Forward Communication Only Placements and their Use for Parallel Program Construction

Martin Griebl[1], Paul Feautrier[2], Armin Größlinger[1]

[1] FMI, University of Passau, Germany. {griebl,groessli}@fmi.uni-passau.de
[2] INRIA, Unité de Recherche de Rocquencourt, France. Paul.Feautrier@inria.fr

Abstract. The context of this paper is automatic parallelization by the space-time mapping method. One key issue in that approach is to adjust the granularity of the derived parallelism. For that purpose, we use tiling in the space and time dimensions. While space tiling is always legal, there are constraints on the possibility of time tiling, unless the placement is such that communications always go in the same direction (*forward communications only*). We derive an algorithm that automatically constructs an FCO placement – if it exists. We show that the method is applicable to many familiar kernels and that it gives satisfactory speedups.

1 Introduction

In the field of automatic parallelization the question of selecting the right granularity is still not completely solved. Especially for imperfectly nested loops or non-uniform dependences (not to talk about irregular programs) many questions remain open.

In this paper, we present a method that allows to freely choose the granularity of the parallelism – if possible. Note that it is not the focus of this paper to *find* the optimal granularity for a given program and actual machine parameters, but to offer a technique that yields a parallel program in which the desired granularity can be set freely.

Our parallelization framework is space-time mapping, based on the polytope model [7, 9, 16]. It is designed for automatic parallelization of imperfect loop nests, and has been extended so as to be widely applicable, e.g., to non-uniform dependences, or, sometimes, with a slight loss in efficiency, even to irregular programs. The main idea is that every instance of every statement is mapped to a virtual point in time (*schedule*) and to a virtual processor (*placement*). In other words, the space-time mapping distributes all computations of the source program to as many processors as required. In order to map the parallel program on a machine with a fixed number of physical processors, we must apply standard tiling techniques.

Note that the initial idea and the technical basis of tiling in our setting is the same as in traditional tiling, namely coalescing iterations, but its application is different: we do not discover parallelism by tiling (this is the task of the preceding scheduling phase), but we limit parallelism to the physically possible amount by applying tiling techniques.

When running the resulting parallel programs on distributed memory systems, we usually find that (even for few physical processors) the granularity is still too fine for being efficient. The reason is that typically there are communications after every single virtual time step.

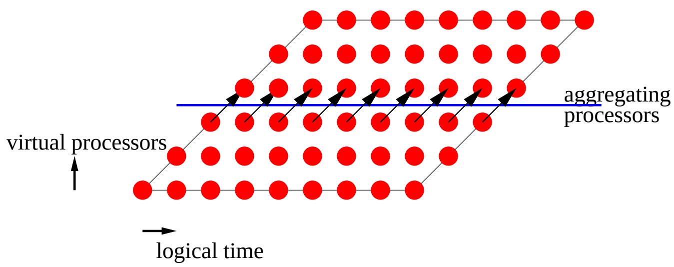

Fig. 1. Target space before tiling the time dimension

Example 1. Consider the program fragment

```
for k=0 to m
   for i=1 to n-1
       A[k,i] = ( A[k,i-1] + 2 * A[k-1,i] ) /3
   end
end
```

After space-time mapping and tiling (partitioning) the one-dimensional processor space, we obtain a space-time mapped iteration domain as in Figure 1. The black arrows represent communications.

The execution times, speedups and efficiency for $(n, m) = (393216, 128)$ are given in Figure 2. The speedups for 2, 4, 8, and 16 processors are 0.94, 1.0, 1.05, and 1.13, which gives poor efficiency values of 0.47, 0.25, 0.13, and 0.07, respectively.

Our solution is to add another tiling phase, which adapts the granularity of the parallelism by coalescing virtual time steps. The idea behind this partitioning of time is (in the setting of distributed memory machines) to postpone and collect all send operations within a time partition and to execute the communications only at the end of the partition. Obviously, this reduces the number of communications. On the other hand, the larger the time partition, the longer the receiver has to wait for its data, i.e., the longer the receiver is delayed. The optimal size for the time partitions depends on the program and on the machine parameters.

Example 2. If we apply this idea to the space-time mapped iteration domain of Figure 1, we obtain the iteration domain in Figure 3, which shows the reduced number of communications and also the increased latency for the upper

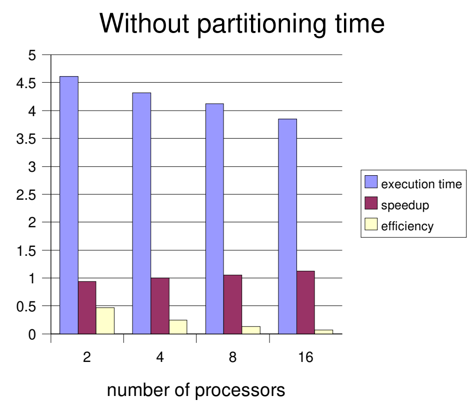

Fig. 2. Execution times, speedup and efficiency after tiling space dimensions only

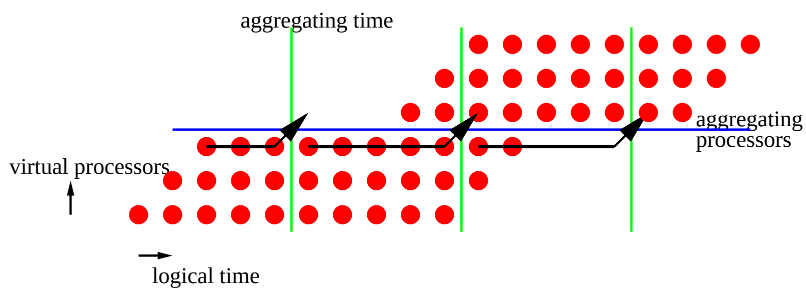

Fig. 3. Target space after partitioning time

processor. The efficiency for the same problem size as above and for different values of the width of the time partitions is depicted in Figure 4. The presence of a maximum in the efficiency curve clearly points to a trade-off between fewer communications and less latency.

The problem is that time tiling may generate deadlocks: suppose that some operation in tile t_1 generates data for a later operation in t_2 while an operation in t_2 generates data for t_1. It is clear that no deadlock can occur if the time is not tiled (since we need at least two operations with different schedules in each tile) or if all communications roughly go into the same direction (e.g. from t_1 to t_2 but not the reverse). A formal definition and proof are given in Section 2. We call this property *forward communications only (FCO)*. A placement satisfying this constraint allows any size for the time partitions [10]. (Note that this constraint is not necessary but sufficient.)

Using FCO placements is not a novel idea. It has been suggested many times as a sure way of avoiding deadlocks. Our aim here is not to advocate the use of FCO placements, but to give an automatic method for building them.

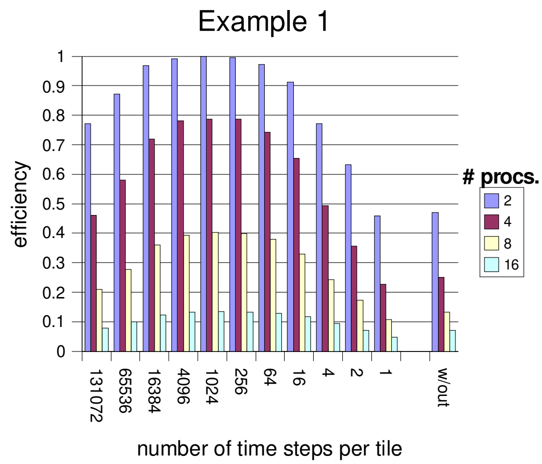

Fig. 4. Execution times after partitioning time

The rest of this paper is organized as follows. Section 2 sets the formal background and derives the FCO placement algorithm. Sections 3 and 4 discuss variants of this algorithm: Section 3 uses a different placement approach, and Section 4 points out some future extensions. Section 5 discusses related work. Section 6 shows some preliminary experimental results and Section 7 concludes.

2 Forward Communication Only Placement

In the presence of loops, every statement S in the body has several instances at run-time. We call them *operations* and denote them by $\langle i, S\rangle$ where the *iteration vector* i is the vector of all loop indices surrounding S. The set of all instances of a given statement S is called the *index set* of S.

In order to use efficient mathematical tools, we require the loop bounds to be affine functions in surrounding loop indices and *structure parameters*, i.e., symbolic constants [7, 16]. (A method avoiding this restriction is given elsewhere [9].)

In our mathematical notation, we often use the *homogeneous representation* of index vectors: we join the l-vector i of surrounding loops indices and the m-vector n of structure parameters in order to obtain the d-dimensional homogeneous index vector. Note that the m-vector of structure parameters shall always contain one entry for the constant 1.

In the affine setting, the c bounds of the loops surrounding a statement S can be expressed as a system of linear inequalities and represented as a $c \times d$ matrix D_S with

$$D_S . \begin{pmatrix} i \\ n \end{pmatrix} \geq 0 \qquad (1)$$

where i is the iteration vector of S, and n is the vector of all structure parameters. For consistency, we take care that the trivial inequality $1 \geq 0$ is always included in D_S.

A *computation placement* π is a function which maps every operation to an integer vector that represents a virtual processor. Again, we require placements to be affine in the loop indices and the structure parameters. Hence, the placement of every statement S, π_S, can be represented by a $p \times d$ matrix Π_S where p is the number of processor dimensions, and $d = l + m$, i.e. d is the dimensionality of the index set of S plus the number of symbolic parameters:

$$\pi_S(i, n) = \Pi_S . \begin{pmatrix} i \\ n \end{pmatrix} \tag{2}$$

Similarly, a *data placement* maps array elements to virtual processors. For each array A, we express this placement as:

$$\pi_A(a, n) = \Pi_A . \begin{pmatrix} a \\ n \end{pmatrix} \tag{3}$$

where a is the vector of A subscripts and n is as above.

Lastly, we need a *schedule* function θ, which maps operations to (virtual) time. Schedules are assumed to be affine in the loop indices and the structure parameters, as this is necessary for subsequent target code generation.

In general, each operation $\langle i, S \rangle$ both reads and writes memory. Our basic assumption is that these accesses are to array cells. Let A be one of the arrays accessed by S. We assume that we have been able to extract from the program text a subscript function f_{AS} such that the cell of A accessed by S is $A[f_{AS}(i, n)]$. Here again we suppose f_{AS} to be affine: there exists a matrix F_{AS} such that:

$$f_{AS}(i, n) = F_{AS} . \begin{pmatrix} i \\ n \end{pmatrix} . \tag{4}$$

In Example 1, the F matrix for the rightmost reference to A is $\begin{pmatrix} 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 \end{pmatrix}$.

Let $A[f_{AS}(i, n)]$ be a read reference to A in S. If this array cell is not on the same processor as operation $\langle i, S \rangle$, a communication is necessary. This communication will be forward if:

$$\pi_A(f_{AS}(i, n), n) \leq \pi_S(i, n). \tag{5}$$

On the other hand, if the distinguished reference is a write, it will be forward if:

$$\pi_S(i, n) \leq \pi_A(f_{AS}(i, n), n). \tag{6}$$

These inequalities are to be understood component-wise. They are to be verified everywhere in the index set D_S of S. The conjunction of these properties for all references in the program defines a forward communication only (FCO) placement. (Note that the definition of the direction is arbitrary: we can always reorder processors independently in each dimension).

A *tile* is a set of operations which are executed atomically by one processor. Operations of a tile are executed sequentially. In this paper, we use a very simple tiling scheme. Let T be the tile size in time and B be the tile size in space[1]. Operation $\langle i, S \rangle$ is executed by physical processor $\pi_S(i,n) \div B$ in its $\theta_S(i,n) \div T$-th time step.

Arrays are tiled according to the same scheme: cell $A[x]$ is in the memory of physical processor $\pi_A(x,n) \div B$. The communication graph has the tiles as vertices; there is an edge from tile a to b if a sends data to b.

Theorem 1. *Any space/time tiling according to an FCO placement is valid.*

Proof. For easier understanding, the proof will be written as if the schedule and placement were one-dimensional. Extension to several dimensions is trivial.

A tiling is valid if there are no cycles in the communication graph. Let us suppose a contrario that such a cycle exists. For $k = 0, \ldots, \ell - 1$, tile (t_k, p_k) sends data to tile (t_{k+1}, p_{k+1}) and tile (t_ℓ, p_ℓ) sends data to tile (t_0, p_0). For each communication, there is an emitter x (a memory cell or an operation) and a receiver y (an operation or a memory cell), each one having a placement function π_e (resp. π_r). The FCO condition implies:

$$\pi_e(x) \leq \pi_r(y),$$

from which follows:

$$p_e = \pi_e(x) \div B \leq \pi_r(y) \div B = p_r,$$

where p_e (resp. p_r) is the name of the (real) processor executing (or holding) x (resp. y). Furthermore, the inequality is strict, since there actually is a communication.

We have just proved that $p_k < p_{k+1}$ for $k = 0, \ldots, \ell - 1$ and $p_\ell < p_0$ which is impossible since $<$ is an order.

Let us now consider one of the FCO conditions, (5) for instance. It can be rewritten as:

$$\forall \begin{pmatrix} i \\ n \end{pmatrix} : D_S . \begin{pmatrix} i \\ n \end{pmatrix} \geq 0 \Rightarrow \mathit{\Pi}_S . \begin{pmatrix} i \\ n \end{pmatrix} - \mathit{\Pi}_A . F_{AS} \begin{pmatrix} i \\ n \end{pmatrix} \geq 0. \tag{7}$$

Farkas' lemma [20] shows how such an affine inequation system can be transformed into an equivalent equation system by adding non negative variables. Thus, (7) is equivalent to:

$$\mathit{\Pi}_S - \mathit{\Pi}_A . F_{AS} = \lambda_{AS} D_S . \tag{8}$$

where the Farkas multipliers λ_{AS} are non negative. In this equation, the $\mathit{\Pi}_S$, $\mathit{\Pi}_A$ and λ_{AS} are unknowns, while F_{AS} and D_S can be deduced from the source program. Similar considerations apply to (6).

[1] When the schedule and/or placement are multidimensional, T and B become vectors, the integer division operator $\div$ being extended componentwise.

Let Π be the vector obtained by concatenating the Π_A and Π_S in some order, and λ be the vector obtained by concatenating the λ_{AS}. (The fact that the entries of Π and λ are p-vectors themselves is irrelevant for the following reasoning.) It is clear that there exist matrices C and D such that the FCO condition is equivalent to:

$$C.\Pi = D.\lambda, \tag{9}$$

$$\lambda \geq 0. \tag{10}$$

The set of solutions of this system (i.e. the set of valid FCO placements) is a cone $\mathcal{C}$ (it is closed both by addition and by multiplication by a non-negative constant). Let $\langle \Pi, \lambda \rangle$ be such a solution; let us consider a specific reference to A in S. There is a part of λ which corresponds to λ_{AS} in (8). If this part is null, then the distinguished reference entails no communication. Let $\langle \Pi_1, \lambda_1 \rangle$ and $\langle \Pi_2, \lambda_2 \rangle$ be two solutions. It is clear that $\langle \Pi_1 + \Pi_2, \lambda_1 + \lambda_2 \rangle$ is another solution whose residual communications are the union of the residual communications of the two initial solutions. This leads us to consider only extremal solutions, which cannot be obtained as a weighted sum of other solutions.

Any cone can be characterized [20] by its extremal rays $r_1, \ldots, r_s$ and its lines $l_1, \ldots, l_t$ in such a way that:

$$\mathcal{C} = \{\sum x_k r_k + \sum y_k l_k \mid x_k \geq 0\}. \tag{11}$$

There are well known algorithms for finding the rays and lines of a cone, and at least one efficient implementation, the Polylib [21].

Let us now consider a line $l_k = \langle \Pi_k, \lambda_k \rangle$. Since l_k is a line, $\langle -\Pi_k, -\lambda_k \rangle$ is also in $\mathcal{C}$. By (10) we obtain $\lambda_k \geq 0$ and $-\lambda_k \geq 0$ which implies $\lambda_k = 0$.

Conversely, if $\langle \Pi_k, \lambda_k \rangle$ is a ray with $\lambda_k = 0$, then $\langle -\Pi_k, -\lambda_k \rangle$ is also a solution and the ray is a line. It follows that lines correspond to communication-free placements, and that rays correspond to FCO placements with residual communications. Furthermore, an analysis of the null components of the λ part of a ray allows one to identify residual communications. If we assign a weight to each reference (e.g. an estimate of the number of transmitted values), we can associate a weight to each ray and select the one with minimum weight (remember that in this context, lines will show up as zero weight solutions).

However, we still have to consider parallelism. Let Π_S be the part of a solution which corresponds to statement S. While up to now we have considered Π_S as a vector, it is in fact a matrix with p rows, where p is the dimension of the processor grid. The set of active processors is the image of the index set of S by Π_S. In order to preserve efficiency, we want this set to have the same dimension as the processor grid (however, this dimension cannot be higher than the dimension of S index set). Finding the dimension of the set of active processors is a simple rank computation.

We can thus propose the following algorithm:

- Build the matrices C and D from the source program.
- Build the rays and lines of the cone $\mathcal{C}$ associated to C and D.

- Filter out rays and lines which do not satisfy the rank condition above.
- Compute the weight of each remaining ray or line.
- Select the ray or line with the smallest weight.

If a line has survived the filtering process, it has zero weight and will be selected, giving a communication free placement. If the selectee is a ray, it will give an FCO placement with minimum communication volume. Lastly, if there are no survivors, then the problem has no FCO placement of the required dimension.

We cannot claim that the placement we find in this way is the best one, in the sense of giving the best speedup. However, if the weights we assign to communications are estimates of the communication volumes, then our algorithm is a greedy solution to the problem of finding a minimum communication FCO placement.

Let us note that the severity of the filtering increases with the dimension of the processor grid. Hence, we can always try again with a grid of a smaller dimension. In general, the higher the dimension, the higher the volume of residual communications, but also the higher the bandwidth of the communication network. Since the relative importance of these two opposite factors depends on details of the architecture, the best choice can only be found experimentally.

3 Another approach: Dependence driven placements

The presented placement algorithm computes one computation placement per statement and one data placement per array. However, there also exists other approaches for the computation of placements. We show how our basic FCO placement algorithm can be adapted accordingly.

One possibility is to drop the notion of ownership and assume that every processor holds the data it computes, and that it sends the data directly to every consumer. We call such a placement method *dependence driven*, in contrast to the original method which we call *ownership driven*.

Note that we have a very strong notion of dependences in this context: we use *direct* dependences for this approach. On the one hand, this requires a precise dataflow analysis, e.g. [2, 4]. On the other hand, the result is as precise as if we had converted the program to single assignment form: we can tell, for every operation, where the accessed data is located – because we know the source of the direct dependence, i.e., the producer in the case of flow dependences.

Note that if some array element $A[x]$ is re-assigned, the new producer holds the new value and, as written above, sends it to those processors that need this new value. Thus, we cannot say that $A[x]$ is owned by some processor, because the "ownership" for $A[x]$ changes. In this aspect, the dependence driven approach is more flexible than the ownership driven approach.

On the other hand, the implicit owner of every element (provided that it exists, e.g., because the program is single assignment) is its producer. There is no possibility that the producer stores the value at some different processor if this would be beneficial. So, in this aspect the ownership driven approach is more flexible [8] .

The construction of a dependence driven FCO placement can be achieved along the same lines as above. There is one placement constraint per dependence in the program.

A dependence d is given as a relation from the source index set to the destination index set:

$$\{\langle i, n, S\rangle \to \langle j, n, T\rangle \mid R_d . \begin{pmatrix} i \\ j \\ n \end{pmatrix} \geq 0\},$$

in which we have assumed that the dependence is representable as one polyhedron. For every such dependence, we require the FCO property:

$$\pi_S(i, n) \leq \pi_T(j, n). \tag{12}$$

This can be rewritten as

$$\forall i, j, n \,:\, R_d . \begin{pmatrix} i \\ j \\ n \end{pmatrix} \geq 0 \;\Rightarrow\; \Pi_T . \begin{pmatrix} j \\ n \end{pmatrix} - \Pi_S . \begin{pmatrix} i \\ n \end{pmatrix} \geq 0. \tag{13}$$

From then on, the algorithm follows the same lines as above. We eliminate quantifiers with the help of Farkas lemma, then find the rays and lines of the solution cone, and select the best one.

4 On the use of redistribution

The ownership driven approach has the drawback that an array has only one placement for all the execution of a program. This is unsatisfactory: many programs can be divided in successive phases with differing access patterns to arrays. Hence, we need the ability to freely determine a data placement, but also to change this data placement during program execution. Let us discuss this on an example.

Example 3. Consider the source program in Figure 5. There, we can avoid any communication due to the two-dimensional (hence, most important) accesses to arrays A and B by the following mapping: $A[x, y] \mapsto x$ and $\langle i, j, T\rangle \mapsto i$ (this eliminates the dependences cycle inside T), and $B[l, k] \mapsto l$ and $\langle l, k, U\rangle \mapsto l$ (this eliminates the dependence from T to U due to A and enables a local store of B).

Furthermore, we map $\langle i, S\rangle \mapsto i$ and $C[z] \mapsto z$ in order to eliminate communications due to accesses of C in S and T. This solution is optimal if we allow one mapping per array and per statement – even if every of the n^2 accesses to $C[l-1]$ in U causes a communication.

A much better solution would be if we could re-map array C between its uses in T and U. If we re-map $C[l-1]$ to l before executing U, then there are no communications caused by U. The cost for the redistribution is one

```
    DO i=0,n-1
S:    C[i] = 42;
      DO j=0, n-1
T:       A[i,j] = A[i,j-1] + C[i]
      END DO
    END DO
    DO l=1,n-1
      DO k=0, n-1
U:       B[l,k] = A[l,k] + C[l-1]
      END DO
    END DO
```

Fig. 5. A source program that needs redistribution

read/re-store per element of C, i.e., the redistribution causes only linearly many communications.

How can we modify our placement algorithm in order to find this solution? The first step is to split the first loop. We then add redistribution points in the source program, i.e., technically, we add artificial statements that read all elements of the array to be redistributed and copy them to a new array (and update the subsequent accesses to the new array). This scheme has the added advantage of limiting the complexity of each elementary placement problem, thus improving the scalability of our approach.

After inserting redistribution points for array C between the loops on S and T, and also between the loops on T and U, and applying our placement algorithm, we obtain:

- between S and T: $C'[z] \mapsto z$
- between T and U: $C''[z] \mapsto z+1$

This means that we should not redistribute C between S and T, but between T and U – the expected result.

The central question for this approach is where to insert redistribution points, and for which arrays. One heuristics is to try redistribution along the edges of the acyclic condensation of the statement dependence graph. On the one hand this allows redistribution between different phases of an algorithm (where redistribution might be most important); on the other hand it guarantees that the expensive re-mapping is not executed too often, esp. not executed repeatedly back and forth, since it forbids redistribution inside dependence cycles. Of course, other strategies can be imagined as well.

In addition, there are other possibilities to make placement algorithms more flexible (e.g., to allow replication of arrays or even redundant computations, or to deal with piecewise affine placements, e.g., via index set splitting [12]). We leave this for our ongoing work.

5 Related Work

Tiling has many applications in program optimization. We will not consider here its use for locality improvement in sequential programs as in the work of Wolfe [23] or Xue et. al. [26]. Tiling may be used as a parallelization method. This approach was first proposed by Triolet [15]. The shape of the tile is first chosen in such a way that deadlocks are avoided. The parallel program is then constructed by a simple application of the hyperplane method. Lastly, the size of the tiles is adjusted for minimum run time [1, 13, 14, 18, 19, 24, 25].

Another approach consists of applying tiling after parallelization in order to adjust the grain of parallelism [22]. This has lead to the definition of fully permutable loop nests. The present paper belongs to this category. It differs from previous proposals in that we do not apply tiling either to arrays or to index sets, but to time and space. In a previous work [11], the first author explained in more detail why the parallelization procedure described in Section 1 can be superior to the traditional tiling approach. The most important reasons are a wider applicability and, at the same time, a possibly better quality of the result.

There also exist multiple papers about placement functions, some of them using the same framework as this paper [3, 6, 17]. However, to the best of our knowledge, this is the first time that automatic construction of FCO placements is considered. In Lim and Lam terminology [17], our methods apply when constant parallelism is not sufficient for taking benefit of all processors.

The use of Farkas lemma for quantifier elimination in formulas like (7) has been first proposed by the second author [5], however in a different application area.

6 Experiments

Our placement algorithm has been implemented as an extension to the LooPo parallelizer and tested on about ten kernels, some real and some artificial. These kernels are available on demand from the authors. We found FCO placements for all examples, and even some communication free placements. The largest examples where "burg" (a signal processing kernel with 22 lines of code) and "LCZOS" (a Lanczos iteration with 60 lines). The algorithm has removed 31 communications out of 44 in the first case and 62 out of 64 in the second case.

We then tested the performances of our target code on an SCI-connected network of 32 nodes, every node (board) with two Pentium 3 processors at 1 GHz and 512 MB of main memory. In order to avoid effects due to the shared memory on the boards, we only used one processor per node. We took gcc-2.96 -O2 for the compilation and SCAMPI as communication library.

Our first experiments show that tiling time is necessary for some cases. As a rule of thumb, these cases arise for loop nests where one dimension goes to space and all other dimensions are covered by the schedule. In this situation, we must reduce the number of communication phases which, before tiling time, take place at every iteration of the sequential loops.

```
DO J=1,M
  DO I=2,N-1
      A(I)=(A(I-1)+
           A(I+1))/2.0
  END DO
END DO
```

SOR 1-dimensional

```
DO k=0, n-1
   sum[n-k] = b[n-k]
   DO l=0, k-1
      sum[n-k]=sum[n-k]-
               a[n-k][n-l]*b[n-l]
   END DO
   b[n-k]=sum[n-k]/a[n-k][n-k]
END DO
```

LUBKSB

Fig. 6. Example programs that need partitioning of time

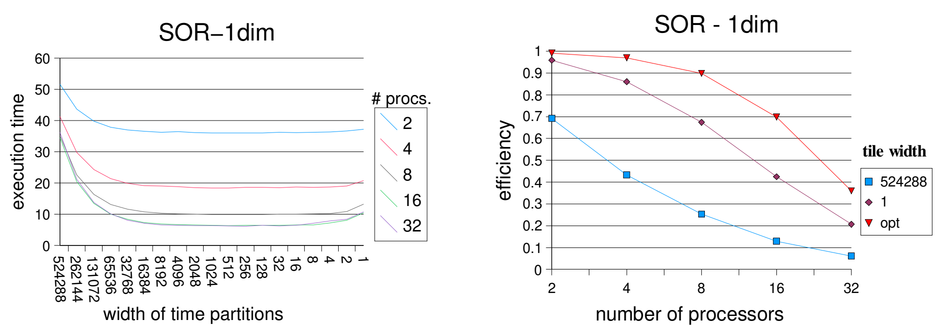

Fig. 7. Execution time and efficiency for SOR

We use the programs in Figure 6. The SOR algorithm has uniform, the LUBKSB (LU Backward Substitution) non-uniform but affine dependences; the complex array indices in LUBKSB result from loop normalization (in the initial program, the loops are counting backward). The schedules for SOR and the three statements of LUBKSB are $2*J+I-4$ and 0, $2*l+2$, $2*k+1$, and the FCO placements generated by our algorithm are J and k, k, k, respectively. We give the execution time and speedup for different numbers of processors and different widths of the time partitions in Figures 7 and 8.

For the SOR experiment, we set M to 6144 and N to 1048576; the resulting original sequential execution time was 180.5 seconds. Due to cache effects, the optimized parallel program on one processor needed only 71.4 seconds. This is an important collateral benefit: the aim of placement algorithms is to improve locality. This results not only in less communications, but also in less cache misses. Figure 7, right, shows the efficiency (with respect to this improved sequential time). We can see that the efficiency for the optimal time partitioning is about 15 to 30 % higher than without partitioning time, i.e., with time partition width of 1. On the other end of the spectrum, long time partitions (width = 524288) give up nearly all parallelism and so do not scale at all. Note that for the chosen value for parameter M, the 32 processors are not fully used; this becomes better

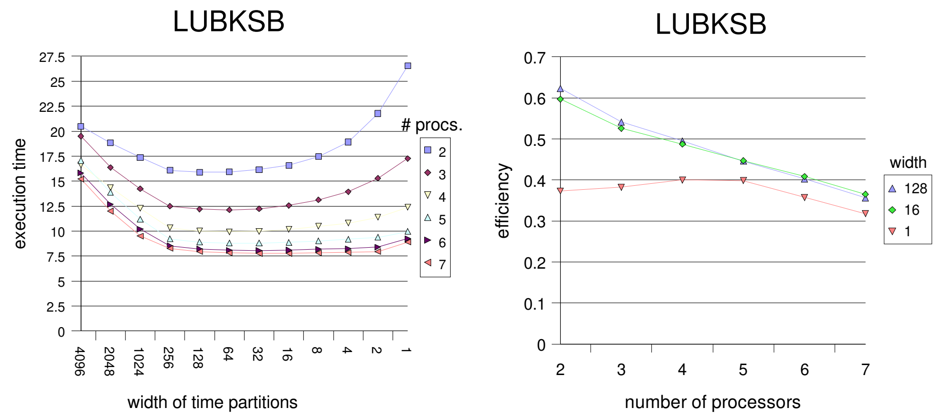

Fig. 8. Execution time and efficiency for LUBKSB

for larger M (but at the same time the importance of partitioning time decreases for the smaller number of processors).

In the LUBKSB experiment we use $N = 10240$ and obtain a sequential execution time of 19.81 seconds. The parallel version executes in 20.69 seconds on a single processor. We do not observe a speedup due to cache effects here since both programs access the array a in a cache friendly way. Figure 8 shows that we achieve the highest speedup with tile sizes between 16 (on 7 processors) and 128 (on 2 processors). This example does not scale as well as the SOR example, because the iteration space is triangular, hence the work is distributed unevenly among the processors. A possible solution is to build tiles with variable size, but we have not worked out all the details of this technique.

7 Conclusions

As we have seen in Section 6, partitioning in the time direction is important in order to obtain good speedups for some kinds of algorithms. However, partitioning time is not always legal. A sufficient condition for legality is that all communications of the parallel program go forward in every dimension (FCO). This condition is also necessary in one dimension.

The main theme of this paper has been the development of an algorithm for the automatic construction of FCO placements. This algorithm has been implemented as an extension to the LooPo parallelizer and used for all the examples in this paper. Experiments show that the transformed programs have satisfactory performances on a cluster of PC, although better load balancing is needed in some cases.

Although we have not emphasized the point, the method can be generalized to handle programs beyond the strict polytope model: modulo and integer division in the subscripts, min and max operators in the loop bounds, tests on the loop indices, union of polytopes in the dependence descriptions, and even infinite iteration domains as in signal processing.

We intend to pursue this work in several directions:

- Analyze the FCO placement algorithm. Can its complexity be reduced? Find examples in which no FCO placement can be found.
- Build a rough cost model for the tiled program, in order to help the selection of a good tile size. Can this model help in the construction of programs with tiles of varying size?
- Compare the ownership driven and the dependence driven approaches as to applicability, complexity and efficiency.
- Explore the redistribution approach, with a view of improving the scalability of the compiler.

Acknowledgments

The first author would like to thank Michael Classen for his support with the experiments, and Max Geigl for his fruitful comments on a draft version of this paper.

All authors acknowledge the help of the French-German exchange program PROCOPE (grant 02969TB on the French side).

References

1. Pierre Boulet, Alain Darte, Tanguy Risset, and Yves Robert. (Pen)-ultimate tiling? *INTEGRATION*, 17:33–51, 1994.
2. Jean-François Collard and Martin Griebl. A precise fixpoint reaching definition analysis for arrays. In Larry Carter and Jeanne Ferrante, editors, *Languages and Compilers for Parallel Computing, 12th International Workshop, LCPC'99*, LNCS 1863, pages 286–302. Springer-Verlag, 1999.
3. Michèle Dion and Yves Robert. Mapping affine loop nests: New results. In Bob Hertzberger and Giuseppe Serazzi, editors, *High-Performance Computing & Networking (HPCN'95)*, LNCS 919, pages 184–189. Springer-Verlag, 1995.
4. Paul Feautrier. Dataflow analysis of array and scalar references. *Int. J. Parallel Programming*, 20(1):23–53, February 1991.
5. Paul Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, 1992.
6. Paul Feautrier. Toward automatic distribution. *Parallel Processing Letters*, 4(3):233–244, 1994.
7. Paul Feautrier. Automatic parallelization in the polytope model. In Guy-René Perrin and Alain Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.
8. Paul Feautrier. 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, March 2000. English translation of TSI vol. 15 pp 529–557, 1996.

9. Martin Griebl. *The Mechanical Parallelization of Loop Nests Containing* while *Loops.* PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, January 1997. Technical Report MIP-9701.
10. Martin Griebl. On the mechanical tiling of space-time mapped loop nests. Technical Report MIP-0009, Fakultät für Mathematik und Informatik, Universität Passau, August 2000.
11. Martin Griebl. On tiling space-time mapped loop nests. In *Thirteenth annual ACM symposium on parallel algorithms and architectures (SPAA 2001)*, pages 322–323, July 2001.
12. Martin Griebl, Paul A. Feautrier, and Christian Lengauer. Index set splitting. *Int. J. Parallel Programming*, 28(6):607–631, 2000.
13. Edin Hodžić and Weijia Shang. On time optimal supernode shape. In *Eighth Int. Workshop on Compilers for Parallel Computers (CPC 2000)*, pages 367–379, 2000.
14. Karin Högstedt, Larry Carter, and Jeanne 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. François Irigoin and Remi Triolet. Supernode partitioning. In *Proc. 15th Ann. ACM Symp. on Principles of Programming Languages (POPL'88)*, pages 319–329. IEEECS, January 1988.
16. Christian Lengauer. Loop parallelization in the polytope model. In Eike Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.
17. Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine partitions. *Parallel Computing*, 24(3–4):445–475, May 1998.
18. Daniel A. Reed, Loyce M. Adams, and Merrell 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.
19. Robert Schreiber and Jack J. Dongarra. Automatic blocking of nested loops. Technical Report CS-90-108, University of Tennessee, Computer Science, May 1990.
20. A. Schrijver. *Theory of Linear and Integer Programming*. Series in Discrete Mathematics. John Wiley & Sons, 1986.
21. Doran K. Wilde. A library for doing polyhedral operations. Technical Report 785, IRISA, December 1993.
22. Michael Wolf and Monica Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, October 1991.
23. Michel Wolfe. Iteration space tiling for memory hierarchies. In G. Rodrigue, editor, *Parallel Processing for Scientific Computing*, pages 357–361. SIAM, 1987.
24. Jingling Xue. Communication-minimal tiling of uniform dependence loops. *J. Parallel and Distributed Computing*, 42(1):42–59, April 1997.
25. Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.
26. Jingling Xue and Chua-Huang Huang. Reuse-driven tiling for improving data locality. *Int. J. Parallel Programming*, 26(6):671–696, December 1998.