# Automatic Parallel Code Generation for Tiled Nested Loops

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

## ABSTRACT

This paper presents an overview of our work, concerning a complete end-to-end framework for automatically generating message passing parallel code for tiled nested for-loops. It considers general parallelepiped tiling transformations and general convex iteration spaces. We address all problems regarding both the generation of sequential tiled code and its parallelization. We have implemented our techniques in a tool which automatically generates MPI parallel code and conducted several series of experiments, concerning the compilation time of our tool, the efficiency of the generated code and the speedup attained on a cluster of PCs. Apart from confirming the value of our techniques, our experimental results show the merit of general parallelepiped tiling transformations and verify previous theoretical work on scheduling-optimal tile shapes.

## Keywords

Nested loops, tiling, supernodes, automatic SPMD code generation, MPI, parallelizing compilers

## 1. INTRODUCTION

During the past decades, explicit parallelism has been achieved using the two general parallel architectures: shared memory and distributed memory. Extensive discussions and practical experience have led to the conclusion that shared memory machines, although easy to program, lack in scalability, while on the other hand distributed memory machines are much more scalable but are definetely more difficult to program. In the past few years it has become obvious, that a two-level hybrid model is the most promising architecture to provide high performance and scalability. Shared memory machines or Symmetric Multi-Processors (SMPs) consisting of 2-16 processors form the lower level of this model. A large number of SMPs is interconnected with a custom or proprietary interconnection network, to form the higher distributed memory level. Thus, all previous experience and research work on both architectures is now combined to enhance the characteristics of this new two-level model.

Programming for two-level parallel architectures can be a very burdensome task, since one has to take into consideration all special characteristics of the underlying hardware (processor speed, cache and cache-line sizes, interconnection network bandwidth and latency etc.). Researchers have proposed a large number of optimizations that can be applied on serial algorithms to optimize performance when running in parallel over a supercomputing platform. Such optimizations mainly concern loop transformations that enhance parallelism [8, 25], exploit cache locality [17, 24] or reduce communication and synchronization needs [21, 23]. Without such optimizations, the real performance of a parallelized algorithm may be far from the peak performance of the parallel platform. However, performing loop transformations by hand is very often impossible even for the most experienced programmers, let alone for the general scientists (physical/chemical scientists, engineers) who are the large majority of parallel programmers. In these cases, a parallelizing compiler is used to both parallelize and optimize the serial code [1, 2, 6, 10]. The automatically generated parallel code best exploits the characteristics of the underlying parallel architecture.

One of the most popular loop transformations is tiling or supernode transformation [16]. Tiling groups neighboring loop iterations together to form a tile or supernode. Researchers have proposed the use of tiling in order to better exploit cache locality in uniprocessors and reduce communication frequency in distributed memory multiprocessors. In the latter case, a lot of research has been done concerning the selection of a proper parallelepiped tile shape which will further reduce communication volume and allow for a more efficient scheduling scheme by minimizing the idle times of processors [5, 7, 13, 14, 15]. When grouping neighboring iterations in orthogonal parallelograms, code generation for the transformed space is quite simple and can be accomplished even by hand. However, when grouping in general parallelepiped tiles, hand coding is impossible. In addition, when the shape of the iteration space is non-rectangular as well, then the compilation process is greatly complicated, while the generated code may result to be very inefficient.

In this paper we present a complete compilation approach for arbitrarily tiled iteration spaces. By “arbitrary” tiling transformations, we mean general parallelepiped groupings (for simplicity also referred to as non-rectangular) in contrast to the simple orthogonal (or rectangular). The generated code complies to the general message-passing programming paradigm, which in our case is implemented with the use of the MPI (Message Passing Interface) library. This code best suits the upper level of the two-level parallel architecture described before, but can be alternatively executed on both levels, using each processor as a distinct processing unit, which communicates with other units via message-passing, regard-

SAC 2004, Nicosia, Cyprus
©2004 ACM 1-58113-445-2/02/03...$5.00

less of the neighboring unit being a processor within the same SMP or a processor in a distant node. In our approach, we implement a new method which greatly decreases the compilation time and simultaneously improves the quality of the generated parallel code. We address all problems concerning the parallelization of an arbitrarily tiled iteration space, including sequential tiled code generation, iteration distribution, data distribution and message-passing code generation.

The rest of the paper is organized as follows: Section 2 presents the algorithmic model and discusses tiling transformation in greater detail. Section 3 presents our approach for generating sequential tiled code, i.e. code that reorders the initial loop order as dictated by the selected tiling transformation. Section 4 presents the parallelization process of the sequential tiled code, including iteration distribution, data distribution and message-passing code generation. In Section 5 we present experimental results which exhibit the acceleration in the compilation process that our method incurs, along with the overall improvement in parallel time due to non-rectangular tiling transformations.

## 2. PRELIMINARIES

### 2.1 The Model of the Algorithms

In this paper we consider algorithms with perfectly nested FOR-loops 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
        $Loop\ \ Body$
      ENDFOR
         ...
  ENDFOR
ENDFOR
```

where $l_1$ and $u_1$ are rational-valued parameters, $l_k$ 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. Therefore, we are not only dealing with rectangular iteration spaces, but also with more general 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 $Z^n$.

### 2.2 Tiling (Supernode) Transformation

When parallelizing a nested for-loop in a distributed memory architecture, it is crucial to mitigate the communication overhead. Despite the development of high-performance System Area Networks used to interconnect the nodes in a distributed memory machine, the communication startup latency has always been a great concern in parallel computing. Coarse-grained parallelism is traditionally proposed as the most suitable form of parallelism when the processing nodes do not physically share a common memory. Several linear loop transformations are considered to elevate coarse-grained parallelism when inherent in an algorithm. Unfortunately, in many cases nested loops cannot be linearly transformed to achieve this kind of parallelism. In these cases, coarse-grained parallelism is forced with the application of a non-linear loop transformation, the tiling or supernode transformation ([20]).

Tiling transformation groups neigboring iterations of a nested loop to form a *"tile"* (or *"supernode"*). Tiles are assigned to processors using a pre-defined mapping scheme and executed as a complete computation unit. Communication occurs before and after the computation of the iterations within the tile. In this way, communication data are packed into larger messages thus reducing communication frequency and the consequent startup overhead. Extensive research has investigated the impact of the tile size and shape on the performance of a parallelized nested loop ([5, 7, 13, 14, 15, 20, 26]). Researchers have proven that analyzing the data dependences of the original algorithm, one can optimally choose between various tile shapes. Optimality in our case means minimum communication and/or minimum idle processor time. Tile shape is proven to affect both communication volume and processor idle times.

Despite the profound benefits of tiling transformation, a number of important issues concerning code generation need to be addressed. Code generation for tiled iteration spaces can be a very burdensome task, especially when non-rectangular tile shapes are concerned. In several cases compilation time may be impractical, while the memory space required to apply the proper compilation techniques may exceed the size of the virtual memory. More importantly, even when compilation succeeds in acceptable time, the generated code may be of poor quality and thus not capable of enhancing the theoretical benefits of tiling transformations. In the rest of this paper we shall describe a complete end-to-end framework, used to produce efficient parallel tiled code in practical compilation times. We have implemented a compiler (Fig. 1), which, according to the dependence constraints of the initial code finds the optimal tiling transformation and, reordering the execution order of iterations, produces the sequential tiled code. In the sequel, sequential code is automatically transformed to MPI (Message Passing Interface) parallel code. In this paper we focus on issues concerning the last two steps of the automatic procedure: the generation of the sequential tiled code and its parallelization.

## 3. SEQUENTIAL CODE GENERATION

When generating code for a tiled nested loop, we essentially need to transform a code segment of $n$ nested for-loops into another one with $2n$ nested for-loops. The $n$ outer loops enumerate the tiles, while the $n$ interior loops traverse the internal points of the tiles.

**Example 1:** Consider the following simple code segment:

```
FOR $j_1 = 0$ TO $11$ DO
  FOR $j_2 = 0$ TO $11$ DO
    A[$j_1$,$j_2$]=1/2*(A[$j_1$-1,$j_2$]+A[$j_1$-1,$j_2$-1]);
  ENDFOR
ENDFOR
```

If we apply tiling transformation to form groups (tiles) of $4 \times 4$ iterations (Fig. 2), then we need to transform the previous code as follows:

```
FOR $t_1 = 0$ TO $2$ DO
  FOR $t_2 = 0$ TO $2$ DO
    FOR $j_1 = 4 * t_1$ TO $4 * t_1 + 3$ DO
      FOR $j_2 = 4 * t_2$ TO $4 * t_2 + 3$ DO
        A[$j_1$,$j_2$]=1/2*(A[$j_1$-1,$j_2$]+A[$j_1$-1,$j_2$-1]);
      ENDFOR
    ENDFOR
  ENDFOR
ENDFOR                                                    □
```

When both the initial iteration space and the tile are rectangular, the

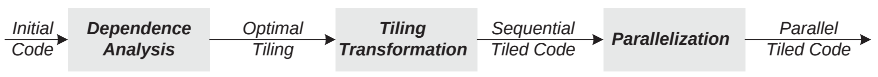


**Figure 1: Automatic Parallel Code Generation for Tiled Iteration Spaces**

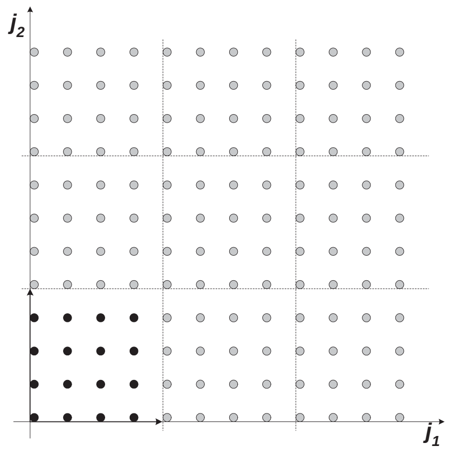


**Figure 2: Rectangular Tiling Transformation**

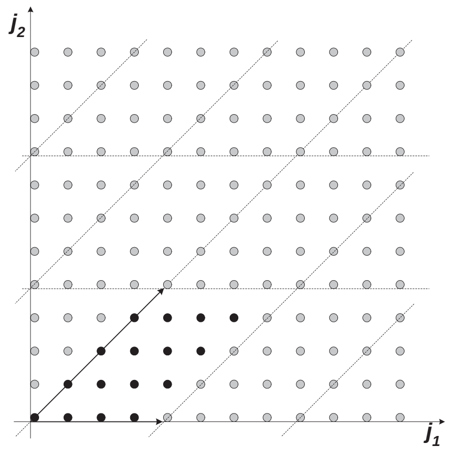


**Figure 3: Non-rectangular Tiling Transformation**

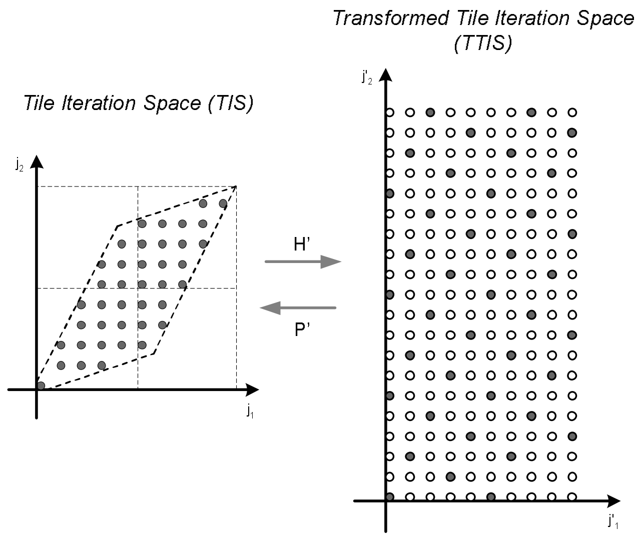
calculation of the loop bounds can be carried out very efficiently, as in Example 1. However, when the tiling transformation and/or the iteration space are not rectangular, the calculation of the new loop bounds can be a very complex and time consuming compiler work. Ancourt and Irigoin in [3], have constructed a proper set of inequalities describing the initial iteration space and the tiling transformation and propose the use of the Fourier-Motzkin elimination (FM) on the set to obtain code that traverses the initial space as dictated by the tiling transformation. Unfortunately, FM is an extremely complex method (doubly exponential) and thus the above method may result to be impractical if special care is not taken.

**Example 2:** For the code segment of Example 1, both communication and scheduling criteria imply that a tiling transformation such as the one shown in Fig. 3, should lead to more efficient parallel code. However, when writing down the transformed nested loops, according to the method described in [3], we get the following code segment:

```
FOR $t_1$=-3 TO 2 DO
  FOR $t_2$=max(0,-1-$t_1$) TO min(2,2-$t_1$) DO
    FOR $j_1$=max(0,4*$t_1$+4*$t_2$) TO
                           min(11,6+4*$t_1$+4*$t_2$) DO
      FOR $j_2$=max(4*$t_2$,-3-4*$t_1$+$j_1$) TO
                            min(4*$t_2$+3,-4*$t_1$+$j_1$) DO
        A[$j_1$,$j_2$]=1/2*(A[$j_1$-1,$j_2$]+A[$j_1$-1,$j_2$-1]);
      ENDFOR
    ENDFOR
  ENDFOR
ENDFOR
```
□

In [11] we have proposed a novel method for calculating the outer loop bounds, which considerably reduces the time required for the generation of the sequential tiled code. This simplification is due to the fact that we managed to cope with the same problem by constructing a system of inequalities with half the number of inequalities and half the number of unknown variables. FM is now applied on this greatly simplified set, which vastly accelerates the compilation time. The tradeoff in constructing such a smaller system of inequalities is the inexact description of the space in consideration. In fact, our method includes some redundant tiles in the enumeration, but this run-time overhead proves to be negligible in practice, since the internal points of these redundant tiles are never accessed.

Note the difference in the tiled code of Examples 1 and 2. In order to scan the internal points of the tiles in Example 2 (non-rectangular tiling), the corresponding loop bounds contain *min* and *max* functions of affine expressions containing the surrounding loop control variables. In some cases these functions may consist of a very large number of expressions which need to be evaluated at run-time. This fact greatly degrades performance and may lead to poor parallel execution times, so that the theoretical benefit from the application of non-rectangular tiling is not revealed in practice. In order to reduce this run-time overhead, we have presented a new method based on the use of a non-unimodular transformation ([11]). Our goal is to reduce the number of affine expressions in the loop evaluation of the loop bounds. The method transforms any non-rectangular tile (Tile Iteration Space-TIS) into a rectangular one (Transformed Tile Iteration Space-TTIS) using transformation $H'$ (Fig. 4), then traverses the rectangular tile and finally accesses the original point by using the inverse transformation $P'$. Note that, the transformed space may contain many points whose anti-image is not an integer point in the initial Tile Iteration Space. These points are depicted with white dots in Fig. 4 and are called "holes". However holes can be easily avoided when scanning TTIS by using specific increment steps in the $n$ inner loop indexes. The appropriate increment steps are directly obtained from the Hermite Normal Form of the transformation matrix, as proved in [19, 9] and discussed in [11]. Thus, we have managed to greatly reduce the number of affine expressions in the loop bounds of the transformed loop, with the overhead of a linear transformation required for each iteration from the transformed iteration space to the initial iteration space. The final outcome of the method is a minor run-time overhead for non-rectangular tiling transformations, since the integer multiplications required for the transformations are efficiently executed by any modern microprocessor.



Transformed Tile Iteration Space (TTIS)
Tile Iteration Space (TIS)
j₂
j₁
H'
P'
j'₂
j'₁

**Figure 4: Traverse the $TIS$ with a non-unimodular transformation**

**Example 3:** If we generate sequential tiled code for the tiling transformation presented in Example 2 and shown in Fig. 3, using our method, we get the following code:

```
FOR $t_1$=-3 TO 2 DO
  FOR $t_2$=max(0,-1-$t_1$) TO min(2,2-$t_1$) DO
    FOR $j'_1$=0 TO 3 DO
      FOR $j'_2$=0 TO 3 DO
        $j_1$=$j'_1$+$j'_2$
        $j_2$=$j'_2$
        A[$j_1$,$j_2$]=1/2*(A[$j_1$-1,$j_2$]+A[$j_1$-1,$j_2$-1])
      ENDFOR
    ENDFOR
  ENDFOR
ENDFOR
```
□

Note in Example 3, that we have greatly reduced the *min* and *max* functions in the evaluation of loop bounds. In fact, the code segment shown in Example 3 is correct only for internal tiles (that is, tiles that do not cross the iteration space boundaries). However, as depicted in Fig. 3, boundary tiles may not be equal to the internal ones. We can cope with this problem by inserting some more control variables only in the few cases of boundary tiles. Since in most real applications the number of boundary tiles is minimal compared to the number of internal tiles, this technique does not add considerable run-time overhead.

## 4. DATA PARALLEL CODE GENERATION

The parallelization of the sequential tiled code involves issues such as computation distribution, data distribution and communication among processors. Tang and Xue in [22] addressed the same issues for rectangularly tiled iteration spaces. We can also generate efficient data parallel code for non-rectangular tiles without imposing any further complexity. We consider that the memory of the underlying architecture is physically distributed among processors, essentially referring to the upper level of a two-level parallel architecture. Processors perform computations on local data and communicate with each other with messages in order to exchange data that reside to remote memories. Practically, we can implement this model with a cluster of PCs (either uniprocessor or multiprocessor) interconnected with a custom or proprietary interconnection network, using a message-passing software platform like MPI. The underlying architecture is viewed by the compiler as a $(n-1)$-dimensional processor mesh. Thus, each processor is identified by a $(n-1)$-dimensional vector denoted $\vec{pid}$.

### 4.1 Computation and Data Distribution

The general intuition in our approach is that since the iteration space is already transformed 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, both of which are rectangular, and properly translates from its local data space to the global one.

Computation distribution determines which computations of the sequential tiled code will be assigned to which processor. We assign to each processor a row of tiles parallel to the longest dimension of the problem space, since previous work [4] 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.

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 thus, communication occurs when one processor needs to read data computed by another. However, with non-rectangular tiling, each processor should own a non-rectangular subset of the original data space. We allocate memory for the data computed by the rectangular transformed space (TTIS), taking care to condense the actual points and ignore the holes. Each processor allocates an $n$-dimensional rectangular space equal to the number of actual points of the Transformed Tile Iteration Space multiplied by the number of tiles assigned to the particular processor (Fig. 5). We call this space Local Data Space (LDS).

### 4.2 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 dependences. 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 Transformed Tile Iteration Space to deduce simple criteria for the com-

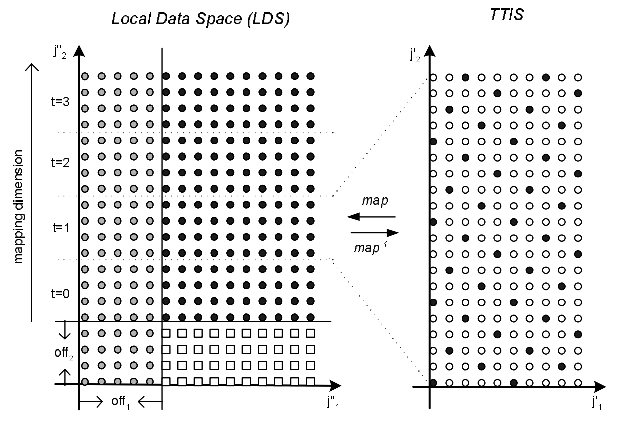


**Figure 5: Local Data Space $LDS$ and Transformed Tile Iteration Space $TTIS$**

munication points at compile time. More specifically, as depicted in Fig. 6, while in the initial iteration space communication points belong to a non-rectangular parallelepiped, in the transformed space they are restricted into a rectangle. Thus they can be easily determined by restricting a loop variable among constants.

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 Local Data Space. Dually, after the completion of a tile, the processor must send part of the computed data to the neighboring processors for later use. Summarizing, the generated data parallel code for the loop of §2.1 would have a form similar to the following:

```
DOACROSS $pid_1$=$l_1^S$ TO $u_1^S$ DO
  ...
  DOACROSS $pid_{n-1}$=$l_{n-1}^S$ 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_DATA(...);
      /*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
          EXECUTE LOOP BODY
        ENDFOR
        ...
      ENDFOR
      /*Send data to neighboring processors*/
      SEND_DATA(...);
    ENDFOR
  ENDDOACROSS
  ...
ENDDOACROSS
```

## 5. EXPERIMENTAL RESULTS

### 5.1 Measuring the compilation time and performance of sequential tiled code

We have implemented both our method (denoted as RI - Reduced Inequalities), which is described in detail in [11], and the one presented in [3] by Ancourt and Irigoin (denoted as AI) in a software

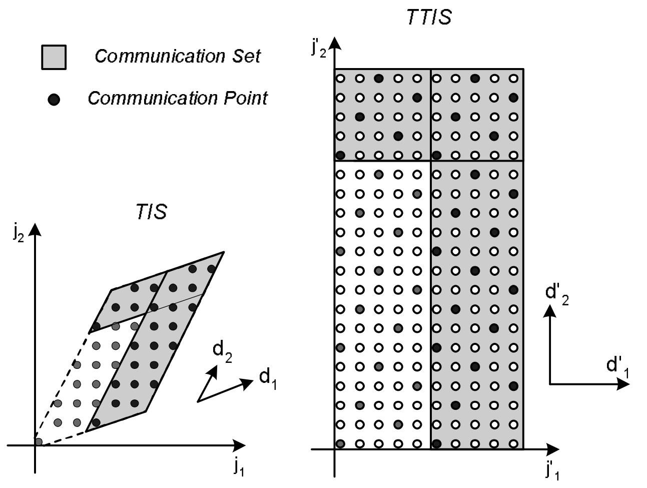


**Figure 6: Determining Communication Sets in the Tile Iteration Space (TIS) and Transformed Tile Iteration Space (TTIS)**

tool which automatically generates sequential tiled C++ code using any tiling transformation. In this section we compare AI and RI methods both in terms of compilation time and generated code efficiency. We applied both AI and RI methods to three real applications: Gauss Successive Over-Relaxation (SOR), Jacobi and ADI integration. We also applied the inequalities of AI method to the Omega calculator [18] and generated sequential code for the same problems. We then measured the compilation time and run time obtained by Omega (the results are denoted as AI-Omega) and compared them with the ones obtained by AI (using our tool) and RI. We performed our experiments on a PIII @ 800MHz processor with 128MB of RAM. The operating system is Linux with kernel 2.4.18. The generated tiled code was compiled using gcc v.2.95.4 with the -O3 optimization flag. We also experimented with lower optimization levels, where the execution times were slower but the relative results for all methods remained the same.

We applied AI and RI methods to tile three real applications: SOR, Jacobi, and ADI integration. For the first two problems there is a skewed and an unskewed version [12], and for each version there are four (communication and scheduling) optimal tiling matrices ($P_1 - P_4$), calculated as described in [14] and [26]. The compilation efficiency of a method is measured by means of the compilation time, which increases as the number of row-operations performed increases. In order to evaluate the runtime overhead due to tiling, we executed all tiled codes of the previous problems and measured their run time. We also executed the original untiled serial code for each problem. We define the Tiling Overhead Factor - TOF, to be the fraction of the runtime of the sequential tiled code to the runtime of the untiled code: $TOF = \frac{\text{Runtime of Sequential Tiled Code}}{\text{Runtime of Untiled Code}}$. Thus, TOF indicates the overhead imposed by the evaluation of the new loop bounds, due to tiling. If TOF is too large, it will aggravate the speedup obtained when we parallelize nested `FOR`-loops using tiling. Ideally, TOF equals to 1, meaning that tiling imposes no overhead to the generated code. Table 1 summarizes the row operations, compilation times and TOFs for each case.

As far as compilation time is concerned, RI method clearly outperforms AI method. This is due to the fact that RI supplies FM with a system of inequalities consisting of $2n$ inequalities with $n$ variables, while AI method supplies FM with a system consisting of

| | | ***Row Operations*** | | ***Compilation Time*** $ms$**)** | | | ***TOF*** | | |
|---|---|---|---|---|---|---|---|---|---|
| | | **AI** | **RI** | **AI-Omega** | **AI** | **RI** | **AI-Omega** | **AI** | **RI** |
| **SOR** | $P_1$ | 99 | 22 | 53.03 | 0.50 | 0.42 | 1.47 | 1.20 | 1.05 |
| | $P_2$ | 107 | 22 | 50.27 | 0.53 | 0.42 | 1.50 | 1.21 | 1.01 |
| | $P_3$ | 118 | 22 | 49.01 | 0.57 | 0.42 | 1.75 | 1.63 | 1.05 |
| | $P_4$ | 165 | 40 | 90.04 | 0.77 | 0.5 | 1.80 | 1.78 | 1.30 |
| **SOR skewed** | $P_1$ | 99 | 22 | 42.09 | 0.53 | 0.41 | 1.59 | 1.29 | 1.06 |
| | $P_2$ | 107 | 22 | 40.60 | 0.53 | 0.42 | 1.60 | 1.29 | 1.06 |
| | $P_3$ | 118 | 22 | 57.9 | 0.57 | 0.42 | 1.90 | 1.73 | 1.12 |
| | $P_4$ | 165 | 40 | 91.97 | 0.77 | 0.51 | 1.95 | 1.86 | 1.34 |
| **Jacobi** | $P_1$ | 645 | 28 | 346.99 | 5.3 | 0.46 | 2.08 | 1.91 | 1.57 |
| | $P_2$ | 645 | 28 | 347.96 | 5.26 | 0.47 | 2.09 | 1.92 | 1.60 |
| | $P_3$ | 800 | 28 | 362.5 | 8.86 | 0.47 | 2.06 | 1.90 | 1.56 |
| | $P_4$ | 3207 | 46 | $1,353.55$ | 194.88 | 0.53 | 5.58 | 5.09 | 2.10 |
| **Jacobi skewed** | $P_1$ | 645 | 28 | 251.885 | 4.93 | 0.48 | 1.99 | 1.88 | 1.44 |
| | $P_2$ | 645 | 28 | 248.27 | 4.98 | 0.47 | 1.98 | 1.87 | 1.46 |
| | $P_3$ | 800 | 28 | 229.34 | 8.19 | 0.48 | 2.02 | 1.89 | 1.45 |
| | $P_4$ | 691 | 28 | 238.82 | 5.95 | 0.47 | 2.01 | 1.88 | 1.43 |
| **ADI** | $P_1$ | 180 | 28 | 47.42 | 0.85 | 0.46 | 1.46 | 1.47 | 1.07 |

**Table 1: Sequential Code Performance for Real Applications**

$4n$ inequalities with $2n$ variables. Despite the reduction in compilation time imposed by RI, it seems that both AI and AI-Omega perform well (compilation times are less than one second). However, in problems of larger dimensions, both AI and AI-Omega present several problems. We executed a number of randomly generated $4-D$ algorithms and observed that, at first, the compilation time of AI becomes impractical (several hours or even days). More importantly, AI failed to generate code for almost half of the problems due to lack of memory. On the other hand, AI-Omega also faced some problems with memory space (to a smaller extent than AI) but here again in almost half of the problems the system rose an overflow exception. Apparently, after a large number of row operations in $4-D$ algorithms, some coefficients exceeded the maximum integer $2^{32}$. In all cases RI method succeeded in generating code within some seconds in the worst case.

As far as run time is concerned, RI also exhibits a significant improvement in performance in all problems. As shown in Table 1, RI's performance is nearly optimal in simple algorithms such as SOR, Jacobi and ADI, since the TOF in these cases is very close to one. The improvement in the quality of the generated code caused by RI, is due to the fact that, although the code to enumerate the tiles is essentially similar in AI and RI, the code to traverse the internal points of the tiles is completely different. Our tool makes a distinction between boundary and internal tiles and generates different code to scan the internal points in each case. In the case of boundary tiles, RI method results in fewer inequalities for the bounds of the Tile Space. Consequently, fewer bound calculations are executed during run time. Finally, note that the enumeration of some redundant tiles does not impose any significant overhead since the number of redundant tiles is negligible. The same holds for the transformation used to access the internal points of the tiles. In this case, the additional operations due to the transformation are simple integer multiplications, while operations on extra variables are integer additions and assignment statements which are all efficiently executed by modern processors and optimized by any backend compiler like gcc.

## 5.2 Measuring the performance of arbitrarily tiled parallel code

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 effect of the tile shape on the overall completion time of an algorithm. In the following set of experiments, we used the same three real problems: SOR, the Jacobi algorithm and ADI integration. In each case, we applied rectangular and non-rectangular tiling transformations. Although, as described in [12], non-rectangular tiling can be directly applied to the initial SOR and Jacobi code, in order to compare rectangular v.s. non-rectangular tiling, we apply them to the skewed version of their iteration space. As far as non-rectangular tiling is concerned, we apply the theoretically optimal tiling transformation, automatically calculated by our tool, as described in [14, 26].

In order to have a theoretical interpretation of the experimental results that follow, let us have a look at Figs. 7, 8. Using non-rectangular tiling, all processors can start their execution simultaneously, while using rectangular tiling, they should wait until the necessary data are transmitted to them. In Figs 7, 8 we have denoted the time step during which each tile will be executed. Since the volume of tiles is equal, the time step in rectangular tiling will be equal to the one in non-rectangular tiling. Thus, the number of time steps required for the completion of the algorithm is proportional to the time required.

We executed a large number of experiments for all problems, applying different sizes of tiling transformations on various iteration spaces. Due to space limitations, we will present the results of one iteration space per problem. However, the results for other iteration spaces were similar. As far as the SOR and Jacobi prob-

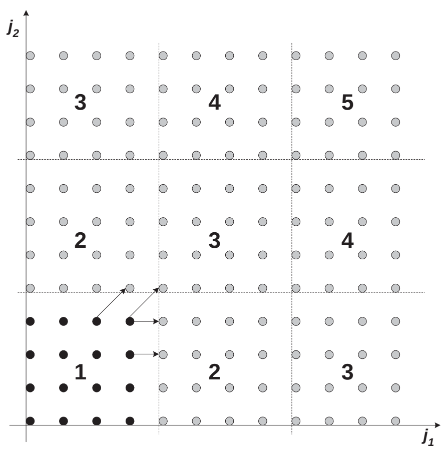


**Figure 7: Timing for Rectangular Tiling Transformation**

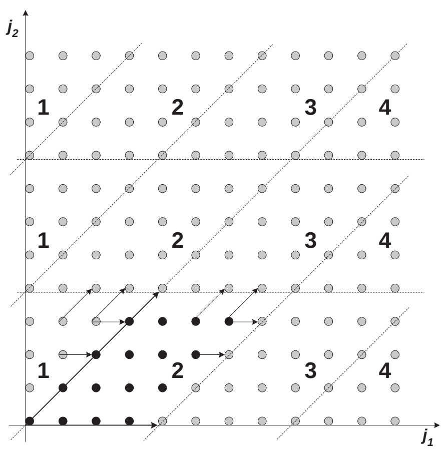


**Figure 8: Timing for Non-rectangular Tiling Transformation**

lems are concerned, we have chosen tiling and mapping parameters so that the communication volume and the number of processors required are the same in both rectangular and non-rectangular tiling. Thus, any differences in execution times are due to the different scheduling schemes imposed by the different tile shapes. In Fig. 9 we present the total execution times for rectangular and non-rectangular tiling for SOR. In this case the minimum execution time is $20.6\%$ smaller and the average execution time is $16.1\%$ smaller in the case of non-rectangular tiling. Applying non-rectangular tiling in Jacobi (Fig. 10), we achieved $27.4\%$ smaller minimum execution time and $36.7\%$ smaller average execution time. Concerning the ADI integration problem, we applied four different tiling transformations: one rectangular, one optimal non-rectangular and two more non-rectangular tiling transformations. In all four cases we have the same tile size, communication and number of processors. In Fig. 9 we show the execution times. In this case the optimal non-rectangular tiling transformation achieved $23.4\%$ smaller total execution time and $72.1\%$ smaller average execution time.

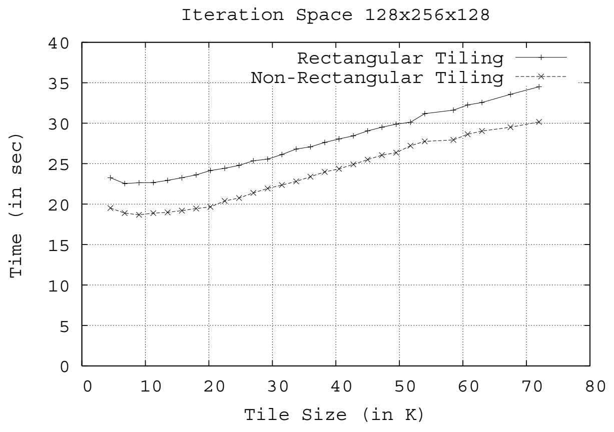


**Figure 9: SOR: Total Execution Times for Rectangular and Non-Rectangular Tiling Transformations**

## 6. CONCLUSIONS

In this paper, we presented and experimentally evaluated a novel approach for the problem of automatically generating code for the parallel execution of tiled nested loops. Our method is applied to general parallelepiped tiles and non-rectangular space boundaries as well. In order to efficiently generate parallel tiled code, we divided the problem into two subproblems: the generation of sequential tiled code and the parallelization of sequential tiled code. In the first case, we extended previous work on non-unimodular transformations in order to produce tiled code solving small systems of inequalities and then transform the parallelpiped tiles into rectangles. In the second case, we exploit the regularity of transformed rectangular tiles in order to easily decide the computation and communication sets to be assigned to each processor. Experimental results show that our method outperforms previous work in terms both of compile and run time. In addition, it can fully exploit the scheduling efficiency of non-rectangular tiles.

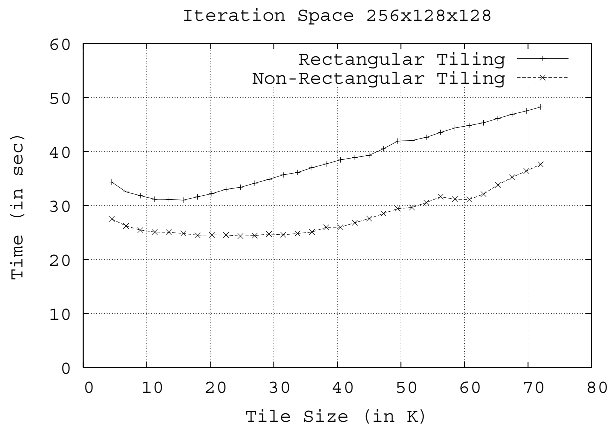


**Figure 10: Jacobi: Total Execution Times for Rectangular and Non-Rectangular Tiling Transformations**

## 7. 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

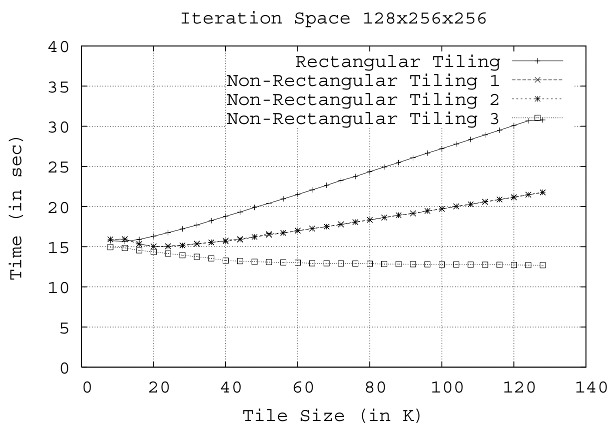


**Figure 11: ADI: Total Execution Times for One Rectangular and Three Non-Rectangular Tiling Transformations (#3 is the theoretically optimal)**

Machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Albuquerque, NM, Jun 1993.

[3] C. Ancourt and F. Irigoin. Scanning Polyhedra with DO Loops. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP)*, pages 39–50, Williamsburg, VA, Apr 1991.

[4] 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.

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

[6] 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.

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

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

[9] A. Fernandez, J. Llaberia, and M. Valero. Loop Transformations Using Nonunimodular Matrices. *IEEE Trans. on Parallel and Distributed Systems*, 6(8):832–840, Aug 1995.

[10] 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.

[11] 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)*, pages 876–881, Madrid, Spain, Mar 2002.

[12] G. Goumas, N. Drosinos, M. Athanasaki, and N. Koziris. Compiling Tiled Iteration Spaces for Clusters. In *Proceedings of the 2002 IEEE International Conference on Cluster Computing*, pages 360–369, Chicago, Illinois, Sep 2002.

[13] 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.

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

[15] K Hogstedt, 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.

[16] F. Irigoin and R. Triolet. Supernode Partitioning. In *Proceedings of the 15th Ann. ACM SIGACT-SIGPLAN Symp. Principles of Programming Languages*, pages 319–329, San Diego, California, Jan 1988.

[17] M. Kandemir, J. Ramanujam, and A. Choudary. Compiler Algorithms for Optimizing Locality and Parallelism on Shared and Distributed Memory Machines. *Journal of Parallel and Distributed Computing*, 60:924–965, 2000.

[18] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. The Omega Library Interface Guide. Technical Report CS-TR-3445, CS Dept., Univ. of Maryland, College Park, Mar 1995.

[19] J. Ramanujam. Beyond Unimodular Transformations. *Journal of Supercomputing*, 9(4):365–389, Oct 1995.

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

[21] J.-P. Sheu and T.-H. Tai. Partitioning and Mapping Nested Loops on Multiprocessor Systems. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):430–439, Oct 1991.

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

[23] P. Tsanakas, N. Koziris, and G. Papakonstantinou. Chain Grouping: A Method for Partitioning Loops onto Mesh-Connected Processor Arrays. *IEEE Trans. on Parallel and Distributed Systems*, 11(9):941–955, Sep 2000.

[24] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Ontario, Jun 1991.

[25] 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.

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