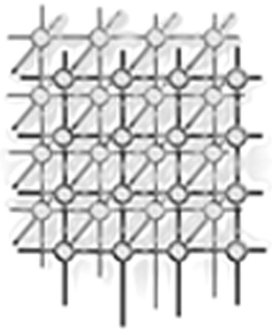
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2003; **0**:1–26 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*

Space-Time Mapping and Tiling – a Helpful Combination



Martin Griebl[1]*, Peter Faber[1], Christian Lengauer[1]

[1] *Fakultät für Mathematik und Informatik*
Universität Passau
D-94030 Passau
Germany

SUMMARY

Tiling is a well-known technique for sequential compiler optimization, as well as for automatic program parallelization. However, in the context of parallelization, tiling should not be considered as a stand-alone technique, but should be applied after a dedicated parallelization phase, in our case after a *space-time mapping*. We show how tiling can benefit from space-time mapping, and we derive an algorithm for computing tiles which can minimize the number of communication startups, taking the number of physically available processors into account. We also present how the use of a simple cost model reduces real execution time.

1. Introduction

Tiling is a well-known technique for (sequential or parallel) compiler optimization which focuses on the efficient execution of nested loops. Each statement in a loop body has multiple instances, one for each loop iteration. Following Feautrier [11], we call these instances *operations*; the set of all operations of a statement is the *index set* of this statement. The general task of tiling is then to coalesce operations, i.e., to partition the index sets, and the application domain determines which operations should be coalesced.

The application domain addressed in this paper is tiling for parallelization. In the past, tiling has either been used *as the* parallelization method *itself*, or it has been applied to the sequential input program in order to increase granularity *before* a parallelization phase [4].

In contrast, we suggest to apply tiling *after* a parallelization phase. In our case, we employ loop parallelization in the polytope model [15, 28], which is based on the concept of *space-time mapping*, and extensions thereof (e.g., the polyhedron model [17]).

*Correspondence to: Fakultät für Mathematik und Informatik, Universität Passau, D-94030 Passau, Germany
griebl@fmi.uni-passau.de

Received September 18, 2001
Revised May 29, 2002
Accepted November 13, 2002

```
DO i=0,n
  DO j=0,n
    A[i,j] = ... B[i-1,j] ...
    B[i,j] = ... A[i,j-1] ...
  END DO
END DO
```

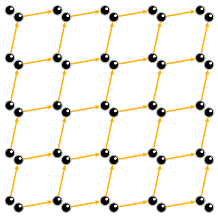


Figure 1. Per-statement view is crucial

The price for tiling after space-time mapping is that we inherit the input constraints from the model: allowed as input programs are arbitrarily nested loops with arrays as the only data structure. All array indices should be affine expressions in the loop indices and structure parameters. (If they are not, the dependence analysis over-approximates the existing communications, and the edge of our method is lost. The latter is also true for programs with arbitrarily dynamic control flow).

However, compared to the usual tiling restrictions (cf. Section 4), these constraints are a relaxation, not an additional limitation.

Technically, we restrict ourselves additionally to *parallelepipeds*, i.e., to tiles in n-dimensional space that are generated by n families of parallel hyperplanes [25].

1.1. Two motivating examples

Let us motivate a first aspect of our approach with a very small and simple example.

Example 1. Consider the code in Figure 1. The program is perfectly nested and has two uniform dependences with distance vectors $(1, 0)$ and $(0, 1)$. The according operation dependence graph is also given in Figure 1. There, every pair of bullets represents the two body statements at some iteration vector (i, j).

For that dependence graph, state-of-the-art tiling techniques determine the rectangle to be a valid tile shape. This solution is even optimal (with respect to communication minimization), if the dependence graph is a uniform dependence grid.

Tiling techniques then construct the tile dependence graph by converting the inter-operation dependences to an inter-tile level, and finally lay out the tile dependence graph in space p and time t, as shown in Figure 2.

With this solution, every processor initiates one communication per executed tile, indicated by the bold arrows in Figure 2. Hence, the total number of communications per processor is linear in the number of tiles per processor.

In contrast, our approach will find a solution without any communication: the dependence grid dissolves into an independent set of dependence chains if we treat the statements individually (cf. Example 4).

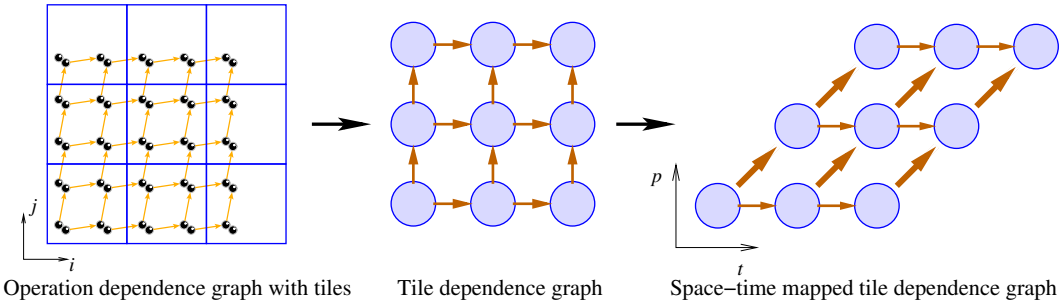


Figure 2. Traditional way of tiling

Example 2. As a second example, we take the LU decomposition algorithm in Figure 3. We chose a syntactic representation of the algorithm that avoids the unnecessary reuse of variables; this reduces the number of dependences. We are not going to analyze the algorithm in detail – its most important property is its imperfectly nested structure.

Typical current tiling techniques do not consider imperfectly nested programs at all (more detail about related work is given in Section 4). Hence, these methods are not suitable for such an implementation of LU decomposition.

To overcome this limitation is one of the main motivations for our approach.

1.2. Goals

The two concrete examples mentioned give rise to the following more general goals, which we present and pursue in two major parts in this paper:

- In the first part (Section 2), we give a brief review of space-time mapping and demonstrate how *existing* tiling methods can benefit from space-time mapping.
- In the second part (Sections 3–7), we reduce communication cost by exploring in more detail the two different types of loops after space-time mapping: we present mechanical methods for loops in space (an optimal solution in Section 5 and a heuristics in Section 6), and a (currently) semi-automatic procedure for loops in time (Section 7).
- Note that, while reducing the number of communication startups, we match the number of required virtual processors with the number of physically available processors – a task which is typically performed by a subsequent brute-force rectangular block or cyclic “tiling”.

```
DO I1 = 1,n
  L(I1,1) = A(I1,1)
END DO
DO J1 = 1,n
  U(1,J1) = A(1,J1)/A(1,1)
END DO
DO J = 2,n
  DO I = J,n
    SUM[J,I] = 0
    DO K1 = 1,J-1
      SUM(J,I) = SUM(J,I) + L(I,K1)*U(K1,J)
    END DO
    L(I,J) = A(I,J) - SUM(J,I)
  END DO
  U(J,J) = 1
  DO I2 = J + 1,n
    SUMM[J,I2] = 0
    DO K = 1,J - 1
      SUMM(J,I2) = SUMM(J,I2) + L(J,K)*U(K,I2)
    END DO
    U(J,I2) = ( A(J,I2) - SUMM(J,I2) )/L(J,J)
  END DO
END DO
```

Figure 3. LU decomposition

2. The Consequences of Tiling after Space-Time Mapping

First, let us briefly review the concept of space-time mapping which, at the outset, has nothing to do with tiling. However, we shall see soon that existing tiling methods for parallelization can take advantage of a previously applied space-time mapping. E.g., the applicability of such methods is extended quite naturally to imperfectly nested loops.

2.1. The concept of space-time mapping in the polytope model

Methods based on the polytope model take as input an arbitrarily nested sequential loop program. Only arrays are allowed as data structures, and only loops and conditional branches as control structures.

They return a pair of two multidimensional and (piecewise) affine functions per statement: a *schedule* and an *allocation*. For every operation in the program, they determine at which logical time and on which virtual processor the operation is executed, respectively. The pair consisting of the schedule and the allocation is called the *space-time mapping*.

The final output of the parallelization is a loop program which enumerates the image of the original operations under the space-time mapping.

```
DO t1=0,3*n-4
  DO PAR p1=1,n
    DO PAR p2=0,n
      IF (0 = t1 .AND. 0 = p2) THEN
       L(p1,1)=A(p1,1)
      END IF
      IF (...) THEN
       U(1,p1)=A(1,p1)/A(1,1)
      END IF
      :
      :

    END DO
  END DO
END DO
```

Figure 4. Target structure of LU decomposition

From the model point of view, this output loop program is perfectly nested and consists of sequential and parallel loops – even if the final program enumerates the same set with imperfectly nested loops in order to be efficient [32].

Example 3. If we apply a space-time mapping to the code in Figure 3 (e.g., with LooPo [21], a prototype parallelizer based on the polytope model), we obtain an (intermediate) target program, which is a perfect loop nest whose body statements are guarded by individual conditions (see Figure 4).

Since this program is perfectly nested, we can apply any existing tiling method (provided that it can deal with the existing affine dependences). In this example, all dependences (except for the first two initialization steps) are contained in the dependence cone $(1+, 0+, 0+)$, thus allowing for, e.g., rectangular tiling.

Note that, in contrast to more brute-force methods, the nesting depth of the target program is only the maximal nesting depth of the imperfectly nested source program (instead of the number of loops in the source program [1]) and, perhaps even more importantly, the merge of the different index sets is based on the dependence structure (instead of the program text – as seen in Example 1). These two aspects are the basis for the effectiveness of our method.

2.2. The strengths, weaknesses and opportunities of tiling after space-time mapping

The previous review of parallelization by space-time mapping already highlights some advantages of tiling after space-time mapping in contrast to traditional techniques:

Unique coordinates. If we tile after the space-time mapping, all operations of the parallelized loop nest are expressed in one unified coordinate system, consisting of space and time dimensions. Hence, we need no special methods for tiling imperfectly nested loops [1, 35] (for more details cf. 4), but may reuse existing tiling techniques that have already proved useful.

Identical tiles. As an immediate consequence of the unified coordinate system, we may restrict ourselves to identical tiles for the entire space-time coordinate system.

A strength of this variant is its simplicity; the weakness is that it ignores the fact that there may be different dependences (communications) in different regions of the space-time mapped index set.

Tiling power. An alternative idea is to exploit that we have direct access to the logical execution time (the indices of the loops in time). This permits time-dependent tilings. E.g., the tile size for the dimensions in space may depend on logical time, enabling some kind of static load balancing. Also the tile shape may be different for different regions of the space-time mapped index set. This opportunity will be explored in future work.

Non-uniform dependences. The space dimensions are always fully permutable (and are extracted by more sophisticated techniques than just traditional loop skewing). Consequently, in our approach any tiling of the space dimensions is legal – also for non-uniform dependences.

Number of processors. The space-time mapping reveals the number of virtual processors required for the parallel execution. Thus, when computing the tiles after space-time mapping, we can take this number into account when determining which virtual processors must be mapped to the same physical processor.

Flexibility of parallelization. If we coalesce operations of the source program, we may give up automatically detectable parallelism unnecessarily, e.g., parallelism which is exposed by iteration graph partitioning [5] (except if we apply some specialized methods [9]), or parallelism exposed only after index set splitting [20]. Our approach can overcome this weakness of traditional techniques.

Another limitation of current tiling techniques is that they view the loop body as indivisible (see Example 1). Tiling on a per-statement basis on the other hand would immediately solve the problem of tiling imperfect loop nests.

In contrast, parallelization methods based on the polyhedron model usually treat each statement of the body individually. It turns out that, if we apply a statement-based parallelization method before tiling, we may end up with programs which require orders of magnitude fewer communications, compared to if we apply tiling without or before another statement-based parallelization method (see Example 4).

Example 4. Consider again the program together with its operation dependence graph in Figure 1. If the body is considered indivisible (as in Example 1), every processor initiates one communication per executed tile. Hence, this incurs linearly many communications in the number of tiles per processor (Figure 2).

However, if we consider the statements individually, the uniform dependence grid dissolves into independent dependence chains parallel to the diagonal (Figure 1, right). So, if we first apply the statement-based space-time mapping method, we can eliminate all communications by placing the operations of the first and the second statement on processor $\pi_1(i, j) = j - i + 1$ and $\pi_2(i, j) = j - i$, respectively (Figure 5). Of course, a subsequent tiling does not introduce any new communications. Hence, we obtain a parallel program with no communication at all.

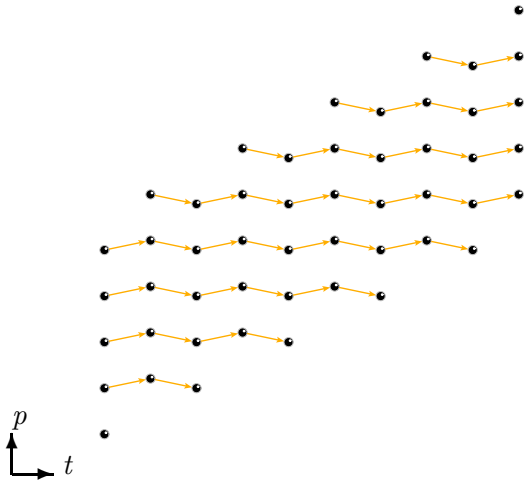


Figure 5. After space-time mapping

Note that the space-time mapping embeds a collection of, in general, differently dimensioned source index sets into a common target coordinate system (of sufficient dimensionality). After this embedding, our method does not distinguish the elements of this collection any more – which is probably its most serious limitation. Hence, in extreme cases, some statements with small source index sets might enforce a globally unsuitable tile shape. In order to avoid this, we could apply index set splitting to the target program, and use different tilings for the different parts of the index set. However, since we have not met such a situation in practice so far, we leave this aspect for future work.

2.3. Summary

Technically, the suggested combination of tiling after space-time mapping just replaces the preprocessing step of the usual tiling procedure: instead of a unimodular transformation (loop skewing) which converts the source loop nest to a fully permutable one (as far as possible), we use the more general affine space-time transformation (including all related sophisticated techniques, e.g., index set splitting [20]).

At the same time, this new combination also improves the applicability (Example 3) and/or the quality (Example 4) of existing tiling techniques.

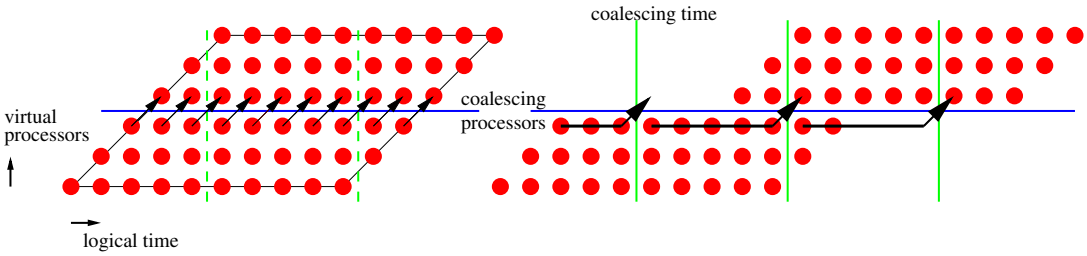


Figure 6. Target space before (left) and after (right) tiling the time dimension

3. Tiling Space and/or Tiling Time

So far, we have treated the mapping to space and time in unison. Let us now consider separately the space and time loops. We shall see that tiling space loops reduces the number of communication *partners* per communication phase, and tiling time loops reduces the number of communication *phases*. Both variants can additionally reduce the total communication *volume*.

Tiling space (processor) dimensions means aggregating neighboring virtual processors in blocks and mapping them to a common physical processor (this is also often called partitioning). Since all dependences are carried by time loops, we have no restrictions on the tiles for the space dimensions. However, there are inter-processor dependences (i.e., communications) between any two successive time steps. In this context, our task is to find tiles which lead to a minimal number of communication partners (cf. Section 5).

Note that, at this point, we ignore the communication volume. The reason is that the preceding space mapping already aims at a global minimum of the amount of data that needs to be communicated. However, it does not take into account startup costs.

A startup occurs every time a processor sends a message and, in the case of point-to-point communications, for every partner. Thus, it is a more precise measure of the communication cost (which we aim to reduce) than the number of synchronizations [4, 27, 29].

Since startup costs have been neglected during the space mapping, we must – and may – focus on this aspect during space tiling. The communication volume will become part of the discussion again in Section 7, since tiling time can, in general, change the communication volume, and this aspect is not yet considered by the scheduler.

Tiling time dimensions seems less intuitive, at first glance. The reason why it is useful is that, even if the granularity is increased by tiling parallel dimensions, we still have to issue communications between any two (logical) time steps (see the arrows in Figure 6, left)!

We avoid this by way of message vectorization: messages originating from the same tile in the time dimensions of the parallelized program are combined into a single message. Aggregating a set of logical time steps to a single step (a kind of “superstep”) reduces the number of communications: they

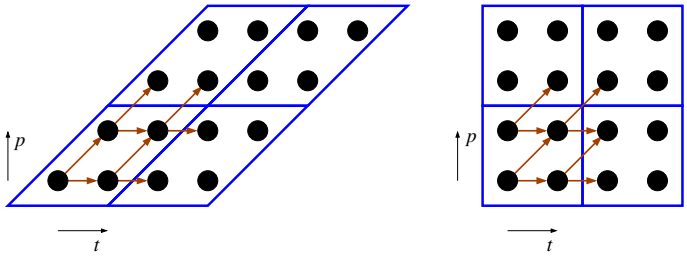


Figure 7. Change of communication startups when skewing the tiles

only take place between two supersteps as indicated in Figure 6, right. This satisfies the key restriction in the definition of tiles by Wolfe [37] which requires that tiles run to completion without preemption.

Of course, this enforces a possible postponement in the scheduling of dependent tiles, compared with the original parallel program; this tile scheduling is a typical subtask of existing tiling methods. In our method, this aspect of the schedule is program-independent: one simply has to skew every space dimension (the physical processors) into time with factor 1. (For more details cf. Section 7.)

Tiling (partitioning) space and time dimensions separately might restrict the search space in comparison to tiling space and time dimensions in one step – however, this is not the case for our approach! Placement algorithms typically aim at reducing the communication cost [10, 14, 16], and we go by the premise that they do so successfully. Hence, we require that the tiling must honor the allocation, i.e.: if the allocation maps two operations to the same (virtual) processor, then the tiling must keep them on the same processor. With this constraint, we have the following property.

Lemma 5. *For every legal parallelepiped tiling τ which respects a given allocation π and which causes c communication startups with a total volume v, there is a tiling τ' in which space and time dimensions are orthogonal to each other and which also causes c communication startups with volume v.*

Sketch of proof. Since τ respects π, we have t spanning vectors $\{v_1, \cdots, v_t\}$ of τ which are only in time, for t the number of time dimensions. We create a new tiling τ' by skewing the remaining spanning vectors of τ (we do not skew the index set) so that they are orthogonal to the subspace spanned by $\{v_1, \cdots, v_t\}$. Note that this skewing does not change the number of communication partners or directions – nor the volume –, but the number of communication startups (see Figure 7). However, due to the correctness criterion for τ, all dependences are inside the pointed cone spanned by the spanning vectors of τ; hence, we can postpone the receiver (without generating a cycle) until all data for it have been computed, and again initiate only a single communication for every pair of communication partners.

Aside: Note that, in general, a parallelepiped tiling can only be replaced by two unimodular transformations and a rectangular tiling in between [40, Section 4.5]. In contrast to that solution, we

need only apply a single unimodular transformation since the framework of space-time mapping allows us to work in a new skewed coordinate system.

Our suggested approach is then as follows: After space-time mapping, we first tile the space dimensions. This minimizes the number of communication partners and maps the operations to the physically available processors. It cannot modify the number of time steps at which communication takes place.

Subsequently, when we know the space tiles, we tile the time dimensions. This reduces the frequency of the communication startups with the just computed communication partners. Note that this second step does not change the communication partners, but it adapts the granularity of the parallelism to the given parallel architecture (for which we need the size of the space tiles).

In general, this separation cannot lead to a globally minimal number of communication startups because the two phases interact: first, sometimes a space tiling may disallow a desired time tiling and, second, the size of the space tiles influences the optimal size of the time tiles.

However, if a given space tiling does not prevent time tiling (as is frequently the case), we obtain the minimal number of communication partners by our separation – for a given number of processors to be used by the parallel program (cf. Section 5).

4. Related Work

In principle, all existing tiling methods, based on whatever cost function, can be applied after as well as before space-time mapping.

The first papers on tiling date back to the mid-eighties [33, 36]. In 1988, Irigoin and Triolet [26] presented a sufficient criterion for the validity of tiling (or supernode partitioning, as they called it). In the Nineties, much research was done to find good (or even optimal) sizes of rectangular or rhomboidal tiles [3, 4, 30], and then to adapt the shape of the tiles to the dependences [6, 23, 25, 34, 38, 39, 41], resulting in parallelepipeds as tiles.

Most of these methods have an initial phase which converts the source loop nest into a fully permutable loop nest, i.e., they first compute a skewing which makes all dependence directions non-negative in every component [6, 36]. This ensures the correctness for the subsequent optimizing tiling phase. I.e., the task of finding optimal tiles can be solved independently of correctness issues. However, this separation may destroy all parallelism in the presence of non-uniform but affine dependences.

In addition, most of the work cited uses dependence polyhedra as the most precise dependence description. This level of abstraction results in a loss of parallelism compared to using, e.g., Feautrier's scheduling algorithm [12, 13] – which works on a precise dependence description for affine index expressions – followed by tiling.

Our main concern with all the work mentioned so far is that it has been limited to perfect loop nests. These methods can only be used for tiling arbitrarily nested loops if some preprocessing – e.g., the proposed space-time mapping – has been applied.

On the other hand, there are specialized methods for the treatment of imperfectly nested loops [1, 35]. Unfortunately, these methods have been developed specifically for cache optimization. However, tiling methods for cache optimization cannot be used directly in tiling for coarse-grain parallelism, even if both aim at improving locality. One of the main differences is the fact that tiling for coarse-grain

parallelism has an additional goal – the distribution of work – which conflicts with locality. Our application domain is tiling for parallelization for distributed memory systems. (Cache optimization on every processor is a possible *subsequent* step.)

Apart from the different purpose of these specialized methods, the approaches are also quite different from ours. Song and Li [35] use a program model which is more restricted than ours: it only allows one enclosing loop whose body contains several loops. In addition, this outermost loop is not subject to tiling. Ahmed, Mateev and Pingali [1] present a method which is technically quite different from ours: it works on a space whose dimensionality is as high as the number of loops in the program (7 for the LU algorithm). This high-dimensional domain is then a subject of tiling. In addition, the embedding of the original index sets into the product space offers a very wide range of choices, from which one is chosen heuristically.

In our approach, the dimensionality is minimal: the maximum nesting depth of the source program (3 for the LU algorithm). The embedding is the space-time mapping, which is (for a restricted application domain) even provably optimal for parallelism. This reduced dimensionality is crucial, since the time complexity of space-time mapping methods usually is exponential in the dimensionality.

In addition, as already stated, these specialized methods are not necessary in our approach.

Furthermore, we are not aware of a tiling method which minimizes the number of communication partners. This is probably due to the fact that we need a notion of real processors in order to determine the number of communication partners, and this notion only enters the picture after some kind of space-time mapping – but previous work on tiling is not based on space-time mapping.

Let us try to classify existing tiling approaches for parallelism.

A first approach which is, among others, intended for distributed systems simply ignores communication cost and minimizes only the idle time instead [8, 24, 25]. On the one hand, this estimate for the execution cost can be far from reality. On the other hand, our task is much simpler because space tiling does not affect the idle time, and for tiling time we have proved that only a very limited form of rectangular tiles is valid (cf. Corollary 14). Note that this is not a restriction, but a feature of the preceding scheduling phase: all operations are already "aligned" to logical time dimensions. Hence, we need neither consider non-rectangular tile shapes nor restrict ourselves to special shapes of iteration domains (such as, e.g., [8]), in order to find time tiles which minimize the idle time – provided the schedule is minimal, i.e., it has minimal latency.

A second approach minimizes the communication *volume* [38], not taking into account the number of startups. But, in state-of-the-art parallel machines, the communication startup cost is much higher than the cost of transferring an additional value, so this should be taken into account. In addition, as stated in Section 3, the preceding space mapping already focuses on reducing the total communication volume.

A third approach reduces the number of necessary *synchronizations* [4, 27, 29], which is a more abstract, and hence, less precise cost function than the number of startups. Some of these approaches [4, 27] are additionally restricted to rectangular partitioning, e.g., due to the limitations of BLOCK distribution in HPF. Furthermore, there are approaches [4] which do not coalesce messages, but issue one communication per variable.

In contrast, our approach computes the optimal tiles w.r.t. the number of startups and guarantees that there is at most one communication between any pair of processors at any time step.

There is also a fourth approach [2] which minimizes the execution time, based on the BSP cost model. This minimization of overall execution time is a very hard task and is therefore limited to two-dimensional iteration domains and uniform dependences with additional constraints on the tile and index set borders. Similarly, we reduce execution time w.r.t. a cost model (cf. Section 7.2), but we cannot guarantee optimality since, first, we separate space and time dimensions and, second, space-time mapping algorithms themselves do not guarantee optimality for the overall execution time. This is the price which we had to accept for extending the applicability to arbitrarily nested loops with affine dependences. Our experiments (cf. Section 7.2) show that our analytical minimum is very close to the real optimum.

5. Optimal Space Tiling

In this section we restrict ourselves to uniform dependences, and we consider only space dimensions. Under these two constraints, we obtain the minimal number of communication startups. (Note that, in our context, optimality can only be proved w.r.t. a given space mapping.) Sections 6 and 7 then drop these limitations, respectively – at the price of the need for heuristics.

5.1. The task of tile optimization

Our goal is now to derive an algorithm that determines a tiling of a space-time mapped loop program, taking into account the number of physically available processors. This tiling leads to the minimal number of communication partners, if we only allow uniform dependences in the space dimensions.

In a subsequent tiling of the time dimensions, described in Section 7, we adapt the granularity, as indicated in Section 3. If the computed space tiles do not prevent a tiling of time, we obtain the minimal number of communication startups for a given number of physical processors in the following sense: if another tiling caused fewer startups, it would be due to fewer communication partners per communication phase (which is not possible since our method constructs tiles with the minimal number of partners), or due to fewer communication phases (which we could mimic by using coarser time tiles – down to one communication per processor, which is minimal for a not embarrassingly parallel program on a parallel computer with a given number of processors).

Note that there is an algorithm which computes, if possible, a space mapping that always allows for tiling of time [19].

Since, after the tiling of time dimensions, we coalesce all messages for a given communication partner, our cost function is equivalent to minimizing the *number* of communications, ignoring the message *size*. This so-called HKT model was first used by Hiranandani et al. [22]. It may appear oversimplified, but has been shown to be quite accurate [4, 30, 31]. As explained in Section 3, this model is sufficient for computing space tiles after a space-time mapping since the space mapping already aimed at reducing the communication volume.

5.2. Technical Overview

In order to obtain the optimal tiling, we determine first the *shape* of the tile, i.e., the directions of its spanning vectors (Sections 5.3 and 5.4). Then, we compute the *form* of the tiles, i.e., constraints on the

relative or – in our case also relevant – absolute lengths of the spanning vectors (Section 5.5). The final component, the *size*, i.e., the volume of the tiles, is easy to determine at this point, since every tile is executed by one physical processor, and the number of processors – and so the number of tiles – is a (parametric) input for our method.

Remark. There are two ways for mapping tiles to a fixed number of physical processors: cyclic distribution or block distribution. Cyclic (or also block-cyclic) distributions are best suited for programs with dynamic control flow since they allow load balancing. However, they cause more communications than a block distribution. The reason is that communications are coalesced and vectorized per tile, and smaller tiles mean more tiles, i.e., more communications.

We decide for the block distribution since, first, our goal is to minimize the number of communications and, second, we focus on programs with a relatively static control flow. Programs with very dynamic control flow should not be tackled with static parallelizing or tiling methods, but with appropriate methods at run time, e.g., inspector-executor schemes.

Note that our desired block distribution requires the number of needed virtual processors as input, in order to produce the target code for the physical processors. Fortunately, this information is provided by the space-mapping.

Technically, we start with "short" dependences, i.e., dependences whose lengths do not exceed the extent of the tile in every dimension. The extensions for long and for non-uniform dependences are given in Section 6.

Also, we ignore the borders of the index space – this aspect is added in Section 5.5. I.e., up to Section 5.5 we assume that the index space contains more than one tile (indeed an unbounded number of tiles) in every dimension.

5.3. The ideal case

Let us first consider the simplest case of n linearly independent uniform dependences, where n is the dimensionality of the index space.

Remark. Dependences expressed in the sequential source coordinates are transformed by the space-time mapping to dependences in target (i.e., spatial and temporal) coordinates. Since only the spatial dimensions determine the communication partners, we simplify our terminology in the rest of this paper: when speaking of a dependence vector d, we mean d projected onto its spatial components.

Observation 6. *Let D be a set of n linearly independent uniform dependence vectors $d_1, \cdots, d_n$ in an n-dimensional index set. Then, a tile whose one-dimensional faces are parallel to $d_1, \cdots, d_n$ incurs the minimal number of communication partners.*

The reason is as follows. A one-dimensional face of a tile is the intersection of $n-1$ hyperplanes forming the tile. Hence, since every dependence d_i is parallel to one one-dimensional face, we have: d_i is parallel to all but one hyperplanes forming the tile. Thus, d_i crosses only one hyperplane. Since we assume short dependences compared to the tile size, d_i causes at most a single communication – covering all existing instances of d_i inside the tile. This local minimum (only one partner per d_i) leads to the global minimum, since all d_i are linearly independent.

Consequently, the optimal tile shape is given directly by the n linearly independent uniform dependence vectors D.

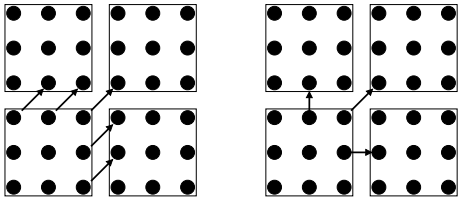


Figure 8. Dependences and communication partners

5.4. Arbitrary number of uniform dependences

Let us now consider the more typical case of m uniform dependences (for $m > n$). Here, from the set of all dependences D, we select a basis $D' = \{d_1, \cdots, d_n\}$ of the n-dimensional space and use these vectors as tile spanning vectors. We express the remaining dependences $D'' = \{d_{n+1}, \cdots, d_m\}$ as linear combinations $L = \{l_{n+1}, \cdots, l_m\}$ of D'. How many communication partners arise from a linear combination?

Since we assume "short" dependences, we can only reach neighbor tiles. Consequently, we may abstract from the precise linear combinations in L and use the vector $\bar{l}_j := (s_1, \cdots, s_n)$ as abstract representation of $l_j \in L$, where s_i is the sign of the i-th component of l_j.

Every such vector describes precisely one dependent neighbor tile (also along the diagonal). We call this representation the *relative neighbor coordinates* w.r.t. a given tile.

Let us now count the number of communication partners caused by a dependence $d_j \in D''$, which is represented by $\bar{l}_j$. For a tile to be a communication partner of the currently considered tile, the following must hold: its relative neighbor coordinates are

- 0 for those dimensions k where $\bar{l}_j[k] = 0$, and
- 0 or x for those dimensions k where $\bar{l}_j[k] = x$ ($x = \pm 1$).

All tiles matching this condition can be communication partners of the actual tile – except for the tile itself, which has neighbor coordinates $(0, \cdots, 0)$. In the worst case, $\bar{l}_j$ has no zero-entries, i.e., we get $2^n - 1$ possible communication partners (all neighbors, and not the tile itself). The maximal number of communication partners over *all* dependences is $3^n - 1$.

Example 7. Let us consider the dependence $\bar{l}_j = (1, 1)$ in Figure 8, left. The first coordinate is 1. Thus, the first dimension of the neighbor coordinate must be 0 or 1. For the second coordinate, we have the same constraint. Hence, the communication partners are (0, 1), (1, 1), (1, 0), as indicated in Figure 8 (right).

The resulting algorithm for computing the optimal tile shape in the case of m uniform dependences (for $m > n$) is as follows.

1. For every possible partitioning p_k of D into D'_k and D''_k as above do:

```
DO J = 2, n
  DO I = J, n
    X(J,I) = X(J-1,I) + Y(J+2, I) + Y(J+1, I)
  END DO
  DO I2 = J + 1, n
    DO K = 1, J - 1
      S(J,I2) = S(J,I2) + ...
    END DO
    Y(J,I2) =  S(J,I2) + Y(J+1, I2-1) + X(J,I2)
  END DO
END DO
```

Figure 9. Source program for Example 8

(a) For every vector $d_j \in D''_k$, compute the set of relative neighbor coordinates $C_{k,j}$ of the possible communication partners.
(b) Compute the union U_k of all $C_{k,j}$, with $d_j \in D''_k$, and the n unit vectors of length n (representing the communication caused by the dependences in D'_k).
(c) Set u_k to the cardinality of U_k.

2. Choose a partition p_k whose u_k is minimal.
3. The optimal tile shape is spanned by the vectors in D'_k.

The step which dominates the time complexity is the computation of the possible communication partners of cost $O(2^n)$, which is executed $\binom{m}{n}$ times. Since n is a very small integer, this is acceptable.

Example 8. Let us consider the code in Figure 9.

After space-time mapping, we obtain a perfect loop nest with one dimension in time and two in space. From the originally nine dependences only four dependence vectors must be considered for tiling (the others are made local by the allocation and, hence, cause no communication). Their projections to the space dimensions are $d_1 = (1, 1)$, $d_2 = (1, -1)$, $d_3 = (-1, 0)$, and $d_4 = (0, 1)$ (Figure 10, middle).

Now we apply the above algorithm in order to select the two optimal basis vectors for D'. As examples for all pairs out of the four dependences, we present the result after Step 1b for two selected pairs in Figure 10. If we take d_1 and d_3 as basis (right), we end up with six communication partners; if we take d_2 and d_3 (left), we have only five, which is minimal over all pairs. Hence, d_2 and d_3 are the tile's spanning vectors.

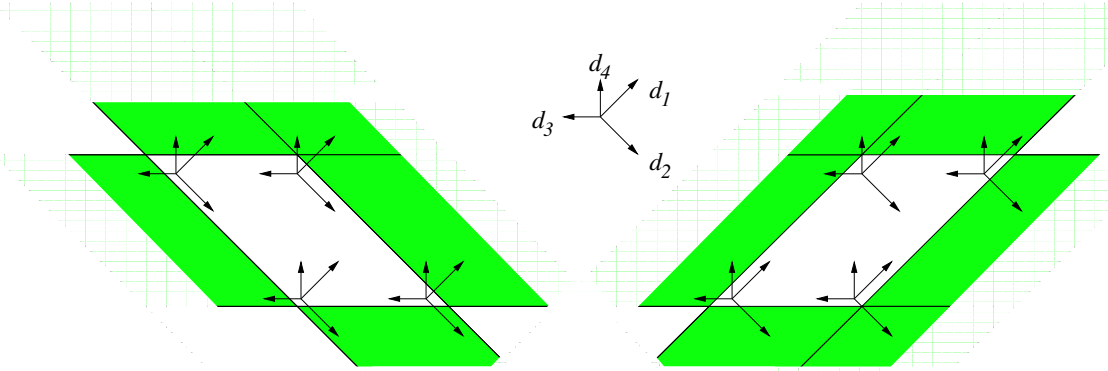


Figure 10. Communication partners for d_2 and d_3 (left) or d_1 and d_3 (right) in D'

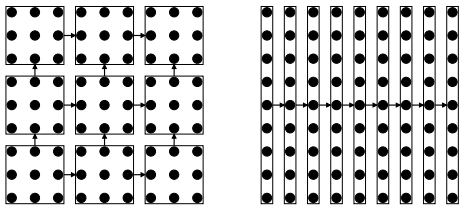


Figure 11. Square and thin but long tiles

5.5. Tile form for uniform dependences

If we do not consider effects at the border of the index set, the form of the tiles has no influence on the number of communication partners in the case of uniform dependences. Thus, the only possibility for avoiding communications due to a different tile form is at the border.

Square vs. thin but long tiles. In the presence of very fine-grained parallelism and/or a small number of processors, tiles can become quite large. In practice, we can frequently collapse a complete dimension of the index set into a single tile and still have enough parallelism (as, e.g., in LU decomposition). It turns out that this way of using thin but long tiles reduces the number of communication startups, compared to nearly square tiles. For an illustration, see Figure 11 with $\sqrt{P} * (\sqrt{P}-1) * 2$ communications in the square and $P-1$ in the non-square case, for P the number of tiles and the number of processors.

Note that, if the tile extends to all of a single dimension i, this can eliminate *multiple* communications: all those communications whose relative neighbor coordinates have a non-zero entry in dimension i.

In order to determine which tile dimensions should be extended to all of the index set and which not, we first compute the communication partners as already described. Then, we count for every dimension i, how many communication vectors have a non-zero entry in dimension i. Startup costs are minimized if we extend the dimension with the highest count.

6. Heuristic Extensions

In this section, we go beyond uniform dependences, at the price of losing optimality. We give hints on how optimality can be reached, but we focus on heuristics which are simple to implement.

First, we look at "long" dependences. The interesting new phenomenon is that the length of the tile may influence the number of communications – not only due to boundary effects. Therefore, we derive an algorithm for choosing this tile length.

6.1. Long dependences

Formally, a dependence with constant direction and a distance which is a symbolic parameter is not uniform. However, such dependences occur in real programs and should be treated more precisely than general affine dependences. These dependences are typical candidates for what we call *long* dependences. Also it might occur that a uniform dependence is longer than our desired tile width. Such a dependence also belongs to the class of long dependences.

First, we might want to consider the direction of a long dependence d as a candidate for D' (the set of tile spanning vectors). Since, per definition of *long*, d is longer than the extent of a tile, we need to cut the length of d to 1, obtaining a vector d', which we then can put into D' (if we put d into D', d is not long anymore).

6.1.1. Communication partners

Let us now compute the number *partners* of communication partners for a long dependence d: $partners = \prod_{i=1}^{n} f_i$, where

- $f_i = 1$ if d maps the hyperplane, which limits the tile in dimensions i, to a parallel hyperplane, which itself limits another tile in dimension i, i.e., starting from a border of a tile, d reaches the equivalent border of another tile, and
- $f_i = 2$ otherwise.

Example 9. Let us count the number of communication partners for the leftmost tile τ in Figure 12. In the left part, the tile width in both directions is 3, and the depicted dependence vector $d = (10, 4)$ (interpreted as a translation function, which is defined by the offset d) maps both the vertical and the horizontal boundary to the interior of another tile; thus, *partners* $= 2*2 = 4$, given by those tiles which intersect with the image of τ under d. (In the general n-dimensional case we have 2^n partners). In the right part, where the tile width in both directions is 4, the horizontal boundary is mapped to another horizontal boundary – *partners* is reduced to $2 * 1 = 2$. If we take a tile width of 2 in both directions (or 2 in the horizontal and 4 in the vertical direction), both boundaries are mapped to boundaries again, resulting in *partners* $= 1 * 1$, i.e., a single communication.

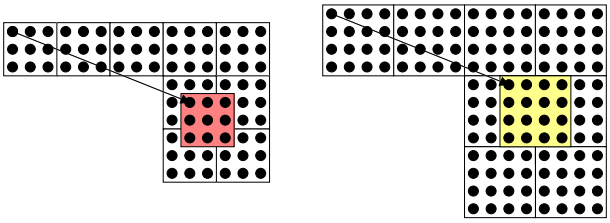


Figure 12. Long uniform dependences

The important difference to the case of Sections 5.3 and 5.4 is that the extents of a tile in the various dimensions have an important impact on the number of startups, as seen in the example.

Fortunately, this aspect can only reduce the number of startups, i.e., for our heuristics, we may first treat long uniform dependences as candidates for the tile shape directions as above, and reduce the number of startups later when we decide the form of the tile (cf. Section 6.1.2).

Remark. We can specify lower and upper bounds for the number of startups without considering the form: a long uniform dependence vector d causes one or two startups if d is a spanning vector of the tile, and between one and 2^n if not. This way, we can sometimes reduce the candidate set before minimizing the number of startups with the help of the form.

A more difficult problem is that we have to check whether any of those communication partners overlap with the communication partners computed in Section 5.3 or with the partners caused by other long uniform dependences. The only precise solution method is to count the number of communication partners using Ehrhart polynomials [7]. With this technique, one can count the number of integer points inside a parameterized polytope; the solution is given by a kind of parameterized polynomial. For simplicity, our heuristics assumes that the communication partners of long dependences do not overlap, and we leave a precise formalization for future work.

6.1.2. Appropriate tiles

Let us now derive an algorithm which computes constraints on the absolute widths of the tiles in order to reduce communication. As indicated above, we can halve the number of communication partners for a long (i.e., non-uniform) dependence d along every tile dimension i in which the dependence expands, if the i-coordinate of d (in the basis of the tile spanning vectors) is an integer multiple of the tile width in dimension i. A suitable extent is computed as follows.

Every dependence d can be expressed as $d = \sum_{i=1}^{n} \lambda_i * d_i$, where $d_i \in D'$, i.e., d_i is one of the tile spanning dependence vectors. Now, we try to factorize λ_i into $\lambda_i = s_i * a_i$, for every $i = 1, \cdots, n$ with $a_i \in \mathbb{Z}$, $d_i * s_i \in \mathbb{Z}$, and $s_i \geq 1$. The reason is as follows: s_i (the value we are interested in) is intended to be the scaling factor of the tile in the direction of d_i, i.e., $d_i * s_i$ will be the – integral – tile width in direction d_i. s_i must not be smaller than 1, because shrinking the spanning vectors would convert some uniform dependences to long dependences. Finally, from Section 6.1 we know that any integral

multiple a_i of the tile length $d_i * s_i$ halves the number of communication partners due to dependence d along dimension i w.r.t. the general case.

Example 10. Let us consider the horizontal dimension in Figure 12, where we assume a tile spanning vector of $d_1 = (1, 0)$ for the horizontal dimension, and a long dependence of $d = 10 * d_1 + 4 * d_2$. We have the following possible factorizations for 10: $(s_1, a_1) \in \{(10, 1), (5,2), (2,5), (1, 10)\}$. If we choose, e.g., $s_1 = 2$, we obtain a horizontal tile width of 2, which halves the number of communications in this direction, as mentioned in Example 9.

In order to find the optimum over all dependences, we enumerate, for every dimension i, the multi-sets S_i of all valid pairs (s_i, a_i) for all dependences. These multi-sets are finite (usually small), since $a_i \in \mathbb{Z}$ and $|a_i| \leq \lambda_i$ (since $s_i \geq 1$), and λ_i is fixed for every dependence. Then we select an $\hat{s_i}$ which occurs most frequently as first component in S_i. The resulting tile vertex $d_i * \hat{s_i}$ maximizes the number of dependences which satisfy the constraint $a_i \in \mathbb{Z}$, i.e., minimizes the number of communication startups (per time step) globally.

Example 11. Suppose we had a second long dependence $d' = (15, 8)$. We factorize 15, and see that $s_1 = 5$ appears both for d' and d, i.e., a horizontal tile width of 5 causes a quarter of the dependences (that are due to the long dependences), compared to the general case.

Note that this reduction can be achieved independently for the various dimensions. However, we need to keep (at least) one dimension which can be extended until the tile has the desired size (volume). In order to determine which dimensions we should keep, we can use a counting scheme for the various directions, which is similar to the one just described for multiple dependences.

6.2. Affine dependences

For the general affine case, we need to compute the precise number of communication partners in order to minimize the number of startups. Again, this can be solved precisely using Ehrhart polynomials: since in our approach for communication generation, we already compute the sets of all processors which send/receive values, we just have to count the number of points in these sets, which are described as parameterized polytopes.

However, since Ehrhart polynomials are difficult to handle, our implementation will use the direction vectors as an approximation and use them as candidates for D' instead – if the number of uniform candidates is less than the number of space dimensions; otherwise, we determine the tile shape based on the uniform dependence vectors.

Example 12. Let us return to our LU decomposition algorithm in Figure 3. Let us apply Feautrier's placement method [14] (as implemented in LooPo [21]), setting the desired dimensionality of the placement to 2. In this case, we obtain a placement which makes 10 out of 19 dependences local, i.e., there are 9 dependences left which cause communications. All of them are non-uniform, some are even not forward communications. We therefore peel off the first two time steps, in which the first and second statement compute some initial values which are broadcasted immediately.

The core of the program then has five remaining dependences. Their direction vectors are (2, 1+, 0), (1, 1+, 0+), (2, 1+, 1+), (1, 1+, 0), and (1, 0, 1+), where the first dimension is time and the others are the two space dimensions.

This leads to the following candidates for spanning vectors of the space tiles: $(1, 0)$, $(1, 1)$, and $(0, 1)$. The communication minimal choice thereof is $(1, 0)$ and $(0, 1)$, (using, e.g., $(1, 0)$ and $(1, 1)$ as a spanning vector leads to an additional communication partner due to the dependence in direction $(0, 1)$). Therefore, we should use rectangular tiling in the space dimensions.

7. A note on tiling time dimensions

7.1. How to tile nested time loops

Let us first check how a nest of time loops can be tiled at all.

Lemma 13. *If, in some time dimension k, at least two successive iterations I and I' are aggregated to one tile $\mathcal{T}$, then all time dimensions k' with $k' > k$, i.e., nested inside the loop at level k, must be contained in $\mathcal{T}$.*

Formally: $\big(I := (i_1, \cdots, i_k, 0, \cdots, 0) \in \mathcal{T} \wedge I' := (i_1, \cdots, i_k + 1, 0, \cdots, 0) \in \mathcal{T}\big) \Rightarrow I'' := (i_1, \cdots, i_k, i_{k+1}, \cdots, i_n) \in \mathcal{T}$ *for every I'' in the index space, where n is the maximal nesting depth.*

Sketch of proof. Multi-dimensional time can be linearized to one-dimensional time (if we allow non-affine expressions). Since, by definition, tiles enumerate successive points, $(I \in \mathcal{T}) \wedge (I' \in \mathcal{T})$ implies that all points between I and I' must belong to $\mathcal{T}$. (There is also a model-oriented proof, which we skip here.)

Note that this lemma does not claim that the tile length is 1 for every time dimension. There usually exists one time dimension which is tiled with a length greater than 1; more details on how to compute the optimal length are given in Section 7.2. A more precise statement is given by the following corollary.

Corollary 14. *There is at most one time dimension with a tile length greater than 1 and less than the maximal extent of the loop.*

Sketch of proof. By contradiction: if there were two such loops, the inner loop l would contradict Lemma 13, if the extent of the tile in the dimension of l were less than the maximum extent of l.

Consequently, we cannot – and need not – apply any sophisticated tiling method for the time dimensions. Once we have chosen a time dimension to be tiled with a length greater than 1, we just have to check that all inner time loops can be *collapsed* into one common tile, i.e., we check whether we can consider every inner loop as enumerating one dimension of the tile.

7.2. An algorithm

The above ideas lead to the following algorithm:

1. Let t be the innermost time dimension.
2. Check whether the dependences allow to *collapse* dimension t (or parts of it) into one tile.
3. If so, check for the granularity of the tile w.r.t. some cost model (see below). (Note that the tile size in the space dimensions is a necessary input for this check.)
   (a) If the granularity is still too small, then let t be the next outer loop. Go to Step 2.

(b) If the granularity is big enough, compute the optimal length of the tile in dimension t w.r.t. the cost model.

4. If the dependences forbid to collapse dimension t, use the maximal permitted length for the tile in dimension t, length 1 in the worst case.

Let us now discuss some aspects of this algorithm in more detail.

7.2.1. Constraints

Step 2 already indicates that tiling time is not always possible. Therefore, we first need to check the interference of tiling space and time dimensions. We define: a tiling has *forward communications only (FCO)*, if all communication directions are contained in the cone $(0+, \cdots, 0+)$.

Note that this limitation is similar to full permutability in the traditional tiling framework but, in our case, we only require it for the space dimensions. For the time dimensions, an even stronger property is already guaranteed by the scheduler.

In this desirable and frequent case, the time dimension for tiling as well as the according tile width can be chosen freely. If we do not have FCO, for two successive time steps two physical processors might cause a dependence cycle, which forbids to coalesce these two time steps.

With this knowledge, we can make step 2 more concrete:

2. Consider only those dependences which are not carried by one of the outer loops $1, \cdots, t-1$. For these dependences, check whether the communication direction is forward (in this case, the dependences allow to collapse dimension t).

Remarks.

- Sometimes the FCO constraint is violated only at the borders of a loop, but is satisfied by the inner iterations. In this case, we may peel off boundary iterations and compute a tiling for the remaining loop iterations.
- Note that FCO is sufficient but not necessary for arbitrary time tiling. E.g., if in some space dimension we *only* have backward communications, we also can tile time. For more details, see [18].
- There exists a placement algorithm that guarantees the FCO constraint (if possible) [19].

7.2.2. Cost model and experiments

The cost model for Step 3 must be more precise than just the number of startups. Otherwise, even in the case of FCO, we may reach the absolute global minimum of startups (one communication per processor), at the price of losing all parallelism.

Our experiments showed good results for the standard linear cost model in which a computation costs cc units of time, each communication startup costs sc and each transferred data element costs vc units of time. We assume that startup and transfer do not overlap.

We ran our experiments on an SCI-connected network of 32 nodes with 512 MByte of main memory and two Intel Pentium-III, 1000 MHz processors on each node (however, we only used one processor

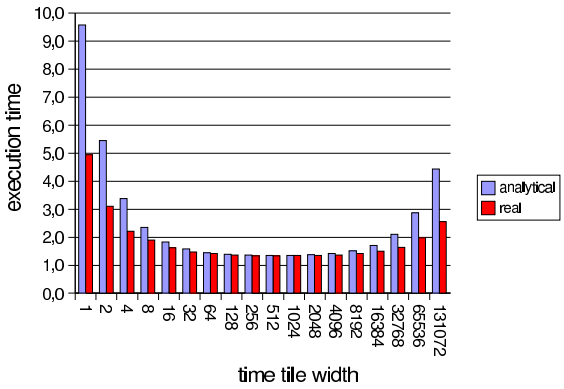


Figure 13. Analytical and experimental results for uniform dependences (Example 15)

per node for our experiments). For this machine, we found as a rough approximation $sc = 100$ and $vc = cc = 1$ (we only need relative costs). Fortunately, it turns out that these parameters need not be computed very precisely; even if the real value of a parameter in the model differs by a factor of almost an order of magnitude, the analytical result is sufficiently precise. The main reason for this behavior is that the optimal execution time usually has a very flat minimum, i.e., around the optimum, the execution time stays nearly constant for different values of the time tile width (see Figure 13). Note that the analytical execution time in Figure 13 is computed in logical units of time, which we scaled such that they fit into the same graphic with the real execution times. Furthermore, this means that the number of computations and communications may be approximated in a similarly coarse manner.

Aside. If a program is very difficult to analyze, so that the approximation of the number of computations and communications is very imprecise, we suggest to execute the first iterations of the parallel target program with the time tile width computed at compile time. During this execution, we measure the real amount of work for computations and communications, and then adapt the time tile width according to the new run-time values. However, in our practical examples, this has never been necessary. So, we leave this aspect for future work.

Example 15. Let us first consider a rectangular index set with uniform distance vectors $(1, 0)$ and $(0, 1)$. The communication graph (after space-time mapping) is given in Figure 6. Let us now apply our algorithm for tiling time:

1. This example has only one dimension in time, so we set t to this dimension.
2. Since the program satisfies the FCO property, collapsing t is permitted.
3. Therefore, we can now compute the optimal tile width. Let m be the number of virtual processors, and n the number of logical time steps per processor. The program then has $n * m$ computations, and every physical processor sends at every time block one package with w pieces of data to its upper neighbor, where w is the time tile width. The computation amount per block is $\frac{n*m}{NP*(n/w)}$, and the number of blocks on the critical path is $n/w + (NP - 1)$, where NP is the number of physical

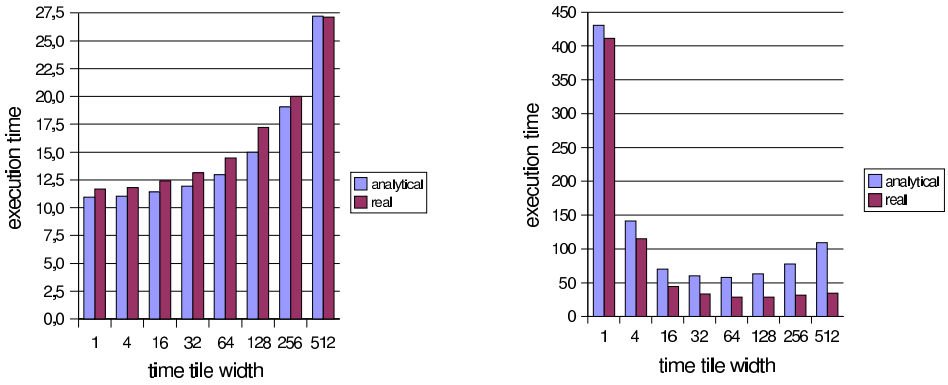


Figure 14. Analytical and experimental results for one-dimensional (left) and two-dimensional (right) allocation for LU decomposition

processors. Hence, the estimated execution time is:

$$C(w) = \frac{n * m}{\mathcal{NP} * (n/w)} * (n/w + (\mathcal{NP} - 1)) * cc + (n/w + (\mathcal{NP} - 1)) * sc + n * vc.$$

By deriving C, we find a minimum at $w_0 \approx \sqrt{\frac{n*sc}{m*cc}} \approx 554$ for our experiment ($n = 393216$, $m = 128$).

The scaled theoretical and the real execution times have already been presented in Figure 13.

In this example, the startup and computation costs are of the same order of magnitude (whereas the volume dependent communication cost is much smaller), so we can see the trade-off between reducing the number of startups vs. reducing the number of computations on the critical path.

Note that the optimal tile width w_0 for the case of $m*cc \geq n*sc$ is between 0 and 1, meaning that in this case tiling time is useless. This fits with our intuition: in this situation we have sufficiently coarse grain parallelism by just mapping the (large number of) m virtual processors to the real processors.

Example 16. We also calculated a space-time mapping defining a two-dimensional placement for our LU decomposition algorithm. This placement guarantees forward communications (except for the first two iterations). Thus, the only time dimension could be collapsed. Our cost model shows that the volume cost dominates the startup cost by orders of magnitude: by code inspection, we were able to determine that the average number of startups per time block is about $\mathcal{NP}/2$, and the transferred volume per broadcast is about $2.25 * n^2$. For simplicity and efficiency reasons, we decided to prefer broadcasts to point-to-point communications in this example. One consequence is that, in contrast to Example 15, the total volume cost decreases for larger time tiles, since our target program distributes a constant number of data elements per broadcast, and the number of broadcasts decreases for increasing tile width. The formula for the computation cost is similar to the one in Example 15, except that we have n^3 instead of $n * m$ computations in the body. In our parameter setting ($n = 1024$, $\mathcal{NP} = 4$ or $\mathcal{NP} = 16$), the volume cost is in the same order of magnitude as the computation cost. Hence we again

have a minimum for the execution time. By evaluating the cost function at several time tile widths (a practical alternative to derivation!), we obtained an optimum of about 64 in both cases, which happens to be the correct global optimum for $NP = 4$ (see Figure 14, right). On 16 processors, the measured minimal execution time is 22.9 seconds for an optimal tile width of 128 (in contrast to 24.5 seconds for a tile width of 64). The sequential execution time is 29.1 seconds.

For a one-dimensional placement of LU decomposition, our model predicted that the computation time (identical to the two-dimensional case) would dominate the constant communication volume cost by orders of magnitude – code inspection revealed about n^2 transferred data elements in total. Hence, the optimal theoretical and actual time tile width is 1, i.e., no time tiling should be done – the computations in the loop body take enough CPU time (in our case, the body contained an additional loop for summing up the values of `SUM` and `SUMM`, respectively).

The measured execution times for 4 and 16 processors and $n = 1024$ with a tile width of 1 are 11.7 and 5.9 seconds, respectively. On 16 processors, the minimal time is 5.5 seconds for a tile width of 2.

Note that for LU decomposition, a two-dimensional instead of a one-dimensional placement is a bad idea (Figure 14). But even if applied to the bad two-dimensional situation, tiling time achieves at least the same order of magnitude of the execution time as in the optimal one-dimensional placement (which is completely different without tiling time, i.e., for a time tile width of 1).

These examples show that our cost model is sufficiently accurate for uniform (Example 15) as well as for affine dependences (Example 16). Furthermore it is stable, regardless of the ratio of the most important cost parameters:

- whether there is a trade-off between the number of communication startups vs. the computation time (Example 15), or
- a trade-off between the volume-dependent communication time vs. the computation time (Example 16 with two-dimensional allocation), or
- whether the computation cost dominates both (Example 16 with one-dimensional allocation).

7.2.3. Practical limitation

The most difficult task is to estimate the number of computations, communication startups and the communication volume from the program text. The derivation of an automatic procedure for this approximation is left for future work. (Note that we could compute these numbers precisely by Ehrhart polynomials, but since we only need a rough guess, we suggest to develop a simpler and faster method.)

8. Conclusions

We have given several reasons for tiling after a space-time mapping. E.g., space-time mapping extends the applicability of existing tiling techniques to imperfect loop nests.

Furthermore, we have drawn the global picture of tiling space-time mapped loop nests, taking into account the resulting two different types of loops: loops in space and loops in time. Additionally, we have focused on the space loops and derived an algorithm which computes tiles with a minimal number of communication partners (if all dependences are uniform). Together with tiling time dimensions, this minimizes the number of communication startups.

Finally, we have shown how we can compute optimal time tiles in order to reduce the total execution time according to a cost model. In our future work, we shall focus on how to extract the communication and computation cost of a program automatically. This task should not be too complicated as we need only rough approximations for our method.

Currently, we are adding our methods to our open source prototype loop parallelizer LooPo. Since schedulers and allocators are already implemented, we hope that the amount of implementation work needed is relatively small. Our (still semi-automatic) experiments have yielded very promising speedups.

ACKNOWLEDGEMENTS

The authors would like to thank Nils Ellmenreich and Max Geigl, as well as some anonymous reviewers, for their fruitful comments on draft and preliminary versions of this paper. Thanks also to Armin Größlinger for his technical assistance. Furthermore, the first author wants to thank the organizers of CPC'01 for a wonderful workshop and the participants, esp. Alain Darte and Rumen Andonov, for their helpful discussions about different views on tiling. Financial Support was granted by the DAAD through PROCOPE.

REFERENCES

1. Ahmed N, Mateev N, Pingali K. Tiling imperfectly-nested loop nests (revised). Technical Report TR 2000–1782, Cornell University, Computer Science Department, Ithaca, NY 14853, 2000.
2. Andonov R, Balev S, Rajopadhye S, Yanev N. Optimal semi-oblique tiling. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA 2001)*. ACM Press, July 2001; 153–162. Extended version available as technical report: IRISA, nr. 1392, dec. 2001.
3. Andonov R, Rajopadhye S. Optimal orthogonal tiling of 2-d iterations. *J. Parallel and Distributed Computing* 1997; 45:159–165.
4. Andonov R, Rajopadhye S, Yanev N. Optimal orthogonal tiling. In Pritchard D, Reeve J, editors, *Euro-Par'98: Parallel Processing*, LNCS 1470. Springer-Verlag, 1998; 480–490.
5. Banerjee U. *Loop Transformations for Restructuring Compilers: The Foundations*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1993.
6. Boulet P, Darte A, Risset T, Robert Y. (Pen)-ultimate tiling? *INTEGRATION* 1994; 17:33–51.
7. Clauss P. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *Proc. Tenth ACM Int. Conf. on Supercomputing*. ACM Press, May 1996.
8. Desprez F, Dongarra J, Rastello F, Robert Y. Determining the idle time of a tiling: New results. *Journal of Information Science and Engineering* 1998; 14:167–190. Also available as technical report: INRIA, Nr. 3272, Oct. 1997.
9. D'Hollander E. Partitioning and labeling of loops by unimodular transformations. *IEEE Trans. on Parallel and Distributed Systems* 1992; 3(4):465–476.
10. Dion M, Robert Y. Mapping affine loop nests: New results. In Hertzberger B, Serazzi G, editors, *High-Performance Computing & Networking (HPCN'95)*, LNCS 919. Springer-Verlag, 1995; 184–189.
11. Feautrier P. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming* Feb. 1991; 20(1):23–53.
12. Feautrier P. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming* 1992; 21(5):313–348.
13. Feautrier P. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming* 1992; 21(6):389–420.
14. Feautrier P. Toward automatic distribution. *Parallel Processing Letters* 1994; 4(3):233–244.
15. Feautrier P. Automatic parallelization in the polytope model. In Perrin G.-R, Darte A, editors, *The Data Parallel Programming Model*, LNCS 1132. Springer-Verlag, 1996; 79–103.
16. Feautrier P. Automatic distribution of data and computation. Technical Report 2000/3, Laboratoire PRiSM, Université de Versailles, URL: http://www.prism.uvsq.fr/rapports/2000/abstract_2000_3.html, Mar. 2000. English translation of TSI vol. 15 pp 529–557, 1996.
17. Griebl M. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, Jan. 1997. Technical Report MIP-9701.

18. Griebl M. On the mechanical tiling of space-time mapped loop nests. Technical Report MIP-0009, Fakultät für Mathematik und Informatik, Universität Passau, Aug. 2000.
19. Griebl M, Feautrier P, Größlinger A. Forward communication only placements and their use for parallel program construction. In *Languages and Compilers for Parallel Computing, 15th International Workshop, LCPC'02*, LNCS, 2002. To Appear.
20. Griebl M, Feautrier P. A, Lengauer C. Index set splitting. *Int. J. Parallel Programming* 2000; 28(6):607–631.
21. Griebl M, Lengauer C. The loop parallelizer LooPo—Announcement. In Sehr D, Banerjee U, Gelernter D, Nicolau A, Padua D, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, LNCS 1239. Springer-Verlag, 1997; 603–604. More details at http://www.infosun.fmi.uni-passau.de/cl/loopo.
22. Hiranandani S, Kennedy K, Tseng C.-W. Evaluating compiler optimizations for Fortran D. *J. Parallel and Distributed Computing* 1994; 21:27–45.
23. Hodžić E, Shang W. On time optimal supernode shape. In *Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, 2000; 367–379.
24. Högstedt K, Carter L, Ferrante J. Determining the idle time of a tiling. In *Proc. 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*, New York. ACM Press, Jan. 1997. Also available as UCSD Tech Report CS96-489.
25. Högstedt K, Carter L, Ferrante J. Selecting tile shape for minimal execution time. In *11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'99)*. ACM Press, June 1999; 201–211. Also available with proofs as UCSD Tech Report CS99-616.
26. Irigoin F, Triolet R. Supernode partitioning. In *Proc. 15th Ann. ACM Symp. on Principles of Programming Languages (POPL'88)*. IEEECS, Jan. 1988; 319–329.
27. Lee H.-J, Fortes J. A. Communication-minimal partitioning and data alignment for affine nested loops. *The Computer Journal* 1997; 40(6):302–310.
28. Lengauer C. Loop parallelization in the polytope model. In Best E, editor, *CONCUR'93*, LNCS 715. Springer-Verlag, 1993; 398–416.
29. Lim A. W, Lam M. S. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing* May 1998; 24(3–4):445–475.
30. Ohta H, Saito Y, Kainaga M, Ono H. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *Proc. 1995 Int. Conf. on Supercomputing*. ACM Press, July 1995.
31. Palermo D, Su E, Chandy J, Banerjee P. Communication optimizations used in the PARADIGM compiler for distributed memory multicomuters. In *International Conference on Parallel Processing*. IEEE, Aug. 1994.
32. Quilleré F, Rajopadhye S, Wilde D. Generation of efficient nested loops from polyhedra. *Int. J. Parallel Programming* 2000; 28(5):469–498.
33. Reed D. A, Adams L. M, Patrick M. L. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. on Computers* July 1987; C-36(7):845–858.
34. Schreiber R, Dongarra J. J. Automatic blocking of nested loops. Technical Report CS-90-108, University of Tennessee, Computer Science, May 1990.
35. Song Y, Li Z. A compiler framework for tiling imperfectly-nested loops. LNCS 1863. Springer-Verlag, 1999; 185–200.
36. Wolfe M. Iteration space tiling for memory hierarchies. In Rodrigue G, editor, *Parallel Processing for Scientific Computing*. SIAM, 1987; 357–361.
37. Wolfe M. More iteration space tiling. In *Supercomputing '89*. ACM Press, 1989; 655–664.
38. Xue J. Communication-minimal tiling of uniform dependence loops. *J. Parallel and Distributed Computing* Apr. 1997; 42(1):42–59.
39. Xue J. On tiling as a loop transformation. *Parallel Processing Letters* 1997; 7(4):409–424.
40. Xue J. *Loop Tiling for Parallelism*. Kluwer, 2000.
41. Xue J, Huang C.-H. Reuse-driven tiling for improving data locality. *Int. J. Parallel Programming* Dec. 1998; 26(6):671–696.