The Minimal Number of Communication Startups when Tiling Space-Time Mapped Programs

Martin Griebl
University of Passau, FMI
griebl@fmi.uni-passau.de

ABSTRACT

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 *space-time mapping* [17]. 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.

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 [7], we call these instances *operations*.

The problem area addressed in this paper is tiling for parallelization. Up to now, 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* another parallelization phase is executed [3].

In contrast, we suggest to apply tiling *after* the parallelization phase. In our case, we employ loop parallelization in the polytope model [8, 17], which is based on the concept of *space-time mapping*, and extensions thereof (e.g., the polyhedron model [9]).

Our goals are as follows:

- We demonstrate how *existing* tiling methods for parallelization can benefit from a previous space-time mapping, e.g., by extending quite naturally the applicability of tiling to imperfectly nested loops – just by using the structure of the space-time mapped program (Sections 1.1–1.3).
- We consider separately the two different types of loops after space-time mapping: tiling space loops reduces the number of communication *partners* per communication phase, and tiling time loops reduces the number of communication *phases* (Section 2).
- We derive an algorithm for computing tiles which leads to the minimal number of communication partners for uniform dependences – in the space-time mapped program (Section 4). Together with tiling time dimensions, this minimizes the number of communication startups. 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 for the communication cost (which we aim to reduce) than the number of synchronizations.
- For non-uniform dependences, we give heuristics for computing tiles which reduce the number of communication startups. E.g., we deal with "long, nearly uniform" dependences, for which a suitable *extent* of the tiles may halve the number of communication startups for every dimension (Section 5.1).
- We give some speedup results for space-time mapped and then tiled LU decomposition, a program with imperfectly nested loops and general affine dependences (Section 5.2).
- Note that, while reducing the number of communication startups, we match the number of virtually needed processors with the number of physically available processors – a task which is typically performed by a subsequent brute-force rectangular block or cyclic "tiling".

1.1 An example

As a running example, we take the LU decomposition algorithm given in Figure 1. We are not going to analyze it in detail – its most important property is its imperfectly nested structure.

```
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 1: LU decomposition

In order to see some immediate advantages of applying tiling after space-time mapping, let us briefly review the concept of space-time mapping and then apply it to the LU program.

1.2 The concept of space-time mapping in the polytope model

Methods based on the polytope model work as follows.

They 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 functions: 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.

In the case of the polytope model, we obtain multidimensional and (piecewise) affine functions as schedule and allocation – one pair per statement.

These functions, together with the index space of the source program, determine at which logical time step and on which virtual processor which operation must be executed. Thus, the final output of the parallelization is a loop program which enumerates those operations.

From the model point of view, this 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 [21]).

Example 1. If we space-time map the original LU decomposition (e.g., with LooPo [11], 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 2).

```
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 2: Target structure of LU decomposition

1.3 Tiling after space-time mapping

This review of parallelization by space-time mapping already highlights some advantages of tiling after space-time mapping.

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, 23], 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.

Non-uniform dependences. The space dimensions are always fully permutable – and are extracted by more flexible techniques than just loop skewing. Consequently, non-uniform dependences are easily integrated, since at least tiling in the space dimensions is possible.

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

Flexibility of parallelization. If we coalesce operations of the source program, we may give up automatically detectable parallelism unnecessarily.

Tiling power. The direct access to the logical execution time (the indices of the loops in time) permits a dynamic modification of the tiles. E.g., the tile size for the dimensions in space may depend on this logical time, enabling a static load balancing. We leave this aspect for future work.

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, all of whose 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).

Technically, we restrict ourselves additionally to *parallelepipeds*, i.e., tiles in n-dimensional space can be generated by n families of parallel hyperplanes [14].

2. Tiling Space and/or Tiling Time

So far, we have (non-trivially) transformed an imperfectly nested program so that existing tiling techniques can be applied, e.g., that of Xue [26] which can deal with affine dependences. Next, we want to benefit from the fact that the loops are either exclusively in space or exclusively in time.

Tiling space (processor) dimensions means aggregating neighboring virtual processors in blocks and mapping them to a common physical processor. 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 4).

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 3, left)!

Our way out is allowing for message vectorization by tiling the time dimensions of the parallelized program. Aggregating a set of logical time steps to a single step (a kind of "superstep") reduces the number of communications: they only take place between two supersteps as indicated in Figure 3, right.

Of course, this requires that dependent tiles start their execution later than in the original parallel program; this is a typical subtask of existing tiling methods ("scheduling the tiles"). In our context, this schedule, if it exists, always corresponds to skewing every space dimension (the physical processors) into time with factor 1.

Summary of our basic idea

After space-time mapping, we first apply tiling to 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.

Afterwards, when we know the space tiles, we apply tiling to 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 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 prevent 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 startups by our separation – for a given number of processors to be used by the parallel program (cf. Section 4.1).

3. 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 [22, 24]. In 1988, Irigoin and Triolet [15] gave 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 [2, 4, 19], and then to adapt the shape of the tiles to the dependences [5, 13, 14, 25–27], resulting in parallelepipeds as tiles. All this work has been limited to perfect loop nests, a part of it even to fixed dimensionality, e.g., [2].

As already noted, after space-time mapping, we can assume a unified coordinate system, i.e., we need no special methods for imperfectly nested loops [1, 23]. In addition, these special methods are developed in the framework of cache optimization. However, tiling methods for cache

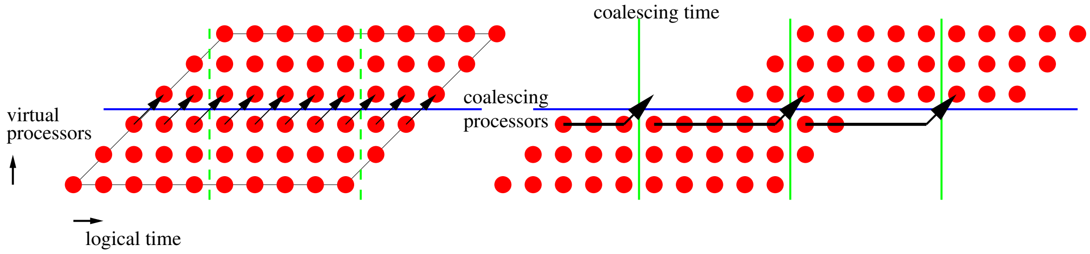

Figure 3: Target space before (left) and after (right) tiling the time dimension

optimization cannot be used directly in tiling for coarse-grain parallelism, even if both try to improve locality. One of the main differences is the (missing) existence of distribution (i.e., parallelism) as a conflicting goal of 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 tiling, the program model of Song and Li [23] is more restricted than ours: it only allows one enclosing loop whose body contains several loops. In addition, this outermost loop is not a subject to tiling. On the other hand, the method of Ahmed, Mateev and Pingali [1] 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). Tiles are searched for this high-dimensional domain. In addition, the embedding of the original iteration domains into the product space offers a very wide range of choices, from which one is taken 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 really important since the time complexity of space-time mapping methods usually is exponential in the dimensionality.

On the other hand, 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 is only given after some kind of space-time mapping – but previous work on tiling is not based on space-time mapping.

Typically, there are two approaches dealing with communication reduction. The first approach minimizes the communication *volume* [25], instead of the number of startups. But, in state-of-the-art parallel machines, the communication startup cost is much higher than the cost for transferring an additional value.

The other approach reduces the number of necessary *synchronizations* [3, 16, 18], which is a more abstract, hence less precise cost model than the number of startups. Some of these approaches [3, 16] are additionally restricted to rectangular partitioning, e.g., due to the limitations of BLOCK distribution in HPF. Furthermore, there are approaches [3] which do not coalesce messages, but issue one communication per variable.

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

4. Optimal Space Tiling

4.1 The task of tile optimization

In the following sections, we derive an algorithm to determine 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. (Extensions for non-uniform dependences are given in Section 5.)

A subsequent tiling of time dimensions adapts the granularity, as indicated above. If the computed space tiles do not prevent tiling time, we obtain the minimal number of communication startups for a given number of physical processors in the following sense: if another tiling would cause less startups, it might 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).

Remark. Note that, in practice, we are not really interested in the absolute global minimum of startups (one communication per processor) since we want to keep parallelism. Thus, a more concrete cost function is used for determining the optimal width of the time tiles. More formal detail and the principal algorithm for computing the optimal time tiles is given elsewhere [10].

Since 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 [12]. It may appear to be oversimplified, but several research papers ([3, 19, 20]) have shown that this cost model it not far from truth.

In order to obtain the optimal tiling, we first determine the *shape* of the tile, i.e., the directions of its spanning vectors (Sections 4.2 and 4.3). 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 4.4). 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. Note that a cyclic distribution of (smaller) tiles causes more communications than a one-to-one mapping of tiles to physical processors.

4.2 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.

In addition, we assume "short" dependences. More precisely: every dependence d must be shorter than the extent of the tile in direction d (this assumption is dropped in Section 5.1). Also, we ignore the borders of the index space (this aspect is added in Section 4.4), i.e., up to Section 4.4 we assume that the index space contains more than one tile (indeed infinitely many) in every dimension.

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

For the ideal case, tiles which minimize the number of communication partners are easily constructed on the basis of the following observation.

Observation 1. *Let D be a set of n linearly independent uniform dependence vectors $d_1, \cdots, d_n$. 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 hyperplane 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 is minimal, since we assume a tile size smaller than the extent of the index space in direction d_i. The 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.

Example 2. Since the LU decomposition algorithm above has some non-uniform dependences, we cannot use it here, i.e., we must construct a new example. Since the structure of the LU algorithm is quite interesting, we keep it and generate the new example out of it by replacing some variables – this is enough to eliminate the undesired complex dependences: we consider L and U as read-only variables and store the computed values in lines 2, 5, 13, 21 in new arrays LL and UU instead. Let us call this modified program *LU-structured*.

If we run LooPo on the LU-structured program, we obtain an allocation which makes all 14 existing dependences local, i.e., we have no communication left! (Aside: This is an example of how tiling can benefit from space-time mapping: the number of relevant dependences is reduced – in this example even to 0.)

In order to keep the example interesting, we add dependences by changing lines 13 and 21 into

```
13': LL(I,J) = A(I,J) - SUM[J,I] +
               LL(I+1,J+1) and
21': UU(J,I2) = (A(J,I2) - SUMM[J,I2])/
       L(J,J) + LL(I2,J) + UU(J+1,I2-1).
```

Note that this introduces a non-uniform dependence due to the access to `LL(I2,J)` in line 21'. Again, this dependence causes no communication since both operations are allocated on the same processor. Consequently, there remain only two dependences, both uniform, with directions $(1, 1)$ and $(1, -1)$, which span the tile that minimizes the number of communication partners. (Aside:

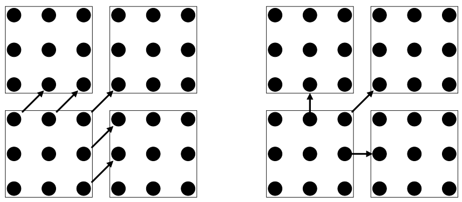

Figure 4: Dependences and communication partners

this elimination of affine dependences is another example of how tiling can benefit from space-time mapping.)

4.3 Arbitrary number of uniform dependences

Let us now consider the more typical case of m uniform dependences (for $m > n$). Here, the critical questions is: which dependence vectors should we use to span the tile, i.e., which vectors should be used as a *basis* of the n-dimensional space? We know: the dependences not selected – let us call them $D'' = \{d_{n+1}, \cdots, d_m\}$ – can be expressed as linear combinations $L = \{l_{n+1}, \cdots, l_m\}$ of the selected ones – let us call those $D' = \{d_1, \cdots, d_n\}$. So we must first check how many communication partners arise from a linear combination.

As above, we assume that the tile size is sufficiently large compared to the dependence distances. More precisely: in every dimension the tile is larger than any dependence in $D' \cup D''$. Therefore, we can only reach neighbor tiles. Consequently, we may abstract from the precise linear combinations in L and use, as abstract representation of $l_j \in L$ the vector $\bar{l}_j := (s_1, \cdots, s_n)$, where s_i is the sign of the i-th component of l_j. Thus, we have $m - n$ vectors of length n with entries $-1, 0$ or 1.

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

At first glance, a vector $\bar{l}_j$ (corresponding to a dependence $d_j \in D''$) might represent precisely that neighbor tile which is the communication partner for dependence d_j. However, a single dependence can lead to several partners (see Figure 4). The correct formula specifying the communication partners for a given dependence $\bar{l}_j$ is as follows. Every tile whose relative neighbor coordinates are

- 0 — for those dimensions k where $d_j[k] = 0$, and
- 0 or x — for those dimensions k where $d_j[k] = x$ ($x \neq 0$, i.e., $x = \pm 1$)

can be a communication partner of the actual tile – except for the tile itself, which has neighbor coordinates $(0, \cdots, 0)$. In the worst case, we have no zero entries, i.e., we have $2^n - 1$ possible communication partners (all neighbors, and not the tile itself).

Example 3. Let us consider the dependence $\bar{l}_j = (1, 1)$ in Figure 4, 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 4 (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:
   - (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 most relevant for the time complexity is the computation of the possible communication partners of cost $O(2^n)$, which is executed $\binom{m}{n}$ times. But, since n is a very small integer, this is acceptable.

Example 4. Again, we modify our LU-structured algorithm in order to obtain this situation:

```
13": LL(I,J) = A(I,J) - SUM[J,I] +
     LL(I+1,J+1) + UU(J-1,I) + UU(J,I+1)
```

and line 21" equals line 21'.

The four dependence vectors (projected on the space dimensions) are $d_1 = (1, 1), d_2 = (1, -1)$, $d_3 = (-1, 0)$, and $d_4 = (0, 1)$. As examples for all pairs out of the four dependences, we present the result after Step 1b for two selected pairs in Figure 5. If we take d_1 and d_3 as basis (right), we end up with 6 communication partners; if we take d_2 and d_3 (left), we have only 5, which is minimal over all pairs. Hence, we have as vertices of the tile: $(0, 0)$, d_2, d_3 and $(0, -1)$.

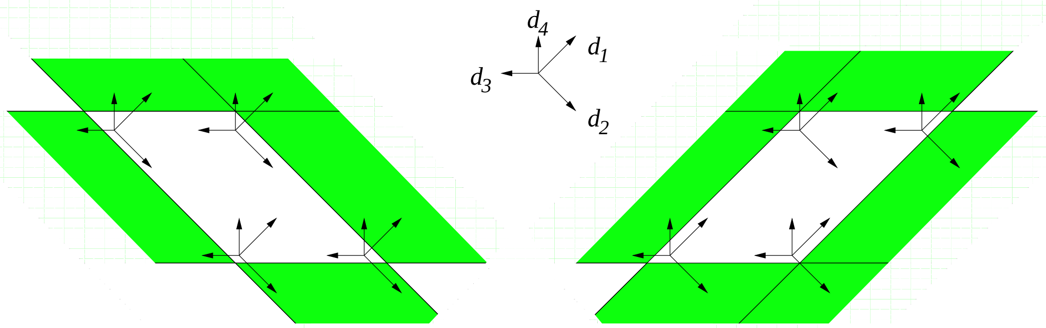

Figure 5: Communication partners for d_2 and d_3 (left) or d_1 and d_3 (right) in D'

4.4 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 is to get rid of communications 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 the LU example). 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 6 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.

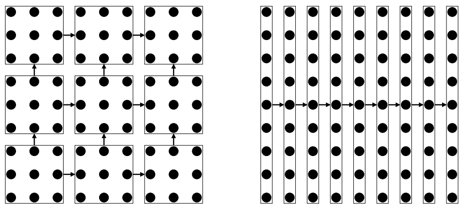

Figure 6: Square and thin but long tiles

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.

5. 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.

5.1 Long dependences

Formally, a dependence with constant distance m is not uniform for a symbolic parameter m. 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 appear 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).

5.1.1 Communication partners

Let us now compute the number 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 hy-

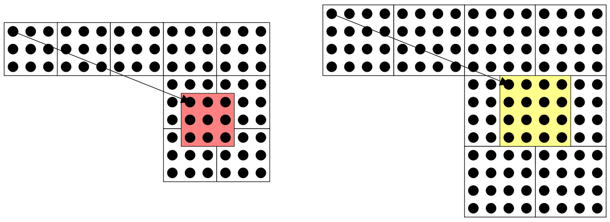

Figure 7: Long uniform dependences

perplane, 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 5. In the left part of Figure 7, the tile width in both directions is 3, and the depicted dependence maps both the vertical and the horizontal boundary to the interior of another tile; the number of communication partners is $2*2 = 4$ (2^n in the general n-dimensional case). In the right part, where the tile width in both directions is 4, the horizontal boundary is mapped to another horizontal boundary – the number of communication 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 $1*1$, i.e., a single communication.

The important difference to the case of Sections 4.2 and 4.3 is that the widths 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 5.1.2).

Remark. We can specify lower and upper bounds for the number of startups without considering the form: a long uniform dependence d causes one or two startups if d is among the vectors spanning 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 4.2 or with the partners caused by other long uniform dependences. The only precise solution method is to count the number of communication partners by using Ehrhart polynomials [6]. 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.

5.1.2 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., not 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 5.1 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 6. Let us consider the horizontal dimension in Figure 7, 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 take, 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 5.

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 7. If we had a second long dependence $d' = (15, 8)$, we factorize 15, and we see that $s_1 = 5$ appears both for d' and d, i.e., a horizontal tile width of 5 causes a quater 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.

5.2 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.

Instead, our implementation will use the direction vectors as an approximation and use them as candidates for D' if there are less uniform candidates than the dimensionality. This case appears for the LU decomposition algorithm, where we end up with rectangular tiling in the space dimensions. The best results (for different time tile widths) of our first experiments with a matrix size of 1024×1024 are given in Table 1; we ran the experiments on a NEC CENJU-4 at GMD which has 64 nodes (MIPS 10000) with 256 MByte main memory per node (a more detailed technical description can be found at `http://filou.gmd.de:8002/popcorn/services/CENJU4/Configuration.html`), and on a network of 96 SCI connected Primergy server nodes, each with two Intel Pentium-II, 450 MHz, 512 MByte memory, at the university of Paderborn (more detail about the architecture at `http://www.uni-paderborn.de/pc2/services/systems/psc/`).

6. Conclusions

We have given a number of reasons for tiling after a space-time mapping. E.g., space-time mapping extends the applicability of existing tiling techniques to imperfect loop nests, or to loop nests with non-uniform dependences.

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 or loops in time. Additionally, we have focussed on the space loops and derived an algorithm which computes tiles with a minimal number of communication partners. Together with tiling time dimensions, this minimizes the number of communication startups.

Acknowledgments

I would like to thank Christian Lengauer, Nils Ellmenreich, Peter Faber, and Max Geigl for their fruitful comments on draft versions of this paper. Thanks also to Armin Größlinger for his technical assistance. Financial Support was granted by the DAAD through PROCOPE.

References

[1] N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests (revised). Technical Report TR 2000–1782, Cornell University, Computer Science Department, Ithaca, NY 14853, 2000.

[2] R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *J. Parallel and Distributed Computing*, 45:159–165, 1997.

[3] R. Andonov, S. Rajopadhye, and N. Yanev. Optimal Orthogonal Tiling. In D. Pritchard and J. Reeve, editors, *Euro-Par'98: Parallel Processing*, LNCS 1470, pages 480–490. Springer-Verlag, 1998.

[4] R. Andonov, S. Rajopadhye, and N. Yanev. Optimal orthogonal tiling. In D. Pritchard and J. Reeve, editors, *Euro-Par'98: Parallel Processing*, LNCS 1470, pages 480–490. Springer-Verlag, 1998.

[5] P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *INTEGRATION*, 17:33–51, 1994.

[6] P. Clauss. 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.

[7] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, Feb. 1991.

[8] P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.

	1d-allocation				2d-allocation			
	4 processors		16 processors		4 processors		16 processors	
	Cenju	SCI	Cenju	SCI	Cenju	SCI	Cenju	SCI
speedup	3.1	2.7	10.9	6.4	1.8	1.3	3.0	1.6
efficiency	0.78	0.67	0.68	0.40	0.45	0.32	0.19	0.1
for time tile width	4	4	1	2	64	96	64	128

Table 1: Experimental results for LU decomposition

[9] M. Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, Jan. 1997. Technical Report MIP-9701.

[10] M. Griebl. On the mechanical tiling of space-time mapped loop nests. Technical Report MIP-0009, Fakultät für Mathematik und Informatik, Universität Passau, Aug. 2000.

[11] M. Griebl and C. Lengauer. The loop parallelizer LooPo—Announcement. In D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing (LCPC'96)*, LNCS 1239, pages 603–604. Springer-Verlag, 1997.

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

[13] E. Hodžić and W. Shang. On time optimal supernode shape. In *Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 367–379, 2000.

[14] K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*, pages 201–211. ACM Press, June 1999. Also available with proofs as UCSD Tech Report CS99-616.

[15] F. Irigoin and R. Triolet. Supernode partitioning. In *Proc. 15th Ann. ACM Symp. on Principles of Programming Languages (POPL'88)*, pages 319–329. IEEECS, Jan. 1988.

[16] H.-J. Lee and J. A. Fortes. Communication-minimal partitioning and data alignment for affine nested loops. *The Computer Journal*, 40(6):302–310, 1997.

[17] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[18] A. W. Lim and M. S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.

[19] H. Otha, Y. Saito, M. Kainaga, and H. Ono. Optimal tile size adjustment in compiling general DOACROSS loop nests. In *Proc. 1995 Int. Conf. on Supercomputing*. ACM Press, July 1995.

[20] D. Palermo, E. Su, J. Chandy, and P. Banerjee. Communication optimizations used in the PARADIGM compiler for distributed memory multicomuters. In *International Conference on Parallel Processing*. IEEE, Aug. 1994.

[21] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *Parallel Processing Letters*, 2001. To Appear.

[22] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: Their influence on the performance of multiple processor systems. *IEEE Trans. on Computers*, C-36(7):845–858, July 1987.

[23] Y. Song and Z. Li. A compiler framework for tiling imperfectly-nested loops. In L. Carter and J. Ferrante, editors, *Languages and Compilers for Parallel Computing, 12th International Workshop, LCPC'99*, LNCS 1863, pages 185–200. Springer-Verlag, 1999.

[24] M. Wolfe. Iteration space tiling for memory hierarchies. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 357–361. SIAM, 1987.

[25] J. Xue. Communication-minimal tiling of uniform dependence loops. *J. Parallel and Distributed Computing*, 42(1):42–59, Apr. 1997.

[26] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[27] J. Xue and C.-H. Huang. Reuse-driven tiling for improving data locality. *Int. J. Parallel Programming*, 26(6):671–696, Dec. 1998.