 229–238. IEEE Computer Society Press, 1997. Extended version available on the WEB at http://www.ens-lyon.fr/~yrobert.

[9] Y-S. Chen, S-D. Wang, and C-M. Wang. Tiling nested loops into maximal rectangular blocks. *Journal of Parallel and Distributed Computing*, 35(2):123–32, 1996.

[10] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also LAPACK Working Note #95).

[11] Ph. Chretienne. Task scheduling over distributed memory machines. In M. Cosnard, P. Quinton, M. Raynal, and Y. Robert, editors, *Parallel and Distributed Algorithms*, pages 165–176. North Holland, 1989.

[12] Michal Cierniak, Mohammed J. Zaki, and Wei Li. Scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

[13] Alain Darte, Georges-André Silber, and Frédéric Vivien. Combining retiming and scheduling techniques for loop parallelization and loop tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[14] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[15] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan-Kaufmann, 1998.

[16] K. Högstedt, L. Carter, and J. Ferrante. Determining the idle time of a tiling. In *Principles of Programming Languages*, pages 160–173. ACM Press, 1997. Extended version available as Technical Report UCSD-CS96-489, and on the WEB at http://www.cse.ucsd.edu/~carter.

[17] François Irigoin and Rémy Triolet. Supernode partitioning. In *Proc. 15th Annual ACM Symp. Principles of Programming Languages*, pages 319–329, San Diego, CA, January 1988.

[18] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 201–214. ACM Press, January 1997.

[19] H. Ohta, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *1995 International Conference on Supercomputing*, pages 270–279. ACM Press, 1995.

[20] Peter Pacheco. *Parallel programming with MPI.* Morgan Kaufmann, 1997.

[21] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[22] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 30–44. ACM Press, 1991.

[23] Michael E. Wolf and Monica S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distributed Systems*, 2(4):452–471, October 1991.