Message-Passing Code Generation for Non-rectangular Tiling Transformations

Georgios Goumas, Nikolaos Drosinos, Maria Athanasaki and Nectarios Koziris

National Technical University of Athens

Computing Systems Laboratory

Abstract

Tiling is a well known loop transformation used to reduce communication overhead in distributed memory machines. Although a lot of theoretical research has been done concerning the selection of proper tile shapes that reduce processor idle times, there is no complete approach to automatically parallelize non-rectangularly tiled iteration spaces and consequently there are no actual experimental results to verify previous theoretical work on the effect of the tile shape on the overall completion time of a tiled algorithm. This paper presents a complete end-to-end framework to generate automatic message-passing code for tiled iteration spaces. It considers general parallelepiped tiling transformations and convex iteration spaces. We aim to address all problems concerning data parallel code generation efficiently by transforming the initial non-rectangular tile to a rectangular one. In this way, data distribution and the respective communication pattern become simple and straightforward. We have implemented our parallelizing techniques in a tool which automatically generates MPI code and run several benchmarks on a cluster of PCs. Our experimental results show the merit of general parallelepiped tiling transformations, and verify previous theoretical work on scheduling-optimal, non-rectangular tile shapes.

Key words: Loop tiling, clusters, data parallel, code generation, MPI.

Preprint submitted to Elsevier Science *26 June 2006*

1 Introduction

Tiling or supernode transformation is one of the most popular loop transformations discussed in literature, proposed to enhance cache locality in uniprocessors and exploit coarse-grain parallelism in multiprocessors. Previous work on tiling for locality in uniprocessor systems addresses issues such as selecting loop transformations which increase locality of references in caches ([29]), as well as determining the appropriate tile size ([19]). The corresponding results are very well established and many of the proposed techniques have been incorporated in research and commercial compilers. On the other hand, when tiling for parallelism is concerned, the application of the related theoretical methods in real compilers is rather limited. In this case, each tile groups together a number of iterations, which are executed uninterruptedly, while communication between processors occurs just before and after the computations within a tile, in order to reduce the communication frequency and volume. Under this scheme, a lot of discussion has been made concerning the selection of an "optimal" tiling transformation. Similar to the case of tiling for locality, controlling the tile size is of the utmost importance, since large tile sizes reduce the overall communication whereas small ones provide more parallelism. Another very important factor is the tile shape. While tiles constructed to achieve locality in the memory hierarchy are always rectangular due to linear memory layouts, it has been shown that this is not always the optimal case when tiling for parallelism.

The tile shape affects inter-processor communication. Ramanujam and Sadayappan in [23] first emphasized on the use of non-rectangular tiles in order to minimize inter-processor communication. Specifically, they showed the equivalence between the problem of finding a set of extreme vectors for a given set of dependence vectors and the problem of finding a tiling transformation H that produces valid, deadlock-free tiles. Since the tiling planes h_i are defined by a set of extreme vectors, they gave a linear programming formulation for the problem of finding optimal shape tiles, thus determining optimal H that reduces communication. Boulet et al. in [6] used a per tile communication function that has to be minimized by linear programming approaches. Based on this function, Xue in [31] presented a complete method to determine the tiling transformation H, which imposes minimum communication.

More importantly, the tile shape also greatly affects the overall completion time of an algorithm. In [16,9] the authors present analytical expressions of the idle time of a processor for 2-dimensional tiled spaces. This idle time is either

Email addresses: goumas@cslab.ece.ntua.gr (Georgios Goumas), ndros@cslab.ece.ntua.gr (Nikolaos Drosinos), maria@cslab.ece.ntua.gr (Maria Athanasaki), nkoziris@cslab.ece.ntua.gr (Nectarios Koziris).

the time a processor is waiting for data from another processor, or the time spent by a processor at a barrier waiting for other processors to accomplish their tasks. It is shown that the idle time depends on a parameter that relates the size of the tile to the size of the iteration space. Hodzic and Shang in [14] discussed the effect of the tile shape and size on the overall completion time of an algorithm taking into account the iteration space bounds. However, they are restricted to rectangular spaces and tiles. In [15], Hodzic and Shang proved that the scheduling-optimal tile shape, i.e. the one that leads to the minimum execution time, is derived from the algorithm's tiling cone. Högstedt et al. in [17] and [18] extend their work from [16] to more deeply nested loops and also confirm that the vectors forming the basic tile shape should be taken from the surface of the tiling cone. This means that if we properly scale n vectors taken from the surface of the tiling cone of an algorithm according to the bounds of the iteration space, we can simultaneously obtain scheduling-optimal and communication-minimal tiling. The problem of determining the tile size (i.e. properly scaling the basic tile shape taken from the surface of the tiling cone in our context) was also attacked by Andonov et al. in [4] and Xue and Cai in [32]. The general problem is an extremely complex one, so the results given in [4,32] suppose some restrictions: the loops are 2-dimensional and one of the tile boundaries is parallel to the iteration space boundaries. Their solution is obtained by formulating and resolving a discrete nonlinear optimization problem.

Despite all these methods for the selection of a proper tiling transformation to minimize communication volume and overall execution time in distributed memory machines, general parallelepiped tiling is not used either in commercial or in research compilers ([2,3,8,11,26]). This is due to the fact that no complete approach has been presented concerning implementation issues for non-rectangular tiling transformations. In general, the parallelizing compiler community has been pessimistic about generating code for non-rectangular tiling transformations. On the other hand, all this theoretical work needs to be experimentally verified. Note that, besides [4] and [32] that present experimental results for 2-dimensional spaces, all previous research is purely theoretical. The main question is whether the overhead imposed by enumerating general parallelepiped tile shapes is annihilated by the theoretically proven gain in overall execution time, since non-rectangular tiles may lead to a reduction in idle time in the context of [9,16,17].

Furthermore, in the majority of the problems where tiling is a candidate transformation to attain higher performance, rectangular tile shapes are not valid. The problem occurs when there are negative coefficients in some dependence vectors. In these cases, the parallel execution of rectangularly tiled iteration spaces would result in a deadlock ([24]). There are two ways to overcome the problem: one can either apply skewing transformation in order to convert all dependence elements into nonnegative (or equivalently transform the loop nest into a fully permutable one [30]), or apply a valid non-rectangular tiling transformation to the original iteration space. Both approaches present similar complexities in the automatic generation of parallel code.

Motivated by the above facts, we aim to parallelize n-dimensional loops that have been transformed by non-rectangular tiling transformations. Our goal is to provide a method to generate parallel code for non-rectangular tile shapes and investigate the effect of the tile shape on real applications in order to verify previous theoretical work. We focus on the tile shape, since the results for the tile size are not yet general and well understood. We present a complete approach to automatically generate data-parallel code for arbitrarily tiled iteration spaces to be executed on distributed memory machines. We address issues such as transformed loop bound calculation, iteration and data distribution and automatic message passing code generation. In order to efficiently generate data-parallel code and keep the relative overhead imposed by non-rectangular tiles as low as possible, we continue previous work on efficient sequential tiled code generation. More specifically, in [12] we presented an approach to drastically reduce the compilation time for tiled iteration spaces. We transformed the generally non-rectangular tile into a rectangular one using a non-unimodular transformation H' directly deriving from the tiling transformation H. We called the transformed (rectangular) tile in the axes origins the Transformed Tile Iteration Space ($TTIS$). We used the Hermite normal form $\widetilde{H'}$ of H' to determine the exact bounds and strides of the loop that will traverse the $TTIS$. The introduction of the $TTIS$ significantly reduces the difficulty brought about by parallelepiped tile shapes, as far as code generation is concerned. In addition, the generated code is much more efficient, avoiding unnecessary computations for boundary calculations. We shall continue using this transformation in the parallelization process. We assign chains of transformed rectangular tiles to each processor and allocate proper local data spaces. Using H', local iteration and data spaces are both rectangular, enabling efficient memory management, while translation between the two local spaces is also simple and straightforward. In addition, following this scheme, we deduce very simple compile-time criteria to determine the communication points. Thus, we parallelize tiled iteration spaces with a negligible compile-time and run-time overhead, completely dwarfed by the considerable gain in parallel execution speedup.

We have implemented a tool that automatically generates data parallel MPI code and run several benchmarks (SOR, Jacobi, ADI) on a cluster of workstations interconnected via FastEthernet. Our goal is to accentuate the merit of non-rectangular tiling transformations and to verify previous theoretical work proposing the selection of a tiling transformation parallel to the tiling cone. Indeed, our experimental results show that a proper non-rectangular tiling transformation can lead to a remarkable increase in execution speedups. Summarizing, this paper makes the following contributions:

- It presents an efficient end-to-end method to generate data-parallel code for non-rectangular tiling transformations.
- It experimentally verifies previous theoretical work on determining proper tile shapes to reduce processor idle times.

The rest of the paper is organized as follows: In Section 2 we define our problem domain along with some notation used throughout the paper, we describe tiling transformation and review previous work on efficient sequential tiled code generation. In Section 3 we present our implementation framework including computation distribution and data distribution, while Section 4 discusses message passing code generation. Section 5 presents experimental results from the application of our method to real problems. Finally, Section 6 summarizes our results. Appendix A gives theoretical proofs of lemmas presented throughout the paper and Appendix B summarizes the notation used.

2 Preliminary Concepts

2.1 Domain of the Algorithms

In this paper we consider problems with perfectly nested `FOR`-loops with uniform and constant dependencies (as in [27]). That is, our algorithms are of the form shown in Algorithm 1.

Algorithm 1 *Algorithmic model*

```
FOR j1 = l1 TO u1 DO
   FOR j2 = l2 TO u2 DO
      ...
      FOR jn = ln TO un DO
         A[fw(j)] := F(A[fw(j - d1)], ..., A[fw(j - dq)]);
```

where: (1) $j = (j_1, \dots, j_n)$, (2) $d_i = (d_{i1}, \dots, d_{in})$, (3) l_1 and u_1 are rational-valued parameters, (4) l_k and u_k ($k = 2, \dots, n$) are of the form: $l_k = max(\lceil f_{k1}(j_1, \dots, j_{k-1})\rceil, \dots, \lceil f_{kr}(j_1, \dots, j_{k-1})\rceil)$ and $u_k = min(\lfloor g_{k1}(j_1, \dots, j_{k-1})\rfloor, \dots, \lfloor g_{kr}(j_1, \dots, j_{k-1})\rfloor)$, where f_{ki} and g_{ki} are affine functions, (5) f_w is a 1-1 function. We are dealing with general and parameterized convex spaces, with the only assumption that the iteration space is defined as the intersection of a finite number of semi-spaces of the n-dimensional space Z^n. The requirement for perfectly nested loops is a trivial one so that loops can be tiled ([6,23,31]). The dependencies are considered uniform and constant, i.e. independent of the indexes of computations, and are expressed by dependence vectors $d_1, d_2, \dots, d_q$.

The rank of the dependence matrix is n, i.e we are considering DOACROSS loop nests. If the rank of the dependence vector matrix is $n_d < n$, then the iteration space contains DOALL parallelism and can thus be divided into independent sets of iterations ([25,10]), which are assigned to different processors and thus coarse-grain parallelism can be achieved without tiling. Although tiling was proposed with dependence notations looser than constant dependence vectors (e.g. dependence cone), we have to strengthen this requirement to constant dependence vectors in order to determine the communication sets. In addition, we require that there are no anti or output dependencies so that each iteration writes to a distinct memory location. Techniques to remove anti and output dependencies are presented in [7]. To simplify our model, we consider single assignment statements with one array variable. Note, however, that this is only a notational restriction, since all of the techniques presented in this paper can be easily adapted to multiple statements on multiple arrays.

Problems that follow the above model very commonly arise from the discretization of Partial Differential Equations (PDEs) using explicit finite-differencing schemes ([22,1,20]) or in image processing functions like smoothing, sharpening, noise reduction, edge detection etc. ([28]). In the majority of the above problems rectangular tiling transformations are invalid, since some dependence vectors contain negative elements (see [24]). This happens because the algorithms under consideration are derived by iterative calculations in each iteration point using information from points that surround it in space (e.g. calculation of the mean value of neighboring points in image processing). Thus, the process presented in this paper can be utilized to automatically generate parallel code and achieve coarse-grain parallelism for this large class of application, that cannot be transformed by a human developer using a rectangular tiling transformation.

2.2 Notation

Throughout this paper the following notation is used: Z is the set of integers, n is the number of nested FOR-loops of the algorithm and q is the number of dependence vectors. If A is a matrix, we denote a_{ij} the matrix element in the i-th row and j-th column. We denote a vector as a or $\vec{a}$ according to the context. The k-th element of the vector is denoted a_k. The dependence matrix of an algorithm is the set of all dependence vectors: $D = [d_1, d_2, \ldots, d_q]$. $J^n \subset Z^n$ is the set of indexes, or the Iteration Space of an algorithm: $J^n = \{j = (j_1, ..., j_n) | j_i \in Z \wedge l_i \le j_i \le u_i, 1 \le i \le n\}$. Each point in this n-dimensional integer space is a distinct instance of the loop body. Accordingly, the Data Space, denoted DS, is defined as: $DS = \{f_w(j) | j \in J^n\}$, where f_w is the write array reference.

2.3 Tiling Transformation

In a tiling transformation, the index space J^n is partitioned into identical n-dimensional parallelepiped areas (tiles or supernodes) formed by n independent families of parallel hyperplanes. Tiling transformation is defined by the n-dimensional square matrix H. Each row vector of H is perpendicular to one family of hyperplanes forming the tiles. Dually, it can be defined by matrix P, which contains the side-vectors of a tile as column vectors. Observe that $P = H^{-1}$. The tile size is given by $|det(P)| = 1/|det(H)|$. Formally, tiling transformation is defined as follows:

$$r : Z^n \longrightarrow Z^{2n}, r(j) = \begin{bmatrix} \lfloor Hj \rfloor \\ j - H^{-1}\lfloor Hj \rfloor \end{bmatrix}$$

where $\lfloor Hj \rfloor$ identifies the coordinates of the tile that iteration point $j \in J^n$ is mapped to and $j - H^{-1}\lfloor Hj \rfloor$ gives the coordinates of j within that tile relative to the tile origin. Thus the initial n-dimensional iteration space J^n is transformed to a $2n$-dimensional one, the space of tiles and the space of indexes within tiles. In the following we define two useful spaces, the Tile Iteration Space, the Tile Space and the Tile Dependence Matrix; all closely associated to the tiling transformation H.

Definition 1 *The Tile Iteration Space (TIS) is defined as:* $TIS(H) = \{j \in Z^n | \lfloor Hj \rfloor = 0\}$

Definition 2 *The Tile Space* J^S *is defined as:* $J^S(J^n, H) = \{j^S | j^S = \lfloor Hj \rfloor, j \in J^n\}$

Definition 3 *The Tile Dependence Matrix* D^S *is defined as:* $D^S = \{d^S | d^S = \lfloor H(j + d) \rfloor, d \in D, j \in TIS\}$

According to the above, $J^n \xrightarrow{H} J^S$. For simplicity reasons we shall refer to $TIS(H)$ as TIS and to $J^S(J^n, H)$ as J^S. Note that TIS contains all points that belong to the tile starting at the axes origin, J^S contains the images of all points $j \in J^n$ according to the tiling transformation and D^S contains the dependencies between tiles.

2.4 Sequential Tiled Code Generation

In [12] we have presented a complete method to efficiently generate sequential tiled code, that is, code that reorders the initial execution of the algorithm according to a general tiling transformation H. The tiled iteration space is now traversed by a $2n$-dimensional loop, the n outermost loops enumerating the tiles and the n innermost ones sweeping the points within a tile. We presented an efficient method to calculate the lower and upper bounds (l_k^S and u_k^S respectively) for a loop control variable j_k^S belonging to the n outer loops. In order to calculate the corresponding bounds for the n innermost loops, we transformed the original non-rectangular tile to a rectangular one, using a non-unimodular transformation H' directly derived from H. Specifically, $H' = VH$, where V is a $n \times n$ diagonal matrix such that $v_{kk}h_k \in Z^n$, and h_k is the k-th row of H ([12]). The inverse of matrix H' is denoted P'. We shall continue using this transformation in the parallelization process.

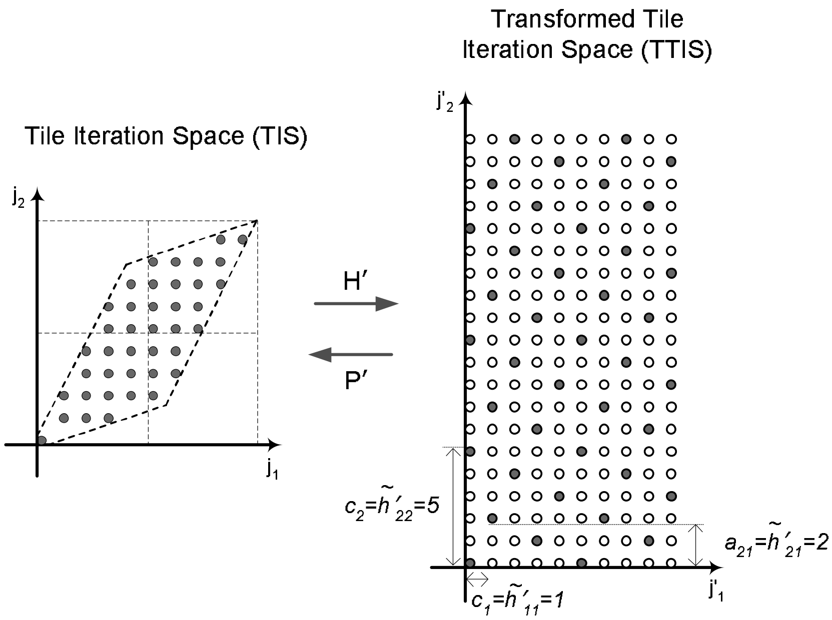

Fig. 1. Traverse the TIS using a non-unimodular transformation and steps, incremental offsets derived from matrix $\widetilde{H'}$

In order to make this paper more self contained, we need to introduce some basic concepts and notations found in greater detail in [12]. Figure 1 shows the transformation of the TIS into a rectangular space called the Transformed Tile Iteration Space $TTIS$ using matrices H' and P'. If $j^S \in J^S$ and $j' \in TTIS$, the corresponding $j \in J^n$ is: $j = Pj^S + P'j'$. Code generation for the loop that will traverse the $TTIS$ is straightforward. The lower and upper bounds of control variable j'_k (l'_k and u'_k respectively) can be easily determined, since it is true that $l'_k = 0$ and $u'_k = v_{kk} - 1$ (for boundary tiles these bounds can be corrected using inequalities describing the original iteration space). Note that each loop control variable may have a non-unitary stride and non-zero incremental offsets. We shall denote the incremental stride of control variable j'_k as c_k. In addition, control variable j'_k may have $k-1$ incremental offsets, one for the increment of each of the $k-1$ outermost control variables, denoted a_{kl} ($l = 1, \ldots, k-1$). In [12] it is proven that strides and initial offsets in our case can be directly derived from the Hermite Normal Form (HNF) of matrix

H', denoted $\widetilde{H'}$. Specifically, $c_k = \widetilde{h'}_{kk}$ and $a_{kl} = \widetilde{h'}_{kl}$ (Figure 1).

3 Computation and Data Distribution

The parallelization of the sequential tiled code involves issues such as computation distribution, data distribution and communication between processors. Tang and Xue in [27] addressed the same issues for rectangularly tiled iteration spaces. We shall generate efficient data parallel code for non-rectangular tiles without imposing any further complexity. The underlying architecture is considered a $(n-1)$-dimensional processor mesh. Thus, each processor is identified by a $(n-1)$-dimensional vector denoted $\vec{pid}$. Note, however, that this is not a physical restriction, but a convention for processor labelling. The memory is physically distributed among processors. Processors perform computations on local data and communicate with each other with messages in order to exchange data that reside to remote memories. In other words, we consider a message-passing environment (like MPI) over a distributed memory architecture. The general intuition in our approach is that since the iteration space is transformed by H and H' into a space of rectangular tiles, each processor can work on its local share of "rectangular" tiles and, following a proper memory allocation scheme, perform operations on rectangular data spaces as well. After all computations in a processor have been completed, locally computed data can be written back to the appropriate locations of the global data space. In this way, each processor essentially works on iteration and data spaces that are both rectangular, and properly translates from its local data space to the global one.

3.1 Computation Distribution

Computation distribution determines which computations of the sequential tiled code will be assigned to which processor. The n innermost loops of the sequential tiled code that access the internal points of a tile will not be parallelized, and thus parallelization only involves the distribution of tiles (traversed by the outermost n-dimensional loop) to processors. Hodzic and Shang in [14] mapped all tiles along a specific dimension to the same processor and used hyperplane $\Pi = [1, \ldots, 1]$ as time schedule vector. In addition to this, previous work [5] in the field of UET-UCT task graphs has shown that if we map all tiles along the dimension with the maximum length (i.e. maximum number of tiles) to the same processor, then the overall scheduling is optimal, as long as the computation to communication ratio is one. Figure 2 displays the overall completion times for different scheduling schemes and mapping dimensions. The upper two cases ((a) and (b)) correspond to the non-overlapping schedule, the one used in [14]. In this schedule we have discrete computation and communication phases (no overlapping occurs). As we can observe, the overall completion time is independent of the mapping dimension. However, if we map along the longest dimension (along j_1 in Figure 2) we can reduce the number of processors required (5 processors instead of 7). In the overlapping scheme, described in [13] (cases (c) and (d)), the processors are able to compute and communicate at the same time. In this scheme, the mapping dimension is of great importance, since mapping along j_1 leads to a smaller execution time than mapping along j_2. Although we do not implement the overlapping schedule in this paper, we have decided to adopt the above mapping approach, first because it reduces the number of processors and second because these advanced overlapping scheduling schemes can be efficiently incorporated in the future.

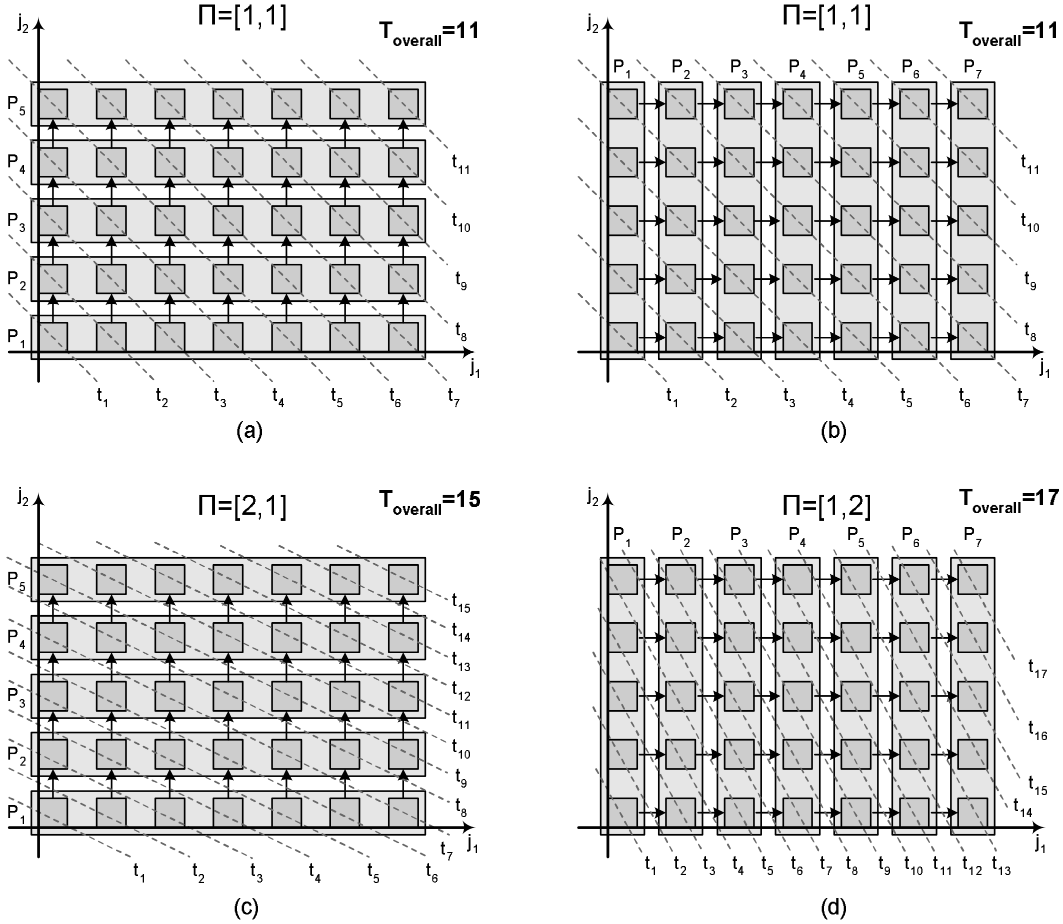

Fig. 2. Overall completion times for different scheduling schemes and mapping dimensions: (a) Non-overlapping Schedule-Mapping along j_1 (b) Non-overlapping Schedule-Mapping along j_2 (c) Overlapping Schedule-Mapping along j_1 (d) Overlapping Schedule-Mapping along j_2

Another criterion for the selection of the mapping dimension could be the minimization of the total communication load. That is, assigning the dimension which imposes most of the communication load to the same processor. When selecting the tile shape with this technique, (i.e see the one presented by Xue in [31]), the communication overheads across tile sides are equal. Since the total communication load of a tiling dimension is equal to the number of tiles along this dimension multiplied by the communication overhead per tile across the respective tile side, in this case, the criteria of selecting as mapping dimension the longest one or the one that minimizes the total communication load, are coincident. In any case, the following theoretical results can be applied with no modification, no matter which the mapping dimension is.

Let us denote the m-th dimension as the one with the maximum total length (or as the mapping dimension selected by any criterion). According to the above, all tiles indexed by $j^S(j_1^S, \ldots, j_m^S, \ \ldots, j_n^S)$, where $j_i^S = const$, $i = 1, \ldots, m-1, m+1, \ldots, n$ and $l_m^S \leq j_m^S \leq u_m^S$ are executed by the same processor. The $n-1$ coordinates of a tile (excluding j_m^S) will identify the processor that a tile is going to be mapped to ($\vec{pid}$). All tiles along j_m^S (denoted also as t^S) are sequentially executed by the same processor, one after the other, in an order specified by a linear time schedule. This means that, after the selection of index j_m^S with the maximum trip count, we reorder all indexes so that j_m^S becomes the innermost index. This corresponds to loop index interchange or permutation. Since all dependence vectors d^S in J^S are considered lexicographically positive, the interchanging or reordering of indexes is valid ([21]).

3.2 Data Distribution

In a distributed memory architecture, the data space of the original algorithm is distributed among the local memories of the various nodes forming the global data space. Data distribution decisions affect the communication volume, since data that reside in one node may be needed for the computations in another. In our approach we follow the "computer-owns" rule, which dictates that a processor owns the data it writes and thus communication occurs when one processor needs to read data computed by another. Substantially, the memory space allocated by a processor represents the space where computed data are to be stored. This means that each processor iterates over a number of transformed rectangular tiles ($TTIS$s) and can locally store its computed data to a rectangular data space. At the end of all its computations, the processor can place its locally computed data to the appropriate positions of the global Data Space (DS). The data space computed by a tile could be an exact image of the $TTIS$, but in this case the holes of the $TTIS$ would correspond to unused extra space. In addition to the space storing the computed data, each processor needs to allocate extra space for communication, that is memory space to store the data it receives from its neighbors. This means that we first need to condense the actual points of the $TTIS$ and second provide further space for receiving data. Since, after all transformations, we finally work with rectangular sets, this Local Data Space (denoted LDS) allocated by a processor, is given by the following definition.

Definition 4 *The Local Data Space (LDS) is defined as:* $LDS = \{j'' \in Z^n | 0 \leq j''_k < off_k + v_{kk}/\widetilde{h}'_{kk}, k = 1, \ldots, n, k \neq m \wedge 0 \leq j''_m < off_m + |t| v_{mm}/\widetilde{h}'_{mm}\}$*, where* $|t|$ *denotes the number of tiles assigned to the particular processor.*

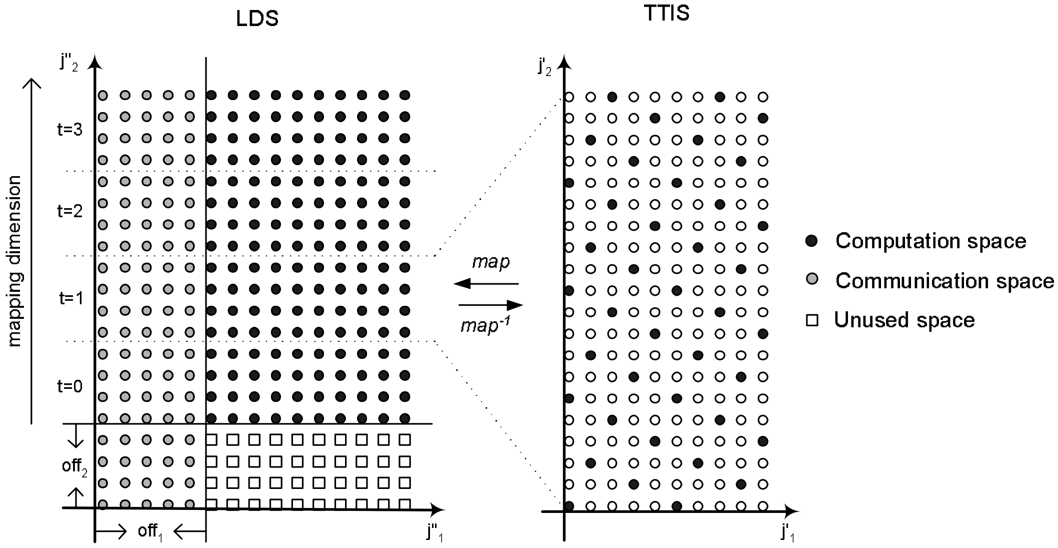

Fig. 3. Local Data Space LDS and Transformed Tile Iteration Space $TTIS$

As shown in Figure 3, the LDS of a processor consists of the memory space required for packing computed data (black dots) and for unpacking received data (grey dots) of a tile, multiplied by the number of tiles assigned to the particular processor. White squares depict unused data. The offset off_k, which expands the space to store received data, derives from the communication criteria of the algorithm, as shown in the next section. Recall that each processor iterates over the $TTIS$ for as many times as the number of tiles assigned to that processor. Lemma 1 determines the translation function from $TTIS$ to LDS, while Lemma 2 determines the inverse translation function from LDS to $TTIS$.

Lemma 1 *If* $j' \in TTIS$*, then its corresponding point in* LDS *is given by the following expressions:*

$$j''_k = j'_k/\widetilde{h}'_{kk} + off_k, k \neq m$$
$$j''_m = (tv_{mm} + j'_m)/\widetilde{h}'_{mm} + off_m,$$

where t *is the current tile.*

PROOF. Given in Appendix A.

Lemma 2 *If $j'' \in LDS$, then its corresponding point in $TTIS$ is given by the following expression:*

$$j' = \widetilde{H'}\vec{x},$$

where $\vec{x}$ is given by:

$$x_k = j''_k - off_k - (\textstyle\sum_{l=1}^{k-1} x_l \widetilde{h'}_{kl})/\widetilde{h'}_{kk},\ k \neq m$$
$$x_m = j''_m - off_m - tv_{mm}/\widetilde{h'}_{mm} - (\textstyle\sum_{l=1}^{m-1} x_l \widetilde{h'}_{ml})/\widetilde{h'}_{mm}$$

where t is the current tile.

PROOF. Given in Appendix A.

Function 1 implements $map(j', t)$ directly from Lemma 1 and thus determines the memory location in LDS where computation for iteration $j' \in TTIS$ is to be stored (Figure 3). Obviously, $map^{-1}(j'')$ (Function 2) implements Lemma 2 and is its inverse. Note that all divisions in the above expressions correspond to integer divisions. Function 3 implements $loc(j)$ which uses $map(j', t)$ in order to locate the processor $\vec{pid}$ and the memory location $j'' \in LDS$, where the computed data of iteration point $j \in J^n$ is to be stored. Inversely, Function 4 shows the series of steps in order to locate the corresponding $j \in J^n$ for a point $j'' \in LDS$ of processor $\vec{pid}$. Thus, $loc^{-1}()$ is called by processors at the end of their computations in order to transit from their LDS to the original iteration space J^n. In the sequel, the corresponding point in the Data Space DS is found via f_w (Figure 4).

Function 1 $map(j', t)$

$$j'' = map(j', t) = \begin{cases} j''_k = j'_k/\widetilde{h'}_{kk} + off_k & k \neq m \\ j''_m = (tv_{mm} + j'_m)/\widetilde{h'}_{mm} + off_m & \end{cases}$$

Function 2 $map^{-1}(j'')$

$j' = map^{-1}(j'')$
$t = (j''_m - off_m)\widetilde{h'}_{mm}/v_{mm}$
$x_k = j''_k - off_k - (\sum_{l=1}^{k-1} x_l \widetilde{h'}_{kl})/\widetilde{h'}_{kk}, k \neq m$
$x_m = j''_m - off_m - tv_{mm}/\widetilde{h'}_{mm} - (\sum_{l=1}^{m-1} x_l \widetilde{h'}_{ml})/\widetilde{h'}_{mm}$
$j' = \widetilde{H'}\vec{x}$

Function 3 $loc(j)$

$j'', \vec{pid} = loc(j)$
$j^S = \lfloor Hj \rfloor$
$j' = H'(j - Pj^S)$
$j'' = map(j', j^S_m - l^S_m)$
$\vec{pid} = (j^S_1, \ldots, j^S_{m-1}, j^S_{m+1}, \ldots, j^S_n)$

Function 4 $loc^{-1}(j'', \vec{pid})$
$j = loc^{-1}(j'', \vec{pid})$
$j' = map^{-1}(j'')$
$j^S = (pid_1, \ldots, pid_{m-1}, t + l^S_m, pid_{m+1}, \ldots, pid_n)$
$j = Pj^S + P'j'$

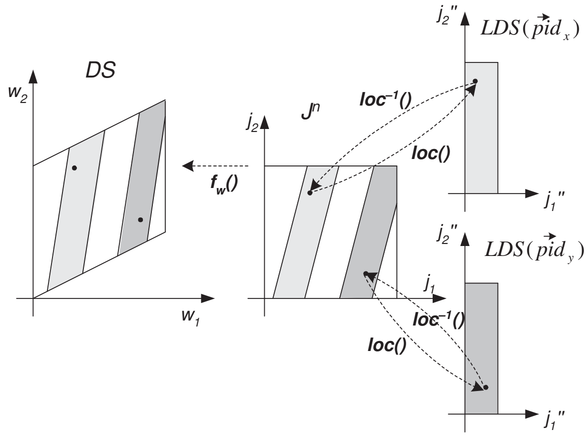

Fig. 4. Relations between DS, J^n and LDS

Under our scheme, each processor allocates exactly the amount of local memory needed for computation and communication (minor over-allocation occurs in the few boundary tiles). Note that direct allocation of a processor's share in the original DS would lead to a waste of memory space, since this generally non-rectangular share would lead to the allocation of the minimum enclosing rectangular memory space. Note also that each processor's share in the original DS (the footprint of a tile because of f_w) is in general non-rectangular, even if a rectangular tiling transformation is applied. Our method, however, forces the local data space of each processor to be rectangular, allowing thus more efficient memory management. In addition, if we also take into account that data spaces for common computationally intensive algorithms are very large, and will probably not fit in each node's memory, the compression of the local space to the LDS is in most cases necessary. Eventually, this leads to a trade-off between computational complexity and allocated memory space, since extra expressions are needed to address the LDS, but this minor overhead does not significantly affect performance. Finally, note that storing data accessed by a non-rectangular tile to a dense rectangular data space also exploits cache locality.

4 Message-Passing Primitives

Using the iteration and data distribution schemes described before, data that reside in the local memory of one processor may be needed by another due to algorithmic dependencies. In this case, processors need to communicate via message-passing. The two fundamental issues that need to be addressed regarding communication are the specification of the processors each processor needs to communicate with, and the determination of the data that need to be transferred within each message.

As far as the communication data are concerned, we focus on the communication points, as defined below:

Definition 5 *Let m be the mapping dimension. Let $d^S \in D^S$ be a tile dependence that implies processor dependence, that is $\exists l \neq m : d^S_l \neq 0$. A point $j' \in TTIS$ is considered a communication point respective to d^S iff the computed data at iteration $j = Pj^S + P'j'$ is needed by tile $j^S + d^S$, where $j^S \in J^S$ and $j^S + d^S \in J^S$.*

Note that since d^S implies processor dependence, tiles j^S and $j^S + d^S$ are essentially computed by different processors. Note also that a communication point is only defined in respect to a specific tile dependence d^S. In other words, communication points in the $TTIS$ correspond to iterations at which data are computed by one processor and need to be sent to another processor in tile direction d^S.

We further exploit the regularity of the $TTIS$ to deduce simple criteria for the communication points at compile time. The following lemma is useful:

Lemma 3 *A point $j' = (j'_1, \ldots, j'_n) \in TTIS$ corresponds to a communication point respective to a tile dependence $d^S = (d^S_1, \ldots, d^S_n) \in D^S$ iff:*

$$j'_k \geq d^S_k(v_{kk} - \max_{d' \in D'}\{d'_k\})$$

where $k = 1, \ldots, n, d' \in D', D' = H'D$.

PROOF. Given in Appendix A.

Definition 6 *The communication vector $\vec{CC}$ is defined as $\vec{CC} = (cc_1, \ldots, cc_n)$ where*

$$cc_k = v_{kk} - \max_{d' \in D'}\{d'_k\}$$

and $k = 1, \ldots, n$, $d' \in D'$, $D' = H'D$.

It is obvious that $\vec{CC}$ can be easily calculated at compile time. According to Lemma 3, $\vec{CC}$ can be used to directly determine the communication points:

Corollary 1 *Let $\vec{CC} = (cc_1, \ldots, cc_n)$ be the communication vector. A point $j' = (j'_1, \ldots, j'_n) \in TTIS$ corresponds to a communication point respective to* *a tile dependence* $d^S = (d_1^S, \ldots, d_n^S) \in D^S$ *iff:*

$$j'_k \geq d_k^S cc_k$$

where $k = 1, \ldots, n$.

We shall directly apply Corollary 1 in our MPI program to pack all data that need to be sent to neighboring processors and unpack the data received by each processor. It should be clear that because of the simplicity of Corollary 1 it is advantageous to identify the communication data in the $TTIS$, as opposed to the other possible alternatives (e.g. the initial iteration space, the TIS etc.) which would complicate the communication procedure unduly. Also, note that the offsets in LDS referenced in §3.2 can easily arise as follows:

$$off_k = \lceil \max_{d' \in D'} \{d'_k\}/c_k \rceil, k \neq m$$

and

$$off_m = v_{mm}/c_m$$

Communication takes place before and after the execution of a tile. Before the execution of a tile, a processor must receive all the essential non-local data computed elsewhere, and unpack these data to the appropriate locations in its LDS. Dually, after the completion of a tile, the processor must send part of the computed data to the neighboring processors for later use. We adopt the communication scheme presented by Tang and Xue in [27], which suggests a simple implementation for packing and sending the data, and a more complicated one for the receiving and unpacking procedure. The asymmetry between the two phases (send-receive) arises from the fact that a tile may need to receive data from more than one tiles of the same predecessor processor, but it will send its data only once to each successor processor, satisfying all the tile dependencies that lead to different tiles assigned to the same successor in a single message. In other words, a tile will receive from tiles, while it will send to processors. Let D^m be the projection of D^S in the $n-1$ dimensions, when the mapping dimension m is collapsed. D^m expresses processor dependencies, meaning that, in general, processor $\vec{pid}$ needs to receive from processors $\vec{pid} - d^m$ and send to processors $\vec{pid} + d^m$ for all $d^m \in D^m$. Algorithms 2 and 3 implement the schemes for receive-unpack and pack-send respectively that have been adopted according to the MPI platform. $d^m(d^S)$ denotes the processor dependence d^m that corresponds to a tile dependence d^S, while $d^S(d^m)$ denotes all tile dependencies d^S that generate processor dependence d^m. Function `minsucc`$(\vec{s}, d^m)$ denotes the lexicographically minimum successor tile of tile $\vec{s}$ in processor direction d^m, while function `valid`$(\vec{s})$ returns true if tile $\vec{s}$ is enumerated. LA denotes an array in local memory which implements the

LDS.

Algorithm 2 *RECEIVE*$(\vec{pid}, t^S, D^S, \vec{CC})$

```
FOR d^S ∈ D^S DO /*For all tile dependencies...*/
    /*...if predecessor tile valid and current tile
      lexicographically minimum successor...*/
    IF(valid((pid,t^S) − d^S) ∧ (pid,t^S)=minsucc((pid,t^S) − d^S,d^m(d^S)))
        /*...receive data from predecessor processor...*/
        MPI_Recv(buffer,Rank(pid − d^m(d^S)),...);
        /*...and unpack it to LDS of current processor.*/
        count:=0;
        FOR j'_1 = max(l'_1, d^S_1 cc_1) TO u'_1 STEP=c_1 DO
          ...
          FOR j'_m = l'_m TO u'_m STEP=c_m DO
            ...
            FOR j'_n = max(l'_n, d^S_n cc_n) TO u'_n STEP=c_n DO
              LA[map(j',t^S − l^S_n)-(d^S_1 v_11/c_1,...,d^S_n v_nn/c_n)]:=buffer[count++];
```

Algorithm 3 *SEND*$(\vec{pid}, t^S, D^m, \vec{CC})$

```
FOR d^m ∈ D^m DO /*For all processor dependencies...*/
    /*...if a valid successor tile exists...*/
    IF(∃d^S(d^m) ∈ D^S :valid((pid,t^S) + d^S(d^m)))
        /*...pack communication data to buffer...*/
        count:=0;
        FOR j'_1 = max(l'_1, d^m_1 cc_1) TO u'_1 STEP=c_1 DO
          ...
          FOR j'_m = l'_m TO u'_m STEP=c_m DO
            ...
            FOR j'_n = max(l'_n, d^m_{n−1} cc_n) TO u'_n STEP=c_n DO
              buffer[count++]:=LA[map(j',t^S − l^S_n)];
        /*...and send to successor processor.*/
        MPI_Send(buffer,Rank(pid + d^m),...);
```

Summarizing, the generated data parallel code for the loop of Algorithm 1 will have a form similar to that shown in Algorithm 4.

Algorithm 4 *Final SPMD code*

```
FORACROSS pid_1 = l^S_1 TO u^S_1 DO
  ...
  FORACROSS pid_{n−1} = l^S_{n−1} TO u^S_{n−1} DO
    /*Sequential execution of tiles*/
    FOR t^S = l^S_n TO u^S_n DO
      /*Receive data from neighboring tiles*/
      RECEIVE(pid,t^S,D^S,CC);
      /*Traverse the internal of the tile*/
```

```
FOR j'_1 = l'_1 TO u'_1 STEP=c_1 DO
   ...
   FOR j'_n = l'_n TO u'_n STEP=c_n DO
      /*Perform computations on Local Data Space LDS*/
      t := t^S - l^S_n;
      LA[map(j',t)] = F(LA[map(j' - d'_1, t)], ..., LA[map(j' - d'_q, t)]);
/*Send data to neighboring processors*/
SEND(pid, t^S, D^m, CC);
```

5 Experimental Results

We have implemented our parallelizing techniques in a tool, which automatically generates C++ code with calls to the MPI library, and run our examples on a cluster with 16 identical 500MHz Pentium III nodes with 256MBs of RAM. The nodes run Linux with kernel 2.4.20 and are interconnected with FastEthernet. We used the gcc v.2.95.4 compiler for the compilation of the sequential programs and mpiCC (which also uses gcc v.2.95.4) for the compilation of the generated data-parallel programs. In both cases the -O2 optimization option was applied. Our goal is to investigate the effect of the tile shape on the overall completion time of an algorithm. We used three real problems, Gauss Successive Over-relaxation (SOR), the Jacobi algorithm and ADI integration. In each case, we applied rectangular and non-rectangular tiling transformations measured the parallel execution times achieved for various tile sizes and iteration spaces.

Results for SOR

The SOR loop nest is shown in Algorithm 5.

Algorithm 5 *SOR algorithm*

```
FOR t=1 TO M DO
   FOR i=1 TO N DO
      FOR j=1 TO N DO
         A[t,i,j]:=w/4(A[t,i-1,j]+A[t,i,j-1]+A[t-1,i+1,j]+A[t-1,i,j+1])+
                    (1-w)A[t-1,i,j];
```

Since the dependencies contain negative coefficients, the loop needs to be skewed in order to be rectangularly tiled. As in [31], we use $T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$ as skewing matrix. The resulting loop nest is shown in Algorithm 6.

Algorithm 6 *Skewed SOR algorithm*

```
FOR t'=1 TO M DO
   FOR i'=t'+1 TO t'+N DO
      FOR j'=2t'+1 TO 2t'+N DO
         t:=t'; i:=-t'+i';j:=-2t'+j';
         A[t,i,j]:=w/4(A[t,i-1,j]+A[t,i,j-1]+A[t-1,i+1,j]+A[t-1,i,j+1])+
                     (1-w)A[t-1,i,j];
```

The dependence matrix of the skewed SOR is $D = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 2 & 0 & 2 & 1 & 1 \end{bmatrix}$ and the corresponding tiling cone is defined by the rows of matrix $\mathcal{C} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -1 & 0 & 1 \\ -2 & 1 & 1 \end{bmatrix}$.

Although non-rectangular tiling can be directly applied to the original loop nest, we choose to apply both rectangular and non-rectangular tiling to the skewed one, so that the comparison is more obvious. For a non-rectangular transformation, we select three vectors parallel to the first three lines of matrix $\mathcal{C}$, i.e. the tiling transformation is of the form: $H_{nr} = \begin{bmatrix} \frac{1}{x} & 0 & 0 \\ 0 & \frac{1}{y} & 0 \\ -\frac{1}{z} & 0 & \frac{1}{z} \end{bmatrix}$, while the rectangular tiling transformation is defined by a matrix of the form: $H_r = \begin{bmatrix} \frac{1}{x} & 0 & 0 \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$, where $x, y, z \in Z^+$. Note that if we select common factors x, y, z for H_{nr} and H_r we have equal tile sizes ($1/|det(H_r)| = 1/|det(H_{nr})| = xyz$). Furthermore, if we map tiles along the third dimension to the same processor, the communication volume and the number of processors required are the same in both cases, since the first two rows of the tiling transformation matrices are identical. Thus, any differences in execution times will be due to the different scheduling schemes imposed by the different tile shapes.

In order to have a theoretical interpretation of the experimental results that follow, let us focus on the following general example. The linear scheduling vector used in our approach is $\Pi = [1, 1, 1]$. We denote the last executed point of the original iteration space as j_{max}. Obviously, this point will belong to tile

$\lfloor Hj_{max} \rfloor$ and will be executed at time step $t = \Pi \lfloor Hj_{max} \rfloor$. In our skewed SOR example $j_{max} = (M, M+N, M+2N)$ and thus, using rectangular tiling, this point will be executed at time step $t_r = \frac{M}{x} + \frac{M+N}{y} + \frac{2M+N}{z}$. Accordingly, using non-rectangular tiling, j_{max} will be executed at $t_{nr} = \frac{M}{x} + \frac{M+N}{y} + \frac{2M+N}{z} - \frac{M}{z} = t_r - \frac{M}{z} < t_r$. Thus, we expect non-rectangular tiling to achieve lower execution times.

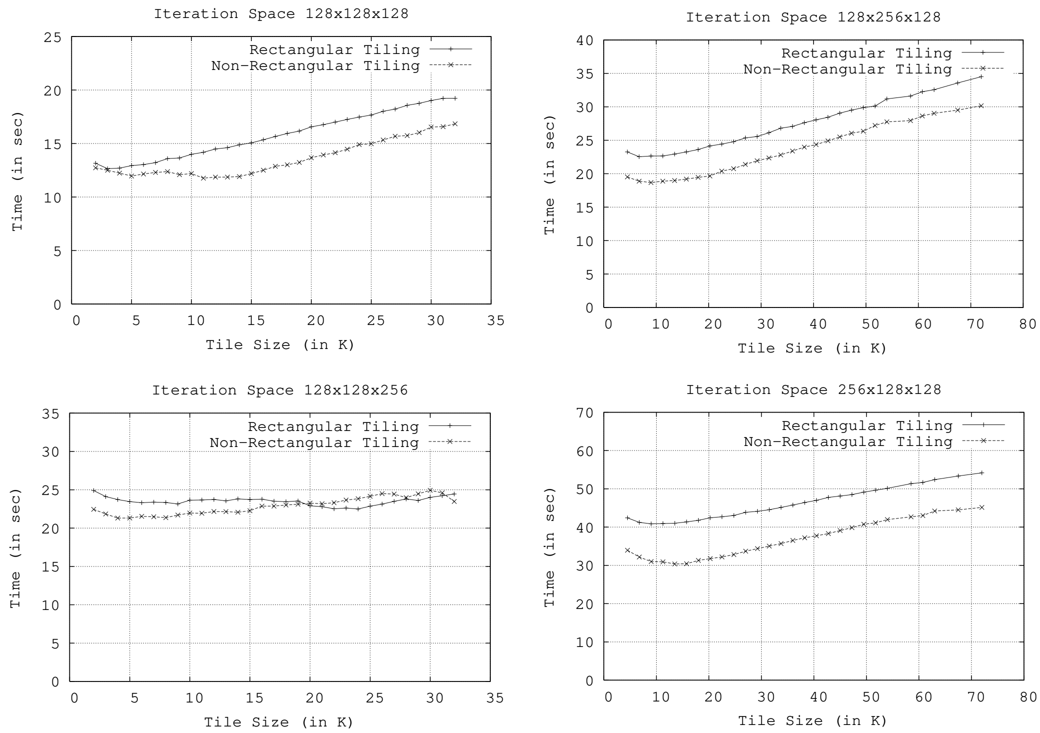

Fig. 5. SOR: Execution times for rectangular and non-rectangular transformations in four iteration spaces

Iteration Space	t_r **min**	t_{nr} **min**	**% diff.**	t_r **avg.**	t_{nr} **avg.**	**% diff.**
$\mathbf{128 \times 128 \times 128}$	12.65	11.76	7.5	15.77	13.56	16.3
$\mathbf{128 \times 128 \times 256}$	22.5	21.31	5.6	23.51	22.89	2.7
$\mathbf{128 \times 256 \times 128}$	22.54	18.68	20.6	24.63	21.21	16.1
$\mathbf{256 \times 128 \times 128}$	40.85	30.37	34.5	41.63	33.15	25.5

Table 1
SOR: Minimum and average execution times (sec) for four iteration spaces

Figure 5 shows execution times for the SOR algorithm for four iteration spaces and various tile sizes. As theoretically expected, non-rectangular tiling transformation achieves lower execution times in the majority of the cases. Table 1 summarizes the improvement attained by the application of a non-rectangular tile shape. Quite impressively, the change of only one element in tiling transformation matrix H leads to an average 25% reduction in execution times for iteration space $256 \times 128 \times 128$.

Results for Jacobi

The Jacobi loop nest is shown in Algorithm 7.

Algorithm 7 *Jacobi algorithm*

```
FOR t=1 TO T DO
   FOR i=1 TO I DO
      FOR j=1 TO J DO
         A[t,i,j]:=0.25(A[t-1,i-1,j]+A[t-1,i,j-1]+A[t-1,i+1,j]+
                     A[t-1,i,j+1]);
```

Note that this loop also needs to be skewed in order to be legally tiled. We use $T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{bmatrix}$ as skewing matrix and thus the skewed loop nest is shown in Algorithm 8.

Algorithm 8 *Skewed Jacobi algorithm*

```
FOR t'=1 TO T DO
   FOR i'=t'+1 TO t'+I DO
      FOR j'=t'+1 TO t'+J DO
         t:=t'; i:=-t'+i'; j:=-t'+j';
         A[t,i,j]:=0.25(A[t-1,i-1,j]+A[t-1,i,j-1]+A[t-1,i+1,j]+
                     A[t-1,i,j+1]);
```

The dependence matrix of the skewed Jacobi is $D = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & 0 & 1 \\ 1 & 2 & 1 & 0 \end{bmatrix}$ and the corresponding tiling cone is $\mathcal{C} = \begin{bmatrix} -3 & 1 & 1 \\ 1 & -1 & 1 \\ -1 & -1 & 1 \\ -1 & 1 & 1 \end{bmatrix}$. In this case, in order to have the same comparison features as in SOR, we applied non-rectangular tiling transformation defined by $H_{nr} = \begin{bmatrix} \frac{1}{x} & -\frac{1}{2x} & 0 \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$. If we choose common x, y, z

factors and map tiles along the first dimension to the same processor, we have the same tile size, communication volume and number of processors required both for rectangular and non-rectangular tiling. Choosing the tile's cutting hyperplanes from the surface of the tiling cone would probably lead to lower total execution times as proven in [15] and [17], but in this case comparison with rectangular tiling would be difficult, since factors like tile size, communication volume and number of processors would differ. In this case, we have $j_{max} = (T, T+I, T+J)$ and, following similar analysis as in the case of SOR, we have $t_r = \frac{T}{x} + \frac{T+I}{y} + \frac{T+J}{z}$, while $t_{nr} = t_r - \frac{T+I}{2x} < t_r$. Again, we expect non-rectangular tiling to achieve better execution times.

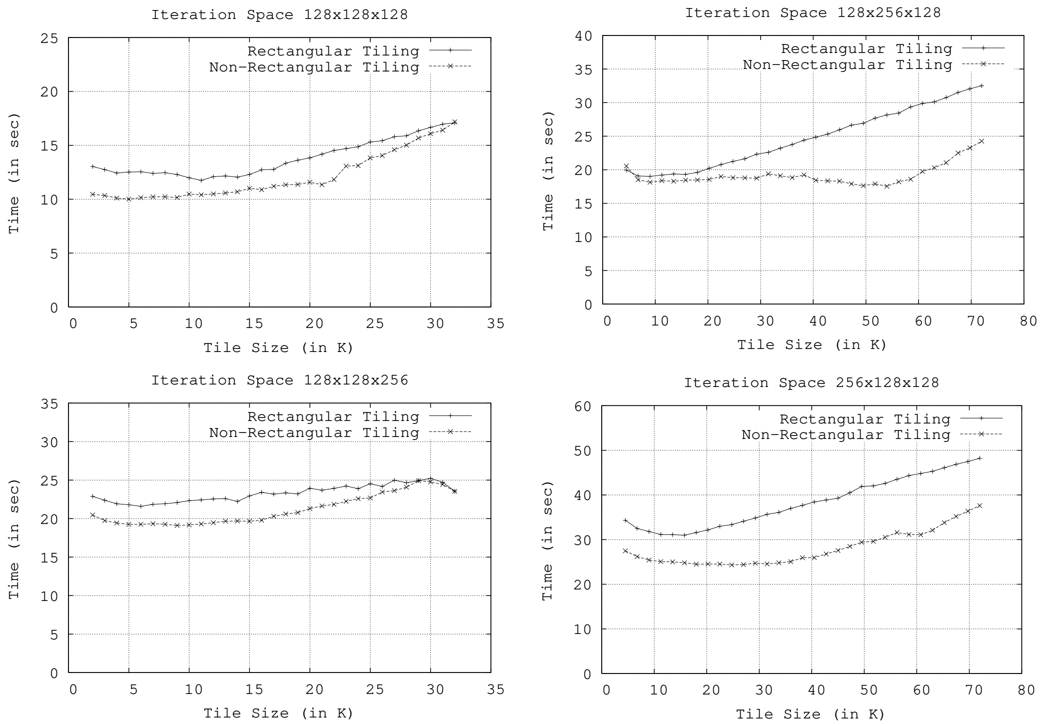

Fig. 6. Jacobi: Execution times for rectangular and non-rectangular transformations in four iteration spaces

In this example, we held y and z factors constant throughout the experiments in each iteration space and varied factor x in order to test different tile sizes. Figure 6 shows the parallel execution times for four different iteration spaces and Table 2 summarizes minimum and average execution times. In this case, non-rectangular tile shapes achieve a minimum execution-time reduction that varies between 8.4% and 27.4% and an average execution-time reduction that varies between 10% and 36.7% throughout the four iteration spaces.

Results for ADI Integration

ADI Integration can be written in the triply nested loop shown in Algorithm 9.

Iteration Space	t_r min	t_{nr} min	% diff.	t_r avg.	t_{nr} avg.	% diff.
$\mathbf{128 \times 128 \times 128}$	11.75	10.01	17.3	13.77	12.06	14.2
$\mathbf{128 \times 128 \times 256}$	21.59	19.11	13	23.26	21.14	10
$\mathbf{128 \times 256 \times 128}$	19.02	17.55	8.4	24.7	19.2	28.6
$\mathbf{256 \times 128 \times 128}$	30.98	24.32	27.4	38.3	28.02	36.7

Table 2
Jacobi: Minimum and average execution times (sec) for four iteration spaces

Algorithm 9 *ADI algorithm*

```
FOR t=1 TO T DO
   FOR i=1 TO N DO
      FOR j=1 TO N DO
         X[t,i,j]:=X[t-1,i,j]+X[t-1,i,j-1]*A[i,j]/B[t-1,i,j-1]-
                     X[t-1,i-1,j]*A[i,j]/B[t-1,i-1,j];
         B[t,i,j]:=B[t-1,i,j]-A[i,j]*A[i,j]/B[t-1,i,j-1]-
                     A[i,j]*A[i,j]/B[t-1,i-1,j];
```

The dependence matrix of ADI integration is $D = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$ and the corresponding tiling cone is $\mathcal{C} = \begin{bmatrix} 1 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$. No skewing is needed in this case, since all dependence vectors are non-negative. In this experiment series we used three different non-rectangular matrices defined by: $H_{nr1} = \begin{bmatrix} \frac{1}{x} & -\frac{1}{x} & 0 \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$, $H_{nr2} = \begin{bmatrix} \frac{1}{x} & 0 & -\frac{1}{x} \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$ and $H_{nr3} = \begin{bmatrix} \frac{1}{x} & -\frac{1}{x} & -\frac{1}{x} \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$. Note that the third one is parallel to the directions of the tiling cone. Again here we map tiles along the first dimension to the same processor. All four transformations applied (the rectangular and the three non-rectangular ones) have the same tile size, communication volume and require the same number of processors. Similar to the analysis in the previous experiments, since $j_{max} = (T, N, N)$, it is true that $t_r = \frac{T}{x} + \frac{N}{y} + \frac{N}{z}$, $t_{nr1} = t_r - \frac{N}{y}$, $t_{nr2} = t_r - \frac{N}{z}$ and $t_{nr3} = t_r - \frac{N}{y} - \frac{N}{z}$. Thus, $t_{nr3} < t_{nr1}, t_{nr2} < t_r$.

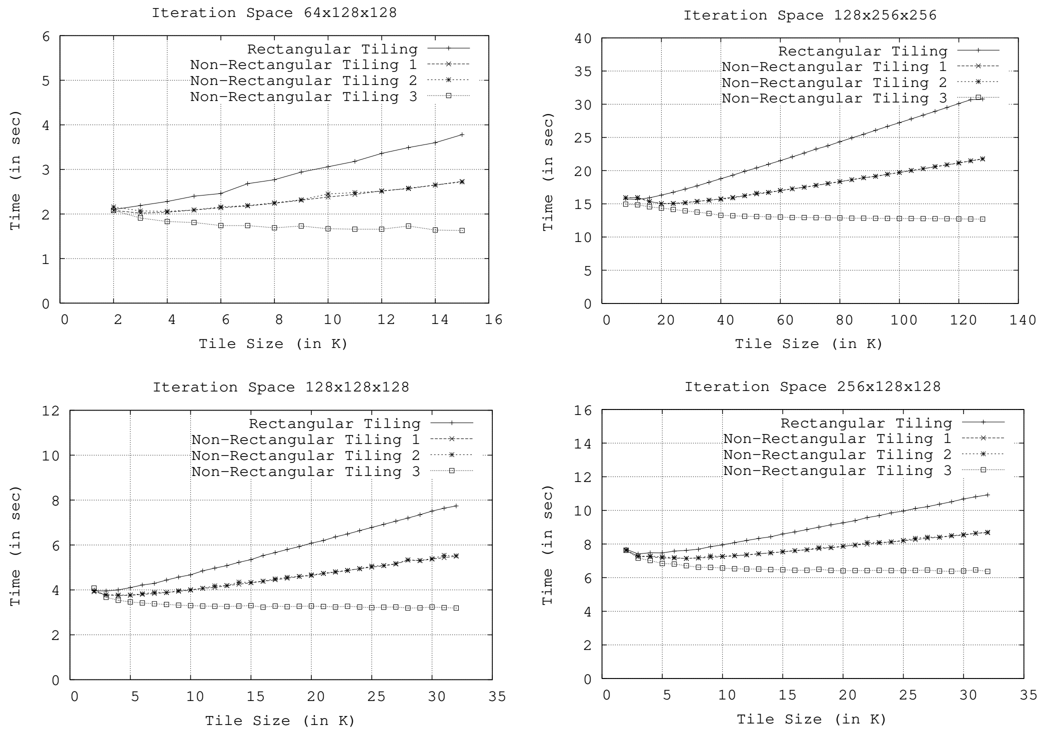

Fig. 7. ADI: Execution times for one rectangular and three non-rectangular 1-3 transformations in four iteration spaces

Iteration Space	t_r **min**	t_{nr1} **min**	t_{nr2} **min**	t_{nr3} **min**	**% diff.**	t_r **avg.**	t_{nr1} **avg.**	t_{nr2} **avg.**	t_{nr3} **avg.**	**% diff.**
$\mathbf{64 \times 128 \times 128}$	2.1	2.02	2.06	1.63	28.8	2.88	2.31	2.33	1.75	64.3
$\mathbf{128 \times 128 \times 128}$	3.95	3.75	3.77	3.19	23.8	5.69	4.51	4.53	3.32	71.5
$\mathbf{128 \times 256 \times 256}$	15.67	15.03	15.03	12.7	23.4	22.8	17.82	17.82	13.25	72.1
$\mathbf{256 \times 128 \times 128}$	7.41	7.15	7.14	6.37	16.3	8.95	7.76	7.78	6.57	36.2

Table 3
ADI: Minimum and average execution times (sec) for four iteration spaces

Figure 7 shows the parallel execution times for four different iteration spaces and Table 3 summarizes minimum and average execution times. In this set of experiments, the verification of the theoretical analysis is even more evident. Indeed, rectangular and non-rectangular transformations attained execution times sorted exactly as expected by theory. Moreover, we observe the the best tiling transformation (H_{nr3}) achieved an impressive reduction in parallel execution times that reached up to 28.8% in minimum execution times and 71.1% in average ones.

6 Conclusions

In this paper we presented a complete approach to generate message-passing code for iteration spaces transformed by general parallelepiped tiling transformations. We thoroughly addressed issues such as data distribution, iteration distribution and automatic message-passing, and generated efficient data-parallel code for a cluster of PCs. Our method is based on transforming the non-rectangular tile into a rectangular one using a non-unimodular transformation. We have implemented our parallelizing techniques using MPI and run several experiments in our cluster. After studying the effect of the tile shape on the overall execution time of an algorithm, we were able to confirm previous theoretical work, which claims that selecting a tiling transformation from the sides of the tiling cone leads to optimal scheduling schemes. The method presented in this paper can be utilized to efficiently execute `DOACROSS` loop nests when rectangular tiling transformations are not valid, or when scheduling criteria propose the application of non-rectangular tile shapes in order to minimize processor idle times.

References

[1] B. D' Acunto. *Computational Methods for Pde in Mechanics*. World Scientific Pub., 2004.

[2] V. Adve and J. Mellor-Crummey. Advanced Code Generation for High Performance Fortran. In *Lecture Notes in Computer Science Series, Compiler optimizations for scalable parallel systems: languages, compilation techniques, and run time systems*, pages 553–596. Springer-Verlag, 2001.

[3] S. P. Amarasinghe and M. S. Lam. Communication Optimization and Code Generation for Distributed Memory Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, USA, Jun 1993.

[4] R. Andonov, P. Calland, S. Niar, S. Rajopadhye, and N. Yanev. First Steps Towards Optimal Oblique Tile Sizing. In *Proceedings of the 8th International Workshop on Compilers for Parallel Computers*, pages 351–366, Aussois, Jan 2000.

[5] T. Andronikos, N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal Scheduling for UET/UET-UCT Generalized N-Dimensional Grid Task Graphs. *Journal of Parallel and Distributed Computing*, 57(2):140–165, May 1999.

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

[7] P. Y. Calland, A. Darte, Y. Robert, and F. Vivien. On the Removal of Anti and Output Dependences. *International Journal of Parallel Programming*, 26(2):285–312, 1998.

[8] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31–50, 1992.

[9] F. Desprez, J. Dongarra, F. Rastello, and Y. Robert. Determining the Idle Time of a Tiling: New Results. *Journal of Information Science and Engineering*, 14(1):167–190, Mar 1997.

[10] E. H. D'Hollander. Partitioning and Labeling of Loops by Unimodular Transformations. *IEEE Transactions on Parallel and Distributed Systems*, 3(4):465–476, Jul 1992.

[11] G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran-D Language Specification. Technical Report TR-91-170, Dept. of Computer Science, Rice University, Dec 1991.

[12] G. Goumas, M. Athanasaki, and N. Koziris. An Efficient Code Generation Technique for Tiled Iteration Spaces. *IEEE Transactions on Parallel and Distributed Systems*, 14(10):1021–1034, Oct 2003.

[13] G. Goumas, A. Sotiropoulos, and N. Koziris. Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS'01)*, San Francisco, Apr 2001.

[14] E. Hodzic and W. Shang. On Supernode Transformation with Minimized Total Running Time. *IEEE Transactions on Parallel and Distributed Systems*, 9(5):417–428, May 1998.

[15] E. Hodzic and W. Shang. On Time Optimal Supernode Shape. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1220–1233, Dec 2002.

[16] K. Högstedt, L. Carter, and J. Ferrante. Determining the Idle Time of a Tiling. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages (POPL)*, pages 160–173, Jan 1997.

[17] K. Högstedt, L. Carter, and J. Ferrante. Selecting Tile Shape for Minimal Execution time. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, pages 201–211, 1999.

[18] K. Högstedt, L. Carter, and J. Ferrante. On the Parallel Execution Time of Tiled Loops. *IEEE Trans. on Parallel and Distributed Systems*, 14(3):307–321, Mar 2003.

[19] M. Lam, E. Rothberg, and M. Wolf. The Cache Performance and Optimizations of Blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 63–74, Santa Clara, California, USA, Apr 1991.

[20] K. Morton and D. Mayers. *Numerical Solution of Partial Differential Equations*. Cambridge University Press, Cambridge, UK, 2005.

[21] D. Padua and W. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, Dec 1986.

[22] W. Press, S. Teukolsky, W. Vetterling, and B. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 1992.

[23] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–120, 1992.

[24] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Multicomputers. *Journal of Parallel and Distributed Computing*, 16:108–120, 1992.

[25] W. Shang and J.A.B. Fortes. Independent Partitioning of Algorithms with Uniform Dependencies. *IEEE Transactions on Computers*, 41(2):190–206, Feb 1992.

[26] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. Hodges, and P. Banerjee. Advanced Compilation Techniques in the PARADIGM Compiler for Distributed Memory Multicomputers. In *Proceedings of the 9th ACM International Conference on Supercomputing (ICS)*, pages 424–433, Madrid, Spain, Jul 1995.

[27] P. Tang and J. Xue. Generating Efficient Tiled Code for Distributed Memory Machines. *Parallel Computing*, 26(11):1369–1410, 2000.

[28] B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[29] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 30–44, Toronto, Ontario, Canada, Jun 1991.

[30] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):452–471, Oct 1991.

[31] J. Xue. Communication-Minimal Tiling of Uniform Dependence Loops. *Journal of Parallel and Distributed Computing*, 42(1):42–59, Apr 1997.

[32] J. Xue and W. Cai. Time-minimal Tiling when Rise is Larger than Zero. *Parallel Computing*, 28(6):915–939, 2002.

APPENDIX A – Theoretical Proofs

Proof of Lemma 1: In order to prove the validity of this transformation, we need to prove that the resulting point $j'' \in LDS$. However, for each $k \neq m$ it is true that $0 \leq j'_k < v_{kk} \Rightarrow 0 \leq \lfloor \frac{j'_k}{\widetilde{h}'_{kk}} \rfloor < \frac{v_{kk}}{\widetilde{h}'_{kk}} \Rightarrow off_k \leq \lfloor \frac{j'_k}{\widetilde{h}'_{kk}} \rfloor + off_k < \frac{v_{kk}}{\widetilde{h}'_{kk}} + off_k \Rightarrow off_k \leq j''_k < \frac{v_{kk}}{\widetilde{h}'_{kk}} + off_k$. In addition, $0 \leq j'_m < v_{mm} \Rightarrow 0 \leq \lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor < \frac{v_{mm}}{\widetilde{h}'_{mm}} \Rightarrow \frac{tv_{mm}}{\widetilde{h}'_{mm}} + off_m \leq \frac{tv_{mm}}{\widetilde{h}'_{mm}} + \lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor + off_m < \frac{(t+1)v_{mm}}{\widetilde{h}'_{mm}} + off_m$ Taking into account that $0 \leq t \leq |t| - 1$, the previous inequality gives $off_m \leq j''_m < \frac{|t|v_{mm}}{\widetilde{h}'_{mm}} + off_m$. Therefore, it is true that $j'' = map(j', t) \in LDS$.

Proof of Lemma 2: We need to prove that map and map^{-1} are indeed inverse functions. Equivalently, we should prove that **(i)** $(j', t) = map^{-1}(map(j', t))$ and **(ii)** $j'' = map(map^{-1}(j''))$.

$$\textbf{(i)}\ (j', t) \stackrel{?}{=} map^{-1}(map(j', t)) \Leftrightarrow \begin{cases} t \stackrel{?}{=} \lfloor \dfrac{((\lfloor \frac{tv_{mm}+j'_m}{\widetilde{h}'_{mm}} \rfloor + off_m) - off_m)\widetilde{h}'_{mm}}{v_{mm}} \rfloor \\ \wedge \\ j'_k \stackrel{?}{=} \sum\limits_{i=1}^{k} \widetilde{h}'_{ki} y_i \end{cases},$$

$$\text{where: } \left\{ \begin{array}{c} y_i = ((\lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor + off_i) - off_i) - \lfloor \dfrac{\sum\limits_{l=1}^{i-1} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor, i \neq m \\ \\ y_m = ((\lfloor \frac{tv_{mm}+j'_m}{\widetilde{h}'_{mm}} \rfloor + off_m) - off_m - \frac{tv_{mm}}{\widetilde{h}'_{mm}}) - \lfloor \dfrac{\sum\limits_{l=1}^{m-1} \widetilde{h}'_{ml} y_l}{\widetilde{h}'_{mm}} \rfloor \end{array} \right\} \Leftrightarrow$$

$$\Leftrightarrow y_i = \lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor - \lfloor \frac{\sum\limits_{l=1}^{i-1} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor$$

However, $t \stackrel{?}{=} \lfloor \frac{((\lfloor \frac{tv_{mm}+j'_m}{\widetilde{h}'_{mm}} \rfloor + off_m) - off_m)\widetilde{h}'_{mm}}{v_{mm}} \rfloor \Leftrightarrow t \stackrel{?}{=} t + \lfloor \frac{\lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor \widetilde{h}'_{mm}}{v_{mm}} \rfloor$. From $0 \leq j'_m < v_{mm} \Rightarrow 0 \leq \lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor < \frac{v_{mm}}{\widetilde{h}'_{mm}} \Rightarrow 0 \leq \lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor \widetilde{h}'_{mm} < v_{mm} \Rightarrow 0 \leq \lfloor \frac{\lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor \widetilde{h}'_{mm}}{v_{mm}} \rfloor < 1 \Rightarrow \lfloor \frac{\lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor \widetilde{h}'_{mm}}{v_{mm}} \rfloor = 0$. Thus, $t \stackrel{?}{=} t + \lfloor \frac{\lfloor \frac{j'_m}{\widetilde{h}'_{mm}} \rfloor \widetilde{h}'_{mm}}{v_{mm}} \rfloor \Leftrightarrow t \stackrel{?}{=} t + 0$, which is always valid.

In addition, from $y_i = \lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor - \lfloor \frac{\sum_{l=1}^{i-1} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor \Rightarrow \lfloor \frac{j'_i}{\widetilde{h}'_{ii}} \rfloor = y_i + \lfloor \frac{\sum_{l=1}^{i-1} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor = \lfloor \frac{\sum_{l=1}^{i} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor \Rightarrow \widetilde{h}'_{ii} \lfloor \frac{\sum_{l=1}^{i} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor \leq j'_i \leq \widetilde{h}'_{ii} \lfloor \frac{\sum_{l=1}^{i} \widetilde{h}'_{il} y_l}{\widetilde{h}'_{ii}} \rfloor + \widetilde{h}'_{ii} - 1$. In this interval, there is one and only one *actual* point j'_i (as $\widetilde{h}'_{ii}$ is the step of j'_i in order to meet another *actual* point), which is $\sum\limits_{l=1}^{i} \widetilde{h}'_{il} y_l$. Therefore it is true that $j'_i = \sum\limits_{l=1}^{i} \widetilde{h}'_{il} y_l$.

(ii)

$$j'' \stackrel{?}{=} map(map^{-1}(j'')) \Leftrightarrow \left\{ \begin{array}{c} j''_k \stackrel{?}{=} \lfloor \frac{\sum\limits_{l=1}^{k} \widetilde{h}'_{kl} z_l}{\widetilde{h}'_{kk}} \rfloor + off_k, k \neq m \\ \wedge \\ j''_m \stackrel{?}{=} \lfloor \frac{tv_{mm} + \sum\limits_{l=1}^{m} \widetilde{h}'_{ml} z_l}{\widetilde{h}'_{mm}} \rfloor + off_m \end{array} \right\}, \tag{1}$$

where: $\left\{ \begin{array}{c} z_l = j''_l - off_l - \lfloor \frac{\sum\limits_{i=1}^{l-1} \widetilde{h}'_{li} z_i}{\widetilde{h}'_{ll}} \rfloor, l \neq m \\ z_m = j''_m - off_m - \frac{tv_{mm}}{\widetilde{h}'_{mm}} - \lfloor \frac{\sum\limits_{i=1}^{m-1} \widetilde{h}'_{mi} z_i}{\widetilde{h}'_{mm}} \rfloor \end{array} \right\} \Rightarrow$

$$\left\{ \begin{array}{l} j''_l = off_l + \lfloor \frac{\sum\limits_{i=1}^{l} \widetilde{h}'_{li} z_i}{\widetilde{h}'_{ll}} \rfloor, l \neq m \\ j''_m = off_m + \frac{tv_{mm}}{\widetilde{h}'_{mm}} + \lfloor \frac{\sum\limits_{i=1}^{m} \widetilde{h}'_{mi} z_i}{\widetilde{h}'_{mm}} \rfloor \end{array} \right. \tag{2}$$

Therefore, (1) $\stackrel{(2)}{\Longleftrightarrow} \left\{ \begin{array}{c} off_k + \lfloor \frac{\sum\limits_{i=1}^{k} \widetilde{h}'_{ki} z_i}{\widetilde{h}'_{kk}} \rfloor \stackrel{?}{=} \lfloor \frac{\sum\limits_{l=1}^{k} \widetilde{h}'_{kl} z_l}{\widetilde{h}'_{kk}} \rfloor + off_k, k \neq m \\ \wedge \\ off_m + \frac{tv_{mm}}{\widetilde{h}'_{mm}} + \lfloor \frac{\sum\limits_{i=1}^{m} \widetilde{h}'_{mi} z_i}{\widetilde{h}'_{mm}} \rfloor \stackrel{?}{=} \lfloor \frac{tv_{mm} + \sum\limits_{l=1}^{m} \widetilde{h}'_{ml} z_l}{\widetilde{h}'_{mm}} \rfloor + off_m \end{array} \right.$, which is obviously always valid, taking into account that v_{mm} is a multiple of $\widetilde{h}'_{mm}$.

After proving claims **(i)** and **(ii)**, it turns out that Lemma 2 is always valid.

Proof of Lemma 3: For j' to be a communication point according to the k-th dimension, we distinguish two cases:

(1) $d^S_k = 0$. Since no tile dependence is enforced in this case, no limitation for j'_k is defined. So it is true that $0 \leq j'_k \leq v_{kk} - 1$.
(2) $d^S_k = 1$. In this case, there must exist a data dependence in the $TTIS$ $d' \in D'$ such, that the k-th component of $j' + d'$ exceeds the respective bound of the $TTIS$, thus incurring need for communication according to the k-th dimension.

According to the above, it must be true that

$$j'_k + d'_k > v_{kk} - 1 \Rightarrow j'_k + d'_k \geq v_{kk} \Rightarrow j'_k \geq v_{kk} - d'_k$$

for some $d' \in D'$ or equivalently

$$j'_k \geq v_{kk} - \max_{d' \in D'}\{d'_k\}$$

The unification of both cases leads to the given condition.

APPENDIX B – Summary of Notations

Z: set of integers
n: loop depth
$\mathbf{J^n}$: original loop iteration space
D: dependence matrix
$\mathbf{d_k}$: a random dependence vector (a column of matrix D)
q: number of dependencies
$\mathbf{f_w}$: write array reference
$\mathbf{l_k}$: lower bound of dimension k in J^n
$\mathbf{u_k}$: upper bound of dimension k in J^n
DS: Data Space
H: tiling transformation matrix
P: inverse tiling transformation matrix ($H = P^{-1}$)
TIS: Tile Iteration Space
$\mathbf{J^S}$: Tile Space
$\mathbf{D^S}$: Tile Dependence Matrix
TTIS: Transformed Tile Iteration Space
$\mathbf{l^S_k}$: lower bound of dimension k in J^S
$\mathbf{u^S_k}$: upper bound of dimension k in J^S
$\mathbf{H'}$: transformation matrix between TIS and $TTIS$
$\mathbf{P'}$: transformation matrix between $TTIS$ and TIS ($P' = H'^{-1}$)
V: a diagonal matrix, $H' = VH$
$\mathbf{l'_k}$: lower bound of dimension k in $TTIS$
$\mathbf{u'_k}$: upper bound of dimension k in $TTIS$
$\widetilde{\mathbf{H}}'$: Hermite Normal Form of H'
$\mathbf{c_k}$: steps in the $TTIS$ ($= \widetilde{h}'_{kk}$)
$\mathbf{a_{kl}}$: incremental offsets in the $TTIS$ ($= \widetilde{h}'_{kl}$)
m: mapping dimension
$\vec{\mathbf{pid}}$: $n-1$-dimensional vector identifying a processor
$\mathbf{j^S_m}$: a coordinate in the mapping dimension of the Tile Space
$\mathbf{t^S}$: a coordinate in the mapping dimension of the Tile Space (equivalent to $j^S_m - l^S_m$)

LDS: Local Data Space
map: a translation function from LDS to $TTIS$
$\mathbf{map^{-1}}$: a translation function from $TTIS$ to LDS
loc: a translation function from LDS to J^n
$\mathbf{loc^{-1}}$: a translation function from J^n to LDS
$\mathbf{D'}$: dependence matrix in $TTIS$ ($D' = H'D$)
$\mathbf{D^m}$: derives from D^S when the m-th line excluded
$\mathbf{d^m}$: a column vector of D^m
$\mathbf{d^S(d^m)}$: all tile dependencies d^S that generate processor dependence d^m
$\mathbf{d^m(d^S)}$: the processor dependence d^m that corresponds to a tile dependence d^S
$\vec{\mathbf{CC}}$: communication vector