Delivering High Performance to Parallel Applications Using Advanced Scheduling

Nikolaos Drosinos [a], Georgios Goumas [a], Maria Athanasaki [a] and Nectarios Koziris [a]

[a] National Technical University of Athens
School of Electrical and Computer Engineering
Computing Systems Laboratory
Zografou Campus, Zografou 15773, Athens, Greece
e-mail: {ndros, goumas, maria, nkoziris}@cslab.ece.ntua.gr

This paper presents a complete framework for the parallelization of nested loops by applying tiling transformation and automatically generating MPI code that allows for an advanced scheduling scheme. In particular, under advanced scheduling scheme we consider two separate techniques: first, the application of a suitable tiling transformation, and second the overlapping of computation and communication when executing the parallel program. As far as the choice of a scheduling-efficient tiling transformation is concerned, the data dependencies of the initial algorithm are taken into account and an appropriate transformation matrix is automatically generated according to a well-established theory. On the other hand, overlapping computation with communication partly hides the communication overhead and allows for a more efficient processor utilization. We address all issues concerning automatic parallelization of the initial algorithm. More specifically, we have developed a tool that automatically generates MPI code and supports arbitrary tiling transformations, as well as both communication schemes, e.g. the conventional receive-compute-send scheme and the overlapping one. We investigate the performance benefits in the total execution time of the parallel program attained by the use of the advanced scheduling scheme, and experimentally verify with the help of application-kernel benchmarks that the obtained speedup can be significantly improved when overlapping computation with communication and at the same time applying an appropriate (generally non-rectangular) tiling transformation, as opposed to the combination of the standard receive-compute-send scheme with the usual rectangular tiling transformation.

1. Introduction-Background

Tiling or supernode transformation is one of the most common loop transformations discussed in bibliography, proposed to enhance locality in uniprocessors and achieve coarse-grain parallelism in multiprocessors. Tiling groups a number of iterations into a unit (tile), which is executed uninterruptedly. Traditionally, only rectangular tiling has been used for generating SPMD parallel code for distributed memory environments, like clusters. In [1], Tang and Xue provided a detailed methodology for generating efficient tiled code for perfectly nested loops, but only used rectangular tiles due to the simplicity of the parallel code, since only division and modulo operations are required in this case. However, recent scientific research has indicated that the performance of the parallel tiled code can be greatly affected by the tile size ([2], [3], [4]), as well as by the tile shape ([5], [6], [7]). The effect of the tile shape on the scheduling of the parallel program is depicted in Figure 1. It is obvious that non-rectangular tiling is more beneficial in this particular case than rectangular one, since it leads to fewer execution steps for the completion of the parallel algorithm. The main problem with arbitrary tile shapes appears to be the complexity of the respective parallel code and the performance overhead incurred by the enumeration of the internal points of a non-rectangular tile. Therefore, an efficient implementation of an arbitrary tiling method and its incorporation in a tool for automatic code generation would be desirable in order to achieve the optimal performance of parallel applications.

Elaborating further more on scheduling, under conventional schemes, the required communication between different processors occurs just before the initiation and after the completion of the com-

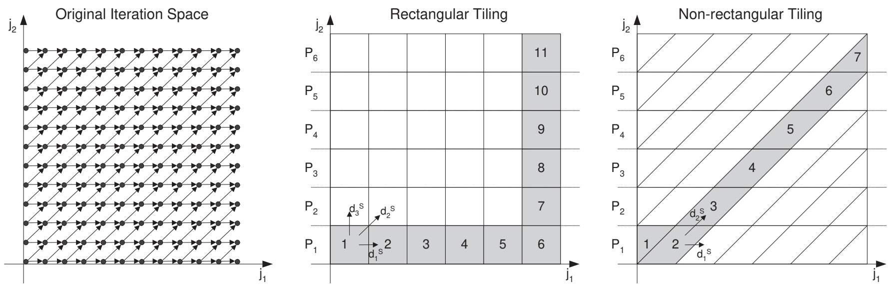

Figure 1. Effect of tile shape on overall completion time

putations within a tile. That is, each processor first receives data, then computes all calculations involved with the current tile, and finally sends data produced by the previous calculations. By providing support for an advanced scheduling scheme that uses non-blocking communication primitives, and consequently allows the overlapping of useful computation with burdensome communication, it is expected that the performance of the parallel application will be further improved. This hypothesis is also established by recent scientific work ([8], [9]). More specifically, the blocking communication primitives are substituted with non-blocking communication functions, which only initialize the communication process, and can be tested for completion at a later part of the program. By doing so, after initializing non-blocking communication the processor can go on with useful computation directly related to the user application. The communication completion can be tested as late as possible, when it will most likely have completed, and thus the processor will not have to stall idle, prolonging the total execution time of the application.

2. Algorithmic Model

Our model concerns n-dimensional perfectly nested loops with uniform data dependencies of the following form:

```
FOR j_1 = min_1 TO max_1 DO
   ...
      FOR j_n = min_n TO max_n DO
         Computation(j_1 , ..., j_n);
      ENDFOR
   ...
ENDFOR
```

The loop computation is a calculation generally involving an n-dimensional matrix A, which is indexed by $j_1, \ldots, j_n$. We assume that the loop computation imposes *lexicographically positive* data dependencies, so that the parallelization of the algorithm with the application of an appropriate tiling transformation is always possible. Also, if the data dependencies are lexicographically positive, an appropriate *skewing* transformation can eliminate all negative elements of the dependence matrix, so that rectangular tiling can be applied, as well.

Furthermore, for the i-th loop bounds min_i, max_i it holds that $min_i = f(j_1, \ldots, j_{i-1})$ and $max_i = g(j_1, \ldots, j_{i-1})$. That is, our model also deals with non-rectangular iteration spaces, under the assumption that they are defined as a finite number of semi-spaces of the n-dimensional space Z^n.

3. Automatic Code Generation

The automatic parallelization process of the sequential program in schematically depicted in Figure 2. The procedure can be divided in three phases, namely the dependence analysis of the algorithm, the application of an appropriate tiling transformation for the generation of intermediate sequential tiled code, and finally the parallelization of the tiled code in terms of computation/data distribution, as well as the implementation of communication primitives.

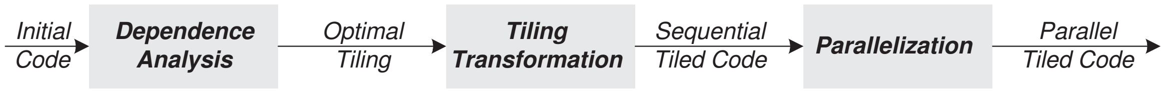

Figure 2. Automatic Parallelization of Sequential Code

The following Subsections elaborate on the automatic parallelization process, emphasizing on the code generation issues.

3.1. Tiled Code Generation

The generation of the sequential tiled code from the initial algorithm mainly implies transforming the n nested loops into $2n$ new ones, where the n outermost loops scan the tile space and the n innermost ones traverse all iterations associated with a specific tile. This equivalent form of the algorithm code is more convenient for the parallelization process, as the computation distribution can be directly applied to the outermost n loops enumerating the tiles.

In case of rectangular tiling and rectangular iteration spaces, the respective sequential tiled code is simple and straightforward, as it is implemented with the aid of integer division and modulo operators ([7]). In the opposite case, if non-rectangular tiling is applied, or a non-rectangular iteration space is considered, the transformation of the initial algorithm into sequential tiled code is a more intricate task, that requires significant compiler work. In [10] we have proposed an efficient compiler technique based on the Fourier-Motzkin elimination method for calculating the outer loops bounds. The efficiency of the proposed methodology lies in that we managed to construct a compact system of inequalities that allows the generation of tiled code, and thus compensates for the doubly exponential complexity of the Fourier-Motzkin method. The simplified system of inequalities enumerates some redundant tiles, as well, but the run-time overhead proves to be negligible in practice, since the internal points of these tiles are never accessed.

As far as the traversing the internal of a tile is concerned, in [10] we further propose a method to transform arbitrary shaped tiles into rectangular ones. By doing so, only rectangular tiles need to be traversed and the expressions required in the n innermost loop bounds evaluation are significantly reduced. Formally, the iteration space of a tile (Tile Iteration Space - TIS) is transformed into a new iteration space, the Transformed TIS ($TTIS$) by using a non-unimodular transformation. The correspondence between the TIS and the $TTIS$ is schematically depicted in Figure 3. It should be intuitively obvious that the $TTIS$ can be more easily traversed in comparison to the TIS, although special care needs to be taken so that only valid points (e.g. black dots in Figure 3) are accessed.

3.2. Parallelization

The sequential tiled code is parallelized according to the SPMD model in order to provide portable MPI C++ code. The parallelization process addresses issues such as computation distribution, data distribution and inter-process communication primitives. We will mainly focus on the communication scheme, as the computation and data distribution are more extensively analyzed in [10].

Each MPI process assumes the execution of a sequence of tiles along the longest dimension of the tile iteration space, as previous work in the field of UET-UCT graphs ([9]) suggests that this

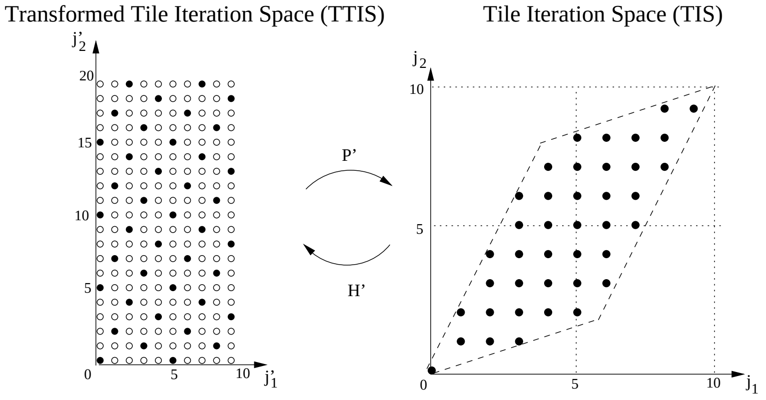

Figure 3. Transformation of arbitrary shaped tile into rectangular

scheduling is optimal. The n outermost loops of the sequential tiled code are reordered, so that the one corresponding to the maximum-length dimension becomes the innermost of the n. Each worker process is identified by an $n-1$ dimensional *pid* vector directly derived from its MPI rank, so that it undertakes the execution of all tiles whose $n-1$ outermost coordinates match *pid*. Also, data distribution follows the *computer-owns* rule, e.g. each worker process owns the data it computes. By adopting the above computation and data distribution, the required SPMD model for the parallelization of the sequential tiled code is relatively simple and efficient, as far as the overall performance is concerned.

Finally, in order for the worker processes to be able to exchange data, certain communication primitives need to be supplied to the parallel code. We have implemented two communication patterns, namely one based on blocking MPI primitives (`MPI_Send`, `MPI_Recv`), and an alternative one based on non-blocking MPI primitives (`MPI_Isend`, `MPI_Irecv`). In the first case (blocking), each worker process initially receives all non-local data required for the computation of a tile, then computes that tile, and finally sends all computed data required by other processes (Table 1). Note that in this case communication and computation phases are distinct and do not overlap. In the second case (non-blocking), each worker process concurrently computes a tile, receives data required for the next tile and sends data computed at the previous tile (Table 2). This communication scheme allows for the overlapping of computation and communication phases.

```
for(tile t){
    MPI_Recv(t);
    Compute(t);
    MPI_Send(t);
}
```

Table 1
Blocking communication scheme

```
for(tile t){
    MPI_Irecv(t+1);
    MPI_Isend(t-1);
    Compute(t);
    MPI_Waitall;
}
```

Table 2
Non-blocking communication scheme

It is obvious that the non-blocking communication scheme allows for overlapping of computation with communication only as long as both the MPI implementation and the underlying hardware infrastructure support it, as well. That is, the MPI implementation should make a distinction between standard and non-blocking communication primitives, so as to exploit the benefits of the advanced communication pattern. On the other hand, the underlying hardware/network infrastructure must also support DMA-driven non-blocking communication. Unfortunately, this is not the case with the used MPICH implementation for ch_p4 ADI-2 device, as indicated by the relative performance of both schemes. In order to evaluate our proposed advanced scheduling scheme also in terms of communication-computation overlapping, we thus resorted to synchronous MPI communication primitives for the blocking scheme (e.g `MPI_Ssend` instead of `MPI_Send`). By doing so we were able to *simulate* the relative performance of both communication patterns, despite the implementation/hardware restrictions.

4. Experimental Results

In order to evaluate the performance benefits obtained by the proposed advanced scheduling scheme, we have conducted a series of experiments using micro-kernel benchmarks. More specifically, we have automatically parallelized the Gauss Successive Over-relaxation (SOR - [11]) and the Texture Smoothing Code (TSC - [12]) micro-kernel benchmarks, and we have experimentally verified the overall execution time for different tiling transformations, blocking and non-blocking communication schemes and various iteration spaces. Our platform is an 8-node dual-SMP cluster interconnected with FastEthernet. Each node has 2 Pentium III CPUs at 800 MHz, 128 MB of RAM and 256 KB of cache, and runs Linux with 2.4.20 kernel. We used g++ compiler version 2.95.4 with -O3 optimization level. Finally, we used MPI implementation MPICH v. 1.2.5, configured with the following options: `--with-device=ch_p4 --with-comm=shared`.

4.1. SOR

The SOR loop nest involves a computation of the form $A[t,i,j] = f(A[t,i-1,j], A[t,i,j-1], A[t-1,i+1,j], A[t-1,i,j+1], A[t-1,i,j])$, while the iteration space is $M \times N \times N$. The dependence matrix of the algorithm is $D = \begin{bmatrix} 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \end{bmatrix}$. Because of the negative elements of D, skewing needs to be applied to the original algorithm for the rectangular tiling to be valid. An appropriate skewing matrix is $T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 0 & 1 \end{bmatrix}$, since $TD \geq 0$, that is the skewed algorithm contains only non-negative dependencies. We will apply both rectangular and non-rectangular tiling to the skewed iteration space, and evaluate both the blocking and the non-blocking communication scheme. More specifically, the rectangular tile is provided by the matrix $P_r = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{bmatrix}$, while the proposed non-rectangular tiling transformation, as obtained from the algorithm's tiling cone, is described by the matrix $P_{nr} = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ x & 0 & z \end{bmatrix}$. Note that in each case, the tile shape can be determined from the column vectors of the respective transformation matrix (P_r or P_{nr}), while the tile size depends on the values of the integers x, y and z. However, both tiles have equal sizes, since $|P_r| = |P_{nr}| = xyz$, so that the per-tile computation volume is equal in both cases. Moreover, since in both cases tiles will be mapped to processors according to the third dimension, the per-tile communication volume and the number of MPI processes required are the same, as process mapping and inter-process communication are implicitly determined by the outermost two dimensions. Consequently, any differences in the overall execution times should be attributed to the different scheduling that results from the two tiling transformations, as well as to the communication pattern (blocking or non-blocking).

Experimental results for the SOR micro-kernel are depicted in Figure 4. In all cases, non-rectangular tiling outperforms the rectangular tiling, while the non-blocking communication pattern is more efficient than the blocking one, at least on a simulation level. In other words, the experimental results comply to the theoretically anticipated performance.

4.2. TSC

TSC algorithm can be written as a triply nested loop with a computation of the form $b[t,i,j] = f(b[t,i-1,j-1], b[t,i-1,j], b[t,i-1,j+1], b[t,i,j-1], b[t-1,i,j+1], b[t-1,i+1,j-1], b[t-1,i+1,j], b[t-1,i+1,j+1])$ (iteration space $T \times N \times N$). The dependence matrix of the algorithm

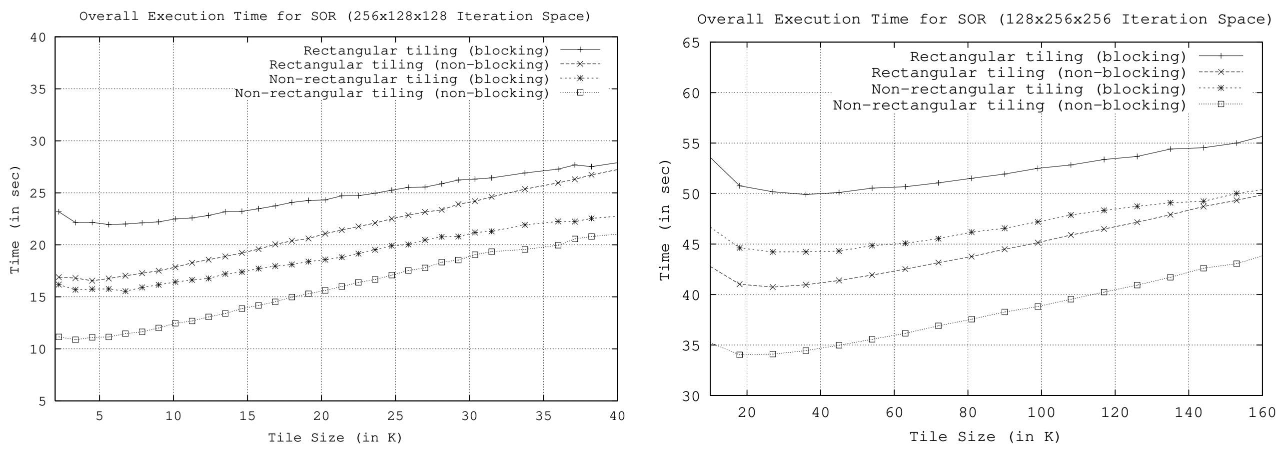

Figure 4. Experimental Results for SOR

is $D = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & -1 & -1 & -1 \\ 1 & 0 & -1 & 1 & -1 & 1 & 0 & -1 \end{bmatrix}$. Since D also contains negative elements, proper skewing needs to be applied for the rectangular tiling transformation to be valid. We consider the skewing matrix $T = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}$. We will apply tiling transformation $P_{nr} = \begin{bmatrix} x & 0 & 0 \\ -x & y & 0 \\ -x & -y & z \end{bmatrix}$ to the original iteration space, and rectangular tiling ($P_r = \begin{bmatrix} x & 0 & 0 \\ 0 & y & 0 \\ 0 & 0 & z \end{bmatrix}$) to the skewed iteration space.

As in the SOR micro-kernel benchmark, in both cases (rectangular and non-rectangular tiling transformation) we have an equal tile size ($|P_r| = |P_{nr}| = xyz$), that results to the same per-tile computation volume. Furthermore, tiles are mapped to MPI processes according to the third dimension.

Experimental results are depicted in Figure 5. We observe that, as in SOR, the non-blocking communication scheme with the application of non-rectangular tiling delivers the best overall performance. In this case however, both tiling transformations deliver similar performance under the blocking communication scheme, while non-rectangular tiling transformation is more beneficial in case of the non-blocking communication scheme.

5. Conclusions

Summarizing, we have combined the notions of arbitrary tiling transformation and overlapping communication and computation and incorporated these aspects into a complete framework to automatically generate parallel MPI code. We have addressed all issues regarding parallelization, such as task allocation, sweeping arbitrary shaped tiles and implementation of appropriate communication primitives. We have experimentally evaluated our work, and supplied simulation results for application-kernel benchmarks that verify the high performance gain obtained by the advanced scheduling scheme.

REFERENCES

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

[2] R. Andonov, P. Calland, S. Niar, S. Rajopadhye, N. Yanev, First Steps Towards Optimal Oblique Tile Sizing, in: 8th International Workshop on Compilers for Parallel Computers, Aussois, 2000, pp. 351–366.

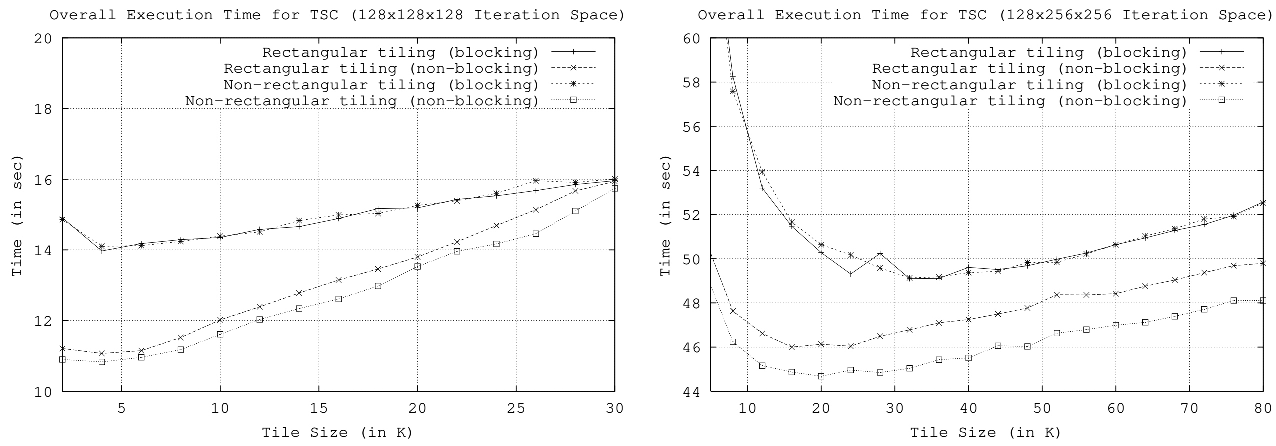

Figure 5. Experimental Results for TSC

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

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

[5] E. Hodzic, 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, 1999.

[6] K. Hogstedt, L. Carter, J. Ferrante, Selecting Tile Shape for Minimal Execution time, in: ACM Symposium on Parallel Algorithms and Architectures, 1999, pp. 201–211.

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

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

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

[10] G. Goumas, N. Drosinos, M. Athanasaki, N. Koziris, Compiling Tiled Iteration Spaces for Clusters, in: Proceedings of the IEEE International Conference on Cluster Computing, Chicago, 2002.

[11] G. E. Karniadakis, R. M. Kirby, Parallel Scientific Computing in C++ and MPI : A Seamless Approach to Parallel Algorithms and their Implementation, Cambridge University Press, 2002.

[12] S. Pande, T. Bali, A Computation + Communication Load Balanced Loop Partitioning Method for Distributed Memory Systems, Journal of Parallel and Distributed Computing 58 (3) (1999) 515–545.