Data Parallel Code Generation for Arbitrarily Tiled Loop Nests

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

Tiling or supernode transformation is extensively discussed as a loop transformation to efficiently execute nested loops onto distributed memory machines. In addition, a lot of work has been done concerning the selection of a communication-minimal and a scheduling-optimal tiling transformation. However, no complete approach has been presented in terms of implementation for non-rectangularly tiled iteration spaces. Code generation in this case can be extremely complex, while parallelization issues such as data distribution and communication are far from being straightforward. In this paper, we propose a complete method to efficiently generate data parallel code for arbitrarily tiled iteration spaces. We assign chains of neighboring tiles to the same processor. Experimental results show that non-rectangular tiling allows better scheduling schemes, thus achieving less overall execution time.

Keywords: Loop tiling, supernodes, non-unimodular transformations, data parallel, code generation.

1 Introduction

Tiling or supernode transformation is a well known loop transformation used to enforce coarse grain parallelism in distributed memory machines. Tiling groups neighboring iterations to form a unique computational unit which is uninterruptedly executed. Communication occurs in larger messages before and after the computation of a tile, reducing both the number of total messages and the amount of data exchanged. A lot of work has been done concerning the selection of a communication-minimal tiling. Ramanujam and Sadayappan in [9] first accented the use of non-rectangular tiles in order to minimize inter-processor communication. Boulet et al. in [3] used a per tile communication function that has to be minimized by linear programming approaches. Based on this function, Xue in [11] presented a complete method to determine the tiling transformation H that imposes minimum communication. On the other hand, different tile shapes can also lead to different scheduling schemes for an algorithm. Hodzic and Shang in [7] presented a method to determine the tile shape that leads to minimum overall execution time. Most significantly, communication-minimal and time-optimal tile shapes are both derived from the tiling cone of the algorithm and thus can be simultaneously achieved. Specifically, in [7] it is shown that, for a given tile size, a time-optimal tile shape can be determined by properly scaling n vectors from the surface of the algorithm's tiling cone.

Despite all this extensive research on tiling iteration spaces for minimum communication and optimal overall execution time, no complete approach has been presented addressing implementation issues such as transformed loop bounds calculation, iteration distribution, data distribution and message passing code generation for arbitrarily tiled loop nests. In other words, there are no actual experimental results to verify the above theory. 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 continue previous work concerning the efficient generation of sequential tiled code [5], based on the transformation of a non-rectangular tile to a rectangular one. We take advantage of the regularity of transformed rectangular tiles in order to effectively distribute data among processors and generate the communication primitives. In this way, the parallelization process of the sequential tiled code is greatly simplified. Our experimental results confirm that the selection of a non-rectangular tiling transformation can lead to a much more efficient scheduling of tiles to processors. The rest of the paper is organized as follows: Section 2 presents some preliminary concepts, along with some notation and intuition from the techniques presented in [5]. Section 3 addresses all parallelization problems such as computation distribution, data distribution and automatic message passing code generation. Section 4 presents our experimental results while Section 5 summarizes our results and proposes future work.

2 Preliminary Concepts

2.1 Program Model-Notation

In this paper we consider algorithms with perfectly nested FOR-loops with uniform and constant dependencies (as in [3]). 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
      $A[f_w(j)] := F(A[f_w(j - d_1)], \ldots, A[f_w(j - d_q)]);$
    ENDFOR
    ...
  ENDFOR
ENDFOR

We are dealing with general 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 (as in [1]). Finally, we assume that there are no anti or output dependencies.

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. 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. 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 n linearly independent vectors, which are the sides of the tiles. Matrix P contains the side-vectors of a tile as column vectors. It holds $P = H^{-1}$. Given a tiling transformation H and an iteration space J^n, we define the following spaces:

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 the 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

In [5] 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 tiling transformation H. The tiled iteration space is now traversed by a $2n$ dimensional loop, where the n outermost loops enumerate the tiles and the n innermost ones sweep the points within tiles. We presented an efficient method to calculate the lower and upper bounds for the n outermost loops, that is l_k^S and u_k^S for the loop control variable j_k^S. 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. 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 [5].

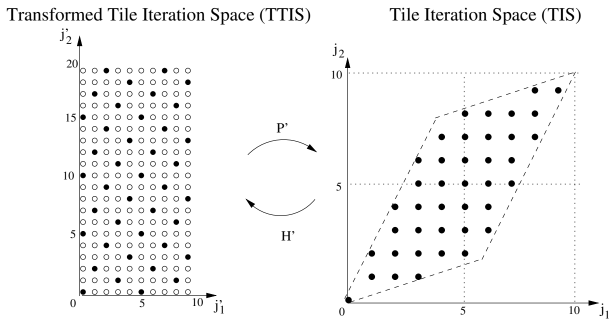

Figure 1. Traverse the TIS with a non-unimodular transformation

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 calculated from the expression $j = Pj^S + P'j'$. Code generation for the loop that will traverse the $TTIS$ is straightforward, since the lower and upper bounds of control variable j'_k (l'_k and u'_k respectively) can be easily determined as follows: $l'_k = 0$ 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 a non-zero incremental offset. 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} ($l = 1 \dots k-1$). In [5] it is proven that strides and initial offsets in our case can be directly obtained 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).

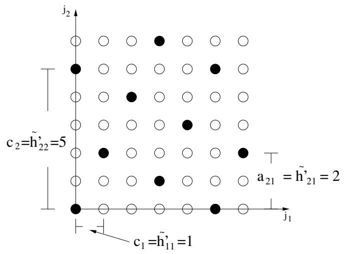

Figure 2. Steps and incremental offsets in $TTIS$ derived from matrix $\widetilde{H'}$

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 [10] 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 over a NUMA architecture. Note, however, that the $(n-1)$-dimensional underlying architecture is not a physical restriction, but a convention for processor naming. 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 perform operations on rectangular data spaces according to a proper memory allocation scheme. 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 to processors. Hodzic and Shang in [6] mapped all tiles along a specific dimension to the same processor and used hyperplane $\Pi = [1, \dots, 1]$ as time schedule vector. In addition to this, previous work [2] 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 $j^S(j_1^S, \dots, j_m^S, \dots, j_n^S)$, where $j_i^S = const$, $i = 1, \dots, m-1, m+1, \dots, n$ 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) 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 one. This corresponds to loop index interchange or permutation. Since all dependence vectors $d^S \in D^S$ are considered lexicographically positive, the interchanging or reordering of indexes is valid (see also [8]).

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 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 communication occurs when one processor needs to read data computed by another. Consequently, 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 iterates over a number of transformed rectangular tiles ($TTIS$s), and locally stores its computed data to a rectangular data space. At the end of all its computations, the processor places its locally computed data to the appropriate positions of the global Data Space (DS). The data space computed by a

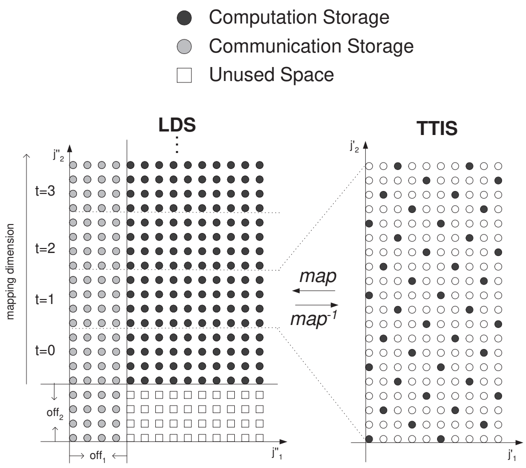

Figure 3. Local Data Space LDS and Transformed Tile Iteration Space $TTIS$

tile could be an exact image of the $TTIS$, but in that 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 received data. Since, after all transformations, we finally work with rectangular sets, this Local Data Space denoted LDS allocated by a processor can be formally 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 unpacking received 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 received data, is derived from the communication criteria of the algorithm as shown in the next subsection.

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 required 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 LDSs 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).

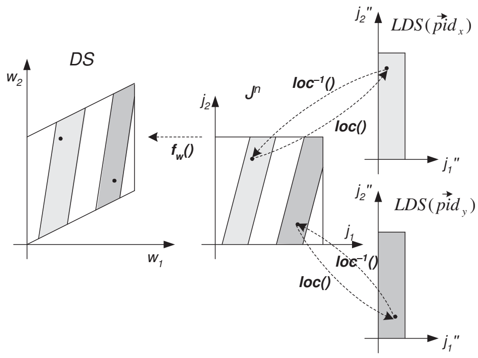

Figure 4. Relations between DS, J^n and LDS

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

$j'' = map(j', t)$
$j''_k = j'_k/c_k + off_k, k \neq m$ $j''_m = (tv_{mm} + j'_m)/c_m + off_m$
$j'', \vec{pid} = loc(j)$
$j^S = \lfloor Hj \rfloor$ $j' = H'(j - Pj^S)$ $j'' = map(j', j^S_m - l^S_m)$ $\vec{pid} = (j^S_1, \ldots, j^S_{m-1}, j^S_{m+1}, \ldots, j^S_n)$

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 is 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

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

$$j' = map^{-1}(j'')$$

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

$$j = loc^{-1}(j'', \vec{pid})$$

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

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 easily be 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 this 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 [10], 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. The two functions are described in detail in [10]. LA denotes an array in local memory which implements the LDS. In the `RECEIVE` routine, we denote as l'_i, u'_i the bounds of the tile $((\vec{pid}, t^S) - d^S)$, from which the current tile $(\vec{pid}, t^S)$ is dependent.

RECEIVE($\vec{pid}, t^S, D^S, \vec{CC}$)
 FOR $d^S \in D^S$ DO /*For all tile dependencies...*/
  /*...if predecessor tile valid and current tile lexicographically minimum successor...*/
  IF(valid($(\vec{pid}, t^S) - d^S$) $\wedge$
      $(\vec{pid}, t^S)$=minsucc($(\vec{pid}, t^S) - d^S, d^m(d^S)$))
   /*...receive data from predecessor processor...*/
   MPI_Recv(buffer,Rank($\vec{pid} - d^m(d^S)$),...);
   /*...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($j', t^S - l^S_n$)-($\frac{d^S_1 v_{11}}{c_1}, \ldots, \frac{d^S_n v_{nn}}{c_n}$)]:=
        buffer[count++];
     ENDFOR
     ...
    ENDFOR
    ...
   ENDFOR
  ENDIF

SEND($\vec{pid}, t^S, D^m, \vec{CC}$)
 FOR $d^m \in D^m$ DO /*For all processor dependencies...*/
  /*...if a valid successor tile exists...*/
  IF($\exists d^S(d^m) \in D^S$:valid($(\vec{pid}, t^S) + d^S(d^m)$))
   /*...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($j', t^S - l^S_n$)];
     ENDFOR
     ...
    ENDFOR
    ...
   ENDFOR
   /*...and send to successor processor.*/
   MPI_Send(buffer,Rank($\vec{pid} + d^m$),...);
  ENDIF

Summarizing, the generated data parallel code for the loop of Section 2 would have a form similar to the following:

```
FORACROSS pid_1 = l_1^S TO u_1^S DO
  ...
  FORACROSS pid_{n-1} = l_{n-1}^S TO u_{n-1}^S DO
    /*Sequential execution of tiles*/
    FOR t^S = l_n^S TO u_n^S 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_n^S;
          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 that automatically generates data parallel code using MPI and run our examples on a cluster with 16 identical 500MHz Pentium nodes with 128MB of RAM. The nodes are interconnected with FastEthernet. The scope of our experiments is to accentuate the selection of non-rectangular tiling transformations. As candidate problems we used SOR as presented in [11] and the Jacobi algorithm as presented in [7]. Both algorithms need to be skewed in order to be rectangularly tiled. Thus, we applied rectangular tiling transformation to the skewed spaces, defined by $H_r = \begin{bmatrix} \frac{1}{x} & 0 & 0 \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$, where $x, y, z \in Z^+$. In addition, in the SOR algorithm we applied non-rectangular transformation defined by $H_{nr} = \begin{bmatrix} \frac{1}{x} & 0 & 0 \\ 0 & \frac{1}{y} & 0 \\ -\frac{1}{z} & 0 & \frac{1}{z} \end{bmatrix}$ and mapped tiles along the third dimension to the same processor, while in the Jacobi algorithm we applied non-rectangular tiling defined by $H_{nr} = \begin{bmatrix} \frac{1}{x} & -\frac{1}{2x} & 0 \\ 0 & \frac{1}{y} & 0 \\ 0 & 0 & \frac{1}{z} \end{bmatrix}$ and mapped tiles along the first dimension to the same processor. In both algorithms, the communication volume, the number of required processes and the tile size are the same for both rectangular and non-rectangular transformations. Figure 5 depicts the maximum speedups in the SOR algorithm, obtained by applying different tile sizes to four iteration spaces for rectangular and non-rectangular tiling (variables M, N represent iteration space bounds), while Figure 6 shows the speedups obtained when varying the tile size for a particular iteration space. Figures 7 and 8 depict the respective speedups for the Jacobi algorithm.

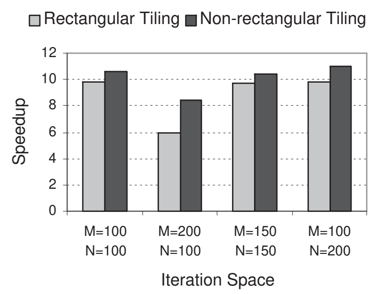

Figure 5. SOR: maximum speedups for different iteration spaces

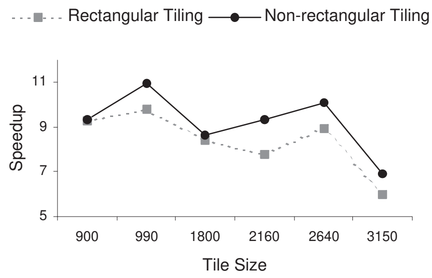

Figure 6. SOR: speedups for various tile sizes (M=100, N=200)

In all cases non-rectangular tiling transformations lead to greater speedups than rectangular ones. Specifically, in SOR we have an average speedup improvement of 17.3%, while in Jacobi an improvement of 9.1%. This confirms the work presented in [7]. Note that all comparison factors between rectangular and non-rectangular tiling are common (tile size, communication volume and number of processors required). Thus, we can safely assert that the reduction in total execution times observed for non-rectangular tiles is due to the more efficient scheduling schemes enabled in this case.

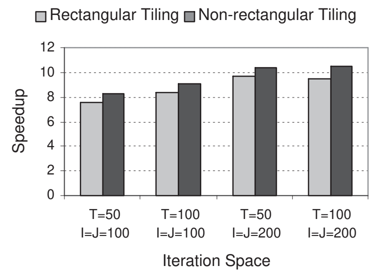

Figure 7. Jacobi: maximum speedups for different iteration spaces

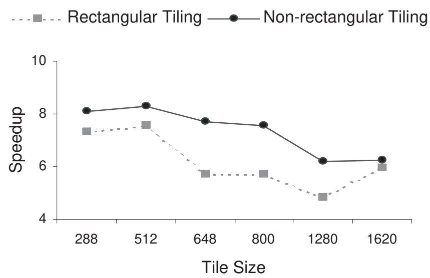

Figure 8. Jacobi: speedups for various tile sizes (T=50, I=J=100)

5 Conclusions-Future Work

In this paper we presented a complete framework to generate data parallel code for arbitrarily tiled iteration spaces. Our work is based on transforming the non-rectangular tile into a rectangular one using a non-unimodular transformation. In this way, we were able to efficiently determine the transformed loop bounds and strides and easily address parallelization issues such as data distribution and automatic message-passing. Our experimental results show that following our approach to execute non-rectangular tiles instead of rectangular ones, we can have a significant increase in speedups, due to more efficient scheduling schemes. Future work includes the combination of our method with advanced scheduling schemes presented in [4].

References

[1] C. Ancourt and F. Irigoin, ”Scanning Polyhedra with DO Loops,” *In Proc. of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP),* pp. 39–50, April 1991.

[2] T. Andronikos, N. Koziris, G. Papakonstantinou and P. Tsanakas, ”Optimal Scheduling for UET/UET-UCT Generalized N-Dimensional Task Graphs,” *Journal of Parallel and Distributed Computing,* vol. 57, no. 2, pp. 140-165, May 1999.

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

[4] G. Goumas, A. Sotiropoulos and N. Koziris, ”Minimizing Completion Time for Loop Tiling with Computation and Communication Overlapping,” *In Proc. of the Int’l Parallel and Distributed Processing Symposium 2001 (IPDPS-2001),* San Francisco, California, April 2001.

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

[6] E. Hodzic and W. Shang, ”On Supernode Transformation with Minimized Total Running Time,” *IEEE Trans. on Parallel and Distributed Systems,* vol. 9, no. 5, pp. 417–428, May 1998.

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

[8] D. Padua and W. Wolfe, ”Advanced Compiler Optimizations for Supercomputers,” *Communications of the ACM,* vol. 29, no. 12, 1986.

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

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

[11] J. Xue, ”Communication-Minimal Tiling of Uniform Dependence Loops,” *Journal of Parallel and Distributed Computing,* vol. 42, no.1, pp. 42–59, 1997. Journal of Parallel and Distributed Computing, vol. 16, pp.108–120, 1992.