Optimal tiling

Rumen Andonov, Sanjay Rajopadhye

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET AUTOMATIQUE

Optimal Tiling

Rumen Andonov, Sanjay Rajopadhye

N° 2135

Janvier 1994

PROGRAMME 1

Architectures parallèles,

bases de données,

réseaux et systèmes distribués

ISSN 0249-6399

Optimal Tiling

Rumen Andonov*, Sanjay Rajopadhye**

Programme 1 — Architectures parallèles, bases de données, réseaux
et systèmes distribués
Projet API

Rapport de recherche n ° 2135 — Janvier 1994 — 26 pages

Abstract: Iteration space tiling is a common strategy used by parallelizing compilers to reduce communication overhead. We address the problem of determining the optimal tile size (which minimizes the total execution time of the program), for a particular program schema. We use a realistic model of the architecture which accounts for coprocessors that permit overlapping of communication and computation, context switching times, etc. Determining the optimal tile size is shown to reduce to a non-linear optimization problem. We solve this analytically, yielding a closed form solution that involves only parameters of the architecture and program that are easily determined at compile time. It can thus be used by a compiler before code generation. Although we solve the problem for a particular schema of programs, our results can be generalized to uniform dependence loops and also to certain classes of loop programs with dynamic dependence vectors.

(Résumé : tsvp)

Supported by the French Coordinated Research Program C^3, and by the Esprit BRA project NANA 2 (No. 6632).

*andonov@irisa.fr (on leave from Center of Computer Science and Technology, Sofia, Bulgaria)
**rajopadhye@irisa.fr (partially supported by NSF Grant No. MIP-910852)

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)

Pavage Optimal

Résumé : Le partitionnement en tuiles de l'espace d'itérations est une stratégie souvent utilisée par les compilateurs paralléliseurs pour réduire le coût des communications. Nous considérons la détermination de la taille optimale de la tuile dans un cas particulier. Nous utilisons un modèle réaliste d'architecture qui prend en compte les coprocesseurs de communication, le recouvrement calculs/communications etc. La détermination de la taille optimale de la tuile se raméne alors à un problème d'optimisation non-linéaire ayant comme fonction objectif le temps total d'exécution du programme. Pour ce problème nous obtenons une solution analytique qui n'implique que des paramètres de l'architecture et du programme facilement déterminés à la compilation. Ceci permet l'utilisation de cette solution analytique avant la génération du code. Bien que nous considérons un schéma particulier de programme, nos résultats peuvent être facilement généralisés au cas des boucles de dépendances uniformes et pour une classe de boucles ayant des vecteurs de dépendences dynamiques.

1 Introduction

Let us consider the problem of computing, $X(c,m)$, where, for all $j,k,\ 0<j\leq c,\ 0<k\leq m$, $X(j,k)$ is given by

$$X(j,k)=f(X(j,k-1),X(j-w_k,k)) \tag{1}$$

for some strict function, $f(x,y)$. The coefficients w_k may take different values in the interval $1,\ldots c$, for different instances of the problem (see Figure 1). The above specification is a recurrence equation of the form

$$\forall z\in D,\quad X(z)=f(\ldots X(Az+a+\boxed{\mathrm{W(z)}})\ldots) \tag{2}$$

where D is a convex polyhedral domain, also called the iteration space. The function, f, is considered atomic, with $\Theta(1)$ complexity and corresponds to a basic block (loop body in a nested loop program). It may require multiple arguments, but each one has the form $X(Az+a+W(z))$. The $Az+a$ is a classical **affine dependency** [RF90], but the presence of a data variable $W(z)$ means that the **dependency itself is not known statically**. Such recurrences are said to have *dynamic* or "run-time" dependencies [Meg93]. Typical

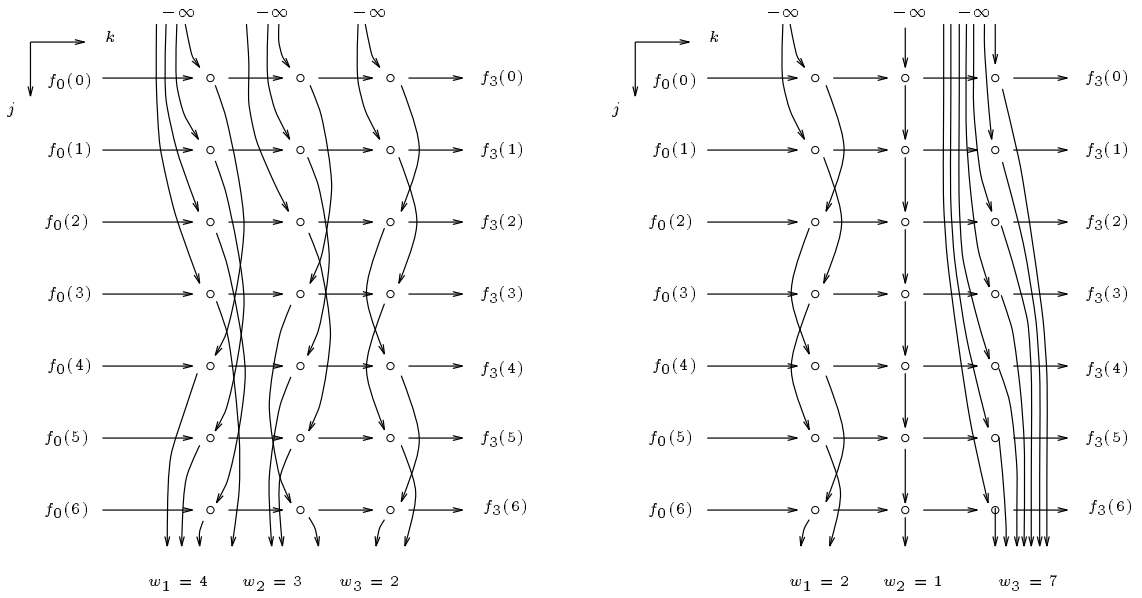


Figure 1: Two instances of (1). Observe how the graphs vary with the w_i's

examples are dynamic programming algorithm(s) for the knapsack problem(s) and their variations [AR94]. Many other problems such as finding longest common subsequence of strings, dynamic time warping, string comparison, etc., also have similar recurrences but without dynamic dependencies.

In this paper, we discuss the implementation of (1) on a distributed memory machine (such as a network of transputers) configured in a ring topology. This class of recurrences has recently been studied from a systolic standpoint [AR94]. But the systolic solution is not efficient on distributed memory architectures because of the startup overhead of inter-processor communications. One obvious approach to improving the performance is to partition the computations into blocks [MF86] so that message startup costs are amortized. In particular, we desire a partitioning of the rectangular domain, $\{j,k \mid j = 0,\ldots,c, k = 0,\ldots,m\}$ into rectangular blocks of size (r,s) which minimizes the total running time as a function of the problem parameters m and c, and the number of processors p. We shall use a realistic model of communication and computation, and show that the total running time of the algorithm is a non-linear function of the tile size and the problem parameters. Determining the optimal tile size is then reduced to a non-linear optimization problem. We solve this analytically, yielding a closed-form solution which depends only on the problem and on certain machine parameters.

The above formulation of our problem is identical to iteration space tiling, a technique used by parallelizing compilers [Wol87, Iri87] (see also [BDRR93, Des94, DV93, KCN90, RS91, Van93]). A nested loop program is compiled for a distributed memory machine in SPMD (Single Program Multiple Data) fashion, and the communication is performed by send/receive calls. A *tile* in the iteration space is a collection of iterations to be executed as a single unit, with the following protocol—all the (non-local) data required for each tile is first obtained by appropriate communication calls. The body of the tile is executed next, and this is repeated iteratively. The code for the body contains no calls to any communication routine.

In all previous work on tiling, only uniform dependencies were considered, and the objective of the tiling was usually to minimize the volume of the residual communication. This is at best, an approximate measure of the performance of the program. Furthermore, much of the work deals with determining the *shape* of the tile. Indeed, Boulet et al. recently gave an eloquent argument for separating the tiling problem into two steps—first determine the tile shape, and then the tile size [BDRR93]. They show how the

first problem can be reduced to a linear optimization problem using an architecture-independent performance model. The second step is not addressed. Our work focuses on the determination of tile size, assuming that the tile shape has already been chosen, and is a logical extension of their work. We are interested in optimizing the global performance—the execution time for the whole program. This approach is similar to that of Miguet and Robert [MR90], who deal with implementing path planning programs on rings. Their result however, is not in closed form but requires a simulation step to determine the optimal tile size.

Our results are also closely related to the work of King et al. [KCN90], who determine a similar optimal tile size, using total completion time as the cost measure. However, their architecture model does not accurately model the overlap between computation and communication (most parallel machines today have separate communication coprocessors, and the communication can be performed concurrently with the computation). In our model, we take this possibility into account. We will see that this complicates the formulation of the optimization problem. Aleksandrov also studies the tile size problem, but for a restrictive model (no overlap between computation and communication) and under some simplifying constraints [Ale93].

Another aspect of our results is that, as mentioned above, our recurrence has dynamic dependencies. We will see that this affects only the tile shape, and our analytical results can be easily adapted for arbitrary uniform dependence programs. We do not however, claim a solution for *arbitrary, dynamic* dependence programs, but only for those with the special form of dependencies exhibited by (1). The remainder of this paper is organized as follows. We first give a model of the computation and communication that accurately describes the overlap permitted by the architecture. In Section 3, we present the tiling implementation of (1) and formulate the optimization problem that is to be solved. We then solve this by a casewise analysis in Sec 4, and give our conclusions in Sec 6.

2 The tile shape and the model

To choose the tile shape, we follow, as far as possible, the systolic synthesis approach [AR94], briefly recapitulated as follows. Two crucial properties allow us to derive a systolic array for (1), despite the dynamic dependencies: the dynamic part of the dependency vector is always **vertical** and **downwards**, i.e., it is a positive multiple of $[0, 1]$. This

implies that if we choose the allocation function to be $a(j,k) = k$, the dynamic part is projected into the processor space, and the interconnections between PEs are purely systolic (nearest neighbor). The PE must manage a certain amount of memory (not just a few registers), and the manner in which it is accessed depends on w_k, but the communication has been localized. Furthermore, $t(j,k) = j + k$ is a valid linear schedule.

Similar considerations can be used to choose the tile shape when mapping the program specified by (1) to a parallel machine. In particular, note that if we use the canonical axes as the tile boundaries, the resulting tile graph also has dependencies whose dynamic components are still vertical and downwards. Thus the tile graph too, can be projected in the $[0,1]$ direction yielding an SPMD program whose communication pattern is nearest neighbor (but whose local memory access pattern may be dynamic). Hence we choose such rectangular tiles of r rows and s columns. The (j',k')-th tile consists of index points $[rj'+1, sk'+1]$ through $[(r+1)j', (s+1)k']$ in the original iteration space.

We now develop an expression for the running time of such a tiled program on a distributed memory machine. For this, we use an accurate model of the underlying machine, using the fact there exist standard system calls that achieve the communication, they involve a context switch and possibly copying of data to/from system buffers, the actual communication is achieved by a coprocessor using a DMA transfer and may be overlapped with the computation, etc. Although the model is fairly elaborate, our results hold for simplified versions of the model. Furthermore, the general idea of setting up an optimization problem and solving it analytically is still a valid approach for other types of machines with other parameters, etc.

We assume that the time to execute one call to f (a single instance of the loop body) is τ_a, and that it can be determined by simple compile time analysis. The time to execute a tile body is then, $\beta_a + \tau_a rs$. Here, β_a is the overhead of setting up the loop for each tile, but it will be ignored in this paper. The arithmetic time is thus given by $t_a = \tau_a rs$.

The communication cost is adapted from the standard model [MR90, GR88, GH86]. At the machine level, the time to transfer v words between two processors in the absence of any contention* is given by $\beta_m + v\tau_c$, where where β_m is the message startup time (and the constant overhead of headers, routing information, etc.) and $1/\tau_c$ is the bandwidth.

*For our problem, communication is contention-free. This is usually the case with most uniform dependencies algorithms when they are properly synchronized.

However, most programmers do not write at this level†, but use system calls to achieve the communication. This additional overhead is similarly modeled by $\beta_s + v\tau_s$—a constant overhead, β_s (the time for switching contexts, allocating message buffers, and setting up DMA channels), plus a cost proportional to data volume, $v\tau_s$ (the cost of transferring the data to/from the system buffers). Thus, in the absence of *any* overlapping whatsoever, the time to transmit a message is $\beta_s + 2v\tau_s + (\beta_m + v\tau_c)$. Note that $v\tau_s$ is counted twice—once for the sender and once for the receiver; but β_s is counted only once. This is because the system call occurs on *different* processors, and when the sender and receiver are properly synchronized they are simultaneous.

King et al. use the same model [KCN90], but with two assumptions. The $\beta_m + v\tau_c$ term is dropped because it can be overlapped with CPU operation. Furthermore, they also assume that the $2v\tau_s$ term can *never* be overlapped because the CPU is involved in transferring data to/from system buffers, and may not simultaneously be used to compute the tile body. Neither of these assumptions is strictly true. First, note that the $\beta_m + v\tau_c$ term can be dropped only if it is smaller than the time to *compute* the tile body, and this may not always be true (depending on the value of τ_a, τ_c, r and s). Second, we can avoid explicitly copying to/from the system buffers as explained below. The code executed for tile $[j', k']$ is as follows:

```
  receive()                        (* dummy *)
repeat
   receive()                       (* data needed for current tile *)
   send()                          (* data produced by previous tile *)
   for j=0:r-1, k=0:s-1 do
       compute f(rj' + j, sk' + k)
 end  repeat
   send()                          (* data produced by last tile *)
```

The `send` is non-blocking, and returns immediately (after initializing the transfer). Hence the tile body may be executed concurrently with the physical data transfer. If the tile does not overwrite into the memory (for an adequate number of previous tiles), the `send` operation does not need to use a separate system buffer. The `receive` is blocking (i.e., the call does not return until all the data has been received). This ensures proper synchronization in the presence of fluctuations in processor speeds, etc. Once all the data

†Indeed, the user programs may be prohibited from accessing the system resources directly (for example, if the machine supports multiple user programs on the same node, managed by an OS kernel).

has arrived (which may even be immediate, since the corresponding `send` was called by the preceding processor when it executed the *previous tile*), the data is copied from the system buffer to the user area before `receive` returns. The only reason for copying is to ensure that the system buffer is not overwritten by the next `send` while its contents are being used in the body of the current tile. This can be avoided if we have two system buffers and use them alternately. While the send is filling up one buffer, the tile body uses the other, and when the `receive` is called in the next iteration, it simply switches a pair of pointers. This simple double buffering strategy is safe if we can show that there is a delay at least equal to the time for the physical communication between two successive calls to `send`, and if between any two calls to `send` by a processor, there is a call to `receive` by the next one. It is easy to show that if these conditions are true for successive tiles on a processor, they will also hold for all subsequent processors because of the dependent nature of the computations. Hence it is up to the host to ensure this condition for the very first processor. With this optimization, the communication time is $\beta_s + \beta_m + v\tau_c$. Note that the very first `send` tile has no meaningful data, but must be matched by a corresponding `receive`. This explains the first line outside the `repeat` loop.

3 Formulating the optimization problem

We now address the problem of choosing the tile size parameters, r and s that minimize the total running time, T, expressed as a function of r, s, the number of processors, p, the problem parameters, m and c, and the system parameters, τ_a, τ_c, β_m and β_s.

Our tile graph has $\lceil c/r \rceil \times \lceil m/s \rceil$ nodes and it is projected to the processor space using a vertical projection (for simplicity, we will drop the ceilings in all formulæ). Thus, each column of the tile graph, henceforth called a *macrocolumn*, is executed sequentially on a single processor. We also consider the case when there may not be sufficient processors, i.e., $p \leq m/s$. In this case, we suppose that the processors are configured in a ring, and the computation is performed in multiple passes. Thus the feasible space of r and s is governed by the constraints $1 \leq r \leq c$ and $1 \leq s \leq m/p$ (if $s > m/p$, the tiles are so large that they cannot be distributed evenly to the processors, even for a single pass, and this is obviously too inefficient). In the multipass case, the host is also responsible for buffering of the results produced by the last processor as needed, in order to ensure correct synchronization.

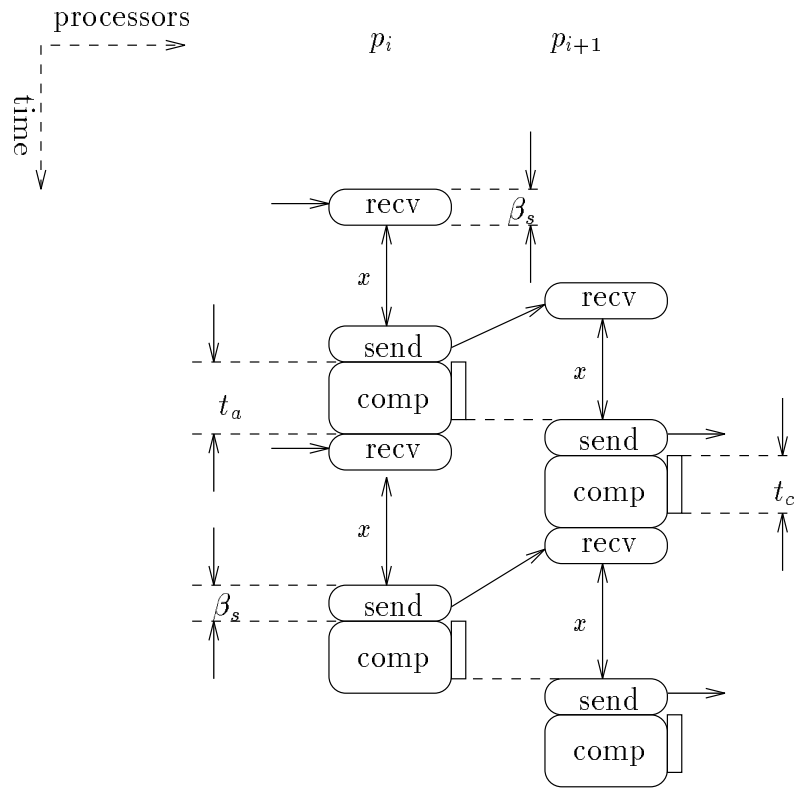


Figure 2: Steady state of four adjacent (internal) tiles. The dashed lines between the macrocolumns illustrate the blocking nature of the `receive`'s: the subsequent `send`'s start only after the data is received.

In our notation, we use the convention that H, L and T denote *throughput*, *latency* and *completion time*, respectively. The subscripts t, m and p denote, respectively, a *single tile*, a *macrocolumn* (i.e., a column of the tile graph) and a *pass* of the ring. Furthermore, we will ignore β_m (the overhead of headers etc, in a message) and β_a, the loop overhead (as already mentioned). We also assume that the communication calls are "atomic", i.e., the physical communication begins β_s time after the `send` is called.

Based on the discussion of the previous section, the steady state activity profile of the tile is as follows. First, a call to `receive` which takes $\beta_s + x$ time (for some, as yet unknown, x which depends on the volume of the communication, and may even be zero), then a call to `send`, which returns in β_s time, and then the tile body, which takes $t_a = \tau_a rs$ time. The physical communication takes $t_c = \tau_c r$ time. This is illustrated in

Fig. 2, where the relative positions of the `send-receive` pairs on adjacent processors is chosen arbitrarily.

Now, x is the smallest value that ensures that at least t_c time elapses between two successive calls to `send` by a single processor, i.e., $t_a + \beta_s + x \geq t_c$. Hence, if $t_a + \beta_s \geq t_c$, then $x = 0$, and the tiles are executed continuously, otherwise there is a non-zero wait (Fig. 3). Thus, the throughput of each tile H_t is given as follows:

$$H_t = \beta_s + \max(t_a + \beta_s, t_c) \tag{3}$$

Assume that at $t = 0$, the first tile starts executing on the first processor (the data it needs has been sent by the host sufficiently in advance). Since the tile throughput is given by (3), and a macrocolumn contains c/r tiles, the throughput and execution time of a macrocolumn, are as follows:

$$H_m = \frac{c}{r} H_t, \tag{4}$$

$$T_m = \begin{cases} H_m, & \text{if } t_a + \beta_s \geq t_c \\ H_m + \epsilon & \text{otherwise} \end{cases} \tag{5}$$

where the term ϵ above is due to the overlapping and represents the difference between the computation and communication times for a *single* tile (i.e., $\epsilon = t_c - t_a$).

There is a latency, L_m, between adjacent tiles mapped to neighboring processors. If a tile starts at time t, its results will be sent during the *next* tile (which itself, starts at $t + H_t$), and the transmission takes $\beta_s + t_c$ additional time (see Figure 3). Since a complete pass involves p macrocolumns, there is a similar latency, L_p between successive passes.

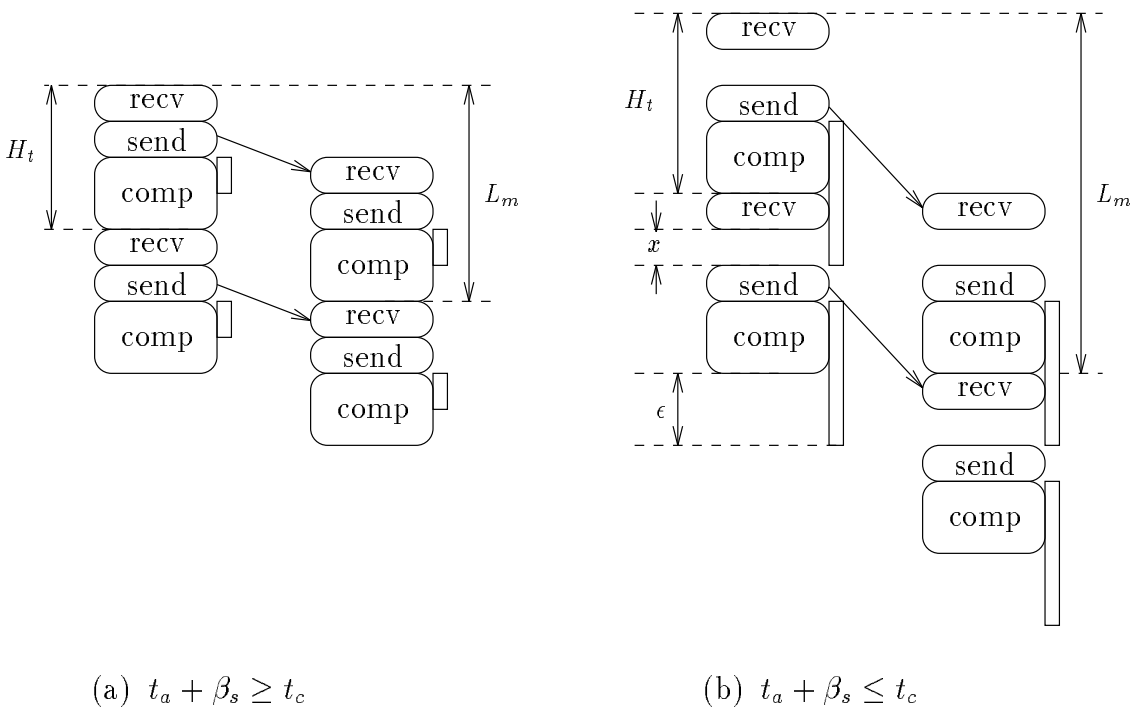
$$L_m = H_t + \beta_s + t_c \tag{6}$$

$$L_p = pL_m = p(H_t + \beta_s + t_c) \tag{7}$$

In a single pass, the last macrocolumn can start only at $(p-1)L_m$, and hence, the execution time of a pass is given by

$$T_p = (p-1)L_m + T_m \tag{8}$$

Observe that there are two constraints, (4) and (7) that determine when the next pass can start. Indeed, the pass-throughput of the ring is



(a) $t_a + \beta_s \geq t_c$ (b) $t_a + \beta_s \leq t_c$

Figure 3: Steady-state profiles when (a) computation, and (b) communication dominates

$$H_p = \max(H_m, L_p) \tag{9}$$

The entire program is executed in m/ps passes, and hence the last pass can only start at $(m/ps - 1)H_p$. Thus, the total running time, T, which is the cost function for our optimization problem, is given by

$$T = \left(\frac{m}{ps} - 1\right) H_p + T_p \tag{10}$$

Our problem thus reduces to minimizing T, subject to $1 \leq r \leq c$ and $1 \leq s \leq m/p$.

4 Solving the optimization problem

Because of the presence of the 'max' operation in (3) and (9), the solution space is divided into four subregions (it is easily seen that the curves $H_m = L_p$ and $t_a + \beta_s = t_c$ intersect at, at most, one point in the positive orthant). Hence need to consider four cases, separated

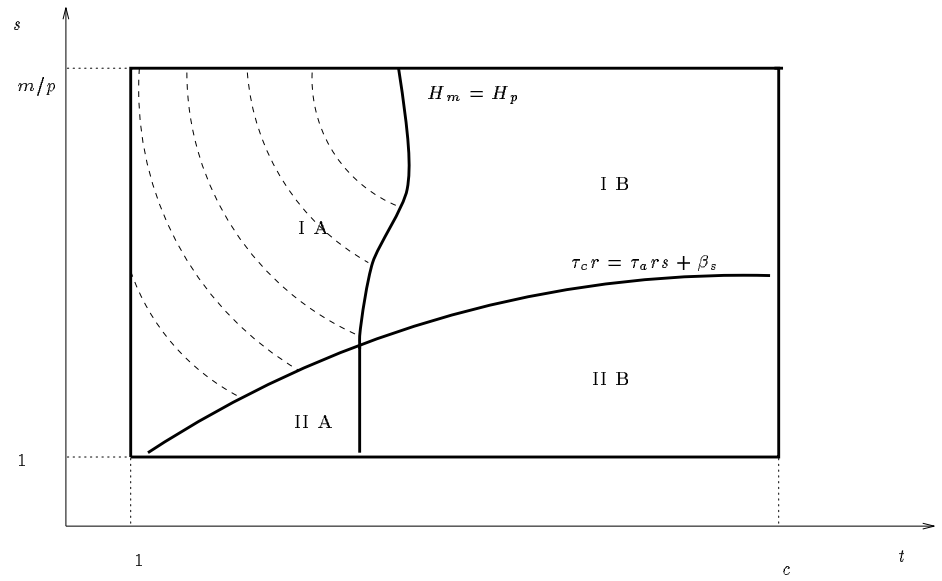


Figure 4: Solution space of the optimization problem. The dashed lines are the family of hyperbolæ used in the proof of Lemma 1.

by $t_a + \beta_s = t_c$ and $H_m = L_p$ (see Figure 4). In each region, we will have a different cost function.

4.1 Computation time dominates ($t_a + \beta_s \geq t_c$)

Our formulæ now simplify to:

$$\begin{aligned} H_t &= 2\beta_s + t_a \\ T_m &= H_m = \frac{c}{r}(2\beta_s + t_a) \\ L_m &= 3\beta_s + t_c + t_a \\ L_p &= p(3\beta_s + t_c + t_a) \\ T_p &= (p-1)(3\beta_s + t_c + t_a) + \frac{c}{r}(2\beta_s + t_a) \end{aligned}$$

We now have to separately consider the subcases $H_m \geq L_p$ and $H_m < L_p$. The execution profile of the program for these two cases is shown in Figure 5 (the first pass is shown lightly shaded and the second one is darker). Observe that under steady state, the tiles are tightly packed (as in Fig 3a, $x = 0$), but at the beginning of the pass there could

be some waiting. Furthermore, note that when $H_m \geq L_p$, the last processor produces its results before the first one is ready to accept them, and the host (or a special I/O process) must buffer them; when $H_m < L_p$, the processors are idle for some time between passes, but no buffering is needed.

Subcase IA ($H_m \geq L_p$)**:**

Therefore, $H_p = H_m = \frac{c}{r}(2\beta_s + t_a)$, and substituting this in (10),

$$T_1(r,s) = \frac{mc}{psr}(2\beta_s + t_a) + (p-1)(3\beta_s + t_a + t_c) \tag{11}$$

If we now set $t_a = \tau_a rs$, and $t_c = \tau_c r$, and simplify, our problem reduces to the following:

Prob. 1: Minimize

$$T_1(r,s) = \frac{2mc\beta_s}{psr} + (p-1)\tau_a rs + (p-1)\tau_c r + 3(p-1)\beta_s + \frac{mc\tau_a}{p} \tag{12}$$

subject to

$$1 \leq r \leq c \tag{13}$$

$$1 \leq s \leq m/p \tag{14}$$

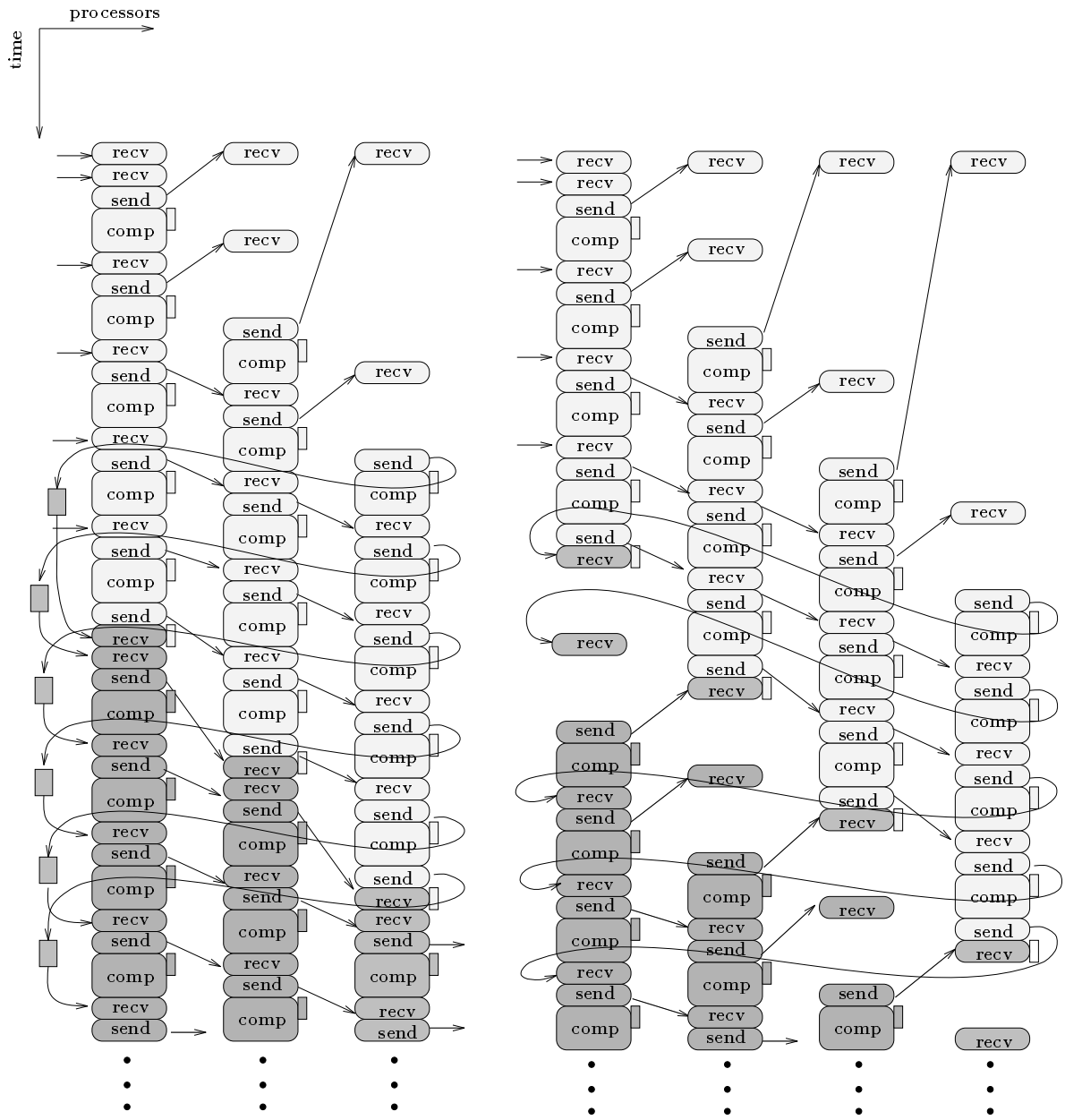
$$\frac{\tau_c}{\tau_a} \leq s + \frac{\beta_s}{r\tau_a} \tag{15}$$

$$p(3\beta_s + \tau_a rs + \tau_c r) \leq c\tau_a s + \frac{2c\beta_s}{r} \tag{16}$$

The objective function and the last two constraints above are nonlinear (see Figure 4). However, we can easily solve **Prob. 1** because of the following observation:

Lemma 1 *The solution of* **Prob. 1** *satisfies either* $r = 1$ *or* $s = m/p$.

Proof: For any point, (r,s), note that T_1 strictly decreases if we reduce r, keeping rs fixed (i.e., move along a hyperbola): the first two terms in (12) remain unchanged, the third term is strictly smaller, and the remaining terms are constants. If we continue to move in this manner, we must eventually satisfy $r = 1$ or $s = m/p$. To see why, observe that if we are on either of these boundaries, any further movement takes us out of the feasible region. On the other hand, if any point (r,s) satisfies (15) or (16), any other point (r',s') hyperbolically to its left (i.e., $r' < r$ and $r's' = rs$) also satisfies it, since in this case the RHS in (15) and (16) increases, and the LHS keeps (or decreases) its values. ∎



(a) $H_m \geq L_p$ (b) $H_m < L_p$

Figure 5: Activity profile of the program when computation dominates

For technical reasons, we also need to consider a variation of **Prob. 1**, (called **Prob. 1e**) which has the same objective function, but without the constraint (16). Its feasible solution is thus defined by constraints (13),(14) and (15), and it specifies the optimization problem with the cost function of region IA, but over the entire region IA $\cup$ IB.

Corollary 1 *The solution of* **Prob. 1e** *also satisfies either* $r = 1$ *or* $s = m/p$.

Subcase IB ($H_m < L_p$)**:**

Now, $H_p = L_p = p(3\beta_s + t_c + t_a)$, and substituting this in (10) and simplifying gives us the following optimization problem.

Prob. 2: Minimize

$$T_2(r,s) = (\frac{m}{s} - 1)(3\beta_s + t_a + t_c) + \frac{c}{r}(2\beta_s + t_a) \tag{17}$$

subject to the constraints (13-15) and

$$\frac{2c\beta_s}{r} + c\tau_a s < p(3\beta_s + \tau_a rs + \tau_c r) \tag{18}$$

It is easy to show that the difference between the two cost functions is given by,

$$T_2(r,s) - T_1(r,s) = \left(\frac{m}{ps} - 1\right)\left[p(3\beta_s + t_a + t_c) - \frac{c}{r}(2\beta_s + t_a)\right] \tag{19}$$

This leads to the following lemma.

Lemma 2 *If the solution of* **Prob. 2** *is not on the* $s = m/p$ *boundary, then it is worse than the solution of* **Prob. 1e**.

Proof: If the solution of **Prob. 2** is not on the $s = m/p$ boundary, then $\frac{m}{ps} - 1 > 0$. Let us denote by (r^*, s^*) the solution of **Prob. 1e**. From (19), (18) and Corollary 1 we have $T_2(r,s) > T_1(r,s) \geq T_1(r^*, s^*)$ for any point (r,s) in the feasible region of **Prob. 2**. ∎

4.1.1 The solution over region I

Now, we need to look for our optimal solution only on the $r = 1$ and $s = m/p$ boundaries. If we substitute $r = 1$ in (12), we get a function of only s:

$$\widehat{T}(s) = \frac{2mc\beta_s}{ps} + (p-1)\tau_a s + (p-1)(\tau_c + 3\beta_s) + \frac{mc\tau_a}{p} \tag{20}$$

Similarly, substituting $s = m/p$ in (12) yields $\widetilde{T}(r)$, a function of only r,

$$\widetilde{T}(r) = \frac{2c\beta_s}{r} + \frac{(p-1)}{p}(m\tau_a + p\tau_c)r + 3(p-1)\beta_s + \frac{mc\tau_a}{p} \tag{21}$$

Hence, our problem now reduces to determining the minima for (20) in the interval $\max(1, \frac{\tau_c - \beta_s}{\tau_a}) \leq s \leq \frac{m}{p}$ and (21) in the interval $1 \leq r \leq c$ (regardless of whether constraint (16) or (18) is valid). The global minimum is at the one with lower cost.

We see that both $\widehat{T}(s)$ and $\widetilde{T}(r)$ have the form, $F(x) = \frac{A}{x} + Bx + C$, the sum of a hyperbola, and a straight line. This is a classic, strongly convex function, which has a unique minimum (for positive x) at $x^* = \sqrt{A/B}$, whose value is $F(x^*) = C + 2\sqrt{AB}$. If we desire to minimize the function at an integer point within an interval bounded by a and b, (where $a < b$, and there is at least one integer between them), this occurs at one of the two integers $\lfloor x^* \rfloor$ or $\lceil x^* \rceil$ (or at $\lceil a \rceil$ or $\lfloor b \rfloor$ if x^* is too small or too large). Formally,

$$x_{opt} = \begin{cases} \lceil a \rceil & \text{if } x^* \leq \lceil a \rceil \\ \lfloor b \rfloor & \text{if } x^* \geq \lfloor b \rfloor \\ \lfloor x^* \rfloor & \text{if } F(\lfloor x^* \rfloor) \leq F(\lceil x^* \rceil) \\ \lceil x^* \rceil & \text{if } F(\lfloor x^* \rfloor) > F(\lceil x^* \rceil) \end{cases} \tag{22}$$

For our problem, the value of s^* and r^* are given as follows.

$$s^* = \sqrt{\frac{2mc\beta_s}{(p-1)p\tau_a}} \tag{23}$$

$$r^* = \sqrt{\frac{2pc\beta_s}{(p-1)(m\tau_a + p\tau_c)}} \approx \hat{r} = \sqrt{\frac{2pc\beta_s}{(p-1)m\tau_a}} \tag{24}$$

The approximation holds because m and c, grow much faster than p, i.e., $m\tau_a \gg p\tau_c$ asymptotically (but note that always, $r^* < \hat{r}$). Following the lines of (22) we can define r_{opt} and s_{opt}. Note that the feasible intervals of the two problems intersect at $[1, \frac{m}{p}]$, and that $\widehat{T}(s)\Big|_{s=\frac{m}{p}} = \widetilde{T}(r)\Big|_{r=1}$.

Now, consider whether s^* and r^* lie in the intervals, $\max(1, \frac{\tau_c - \beta_s}{\tau_a}) \leq s \leq \frac{m}{p}$, and $1 \leq r \leq c$, respectively. Asymptotically, $s^* \geq 1$, and $r^* \leq c$ (since $mc \gg \frac{(p-1)p\tau_a}{2p\beta_s}$ and $\frac{2p\beta_s}{(p-1)\tau_a}$). It is easy to see that $\hat{r} \geq 1$ iff $s^* \geq \frac{m}{p}$. Then s^* is outside its feasible interval, and $s_{\text{opt}} = \frac{m}{p}$. But this puts $[1, s_{\text{opt}}]$, i.e., $[1, \frac{m}{p}]$ into the feasible interval of $\widetilde{T}(r)$, and the final solution is at $[r_{\text{opt}}, \frac{m}{p}]$. Otherwise, $r^* < 1$, and the optimal solution is at $[1, s_{\text{opt}}]$. Based on this discussion, we have the following result.

Theorem 1 *The optimal tile size is unique and given by*

$$\begin{cases} [r_{\text{opt}}, \frac{m}{p}] & \text{if } 2pc\beta_s \geq (p-1)m\tau_a \\ [1, s_{\text{opt}}] & \text{otherwise} \end{cases}$$

where s_{opt} *minimizes* $\widehat{T}(s)$ *in the interval* $[1, \frac{m}{p}]$*, and* r_{opt} *minimizes* $\widetilde{T}(r)$ *in* $[1, c]$*.*

We have thus obtained a single closed form expression, involving constants that are easily determined at compile time, that gives us the tile size that optimizes the running time of the program. Note that the condition $2pc\beta_s \geq (p-1)m\tau_a$ is simply a constraint on the aspect ratio of the iteration space, $\frac{m}{c}$, relative to a constant, $\frac{2p\beta_s}{(p-1)\tau_a}$. For certain aspect ratios, a single pass program with r_{opt} rows in each tile is optimal, and for others, we use a multi-pass program, with one only row per tile.

4.2 Communication Time Dominates ($t_a + \beta_s \leq t_c$):

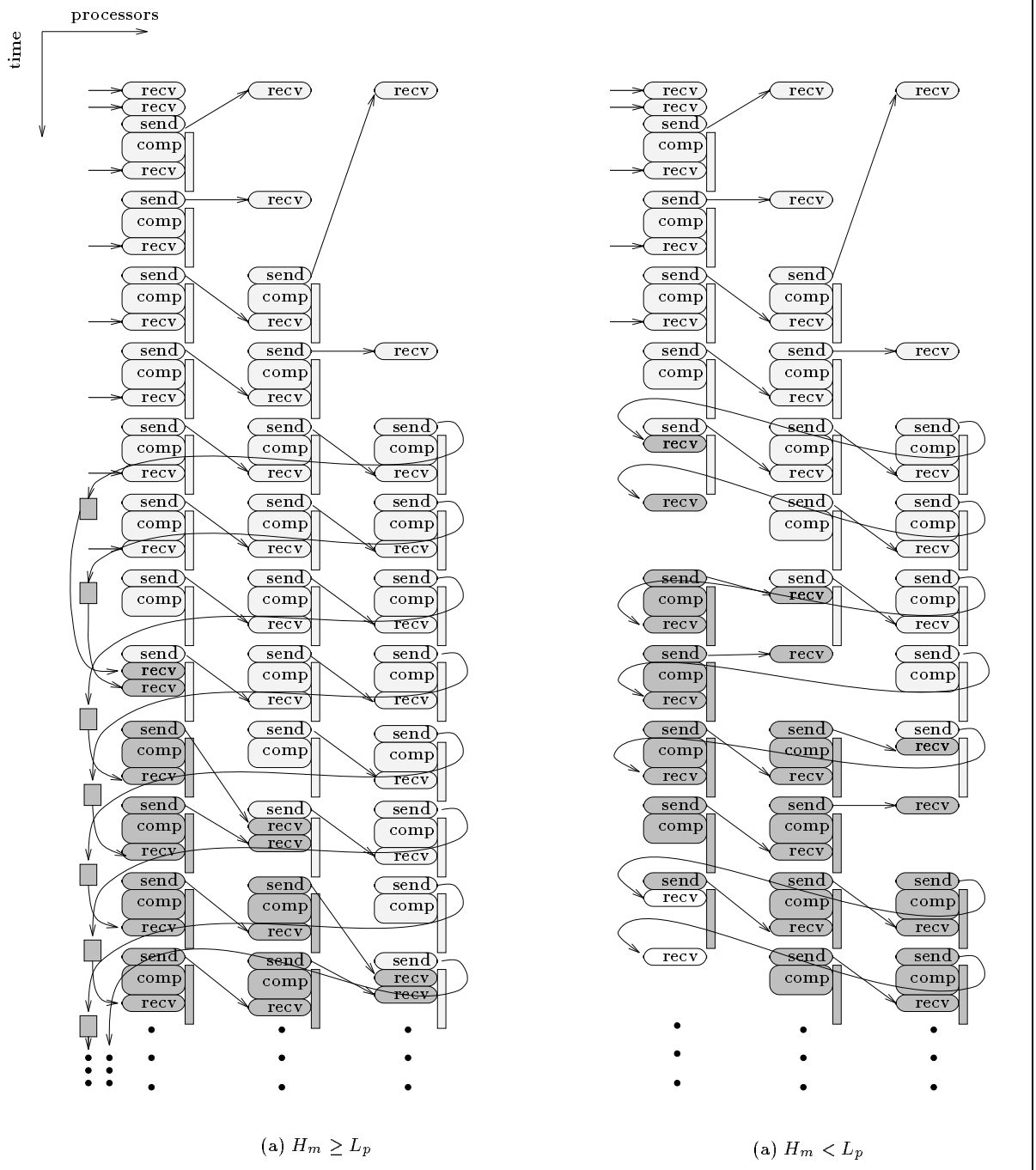
Let us now consider what happens if the physical communication time for the data sent by a tile is larger than the time to perform the computation in the tile body. This corresponds to a domain, bounded from above by the hyperbola

$$s \leq \frac{\tau_c}{\tau_a} - \frac{\beta_s}{\tau_a r} \tag{25}$$

We assume $\frac{\tau_c}{\tau_a} - \frac{\beta_s}{\tau_a c} > 1$, otherwise the feasible space in this case becomes vacuous. Again we have to the consider the both subcases $H_m \geq L_p$ and $H_m < L_p$. Now, the $H_m = L_p$ boundary simplifies to $\frac{c}{r} = 2p$. The activity profile of the processors for the first two passes is illustrated in Figure 6. Observe how $x \neq 0$, and there is idle time between successive tiles, even under steady state. Also note that buffering between passes is needed for the $H_m \geq L_p$ case, but not for the $H_m < L_p$ case.

Our formulæ simplify to:

$$\begin{aligned} H_t &= \beta_s + t_c \\ H_m &= \frac{c}{r}(\beta_s + t_c) \\ T_m &= H_m + \epsilon = \frac{c}{r}(\beta_s + t_c) + t_c - t_a \\ L_m &= 2(\beta_s + t_c) \\ L_p &= 2p(\beta_s + t_c) \\ T_p &= \left(\frac{c}{r} + 2(p-1)\right)(\beta_s + t_c) + \epsilon \end{aligned}$$



(a) $H_m \geq L_p$ (a) $H_m < L_p$

Figure 6: Activity profile of the program when communication dominates

Subcase IIA ($H_m \geq L_p$):

Now, $H_p = H_m$, and the execution time becomes:

$$T = \left(\frac{mc}{psr} + 2(p-1)\right)(\beta_s + t_c) + t_c - t_a$$

Substituting $t_c = \tau_c r$ and $t_a = \tau_r s$ we get

$$T = \left(\frac{mc}{psr} + 2(p-1)\right)(\beta_s + \tau_c r) + \tau_c r - \tau_a rs \tag{26}$$

We see that in the cost function s occurs only in the denominator and in the linear part with negative coefficient. Hence, for any fixed r the value of s should be as large as possible in order to minimize T. Therefore, the optimal solution should satisfy (25) as equality, i.e., $s = \frac{\tau_c}{\tau_a} - \frac{\beta_s}{\tau_a r}$. However, with this value of s we move to the feasible region of Subcase IA, and the solution given by Theorem 1 holds.

Subcase IIB ($H_m < L_p$):

Now, $H_p = L_p$ and the total execution time becomes:

$$T = \left(\frac{2m}{s} + \frac{c}{r} - 2\right)(\beta_s + \tau_c r) + \tau_c r - \tau_a rs \tag{27}$$

The same argument as in **subcase IIA** holds. Once again, s should be as large as possible in the optimal solution, i.e., $s = \frac{\tau_c}{\tau_a} - \frac{\beta_s}{\tau_a r}$. This moves us into the feasible region of subcase IIA, and again, the optimal solution given by Theorem 1 holds.

5 Discussion and Illustration

To recapitulate, we developed an analytic model of the behaviour of a tiled program for (1) on a distributed memory machine (configured as a ring). The model accounts for coprocessors that permit overlapping of communication and computation, OS calls for communication and the associated transfer context switching times, etc. Using this model, we formulated the problem of minimizing the total running time of the program as a (non-linear, discrete, two-dimensional) optimization problem over the space of feasible tile sizes. We explored the *entire* feasible region, consisting of four subregions and showed that the optimal solution must always lie on one of two boundaries, reducing our problem

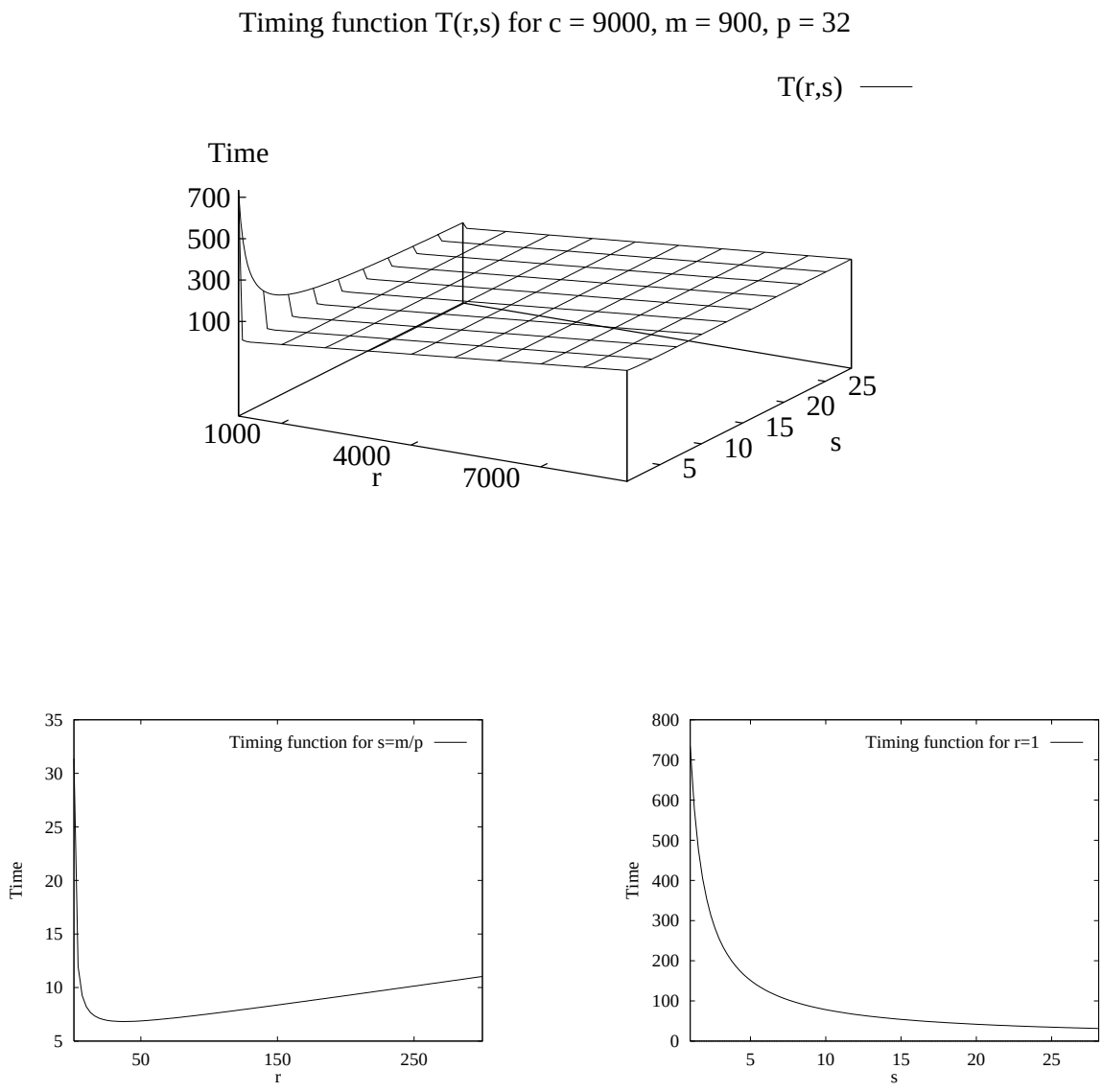


Figure 7: Illustration of the case when the solution is on $s = m/p$ boundary

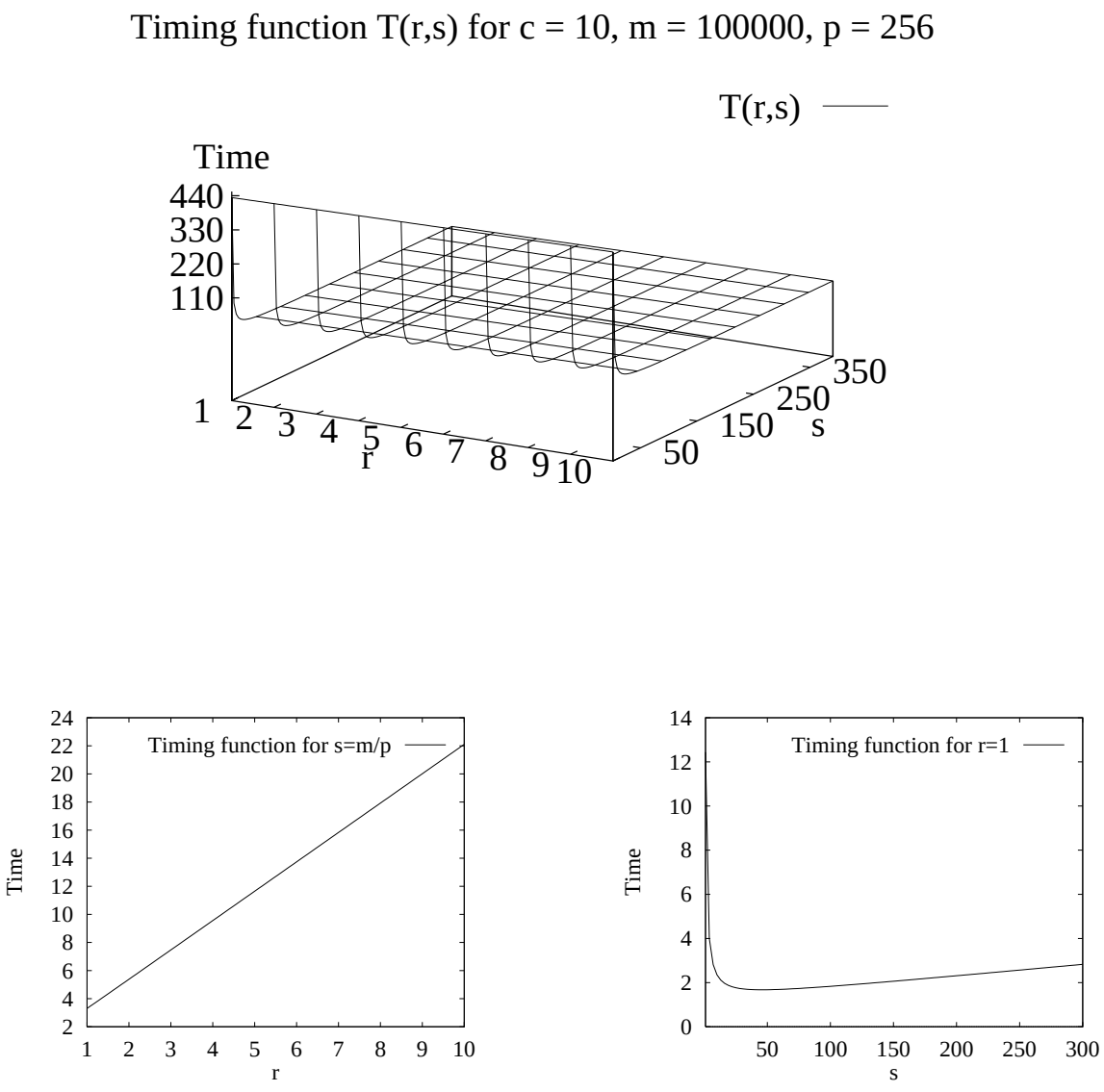


Figure 8: Illustration of the case when the solution is on $r = 1$ boundary

to a one-dimensional (still non-linear, discrete) optimization problem. Fortunately, the cost function is now strongly convex, and so we are able to analytically obtain the exact solution to the problem.

Figs 7 and 8 show the predicted behavior (obtained using gnuplot) for two examples respectively: one for the case when the solution is on $s = m/p$, and the other when it is on $r = 1$. The parameters used are taken from Aleksandrov [Ale93] who investigates the knapsack problem running on a Transputer T800 based machine under HELIOS. The values are $\beta_s = 1.440$ milliseconds, $\tau_c = 0.56$ microseconds, and $\tau_a = 21$ microseconds. These values are only for illustrating the results, and similar parameters for other machines can easily be substituted.

We now present an intuitive explanation of the results. Considering the four subregions of the feasible space that we have explored, one expects intuitively, that the solution should be in region IA because in each of the other regions there are idle processors as explained below. If communication dominates the computation in a single tile (i.e., $t_a + \beta_s < t_c$), the PE is idle for a short time (shown by x in Fig 2) in *each* tile; and whenever $H_m < L_p$, each processor idles between successive passes. Our results in the previous section formally confirm this intuition. Indeed, some researchers simply assume that $H_m \geq L_p$ is a constraint of the feasible region [Ale93, MR90]. This is not strictly accurate. Even intuitively, one should allow the latency of a pass to be greater than the throughput of a macrocolumn if the program is executed in a single pass. Our solution correctly permits H_m to be less than L_p if we are on the $s = m/p$ boundary.

Indeed, by substituting (24) in (18) and simplifying, we can see that the optimal solution is in region IB iff

$$(p-2)\sqrt{\frac{2pc\beta_s(m\tau_a + p\tau_c)}{p-1}} < 3p\beta_s^2 - mc\tau_a \tag{28}$$

Hence, for example, when $p = 2$, $m = 10$ and $c = 75$ (using our previous parameters) the optimal solution is at $r = 45$ but the boundary of region IA is at $r = 30$. This solution would not have been discovered by the other techniques. Asymptotically however, the constraint that $H_m \geq L_p$ is justified: note that m and c, the problem sizes increase much faster than p^2, the number of processors, and (28) will not hold.

We point out an interesting connection between the tiling problem, as posed in this paper, and the allocation of arrays in parallelizing compilers. Indeed, we assign the columns of the iteration space to processors, using a *block cyclic* allocation (blocks of size

s to p processors in a round robin fashion, using as many passes as needed). We seek to determine the value of s, the number of contiguous columns in each block. Moreover, we also seek the intervals when the computation of each block should be interrupted for communication, so as to achieve optimal performance. Our optimization problem solves for both of these unknowns.

6 Conclusions

We have presented an analytic solution of optimal tiling problem, namely the problem of determining the size of iteration space tiles so as to minimize the running time of the program. The source programs that we investigated have a particular schema—the dynamic programming formulation of the knapsack and related problems. Although this is a small subset of nested loop programs, it includes many interesting problems such as longest common subsequence, dynamic time warping, string comparison, etc. The program has a two dimensional, rectangular iteration space, and exactly two dependencies, and are implemented on a ring topology. However, one of the dependence vectors is dynamic (not known at compile time) and this complicates the problem. We have shown how this can be addressed by properly selecting the tile shape, leaving the size optimization problem unaffected.

To solve the problem we have developed an accurate model of the performance of a tiled program on distributed memory machines. This accounts for the fact that communication is acheived by calls to the operating system, which is a fairly large overhead, but that the physical communication can very often be overlapped with the computation because the communication is set up as a DMA transfer. Our model however, does not account for caches, and this could be a source of anomalies. Furthermore, the model as we have developed is not strictly valid for machines such as the Intel Paragon where an OS kernel runs on each node (this implies that for any call to a communication routine, the copy cannot be avoided). However, Desprez notes that empirically the times are very similar to those predicted by our model [Des94], and it would be interesting to extend the model to such machines, as well as to SIMD machines.

Although our results have been developed for a specific program (schema), the same approach can be used for other programs and for higher dimensions. When there are additional dependencies, there will be additional send-receive calls in the tile body. However,

if they are all being sent to the same processor, one may combine the communication into a single call. Then the model that we have used still holds, except that τ_c will be different. For higher dimensional iteration spaces, the model needs to be extended to two dimensional arrays (either meshes or tori), and indeed, one can determine which architecture (torus/mesh or ring/linear-array) yields a better performance.

Since our final solution is in terms of a closed form expression involving program and machine parameters that can be statically determined, it can guide the compiler in choosing the tile size automatically.

It is clear that what we have presented is an analytic solution, and it needs experimental validation, and is the subject of our ongoing investigation. Another problem for future work is the generalization of the aproach to arbitrary uniform dependence programs.

Acknowledgments

The authors are thankful to S. Miguet who gave the initial motivation for this study, to members of the API team and V. V. Dongen for useful feedback and many helpful discussions.

References

[Ale93] Vassil Aleksandrov. *Parallel Algorithms for Discrete Optimization Problems.* PhD thesis, Center of Computer Science and Technology, Acad. G. Bonchev st., bl. 25a, Sofia 1113, Bulgaria, 1993. Preliminary Project.

[AR94] R. Andonov and S. Rajopadhye. *An Optimal Algo-tech-cuit for the Knapsack Problem.* Technical Report PI-791, IRISA, Campus de Beaulieu, Rennes, France, January 1994. (submitted to IEEE Transactions on Parallel and Distributed Systems).

[BDRR93] P. Boulet, A. Darte, T. Risset, and Y. Robert. *(Pen)-ultimate tiling?* Research Report 93-36, ENS de Lyon, 46, Allée d'Italie, 69364 Lyon Cedex 07, France, November 1993.

[Des94] Frédéric Desprez. *Procédures de Base Pour le Calcul Scientifique sur Machines Parallèles à Mémoire Distribuée*. PhD thesis, Institut National Polytechnique de Grenoble, 1994.

[DV93] W. Ding and V. Van Dongen. Border tiling: for efficient loop execution on distributed memory machines. In *World Transputer Congress'93 Aachen*, 1993.

[GH86] A.G. Geist and M.T. Heath. Matrix factorization on a hypercube multiprocessor. In *Hypercube Multiprocessors*, pages 161–180, 1986.

[GR88] D. C. Grunwald and D. A. Reed. Networks for parallel processors: measurements and prognostications. In *Third Conference on Hypercube Concurrent Computer Applications*, pages 610–619, January 1988.

[Iri87] François Irigoin. *Partitionnement des boucles imbriquées, une technique d'optimisation pour les programmes scientifiques*. PhD thesis, Ecole Nationale Supèrieure des Mines de Paris, 1987.

[KCN90] C-T. King, W-H. Chou, and L. Ni. Pipelined data-parallel algorithms: Part II–design. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):486–499, October 1990.

[Meg93] G. M. Megson. Mapping a class of run-time dependencies onto regular arrays. In *International Parallel Processing Symposium*, pages 97–107, IEEE, Newport Beach, CA, April 1993.

[MF86] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transaction on Computers*, C-35(1):1–12, January 1986.

[MR90] S. Miguet and Y. Robert. Path planning on a ring of processors. *Intern. J. Computer Math.*, 32:61–74, 1990.

[RF90] S. V. Rajopadhye and R. M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, June 1990.

[RS91] J. Ramanujam and P. Sadayappan. Tiling multidimensional itereation spaces for non shared-memory machines. In *Supercomputing 91*, pages 111–120, 1991.

[Van93] V. Van Dongen. Loop parallelization on distributed memory machines: problem statement. In *Proceedings of EPPP*, 1993.

[Wol87] M. Wolfe. Iteration space tiling for memory hierarchies. *Parallel Processing for Scientific Computing (SIAM)*, 357–361, 1987.

Unité de recherche INRIA Lorraine, Technôpole de Nancy-Brabois, Campus scientifique,
615 rue de Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur

INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)

ISSN 0249-6399