Two-Dimensional Orthogonal Tiling: from Theory to Practice

Rumen Andonov & Hafid Bourzoufi
ISTV, Valenciennes, France
[bourzouf|andonov]@univ-valenciennes.fr

Sanjay Rajopadhye
IRISA, Rennes, France
rajopadh@irisa.fr

Abstract

In pipelined parallel computations the inner loops are often implemented in a block fashion. In such programs, an important compiler optimization involves the need to statically determine the grain size. This paper presents extensions and experimental validation of the previous results of Andonov and Rajopadhye on optimal grain size determination.

Keywords: *iteration space tiling, coarse grain pipelining, SPMD programs, distributed memory machines, communication-computation overlap, nonlinear optimization, supernode partitioning, automatic parallelization, macro-systolic arrays.*

1. Introduction

Tiling the iteration space [7, 12, 15] is a common method for improving the performance of parallel loop programs executed in SPMD (Single Program Multiple Data) fashion on a DMM (distributed memory machine). It may be used as a technique in parallelizing compilers (see [6, 11] where it is also called coarse-grain pipelining) as well as in performance tuning of parallel codes by hand (see also [9, 13, 14]). 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 communications are performed by send/receive calls and the code for the body contains no calls to any communication routine.

The *tiling problem* can be broadly defined as the problem of choosing the tile parameters (notably the shape and size) in an optimal manner. It may be decomposed into two subproblems: choosing an "optimal" tile shape [3, 4], and finding an "optimal" tile size [1, 6, 8, 11] (some authors attempt to resolve both problems under some simplifying assumptions [12, 13]). For the former problem, the communication cost is approximated by the number of dependency vectors crossing a tile boundary. The latter problem assumes that the tile shape is first given and then seeks to minimize the total execution time. In general, this appears to be a hard discrete non-linear optimization problem and researchers use different heuristics. Currently, there is no consensus on how to automatically obtain the "ideal" solution in the general case. This does not however, exclude particular classes where optimal solutions can be found analytically.

One such class is that of two-dimensional uniform dependency loops, whose iteration space is a parallelogram, and which admit *orthogonal tiling* (i.e., whose dependencies are such that tiles whose boundaries are parallel to the domain boundary are valid). Although this may seem very restrictive at first sight, similar assumptions are tacitly made by most researchers treating the problem. For example, the Fortran 90 and Paradigm compilers automatically determine the block size for coarse grain pipelined loops [6, 11], assuming that the iteration space is rectangular. Moreover, although they treat arbitrarily nested loops, it is only the two innermost loops that are tiled. These assumptions thus impose similar restrictions.

Recently, Andonov and Rajopadhye [1] modeled such programs as uniform recurrence equations (UREs), and showed how the methods of systolic array synthesis can be profitably used for optimal tiling. The key observation is that independent of the original dependencies, the tiled computation for *all* programs in this class can be abstracted by a single, specific URE. An analysis of this URE using a realistic model of the communication costs of general purpose multiprocessors yields a particular nonlinear optimization problem. This problem admits an analytic solution and gives a closed form formula for the optimal tile size.

In this paper we extend these results and also report on the experimental validation of the techniques. The main contributions are as follows.

- **Optimal processor count:** We show that in addition to the optimal tile size, the number of processors that minimizes the running time can also be determined analytically.
- **Generalization:** The machine model of [1] considers only a specific kind of system call (the so called *block-*

ing send and receive). For better performance, latency hiding mechanisms such as overlapping communication and computation [5], copy avoidance, etc. are often used. We extend the method to address such variations of the code.

We also show that the model used by the Fortran 90 and Paradigm compilers is a particular instance of that of [1].

- **Experimental Validation:** We perform detailed experiments on the Intel Paragon that validate the theoretical results of Andonov and Rajopadhye [1] and the new extensions presented here. The results show that the three models are very close in practice, and highlight the importance of accurately estimating machine parameters.

In the remainder of this paper is organized as follows. We first summarize (in Section 2) the Andonov-Rajopadhye results [1]. The following three sections address each of the above points, and we conclude in Section 6.

2. Optimal Orthogonal Tiling of 2-D UREs

The approach is an extension of systolic array synthesis techniques.

1. Partition the iteration space into orthogonal tiles of size $r \times s$. Leave r and s as free variables.
2. "Cluster" the recurrence into tiles, yielding a new uniform recurrence over a new domain. Each tile is considered as an atomic computation.
3. Apply systolic space-time transformations, yielding a virtual (linear) systolic array. Implement this array on a fixed size array (or ring) of p processors (by performing multiple passes if necessary).
4. Relax the systolic timing model to account for practical machines, and obtain a formula for the total running time of the final implementation. This formula will be expressed as a function of r and s, as well as other parameters of the program and architecture.
5. Choose the tile size that minimizes this running time by solving the corresponding optimization problem. This is a discrete (non-linear) optimization problem with two variables, and special techniques are used to solve it.

Given that we address programs that permit orthogonal tiling, it is easy to show that the the result of Steps 1 and 2 above can always be transformed to the following URE (under the standard assumption that the tile size is large compared to the dependency vectors).

$$\begin{aligned} &\forall\ [j,k]\ \text{ s.t. } 0 < j \leq M',\ 0 < k \leq N' \\ &X[j,k] = f(X[j-1,k], X[j,k-1]) \end{aligned} \tag{1}$$

Each point, $[j,k]$ represents a tile, $M' = M/r$ and $N' = N/s$ (the original recurrence being defined over an $N \times M$ parallelogram. There may be an additional (diagonal) dependency, but it can always be combined with one of the others. We call the dependency graph of this URE the *tile graph*.

It is well known that the space of systolic solutions for a given algorithm is not usually very large and can be exhaustively explored even for arbitrary UREs. Our URE above is particularly simple. Its optimal schedule is given by the vector $[1,1]$, i.e., tile $[j,k]$ is computed at step $j+k$. Moreover, in order to have nearest neighbor interconnections, it is known [16] that there are only four possible projection directions (i.e., allocations of tiles to processors). Of these, the $[1,-1]$ direction conflicts with the schedule and $[1,1]$ yields an array with only 50% efficiency. Hence, we really have only two choices: projection along either $[1,0]$ or $[0,1]$. Both of these yield very similar arrays. They are both linear and each processor repetitively executes the tile body, each time keeping one set of results, and sending another set to the next processor.

In studying the behavior of such a (virtual systolic) array on a DMM we now model the performance of a real parallel machine, where one computation does not take the same time as one communication. To do this, we will use the following terminology.

- Two tiles are said to be *successive* if one depends directly on the other.
- The *tile period*, P_t is the time between corresponding instructions in two successive tiles executing on the *same* processor.
- The *inter-tile latency*, L_t is the time between corresponding instructions in two successive tiles on *adjacent* processors.

The total running time, $T(r,s)$, of the parallel tiled code can now be formulated using some simple analysis. The optimal tile size is obtained by minimizing this execution time. More precisely, we have the following optimization problem.

Minimize $T(r,s)$, defined as

$$T(r,s) = \begin{cases} M'P_t \geq pL_t & \Rightarrow \quad \dfrac{N'M'}{p}P_t + (p-1)L_t \\ M'P_t \leq pL_t & \Rightarrow \quad M'P_t + (N'-1)L_t \end{cases} \tag{2}$$

subject to

$$1 \leq r \leq M \text{ and } 1 \leq s \leq N/p \tag{3}$$

Observe that our optimization problem has been formulated even before we consider the particular machine model or the details of the code executed on each tile.

This separation will play a crucial role in our extension of the method to different variations of code, and also in our ability to model the approaches proposed by other authors.

2.1. A Closed Form Solution

If the tile code is a (`receive-compute-send`) loop with blocking communication calls, and we use a refinement of the standard affine model for communication cost [2], the tile period is given by

$$\begin{aligned} P_t &= \beta_r + \tau_c r + \tau_a rs + \beta_s + \tau_c r \\ &= \beta_r + \beta_s + 2\tau_c r + \tau_a rs \end{aligned} \tag{4}$$

Here, β_r and β_s account for the two OS calls for receive and send respectively, $\tau_c r$ is the time to copy a message of size r from (to) user memory to (from) system memory, and $\tau_a rs$ is the time to execute rs iterations of the loop (one tile body).

Now, consider the latency between tiles, L_t. Assume that two (successive) tiles start simultaneously on adjacent processors. Because of the dependency between the tiles, the `receive` on the second tile will block until the first tile is finished, its results are sent, they arrive at the other, and are copied into user memory (see [1] for details). Hence,

$$L_t = P_t + \tau_t r - \beta_r = \beta_s + 2\tau_c r + \tau_a rs + \tau_t r \tag{5}$$

Observe that because the receive call is overlapped, we subtract β_r, and the $\tau_t r$ accounts for the transmission time. Substituting Equations 4 and 5 into 2, $T(r,s)$ is as given in Eqn. 6 below, yielding the subsequent result.

$$\begin{aligned} T(r,s) &= \frac{MN(\beta_s+\beta_r)}{psr} + (p-1)(\tau_a s + \tau_t + 2\tau_c)r \\ &\quad + \frac{2MN\tau_c}{ps} + (p-1)\beta_s + \frac{MN\tau_a}{p} \end{aligned} \tag{6}$$

Theorem 1 *[AR'94] The optimal tile size is unique and given by*

$$\begin{cases} [r_{\text{opt}}, \frac{N}{p}] & \text{if } pM(\beta_r+\beta_s) \geq (p-1)N\tau_a \\ [1, s_{\text{opt}}] & \text{otherwise} \end{cases}$$

where s_{opt} and r_{opt} are the integer approximations of s^* and r^*, respectively defined as follows.

$$s^* = \sqrt{\frac{MN(\beta_r+\beta_s+\tau_c)}{(p-1)p\tau_a}} \tag{7}$$

$$r^* = \sqrt{\frac{pM(\beta_r+\beta_s)}{(p-1)(N\tau_a + p\tau_t + 2p\tau_c)}} \tag{8}$$

$$\approx \hat{r} = \sqrt{\frac{pM(\beta_r+\beta_s)}{(p-1)N\tau_a}} \tag{9}$$

3. Optimal Processor Count

Theorem 1 determines the optimal tile size as a function of some machine and program parameters. We now show how the number of PEs can be chosen optimally.

Let us start by the analysis of Theorem 1. Note that the condition $pM(\beta_r+\beta_s) \geq (p-1)N\tau_a$ is simply a constraint on the aspect ratio of the original index space, $\frac{N}{M}$, relative to the constant, $\frac{p(\beta_r+\beta_s)}{(p-1)\tau_a}$. For certain aspect ratios, a single pass program with r_{opt} rows in each tile is optimal, and for others, we should use a multi-pass program, with one only row per tile. However, usually, $\beta_r + \beta_s \gg \tau_a$, and the latter case is rare. Moreover, our experimental results later show that even the optimal solution requires multiple passes, the benefits are not significant. We therefore assume that the optimal tile size satisfies $s = N/p$.

We further assume that that $(p-1)/p \approx 1$. This is reasonable, since it is here that optimizing the number of processors is practical (otherwise, p is relatively small, and one would probably need as many processors as possible). Now, Eqn. (9) can be further approximated as $\hat{r} \approx \widetilde{r} = \sqrt{\frac{M(\beta_r+\beta_s)}{N\tau_a}}$. As a result, the optimal running time can be expressed as a function of p as follows.

$$\begin{aligned} \widetilde{T}(p) &= \frac{M(\beta_r+\beta_s)}{\widetilde{r}} + \frac{MN\tau_a}{p} + (\tau_t\widetilde{r} + 2\tau_c\widetilde{r} + \beta_s)p \\ &\quad + N\tau_a\widetilde{r} - \beta_s + 2M\tau_c \end{aligned} \tag{10}$$

$\widetilde{T}(p)$ is a strongly convex function and attains its minimum at

$$p^* = \sqrt{\frac{MN\tau_a}{2\tau_c\widetilde{r} + \beta_s + \tau_t\widetilde{r}}} \tag{11}$$

4. Extensions

As noted before, the theoretical results of Section 2 were formulated only in terms of two functional parameters of a tile namely its *period*, P_t, and its *latency*, L_t. In other words, the global optimization problem to solve is always that of equations (2-3); all variations and extensions can be resolved by appropriately determining these two functions. We now illustrate the advantages of this modular approach by first showing that the models of the Fortran D and Paradigm compilers are subsumed by an appropriate choice of our machine parameters. We then illustrate how an optimized version of the code can also be described in the same framework.

Hiranandani et al. [6] present one of the earliest work on incorporating such analysis into a parallelizing compiler. They use a simple model in which all communication costs are modeled by a fixed overhead, β (independent of the message volume). They also assume that the iteration space is square ($N = M$). This corresponds to a particular case in our model where $\tau_c = \tau_t = \beta_r = 0$, and $\beta_s = \beta$. Then, Equation 8 becomes $r^* = \sqrt{\frac{pM\beta}{(p-1)N\tau_a}}$, which is precisely their result (extended to rectangular iteration spaces).

Palermo et al. [11] present a more refined model which accounts for the transmission rate but they do not make a distinction between time spent on the main processor from the time by the communication co-processor (i.e., they do not separate the τ_t and τ_c terms). Their model is also more general in that they consider the possibility of an outer loop (their iteration space may have more than two dimensions, but they tile only the innermost two loops). They do not allow a multiple pass solution, but do consider the effects of *chaining,* namely the case where the final results of the pipelined computation are used in the next outer iteration. For the case when there are only two loops to pipeline, their running time can be simplified as follows.

$$T(r) = \frac{2M\beta}{r} + (p-1)(N\tau_a + p\tau_t)r + (p-2)\beta + \frac{MN\tau_a}{p}$$

Note that if we neglect one β term above, this is identical to Equation 6, if we substitute $\tau_c = \beta_r = 0$ and $\beta_s = \beta$. Hence their model is a particular case of the Andonov-Rajopadhye model, where the cost of copying data to/from system memory is ignored.

4.1. Modeling Code Optimizations

Copying data to/from system memory can be expensive. Some machines provide an alternative library of communication routines (the so-called non-blocking or asynchronous calls) which perform the data transfer directly from user memory. It is then up to the programmer (typically by using double-buffering) to ensure that data are not inadvertently corrupted. If this strategy is used in conjunction with tiling, the tile code typically has the form shown below. For simplicity, we consider the case where copy-avoidance is used only for the `send` calls, but other variants can be similarly modeled.

```
receive
wait(a) (* for buffer A *)
compute (* into buffer A *)
a = isend (* buffer A *)
receive
wait(b) (* for buffer B *)
compute (* into buffer B *)
b = isend (* buffer B *)
```

The period P_t of such a tile can be determined easily, and by a careful analysis of the code running on two adjacent processors (a figure would help but is omitted due to space constraints), we can also obtain the new tile latency (we use primes in the new formulæ to avoid confusion).

$$P'_t = \beta'_r + \tau'_c r + \beta'_w + \tau'_a rs + \beta'_s \tag{12}$$
$$L'_t = P'_t - \beta'_r + (\tau'_c + \tau'_t)r \tag{13}$$

This is identical to the model of Section 2, if we substitute $\tau_c = 0$, $\beta_s = \beta'_s + \beta'_w$, and $\tau_t = \tau'_t + \tau'_c$.

5. Experimental results

We now summarize the results of our experimentation on the Intel Paragon at IRISA, using the vendor-supplied NX communication library. We performed three types of experiments: measurement of machine parameters; validation of the basic Andonov-Rajopadhye model; and validation of the extensions.

5.1. On Ping-Pong Messages

From Equations (8-9) we see that the optimal tile size is sensitive to β_s and β_r, but is not critically affected by τ_c and τ_t. Previous work on tile sizing [6, 11] does not make a distinction between β_r and β_s, but the authors do acknowledge the need to measure the time for a *complete* communication event. Typically, this is measured by the so-called "ping-pong" experiments [2] where one processor sends a message to another, and waits to receive a "reply" from the other. It is argued that this measures (twice) the time of a complete message exchange and is used to obtain the standard affine formula ($\beta + \tau r$). However, since the system calls on different processors are overlapped, this does not give accurate measurements. Indeed, if β_s and β_r are comparable one can show by carefully analyzing the behavior and the dependencies (illustrated for a particular case in Figure 1) that the time to execute one send-receive pair is actually (twice) $\beta_s + (\tau_c + \tau_t)r$: the receive system call is *completely* hidden.

As a result, we designed a series of experiments to accurately measure the machine parameters. We first ensured that the Intel Paragon had an accurate and unobtrusive system call for timing measurements (`dclock`) and that the

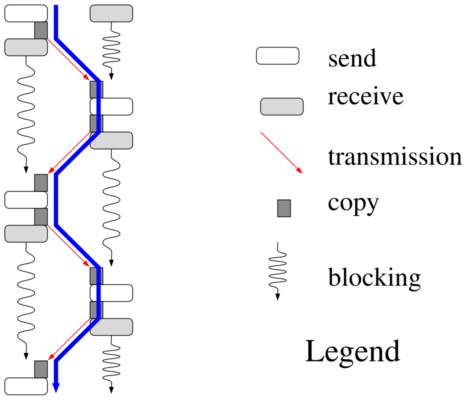

Figure 1. Ping-pong anomaly: the receive call does not appear in the critical path (in bold)

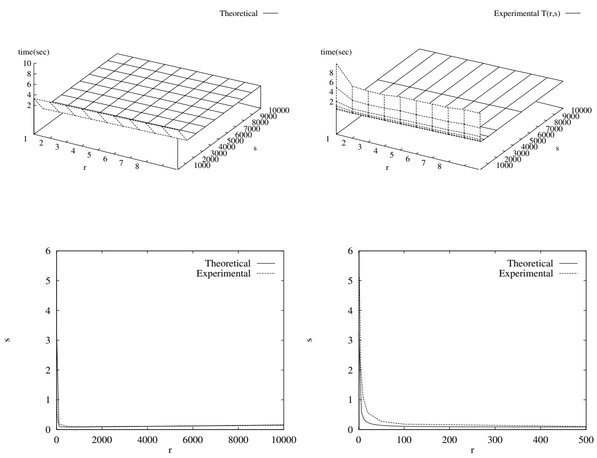

Figure 3. Multiple passes ($M, p = 10, N = 10^4$)

clock was accurately synchronized across all processors. We then used "one-sided" communications (the master only sends and the slave only receives data). We observed [10] that on the Paragon the parameters were difficult to measure accurately because of variations of the system (each node has an OS kernel running on it and many random and inexplicable factors come into play). For example if we performed one-sided communications without synchronization between the iterations the throughput was *halved* (the linear part doubled), although the β remained relatively stable. We added synchronization, and this made the τ values comparable, but changed the β values. Modeling the machine precisely remains an unresolved issue. For the purposes of this paper, we took the β values indicated by the non-synchronized experiments (the tiled program does not use explicit synchronization). Thus, $\beta_s = 32\mu$sec, $\beta_r = 47\mu$sec (for messages smaller than 8 Kbytes), $\tau_t = 0.012\mu$sec per byte and $\tau_c = 0.012\mu$sec per byte.

5.2. Validation of the Model

Next, we performed a series of experiments, with a particular parallel program (string comparison using dynamic programming) in order to validate the Andonov-Rajopadhye model. For this program, we first determined that $\tau_a = 0.98\mu$sec. We experimented with a wide range of problem size parameters, and observed that for the single pass case, (i.e., whenever $\frac{N}{M} \geq \frac{p(\beta_r+\beta_s)}{(p-1)\tau_a}$), the theoretical predictions matched the experimental results very closely. Fig. 2 (this figure is on the next page, due to a bug in the .sty file) shows a typical case. The difference in the two values is less than 5%, which we conjectured was due to cache effects. To confirm our conjecture, we ran our experiments on a contrived example that always accessed the same memory locations, and this had closer accordance with the predictions (Fig. 2 second example). In order to clearly see the effects of the other parameters, our subsequent experiments use this example.

We next performed experiments for the case when the theoretical predictions indicated the optimal solution required multiple passes with $r = 1$ ($N = 10$, $M = 10^5$), as shown in Figure 3 (the upper graphs show predicted and observed running times for a range of values of r and s, and the other two show a slice of the same data at $r = 1$ at two scales). Observe that there is agreement between the theoretical and experimental times, except in the vicinity of small tile sizes (we conjecture that this is because the instrumentation of the code is no longer unobtrusive), but that the optimal value (both theoretical and experimental) is far from the area of disagreement. This was found to hold over a wide range of problem sizes.

Note that though there is good agreement between the theoretical and experimental values, the theoretical and experimental curves are almost horizontal (the running time is almost independent of s near the optimal solution). This implies that in practice, not only is it rare to have problem size parameters which require multiple passes, but the improvement gained by this is also not significant. Hence, although our model allows us to more completely explore the solution space, it may not be practically significant. The choice made by other researchers [6, 11] to only consider one-pass tiling is pragmatic and sound.

5.3. Validation of Extensions

We conducted similar experiments on the extensions of the model to account for the code that avoids copying on sends. The experiments confirmed that the theoretical model was valid in terms of the accuracy of the predictions. One apparently surprising observation was that the

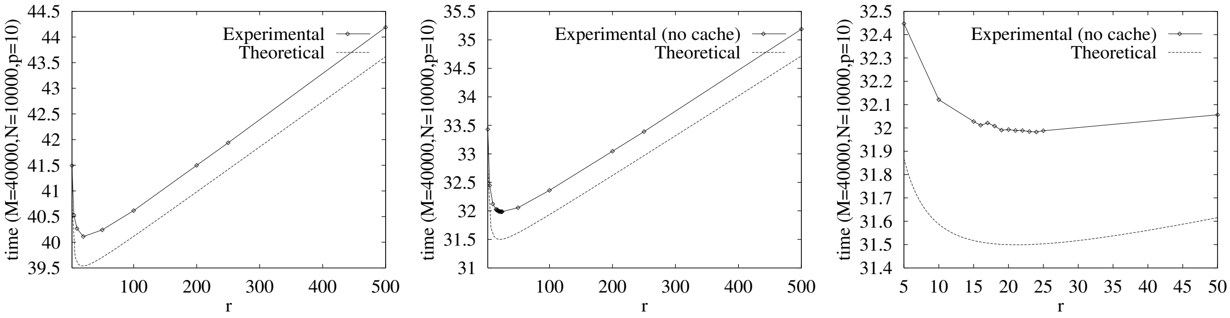

Figure 2. Running time vs tile size (for two examples), and a blown up view of the second example

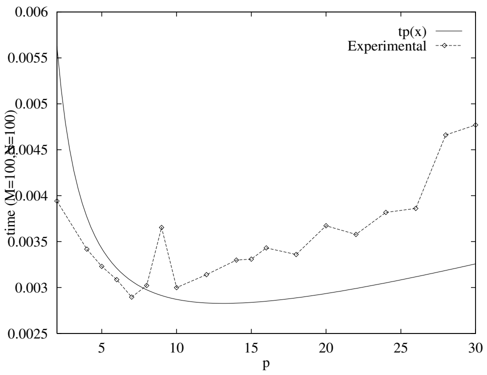

Figure 4. Optimal Processor Count

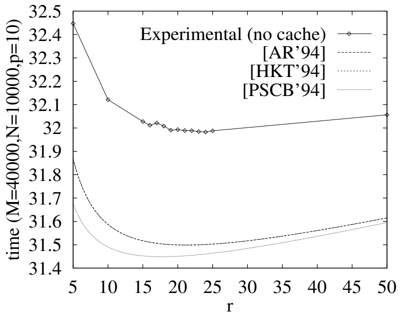

Figure 5. Comparison of the Models

proposed "improvements" do not actually reduce the running time. The straightforward code with blocking sends and receives has better performance (and this is *predicted* by the theoretical model). The reason for this is that the overhead for the additional system call (to `wait`) is far more expensive than the time for the copy that it saves.

The result on optimal number of processors turns out to be of only theoretical interest. For large enough problems sizes we can and should always use more processors. It is only in fairly contrived cases, such as solving a 100×100 problem on a few tens of processors that the theoretical predictions show an increase in the running time (Fig. 4). The predicted optimal value is around 13. It is not too close to the experimental values, which are moreover, greatly perturbed (possibly due to system events page faults, interrupts, etc.) Furthermore, the running time is just a few microseconds, and this is hardly worth optimizing.

5.4. Discussion and Comparison of Models

Fig. 5 shows, for the blown up region of Fig. 2 the experimental running time as well as the predictions based on the three models [1, 6, 11]. Note that the HKT'94 and PSCB'94 curves are *completely* indistinguishable. Moreover, the only reason that the AR'94 curve is different is because of different communication startup time. If we substitute the more accurate β values into the HTK'94 formula, it becomes indistinguishable from the AR'94 one.

This illustrates that refining the communication model to account for the network throughput, memory copy, etc., *does not significantly improve the predictions* (on current generation machines). Indeed, as even Palermo et al. acknowledge, the improvements they obtained were attributable to their better modeling of *chained* pipelines (with an outer loop) rather than due to the precision of the communication model, and this has been confirmed. We also obtained better predictability with the Andonov-Rajopadhye model, but this is attributable to resolving the anomalies of the ping-pong style measurement of communication overhead, and *not* due to the added precision of the model.

It must be noted that the effort of going through the complete development of a more general model is useful in its own right, since the relative importance of the parameters is not known beforehand. It is only after carrying the analysis through to the end that one can decide unequivocally which parameters are important. Moreover, the model is also stable if the relative importance of the parameters changes. For example, the dominance of message startup time is valid for current generation machines. If in the future, manufacturers succeed in reducing the startup costs to the point that throughput becomes critical, the model would remain valid. This is not as far-fetched as one might think. The Intel iWarp had communication processors, where the channels were allocated at compile time, and there was *no* run-time overhead for starting messages.

6. Conclusions

This paper presented extensions and experimental validation of previous results of Andonov and Rajopadhye [1]. We show how the number of processors can be chosen optimally, though this result is only of theoretical interest. We also extend the method to deal with different coding strategies to overlap computation with communication, avoidance of copying between system and user buffers, etc. Furthermore, we also experimentally validate the theoretical model as well as its extensions. Another pragmatic implication is that the code generation can be considerably simplified in practice, since a multiple pass solution, although theoretically possible, is often not worth the effort.

Pipelining techniques are applicable a fairly narrow class of programs, but for which the performance benefits are considerable. Our results highlight the fact that message startup costs are crucial and it is important to measure them accurately if one desires a model with good predictability. Similarly, caches must be accounted for by measuring τ_a precisely.

In conclusion, orthogonal tiling has nice theoretical properties (simplicity of the code generated, and existence of an analytic solution to the optimization problem) which were experimentally confirmed in this paper. Open questions are the extensions of these results to more than two dimensions, and also to more difficult cases when the orthogonal tiling is not possible theoretically. Our experimental observation on the relative importance of machine parameters may enable realistic simplifications.

References

[1] R. Andonov and S. Rajopadhye. Optimal tiling of two-dimensional uniform recurrences. Technical Report PI-999, IRISA, Campus de Beaulieu, Rennes, France, April 1996. (submitted to JPDC). This report supersedes IRISA PI-792, January 1994 ("Optimal Tiling"). Preliminary results presented at CONPAR '94.

[2] S. Bokhari. Communication overheads on the Intel Paragon, IBM SP2 and Meico CS-2. NASA ICASE Interim Report 28, 1995.

[3] P. Boulet, A. Darte, T. Risset, and Y. Robert. (pen)-ultimate tiling? *INTEGRATION, the VLSI journal*, 17:33–51, Nov? 1994.

[4] P.-Y. Calland and T. Risset. Precise tiling for uniform loop nests. In P. Cappello, C. Mongenet, G.-R. Perrin, P. Quinton, and Y. Robert, editors, *Application Specific Array Processors*, pages 330–337, Strasbourg, France, July 1995. IEEE.

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

[6] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluating compiler optimizations for Fortran D. *Journal Of Parallel and Distributed Computing*, 21:27–45, 1994.

[7] F. Irigoin and R. Triolet. Supernode partitioning. In *15th ACM Symposium on Principles of Programming Languages*, pages 319–328. ACM, Jan 1988.

[8] C.-T. King, W.-H. Chou, and L. Ni. Pipelined data-parallel algorithms: Part I–concept and modelling. *IEEE Transactions on Parallel and Distributed Systems*, 1(4):470–485, October 1990.

[9] 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.

[10] V. Nicolas. Etude experimentale des parametres de communication sur le intel paragon (in preparation). Master's thesis, Université de Rennes, Sept 1996.

[11] D. Palermo, E. Su, J. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed memory multicomputers. In *International Conference on Parallel Processing*, pages xx–yy, St. Charles, IL, August 1994. IEEE.

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

[13] R. Schreiber and J. Dongarra. Automatic blocking of nested loops. Technical Report 90.38, RIACS, NASA Ames Research Center, Aug 1990.

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

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

[16] X. Zhong, S. V. Rajopadhye, and I. Wong. Systematic generation of linear allocation functions in systolic array design. *Journal of VLSI Signal Processing*, 4(4):279–293, Nov 1992.