On the Mechanical Tiling of Space-Time Mapped Loop Nests

Martin Griebl

Fakultät für Mathematik und Informatik
Universität Passau, D–94030 Passau, Germany
email: griebl@fmi.uni-passau.de
WWW: http://www.fmi.uni-passau.de/~griebl

Abstract. There exist many methods for extracting automatically parallelism (sometimes even a provably maximal amount of parallelism) out of a sequential imperative loop program. However, for performance reasons, the granularity of parallelism must be coarse enough in order to get a useful ratio between the number of computations and the number of communications. Usually, tiling techniques are applied for obtaining coarser parallelism. Unfortunately, those tiling techniques designed for limiting parallelism can only deal with perfectly nested loops so far (even if there is some recent work which deals with tiling imperfect loop nests for cache optimization; cf. Sectionsec:relwork). Thus, the goal of this paper is to provide a technique which allows imperfectly nested programs as input and produces a well-performing tiled parallel program as output.
In contrast to other approaches, we apply tiling techniques not to a (sequential) source program but to its derived parallel, i.e., *space-time mapped* target program. Therefore, we need no sophisticated tiling techniques for imperfect loop nests, we do not limit the power of the parallelization phase, i.e., the space-time mapping phase, and we can directly choose the granularity dependent on the number of physically available processors.

1 Introduction

Tiling is a well-known loop transformation technique for (sequential or parallel) compiler optimization. In programs containing nested loops, there are multiple instances of every statement. We call them *operations* [11].

The principal idea of tiling is to group together sets of operations. Depending on the goal, these sets must satisfy different requirements: they must be small enough to satisfy some resource limitation, e.g., on cache or memory size, or they must be large enough to contain enough computation, e.g., to increase granularity in an automatic parallelization.

For technical reasons, these sets are congruent, non-intersecting polytopes, so-called *tiles*, the union of which covers all operations. In many cases, also in this paper, an additional restriction is that the tiles must be *parallelepipeds*, i.e., tiles in d-dimensional space can be generated by d families of parallel hyperplanes [17].

The focus of this paper is on tiling for automatic parallelization, esp. loop parallelization in the polytope model [12, 19], which is based on space-time mapping. The methods of this model typically search for optimal parallelism, not optimal performance, since the latter is mathematically too complex to be computed during compilation. Consequently, the granularity of the automatically derived parallelism is very fine and must be coarsened in order to reduce communication cost and get good performance. So the goal of this paper is to examine the application of tiling to space-time mapped programs so as to obtain well performing parallel programs. (Aside: This way of building parallel programs corresponds to the procedure of Foster [13].)

In addition, we shall see that, in our context, we need no special techniques for tiling imperfect loop nests, even if the sequential source program is imperfectly nested. This is important since, as long as the loop nest to be tiled is perfect (i.e., every loop, except for the innermost, contains precisely one further loop as its body), and the dependences between the operations are uniform, there exist sophisticated techniques which compute the optimal tile shape and size (cf. Section 2), whether the application domain is cache optimization or reducing parallelism. However, up to now, there is only little work on automatic tiling of imperfect loop nests, and none with a focus on parallelism (cf. Section 2).

Note that the extension of our technique towards cache optimization or hierarchical memory systems is a subsequent step in our mind: first, we determine which operations are placed on which (virtual) processor, before we then try to cache-optimize the program which runs on every processor.

The basid idea of this paper is to apply tiling techniques to the parallelized program – not to the sequential source program. This is useful for the following reasons:

- If we tile after the space-time mapping phase, all operations of the parallelized loop nest are expressed in one unified coordinate system, consisting of space and time dimensions. Hence, in this paper, we may restrict ourselves to a single tile shape and size for the whole space-time coordinate system.
- If we apply tiling to the source program, we first need techniques for imperfect loop nests and, second, we tile, i.e., group together operations before we have any idea about how the operations could/should be distributed in space and time; this may lead to unnecessary loss of flexibility of the space-time mapping methods and, hence, in a loss of performance. For example, space-time mapping covers iteration graph partitioning [7], but if tiling of the source code caused the independent iterations to be inside one tile, this parallelism is lost.
- Another advantage of tiling the parallelized loop nest is that, only after space-time mapping, we know the maximum number of required virtual processors. Thus, for the computation of our tile shape and size, we can take into account the number of virtual processors which must be mapped to one physical processor.

In addition, we see that parallelization and tiling fit well together and can be treated in the same model. Hence, all advantages of model-baseed parallelization still remain valid; especially, we need not find heuristically a suitable set and ordering of transformations for finding a tiling, but we get the tiling in a straight forward manner from the model.

After a brief review of related work in Section 2, we explain our idea in more detail in Section 3 and provide the technical solution in Section 4. Finally, Section 5 concludes.

2 Related Work

The first papers on tiling date back to the mid-Eighties [21, 23]. In 1988, Irigoin and Triolet [18] 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 [4, 5, 20], and then to adapt the shape of the tiles to the dependences [8, 16, 17, 24–26], resulting in parallelpipeds as tiles. All this work has been limited to perfect loop nests, a part of it even to fixed dimensionality, e.g., [4].

To the best of our knowledge, Song and Li [22] are the first authors who do tiling for imperfect loop nests. Their program model is one enclosing loop whose body contains several loops, and their goal is to tile for cache optimization between array accesses in the different inner loops. They achieve this goal to a large extent. Unfortunately their method cannot be used for our purpose, since their technique restructures the code, which would invalidate our space-time mapping. In addition, the outermost loop is not partitioned, which is usually necessary for our purposes (cf. Section 3.4).

Recently, Ahmed, Mateev and Pingali [1, 2] presented a method for tiling imperfeclty nesteded loop nests. The main difference to our approach is that their work treats cache optimization only and does not take parallelism into account. Also technically, the approaches are quite different: their method uses the product space af all iteration domains, i.e., they work on a space whose dimensionality is as high as the number of loops in the program. 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, and 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.

Furthermore, we shall see that in our setting it is sometimes useful to try to tile dimensions which are not fully permutable (cf. Section 3.4).

From the application point of view, [16] is one of the most closely related papers – however, it applies only to perfect loop nests.

3 Combining Tiling with the Polytope Model – The Idea

3.1 Tasks of tiling

In the setting of parallelization, tiling has three goals:

- to minimize the number of communications by choosing an appropriate tile shape – note that communications appear only at the borders of the tiles;
- to adapt the granularity to the relative communication speed (computations per communication) by choosing the right tile size;
- to map the virtually enumerated processors to the physical processors.

For the first two goals, there are sophisticated techniques, as mentioned in Section 2, some of which even address the third goal [4, 5, 20]. However, if we tile *after* space-time mapping, we have a much clearer picture of what we are doing: with different tilings we can directly control the number, size and direction of communication, the computation volume per tile and the number of needed real processors per dimension. Note that we cannot and do not focus on theoretical optimality, since we combine heuristic solutions for space-time mapping and tiling. Our goal is to present how far we can get by a new combination of well-understood techniques.

3.2 The framework of space-time mapping

Space-time mapping is one way of parallelizing programs, and the polytope model, which we use throughout this paper, is a mathematical model developed for space-time mapping loop programs.

Input: Methods based on the polytope model take as input an arbitrarily nested sequential loop program. The only allowed data structure is the array, and the supported control structures are loops and conditional branches.

Note that, in order to get the full benefit out of the polytope model, all array indices must be affine expressions in surrounding loop indices and structure parameters; equivalently, the loop bounds and conditionals should be affine expressions in the program; the existing dependences need not be uniform, but should also be affine. The methods of the polytope model can be applied without these requirements, but at the price of conservative approximations, which reduce the parallelism.

Since our method applies tiling after space-time mapping, it inherits the input requirements from the polytope model.

Output: The result of any space-time mapping are two functions: a *schedule* and an *allocation*, which determine, respectively, for every operation in the program, at which logical time and on which logical processor the operation is executed.

```
for j=2 to n do
    for i=j to n do
        for k1=1 to j-1 do
            L[i,j] = L[i,j] - L[i,k1]*U[k1,j];
        enddo
    enddo
    for i2=j+1 to n do
        for k=1 to j-1 do
            U[j,i2] = U[j,i2] -  L[j,k]*U[k,i2];
        enddo
        U[j,i2] = U[j,i2] / L[j,j];
    enddo
enddo
```

Fig. 1. Core of LU decomposition – the source code

In the case of the polytope model, we obtain separate schedules and allocations for every statement, and all these functions are multidimensional and (piecewise) affine.

For the rest of this paper, it is important to realize that, without loss of generality, the allocations (and also the schedules) have the same dimensionality for all statements:

- For the allocation, this is guaranteed by the allocation method, since, in a d-dimensional processor space, every allocation must determine the processor coordinates in all d dimensions.
- The schedule may, a priori, have different dimensionalities for different statements. But a schedule with insufficient dimensionality may always be extended by suitable coordinates: for every time dimension i determine its minimal value m_i (this exists and is finite!) and set the time coordinate to $m_i - 1$ for all those schedules in which time dimension i is not specified explicitly.

In other words: after space-time mapping, all operations can be viewed as being embedded in a perfect nest of time and space loops. I.e., the input to the tiling phase is always perfectly nested. Note that this is true, regardless of whether the target program is indeed perfectly nested (as in typical naïve code generation methods, e.g., the run-time method of [15]) or not (as in more sophisticated code generation algorithms, such as the compile-time method of [15] or Quilleré's method [6, page 24]).

Example 1. As a running example, we take the core of an LU-decomposition algorithm, whose imperfectly nested source code is given in Figure 1. After space-time mapping and pretty printing we obtain a loop nest (with communication statements omitted) as depicted in Figure 2.

By some more sophisticated code generation techniques (as mentioned above), we obtain an imperfectly nested target loop nest (see Figure 3), which is, however,

```
seqfor t0=0 to n-2 do
    parfor p1=t0+2 to n do
        parfor p2=min(p1,t0+3) to n do
            if(p1 <= p2) then
S1:             L[p2,p1]=L[p2,p1]-L[p2,t0+1]*U[t0+1,p1];
            endif
            if(t0<=n-3 && p1 <= n-1 && 1+p1 <= p2) then
S2:             U[p1,p2]=U[p1,p2]-L[p1,t0+1]*U[t0+1,p2];
            endif
            if(t0 <= n-3 && t0+2 == p1 && t0+3 <= p2) then
S3:             U[t0+2,p2]=U[t0+2,p2]/L[t0+2,t0+2];
            endif
         enddo
    enddo
enddo
```

Fig. 2. The perfectly nested target code

only a different representation. Thus, for tiling, we may (and do) use the perfect nest of Figure 2. We shall revisit this example in Section 4.3.

Remark 1. Note that tiling creates new sequential loops, i.e., it modifies the schedule. Nevertheless, there is no cycle in our suggested method, since this modification of the schedule is either just a refinement (if we tile space dimensions) or a simple skewing (if we tile time dimensions), which is independent of the program and, thus, does not require a new scheduling task.

Remark 2. Note that space-time mapping methods lay out independent statement instances in space. In other words, if instances are laid out in time, this is due to (assumed) dependences.

The following sections present the idea and interpretation of tiling space and time dimensions. Note that, in this paper, we restrict ourselves to tiling space and time dimensions separately. If tile bounds of space dimensions depend on time dimensions, we reintroduce, in general, dependences, i.e., communications, which the allocator has had already eliminated. More analysis on whether this is useful is left for future work.

3.3 Tiling space (processor) dimensions

Tiling space dimensions means aggregating neighbouring 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 shape and size of tiles for the space dimensions. Note that this is valid for `DOALL` as well as for `DOACROSS` loops. We need not (and do not) distinguish between these two types of loop in the rest of this paper.

```
seqfor t0=0 to n-3 do
    L[t0+2,t0+2]=L[t0+2,t0+2]-L[t0+2,t0+1]*U[t0+1,t0+2];
    parfor p2=t0+3 to n do
        L[p2,t0+2]=L[p2,t0+2]-L[p2,t0+1]*U[t0+1,t0+2];
        U[t0+2,p2]=U[t0+2,p2]-L[t0+2,t0+1]*U[t0+1,p2];
        U[t0+2,p2]=U[t0+2,p2]/L[t0+2,t0+2];
    enddo
    parfor p1=t0+3 to n-1 do
        L[p1,p1]=L[p1,p1]-L[p1,t0+1]*U[t0+1,p1];
        parfor p2=p1+1 to n do
            L[p2,p1]=L[p2,p1]-L[p2,t0+1]*U[t0+1,p1];
            U[p1,p2]=U[p1,p2]-L[p1,t0+1]*U[t0+1,p2];
        enddo
    enddo
    L[n,n]=L[n,n]-L[n,t0+1]*U[t0+1,n];
enddo
L[n,n]=L[n,n]-L[n,n-1]*U[n-1,n];
```

Fig. 3. The imerpectly nested version of the same LU program

On the other hand, between *different* time steps, there can be interprocessor dependences (i.e., communications). Thus, the typical optimization task arises in the search for the tile shape which incurs the fewest communications. For this purpose, already existing tiling methods, can be reused (but note that, in general, we need tiling techniques which can deal with affine dependences, such as [25].) However, tile shape optimization is not the focus of this paper.

3.4 Tiling time dimensions

At first glance, tiling time dimensions seems less intuitive. So, let us first focus on *why* it is useful. Let us look again at space-time mapped programs: typically they contain fine-grained parallelism. Note 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 4)![1]

One way out has been to introduce tiling as a kind of preprocessing phase (e.g., [5]). This has several disadvantages: the most important one is that, so far, there are no tiling methods for imperfect loop nests; others are the difficulties arising for dynamic programs [14] or affine dependences, or the fact that tiling as a postprocessing phase for matching the physical number of processors is needed anyway (see the third goal of Section 3.1).

Basic idea of time tiling. Our approach is to tile the time dimensions of the parallelized program: by aggregating a set of logical time steps to a single step

[1] This is the reason for a phenomenon that erverybody who works on parallelization in the polytope model has encountered at least once: the parallelized program suffers a linear slowdown, due to the communication startup times which are usually orders of magnitude higher than the time of a basic computation.

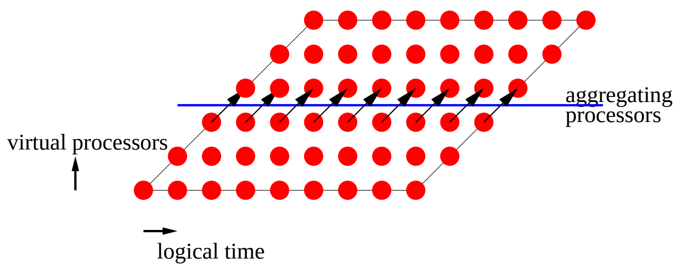

Fig. 4. Target space before tiling the time dimension

(a kind of "superstep"), we can reduce the number of communications: they are only allowed take place between two supersteps! We achieve this by *collecting* all results during the computation of the iterations within one tile, and by *postponing* the communication until the end of all computations of the tile, as indicated in Figure 5.

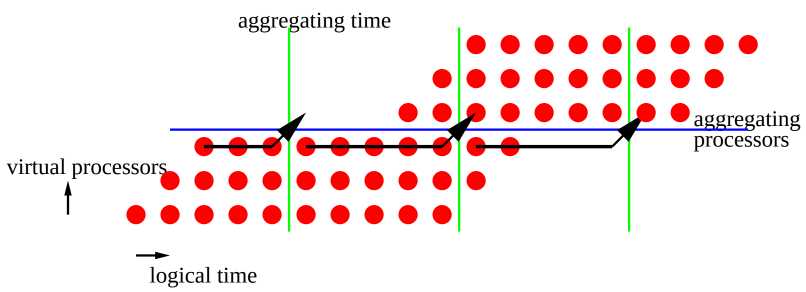

Fig. 5. Target space after tiling the time dimension

Of course, this requires that dependent tiles start their execution later than in the original parallel program, because their execution must be delayed until the data from the tile they depend on have been sent. In Section 4, we derive formally when the application of this technique is possible without introducing a dependence cycle between the tiles.

Remark 3. In asynchronous systems, our typical target architecture, this postponement happens automatically by using blocking receive commands. In synchronous systems, we have to rewrite the loop nest accordingly, so that every tile starts executing after all predecessors have finished.

So, the idea is as follows: by increasing the tiles, we reduce the number of startups, at the price of increasing the communication volume per message and also the computation volume of a tile. In addition, we pay a two-fold price, due to postponing the start of the execution of the first tile for all but the first processors: a larger number of logical time steps and a worse processor utilization.

4 Formal Solution

In this section, we answer the two most important remaining technical questions: "What are the application restrictions of our approach?" and "Does the increased overhead pay off?"

4.1 Possible shapes for tiling space and time loops

First, we have to ensure correctness by finding valid tiles – with the same methods as for tiling sequential programs. This validity constraint guarantees that there is no cyclic dependence between tiles, i.e., we *can* postpone the execution of a tile until its predecessors have finished execution – without introducing dependence cycles between the tiles. In the setting of perfectly nested loops and parallelpipeds as tiles, this is guaranteed if the angle between the tile-generating hyperplanes contains all occurring angles between the dependence vectors [18].

Let us now apply this correctness constraint to space-time mapped loops. For tiling parallel dimensions, we already have established the following lemma.

Lemma 1. *Tiles with length 1 in every time dimension and an arbitrary shape and size in the space dimensions are valid.*

Proof sketch. There are no dependences in the parallel dimensions.

Let us now analyze the situation for time dimensions.

Lemma 2. *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: $(I := (i_1, \cdots, i_k, 0, \cdots, 0) \in \mathcal{T} \wedge I' := (i_1, \cdots, i_k + 1, 0, \cdots, 0) \in \mathcal{T}) \Rightarrow I'' := (i_1, \cdots, i_k, i_{k+1}, \cdots, i_d) \in \mathcal{T}$ *for every I'' in the index space, where d is the maximal nesting depth.*

Proof sketch. 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 shall be tiled with a length greater than 1; more details about how to compute the optimal length are given in Section 4.2 and 4.3. A more precise statement is given by the following corollary.

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

Proof sketch. By contradiction: if there were two such loops, the inner loop l would contradict Lemma 2 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. An algorithm for tiling sequential dimensions is presented in Section 4.3.

Remark 4. Note that the lemmata just presented do not *assume* rectangular tiling – they establish it (for the time dimensions)!

For the rest of this paper, we restrict ourselves to rectangular tiling also for the space loops. In our ongoing research, we have seen that the ideas can, in principle, be transferred to the general case; details and proofs, however, are left for future work.

4.2 Correctness for rectangular tiling

Rectangular tiling is valid if all dependence direction vectors are contained in the hyperplanes generating the rectangles [18]. Thus, these dependence direction vectors are non-negative in every dimension, i.e., they are of the form $(0+, \cdots, 0+)$. Due to the model, this is satisfied for all time dimensions. For dimensions in space, it is an additional constraint.

If this constraint is satisfied for all dependences, then any rectangular tiling is valid. Especially, we can collapse as many of the inner time loops as we need to obtain the desired coarse granularity.

If this constraint is only satisfied for dependences which are carried by one of the inner loops, i.e., they have fixed outer loop indices for dimensions $1, \cdots, k$, then we may collapse all inner loops, but not one of the outer k loops.

If this constraint is not satisfied for the innermost time dimension, our method cannot find a tiling for the time dimensions.

Remark 5. Sometimes the constraint is violated only at the borders of a loop, but is satisfied by the inner iterations. In this case, we can peel off boundary iterations and compute a tiling for the remaining loop iterations.

Remark 6. Note that a direction vector entry of $0-$ in some (or all) of the space components is also valid; we simply reverse enumeration order of the respective processor dimension. Problems arise only if the sign of the direction vector in at least one space component changes.

4.3 The algorithm

The above ideas lead to the following algorithm.

Algorithm.

1. Let t be the innermost time dimension.
2. Check whether the dependences allow to *collapse* dimension t into a single tile. More formally: consider only those dependences which are not carried by one of the outer loops $1, \cdots, t-1$. For them, check whether the direction vector is of the form $(0+, \cdots, 0+)$.
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.
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.

In order to compute the optimal granularity in Step 3, we need some cost function which measures at least the communication startup costs and the increased latency due to the postponed start of some tiles. Once having constructed the cost function for the program under consideration, we can use simple mathematical techniques (derivation) to compute this optimum.

One sufficient standard cost model is as follows: we assume that communication costs $s + v * d$, where s is a constant startup cost, d is the cost for transferring one element of data, and v is the communication volume; a body computation shall have constant cost c in our model. Then, the cost for executing one tile is the volume V of the tile times c for the computation, plus the communication volume v of the tile times d plus a single startup cost s for every processor with which the tile must exchange data.

This model enables an accurate analysis, but at the high cost of counting the number of integer points inside the target polyhedra precisely in order to compute the volume V [9]. In our practical examples, we performed only vague approximations and got reasonably good results.

Example 2. Let us first consider the target program in Figure 2, which has two dimensions in space. We only present the results here:

- Step 2 of the algorithm: all dependences (apart from the initialization iterations) are componentwise non-negative in the two space dimensions [3]. I.e., we may tile the only time dimension arbitrarily.
- To match the virtual to real processors, the tile size along every space dimension must be set to $\lceil n/\sqrt{NP} \rceil$

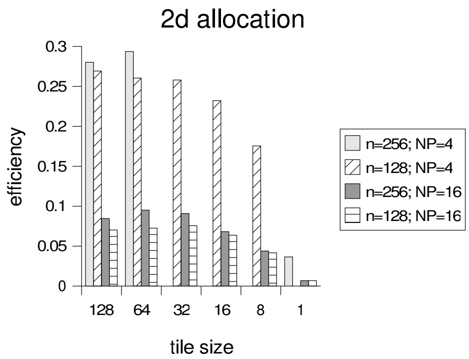

Fig. 6. Efficiency with two-dimensional allocation

- Step 3: with the presented cost model, we computed optimal tile lengths of 10 to 50 along the time dimension, depending on the number of processors (16 or 64) and the matrix size (64, 128, or 256).
- We ran our practical experiments on a PARSYTEC transputer network of 64 T801 processors with 4MB RAM, since these processors have no cache, and we did not want to mix cache effects with the communication effects. We believe that combining the presented method with cache optimization techniques will work well, but details are left for future work.
- The results of the experiments in Figure 6 show that, without time tiling (tile size equals 1), we have less than 5% efficiency in the best case. With time tiling, we can achieve 30% on 4 and 10% on 16 processors. (Missing bars are due to technical problems with our machine.)

The reason for this still inacceptable result is a bad use of the processor array (on average, only a fourth of the processors is active). One way to avoid this problem is by computing a one-dimensional allocation. The resulting program is given in Figure 7.

Again, we apply our method. As before, we may apply time tiling. However, the optimum – in theory and practice – is a tile length of 1. In other words, we may use time tiling, but we should not (see Figure 8). The reason is that, first, at every time step, every processor has a loop to execute, i.e., the granularity is already quite high and, second, for the one-dimensional allocation, only the computed values of U must be communicated (in contrast to L *and* U in the two-dimensional case).

5 Conclusions and Future Work

We have shown that mechanical tiling is possible for imperfectly nested loops. Our approach is to first apply space-time mapping methods for loop parallelization, and then use tiling techniques.

```
 seqfor t1=0 to n-2 do
    parfor p1=t1+2 to n do
       seqfor t2=t1+2 to n do
          if t2 <= p1 then
             L[p1][t2]=L[p1][t2]-L[p1][t1+1]*U[t1+1][t2];
          endif
          if (t1 <= n-3 && p1 <= n-1 && 1+p1 <= t2) then
             U[p1][t2]=U[p1][t2]-L[p1][t1+1]*U[t1+1][t2];
          endif
          if (t1 <= n-1 && p1 == t1+2 && t1+3 <= t2) then
S:           U[t1+2][t2]=U[t1+2][t2]/L[t1+2][t1+2];
          endif
       enddo
    enddo
 enddo
```

Fig. 7. Target program for one-dimensional allocation

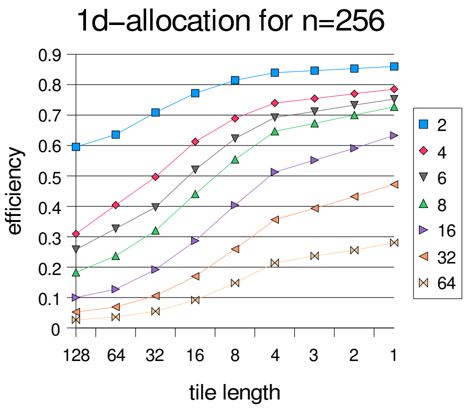

Fig. 8. Efficiency with 1d allocation and various values of $\mathcal{NP}$

We have seen that we have little freedom in the tiling of time dimensions. This lack of choice enabled us to derive an algorithm which computes the optimal tiling of time dimensions for a given tiling of the space dimensions and a given cost model.

We did not focus on finding good tile shapes and sizes for the processor dimensions. In principle, most of the standard tiling techniques can be reused in this framework. A more detailed analysis on this issue is ongoing research.

Future work will also

- deal with load balancing aspects (in the polytope model for parallelization, processor number 1 has often more work to do than the others);
- take into account the subdomains in which the various dependences exist, in order to find a better tiling of the processor dimensions (e.g., by weighting the dependences with the size of their domain or range [10]);
- try piecewise tiling.

Acknowledgment

I would like to thank Christian Lengauer, Nils Ellmenreich, and Peter Faber for their fruitful comments on draft versions of this paper. Thanks also to Sven Anders and Stefan Schuster for their technical assistance and to Jens Knoop for his encouraging mails.

References

1. N. Ahmed, N. Mateev, and K. Pingali. Synthesizing transformations for locality enhancement of imperfectly-nested loop nests. In *Conference proceedings of the 2000 International Conference on Supercomputing*, 2000. To appear.
2. 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.
3. S. Anders. Analyse und Optimierung des Kommunikationsaufwands in automatisch parallelisierten Programmen. Master's thesis, Fakultät für Mathematik und Informatik, Universität Passau, 2000. To appear.
4. R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *J. Parallel and Distributed Computing*, 45:159–165, 1997.
5. 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.
6. D. K. Arvind, K. Ebcioglu, C. Lengauer, and R. S. Schreiber, editors. *Instruction-Level Parallelism and Parallelizing Compilation*. Schloß Dagstuhl, 1999. Report 237.
7. U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations*. Series on Loop Transformations for Restructuring Compilers. Kluwer, 1993.
8. P. Boulet, A. Darte, T. Risset, and Y. Robert. (Pen)-ultimate tiling? *INTEGRATION*, 17:33–51, 1994.
9. 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.
10. P. Clauss. Méthodes polyédriques pour la parallélisation et l'optimisation des programmes. Habilitation thesis, Université Louis Pasteur, Strasbourg, 2000.
11. P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, Feb. 1991.
12. 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.
13. I. Foster. *Design and Building Parallel Programs*. Addison-Wesley, 1995.
14. M. Griebl and C. Lengauer. Classifying loops for space-time mapping. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96: Parallel Processing, Vol. I*, LNCS 1123, pages 467–474. Springer-Verlag, 1996.
15. M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Proc. Int. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111. IEEE Computer Society Press, 1998.
16. 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.
17. K. Högstedt, L. Carter, and J. Ferrante. Selecting tile shape for minimal execution time. In *Proc. 11th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '99)*. ACM Press, June 1999. Also available with proofs as UCSD Tech Report CS99-616.
18. 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.
19. C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

20. 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.
21. 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.
22. Y. Song and Z. Li. A compiler framework for tiling imperfectly-nested loops. In *Languages and Compilers for Parallel Computing (LCPC'99)*, LNCS. Springer-Verlag, 1999. To appear.
23. M. Wolfe. Iteration space tiling for memory hierarchies. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 357–361. SIAM, 1987.
24. J. Xue. Communication-minimal tiling of uniform dependence loops. *J. Parallel and Distributed Computing*, 42(1):42–59, Apr. 1997.
25. J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.
26. J. Xue and C.-H. Huang. Reuse-driven tiling for improving data locality. *Int. J. Parallel Programming*, 26(6):671–696, Dec. 1998.