# Compiling Tiled Iteration Spaces for Clusters

Georgios Goumas, Nikolaos Drosinos, Maria Athanasaki and Nectarios Koziris
*National Technical University of Athens*
*Dept. of Electrical and Computer Engineering*
*Computing Systems Laboratory*
e-mail: {goumas, ndros, maria, nkoziris}@cslab.ece.ntua.gr

## Abstract

*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 general 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 communication become simple and straightforward. We have implemented our parallelizing techniques in a tool which automatically generates MPI code and run several experiments on a cluster of PCs. Our experimental results show the merit of general parallelepiped tiling transformations, and confirm previous theoretical work on scheduling-optimal tile shapes.*

**Index Terms:** Loop tiling, clusters, data parallel, code generation, MPI.

## 1 Introduction

Using clusters to deliver high performance to scientific and commercial applications is considered as the state of the art in high performance computing. Clusters are becoming increasingly popular due to their affordable scalability, low cost and high aggregate network bandwidth. On the other hand, it is generally accepted that they are difficult to program, since they lack a virtually shared view of global memory. Clusters, in general, comply to the distributed-memory model, since each node has its own local memory, and require therefore communication via message passing, using libraries such as MPI or PVM. This means that, in order to program clusters, one must be acquainted with such message-passing libraries. Although this may be quite likely in the scientific community, it is definitely not the usual case in the commercial one. More importantly, writing efficient hand-made message-passing code is a very intricate task requiring deep knowledge of both the algorithm and the underlying architecture. Evidently, in order to achieve acceptable execution speedups and further extend the popularity of clusters to the commercial community, the programmer needs to be released from the tasks of both writing message-passing code as well as optimizing it, by assigning these tasks to the compiler.

Tiling transformation is one of the most popular loop transformations discussed in the literature, proposed to enhance locality in uni-processors and achieve coarse-grain parallelism in multiprocessors. Tiling groups a number of iterations into a tile, which is executed uninterruptedly, while communication between processors occurs just before and after the computations within a tile. A lot of discussion has been made concerning the selection of an optimal tiling transformation. Ramanujam and Sadayappan in [12], Xue in [15] and Boulet et al. in [4] studied the effect of the tile shape on the communication imposed by a tile, and proved that the communication-minimal tiling can derive from the algorithm's tiling cone. Moving one step further, Hodzic and Shang in [9] discussed the effect of the tile shape and size on the overall completion time of an algorithm taking into account the iteration space bounds. In [10] Hodzic and Shang proved that the scheduling-optimal tile shape, i.e. the one that leads to minimum execution time, is derived from the algorithm's tiling cone similarly to the communication-optimal tiling, as described in [4, 15, 12].

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 by commercial and research compilers ([1, 2, 5, 6, 13]). Furthermore, no complete approach has been presented concerning implementation issues for non-rectangular tiling transformations. Performing proper data distribution and determining communication sets in this case are far from being straightforward. In this paper we present a complete approach to generate data-parallel code, for arbitrarily tiled iteration spaces to be executed onto distributed memory machines. We address issues such as transformed loop bound calculation, iteration and data distribution and automatic message passing code generation. We continue previous work on efficient sequential tiled code generation. More specifically, in [7] we presented an approach to drastically reduce the compilation time for tiled iteration spaces. We transformed the 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. 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 transition 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 examples on a cluster of workstations interconnected via FastEthernet. Our goal is to accentuate the merit of non-rectangular tiling transformations and to confirm 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 remarkable increase in speedups.

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, data distribution and message passing code generation. Section 4 presents experimental results from the application of our method to real problems. Finally, Section 5 summarizes our results and proposes future work.

# 2 Preliminary Concepts

## 2.1 Domain of the Algorithms - Notation

In this paper we consider algorithms with perfectly nested FOR-loops with uniform and constant dependencies. That is, our algorithms are of the form:

```
FOR j_1 = l_1 TO u_1 DO
    FOR j_2 = l_2 TO u_2 DO
        ...
        FOR j_n = l_n TO u_n DO
            A[f_w(j)] := F(A[f_w(j - d_1)], ..., A[f_w(j - d_q)]);
        ENDFOR
        ...
    ENDFOR
ENDFOR
```

where: (1) $j = (j_1, ..., j_n)$, (2) $d_i = (d_{i1}, \ldots, d_{in})$, (3) $l_1$ and $u_1$ are rational-valued parameters, (4) $l_k$ and $u_k$ ($k = 2...n$) are of the form: $l_k = max(\lceil f_{k1}(j_1, \ldots, j_{k-1})\rceil, \ldots, \lceil f_{kr}(j_1, \ldots, j_{k-1})\rceil)$ and $u_k = min(\lfloor g_{k1}(j_1, \ldots, j_{k-1})\rfloor, \ldots, \lfloor g_{kr}(j_1, \ldots, j_{k-1})\rfloor)$, where $f_{ki}$ and $g_{ki}$ are affine functions. We are dealing with general and parameterized convex spaces, with the only assumption that the iteration space is defined as the bisection 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 ([12, 4, 15]). The dependencies are considered uniform and constant, i.e. independent of the indexes of computations and are expressed by dependence vectors $d_1, d_2, \ldots, d_q$. 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.

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: $J^n = \{j(j_1, ..., j_n) | j_i \in Z \wedge l_i \leq j_i \leq u_i, 1 \leq i \leq n\}$. 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.

## 2.2 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. It holds

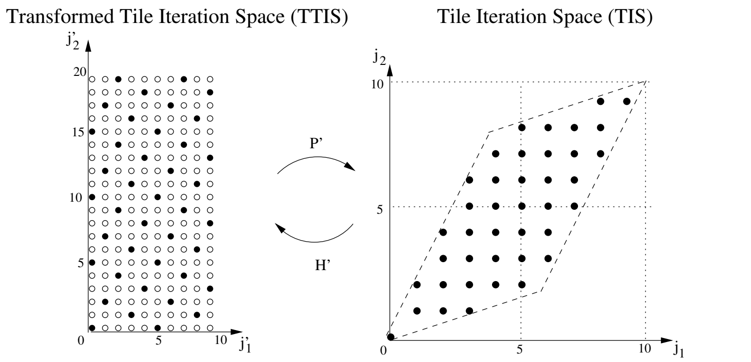


**Figure 1. Traverse the $TIS$ with a non-unimodular transformation**

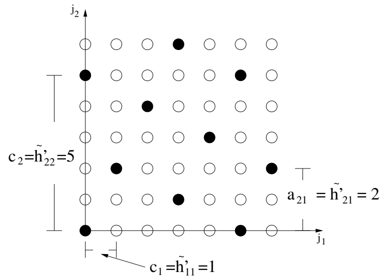


**Figure 2. Steps and incremental offsets in $TTIS$ derived from matrix $\widetilde{H'}$**

$P = H^{-1}$. The tile size is given by $|det(P)| = 1/|det(H)|$. The following spaces are derived from a tiling transformation $H$, when applied to an iteration space $J^n$.

1. The Tile Iteration Space $TIS(H) = \{j \in Z^n | \lfloor Hj \rfloor = 0\}$, which contains all points that belong to the tile starting at the axes origins.

2. The Tile Space $J^S(J^n, H) = \{j^S | j^S = \lfloor Hj \rfloor, j \in J^n\}$, which contains the images of all points $j \in J^n$ according to tiling transformation.

3. The Tile Dependence Matrix $D^S = \{d^S | d^S = \lfloor H(j+d) \rfloor, d \in D, j \in TIS\}$, which contains the dependencies between tiles.

### 2.3 Sequential Tiled Code Generation

In [7] 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 $n$ innermost sweeping the points within tiles. 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, it holds $H' = VH$, 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 $H$ ([7]). The inverse of matrix $H'$ is denoted $P'$. We shall continue using this transformation in the parallelization process presented in this paper and thus we need to introduce some basic concepts and notations found in greater detail in [7]. 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 found by the expression: $j = Pj^S + P'j'$. 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: it holds: $l'_k = 0$ 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 [7] 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, it holds: $c_k = \widetilde{h'}_{kk}$ and $a_{kl} = \widetilde{h'}_{kl}$ (Fig. 2).

## 3 Data Parallel Code Generation

The parallelization of the sequential tiled code involves issues such as computation distribution, data distribution and communication between processors. Tang and Xue in [14] 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 $\vec{pid}$. 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 NUMA architecture. Note, however, that the $(n-1)$-dimensional underlying architecture is not a physical restriction, but a convention for processor naming. It is clear, that, this abstract model can be easily implemented with a cluster of computers, interconnected with a commercial interconnection network. The general intuition in our approach is that since the iteration space is transformed by $H$ and $H'$ into a space of rectangular tiles, then 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 and Data 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 involves the distribution of tiles (traversed by the outermost $n$-dimensional loop) to processors. Hodzic and Shang in [9] 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 [3] 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. We follow this approach in order to map tiles to processors trying to adjust tile size properly. Let us denote the $m$-th dimension as the one with the maximum total length. According to this, 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 (see also [11]).

In a NUMA architecture, the data space of the original algorithm is distributed to the local memories of the various nodes forming the global data space. Data distribution decisions affect the communication volume, since data

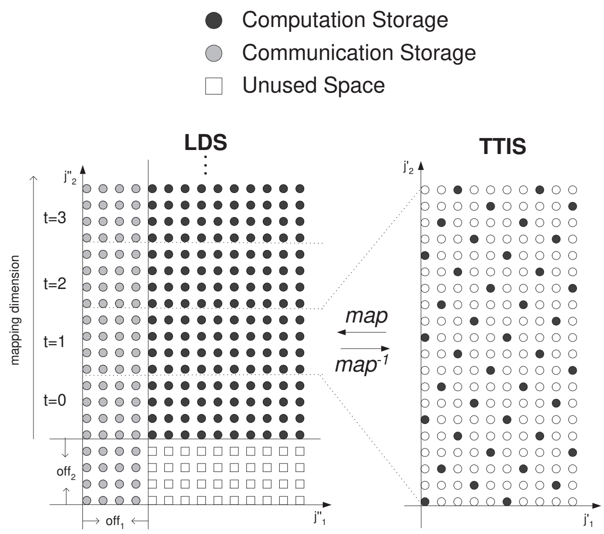


**Figure 3. Local Data Space $LDS$ and Transformed Tile Iteration Space $TTIS$**

that reside in one node may be needed for the computation 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. So, the original location of an array element is where it is computed. Substantially, the memory space allocated by a processor represents the space where computed data are to be stored. This means that each processor iterating over a number of transformed rectangular tiles ($TTIS$s), 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 need to condense the actual points of the $TTIS$ and 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 can be easily defined as follows: $LDS = \{j'' \in Z^n | 0 \leq j''_k < off_k + v_{kk}/c_k, k = 1, \ldots, n, k \neq m \wedge 0 \leq j''_m < off_m + |t| v_{mm}/c_m\}$, where $|t|$ denotes the number of tiles assigned to the particular processor. As shown in Figure 3, $LDS$ consists of the memory space required for storing computed data (black dots) and for buffering receiving data (grey dots) of a tile, multiplied by the number of tiles assigned to the processor. White squares depict unused data. The offset $off_k$, which expands the space to store receiving data, derives from the communication criteria of the algorithm as shown in the next subsection.

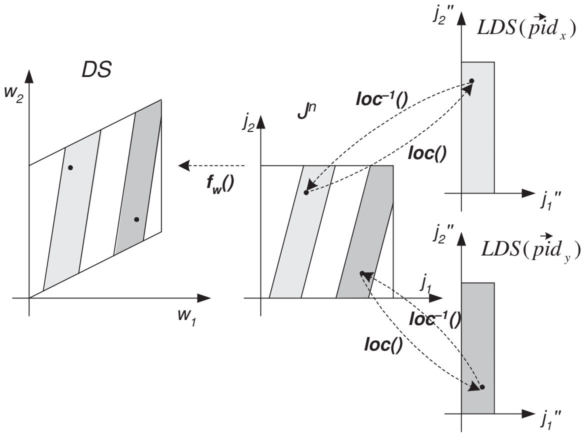


**Figure 4. Relations between $DS$, $J^n$ and $LDS$**

Recall that each processor iterates over the $TTIS$ for as many times as the number of tiles assigned to the processor. If $t$ is the current tile and $j' \in TTIS$ the current instance of the inner $n$-dimensional loop, function $map(j', t)$ determines the memory location in $LDS$ where the computation for this iteration is to be stored (Figure 3). Apparently, $map^{-1}(j'')$ is its inverse. Function $loc(j)$ in Table 1 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, Table 2 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). All expressions in Tables 1 and 2 derive from the properties of the two spaces $LDS$ and $TTIS$ and the Hermite Normal Form of matrix $H'$ denoted $\widetilde{H'}$.

Under our scheme, each processor only 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.

| $j'' = map(j', t)$: |
|---|
| $j''_k = j'_k / c_k + off_k, k \neq m$<br>$j''_m = (tv_{mm} + j'_m)/c_m + off_m$ |

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

**Table 1. Using function $loc()$ to locate $j \in J^n$ in the $LDS$ of a processor**

| $j' = map^{-1}(j'')$: |
|---|
| $t = (j''_m - off_m)c_m / v_{mm}$<br>$j'_k = c_k(j''_k - off_k) + (\sum_{l=1}^{k-1} \widetilde{h'}_{kl} j'_l) \% c_k, k \neq m$<br>$j'_m = c_m(j''_m - off_m) - tv_{mm} + (\sum_{l=1}^{m-1} \widetilde{h'}_{ml} j'_l) \% c_m$ |

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

**Table 2. Using function $loc^{-1}()$ to locate $j'' \in LDS$ of processor $\vec{pid}$ in $J^n$**

### 3.2 Communication

Using the iteration and data distribution schemes described before, data that reside in the local memory module 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 in each message.

As far as the communication data are concerned, we focus on the communication points, e.g. the iterations that compute data read by another processor. We further exploit the regularity of the $TTIS$ to deduce simple criteria for the communication points at compile time. More specifically, a point $j' \in TTIS$ corresponds to a communication point according to the $k$-th dimension if the $k$-th coordinate of $j' + d'_l$ is greater than the respective $TTIS$-bound at the $k$-th dimension for some transformed dependence vector $d'_l \in D'$ ($D' = H'D$). In other words, $j'$ is a communication point respective to the $k$-th dimension when it holds $j'_k + max(d'_{kl}) > v_{kk} - 1$ or equivalently $j'_k \geq v_{kk} - max(d'_{kl})$. We define the communication vector $\vec{CC} = (cc_1, \ldots, cc_n)$ where $cc_k = v_{kk} - max(d'_{kl})$. It is obvious that $\vec{CC}$ can be easily determined at compile time. Note that the offsets in $LDS$ referenced in §3.1 can easily arise as follows: $off_k = \lceil max(d'_{kl})/c_k \rceil, k \neq m$ 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 [14], 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 $d^m \in D^m$. The following schemes for receive-unpack and pack-send 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 $LDS$.

```
RECEIVE(pid⃗, t^S, D^S, CC⃗)
  FOR d^S ∈ D^S DO /*For all tile dependencies...*/
   /*...if predecessor tile valid and current tile
    lexicographically minimum successor...*/
   IF(valid((pid⃗, t^S) - d^S) ∧
                (pid⃗, t^S)=minsucc((pid⃗, t^S) - d^S, d^m(d^S)))
     /*...receive data from predecessor processor...*/
     MPI_Recv(buffer,Rank(pid⃗ - 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++];
         ENDFOR
         ...
       ENDFOR
       ...
     ENDFOR
   ENDIF
```

```
SEND(pid⃗, t^S, D^m, CC⃗)
  FOR d^m ∈ D^m DO /*For all processor dependencies...*/
   /*...if a valid successor tile exists...*/
   IF(∃d^S(d^m) ∈ D^S:valid((pid⃗, 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)];
         ENDFOR
         ...
       ENDFOR
       ...
     ENDFOR
     /*...and send to successor processor.*/
     MPI_Send(buffer,Rank(pid⃗ + d^m),...);
   ENDIF
```

Summarizing, the generated data parallel code for the loop of §2.1 would have a form similar to the following:

```
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(pid⃗, t^S, D^S, CC⃗);
      /*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)]);
```

```
      ENDFOR
      ...
    ENDFOR
    /*Send data to neighboring processors*/
    SEND(pid, t^S, D^S, CC);
  ENDFOR
ENDFORACROSS
  ...
ENDFORACROSS
```

# 4 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 128MB of RAM. The nodes run Linux with kernel 2.2.17 and are interconnected with FastEthernet. We used the gcc v.2.95.2 compiler for the compilation of the sequential programs and mpiCC (which also uses gcc v.2.95.2) for the compilation of the generated data-parallel programs. In both cases the -O2 optimization option was applied. Our goal is to investigate the tile shape effect 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. We present the speedups obtained for various tile sizes and iteration spaces.

## 4.1 Results for SOR

The SOR loop nest is as follows:

```
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];
    ENDFOR
  ENDFOR
ENDFOR
```

Since the dependencies contain negative coefficients, the loop needs to be skewed in order to be rectangularly tiled. As in [15], we use $T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$ as skewing matrix. The resulting loop nest is:

```
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];
    ENDFOR
  ENDFOR
ENDFOR
```

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 for the comparison to be 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 $\Pi = [1, 1 \ldots 1]$. We denote the last executed point of the original iteration space as $j_{max}$. Apparently, this point will belong to tile $\lfloor Hj_{max} \rfloor$ and will execute 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 execute at time step $t_r = \frac{M}{x} + \frac{M+N}{y} + \frac{2M+N}{z}$. Accordingly, using non-rectangular tiling $j_{max}$ will execute 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 exhibit lower total execution times.

We performed our experimental results for four different iteration spaces. In each iteration space we held factors $x$ and $y$ constant, such that the required number of MPI processes would be 16 (one process per processor). We then varied factor $z$ in order to test different tile sizes. Figure 5 shows the maximum speedups obtained in each iteration space, while Figure 6 shows the speedups obtained in one iteration space for various tile sizes.

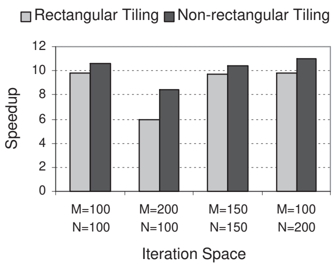


**Figure 5. SOR: maximum speedups for different iteration spaces**

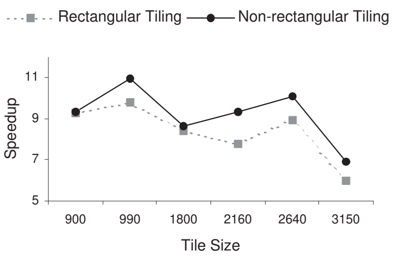


**Figure 6. SOR: speedups for various tile sizes ($M = 100$, $N = 200$)**

### 4.2 Results for Jacobi

The Jacobi loop nest is:

```
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]);
    ENDFOR
  ENDFOR
ENDFOR
```

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:

```
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]);
    ENDFOR
  ENDFOR
ENDFOR
```

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 [10], 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 here we expect non-rectangular tiling to achieve better execution times.

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 7 shows the maximum speedups obtained in each of the four iteration spaces, while Figure 8 shows the speedups obtained in one iteration space for various tile sizes.

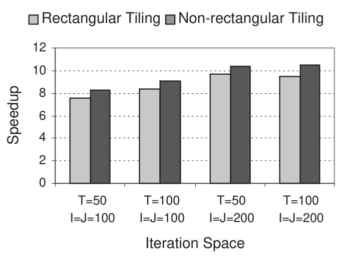


**Figure 7. Jacobi: maximum speedups for different iteration spaces**

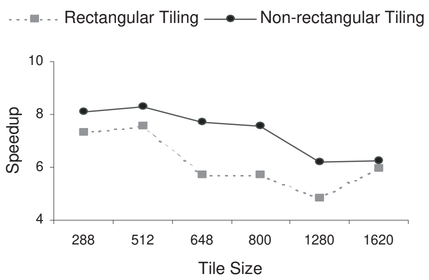


**Figure 8. Jacobi: speedups for various tile sizes ($T = 50$, $I = J = 100$)**

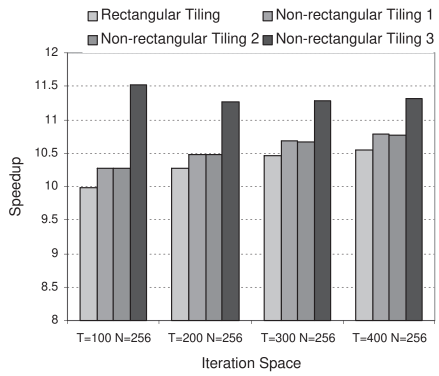


**Figure 9. ADI Integration: maximum speedups for different iteration spaces**

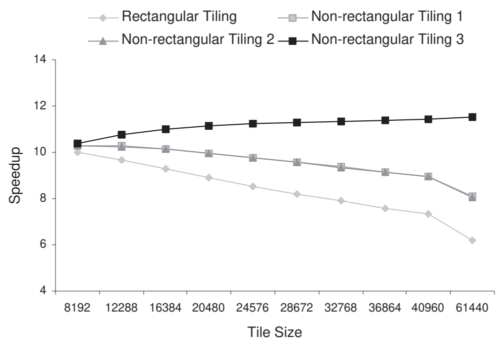


**Figure 10. ADI Integration: speedups for various tile sizes ($T = 100$, $N = 256$)**

### 4.3 Results for Adi Integration

Adi Integration can be written in a triply nested loop as shown in Table 3. 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 holds 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, $t_{nr3} < t_{nr1}, t_{nr2} < t_r$. Figure 9 shows the maximum speedups obtained in each of the four iteration spaces, while Figure 10 shows the speedups obtained in one iteration space for various tile sizes.

### 4.4 Comments on the Results

The first conclusion easily drawn from all sets of experiments is that, as expected, in all cases (i.e. for each algorithm, iteration space and tile size) non-rectangular tiling leads to better execution speedups than rectangular. In SOR we had an average speedup improvement of $17.3\%$, in Jacobi $9.1\%$ and in ADI Integration $10.1\%$. This is a remarkable result since the improvement only arose from a slight change in the rectangular tiling transformation matrix $H$. In fact, in the first two sets of experiments only one element of matrix $H$ was changed, while in the last set only two. Note also that in the last set the gradual improvement from the rectangular tiling to the non-rectangular one taken from the tiling cone is much more obvious. This is to confirm the theoretical work in [10], where it is proven that if any of the row vectors of $H$ lies in the interior of the tiling cone, then the corresponding tiling transformation is not optimal. Non-rectangular tiling defined by $H_{nr1}$ and $H_{nr2}$ exhibit the same speedups as expected (we used equal $y$ and $z$ factors) and lower than the one defined by $H_{nr3}$ (but still higher than the rectangular one). We also need to point out that our tool is not yet optimized for performance. Our main goal was to compare the total execution times imposed by

```
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];
    ENDFOR
  ENDFOR
ENDFOR
```

**Table 3. Code of Adi Integration**

rectangular and non-rectangular tiling transformations for various algorithms. We believe that we can achieve better speedups by using certain technical optimizations.

# 5 Conclusions-Future Work

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 tile shape effect 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. Future work includes the incorporation of the computation and communication overlapping scheduling schemes presented in [8] into our method, and the further analysis of the tile shape effect on the overall completion time of an algorithm when these advanced scheduling schemes are applied.

# References


[1] V. Adve and J. Mellor-Crummey. Advanced Code Generation for High Performance Fortran. In *Languages, Compilation Techniques and Run Time Systems for Scalable Parallel Systems*, chapter 18, Lecture Notes in Computer Science Series. Springer-Verlag, 1997.

[2] 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, NM, Jun 1993.

[3] 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.

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

[5] B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 121–160, Jul 1992.

[6] 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.

[7] G. Goumas, M. Athanasaki, and N. Koziris. Automatic Code Generation for Executing Tiled Nested Loops Onto Parallel Architectures. In *Proceedings of the ACM Symposium on Applied Computing (SAC 2002)*, Madrid, Mar 2002.

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

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

[10] E. Hodzic and W. Shang. On Time Optimal Supernode Shape. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, CA, Jun 1999.

[11] D. Padua and W. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12), 1986.

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

[13] 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 ACM International Conference on Supercomputing (ICS)*, Madrid, Spain, Jul 1995.

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

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