Automatic Code Generation for Distributed Memory Architectures in the Polytope Model

Michael Claßen
FMI, University of Passau, Germany
Email: michael.classen@uni-passau.de

Martin Griebl
FMI, University of Passau, Germany
Email: martin.griebl@uni-passau.de

Abstract— **The polytope model has been used successfully as a tool for program analysis and transformation in the field of automatic loop parallelization. However, for the final step of automatic code generation, the generated code is either only usable on shared memory architectures or severely restricts the parallelization methods that can be applied. In this paper, we present a fully automated method for generating efficient target code, which is executable on clusters that are based on a distributed memory architecture. We also provide speedup results of experiments on a local cluster.**

I. Introduction

The polytope model has been valuable for model based program analysis in the field of automatic loop parallelization [15], [17], [18]. Methods have been developed to transform the original input program (in its polytope representation) into a target program (also in polytope representation) with minimal communication [3], [5], [8], [14], [17].

The final step however – generating executable code from the polytope representation – has only been fully implemented for shared memory architectures [4]. In this paper, we present an approach for automatically obtaining efficient target code for distributed memory architectures, i.e., we include the generation of the necessary communication code.

The paper is organized as follows. First, we give a very brief description of the polytope model. Then, we illustrate a first approach to determining the required communications and representing them in the model. Since it turns out that this first approach leads to inefficient communications, we refine the model-based description of the communications, before we generate code.

II. Model-based parallelization for distributed memory architectures

Let us start with a presentation of the mathematical model of a program.

A. Different instances of statements

This paper deals with parallelizing loop programs. Every iteration of a loop generates an individual instance of its body. Thus, we model a statement S by the set of all iterations that are enumerated by the surrounding loops of S.

Definition 1: The set of iterations that are enumerated by the loops surrounding a statement S is called the *index space* (ISPC) of S. Every vector $\vec{\imath}$ in the ISPC represents a single iteration of the surrounding loops. We call $\vec{\imath}$ the *iteration vector*.

Definition 2: Each instance of a statement is called an *operation*. It is fully described by the iteration vector $\vec{\imath}$ and the name of the statement S. We denote an operation o by the pair

$$o = \langle \vec{\imath}, S \rangle. \tag{1}$$

In the context of this paper, we deal only with loops whose bounds are affine linear expressions in indices of surrounding loops and in (symbolic) constants. Using some additional mathematical definitions, this allows for a more concrete description of index spaces.

Definition 3: In an n-dimensional vector space, a *hyperplane* is an $(n-1)$-dimensional subspace. It divides the vector space into two *half spaces*.

Definition 4: A *polyhedron* is the intersection of finitely many half spaces. A *polytope* is a bounded polyhedron. A polyhedron (polytope) can be defined by an affine inequality system, which we usually represent in matrix form:

$$M\vec{\imath} \leq \vec{c}, \tag{2}$$

where M is the coefficient matrix of the inequality system and $\vec{c}$ is the constant vector.

In our application, every loop surrounding a statement S corresponds to one dimension in the vector space, i.e., the index space, in the model. Every affine loop bound is modeled by an inequality. Thus, the n-dimensional index space of a statement S can be represented as m inequalities, using a $m \times n$ matrix B and a constant vector b of length m:

$$ISPC \ : B\vec{\imath} \leq \vec{b} \tag{3}$$

When we are not interested in the inequalities but only in the dimensions in which the polytope lies, we denote a polytope P by a sequence of sub-vectors which represent the names of the variables of the relevant dimensions:

$$P : \langle \vec{d_1}, \ldots, \vec{d_n} \rangle \tag{4}$$

B. Dependences

As the final step of modeling a loop nest, we must integrate the interactions of different operations into the model.

Definition 5: A *dependence* δ is defined by a common data access between two operations s and d. If s is executed before

d, we call s the *source* and d the *destination* of δ. The source and destination of a given dependence δ shall be given by functions $src(\delta)$ and $dst(\delta)$, respectively.

The set of all dependences Δ in the original program defines a partial order on the execution of the statements.

Definition 6: Four types of dependences are defined, depending on the type of memory access at the source and destination:

	source	
destination	reads	writes
reads	input	true
writes	anti	output

True dependences make communication of data necessary; anti and output dependences require only synchronization between processors. Input dependences are ignored in our context.

At this point, any relevant information of the original loop nest is represented in the model. Thus, we can now start parallelizing

C. Space-time mapping

The essence of parallelization is to map every operation to a logical time ($\vec{t}$) and space, i.e., processor coordinate ($\vec{p}$).

Definition 7: The (multi-dimensional) function θ that maps every operation to its logical execution time is called *schedule*. It needs to respect the partial order defined by the set D of all dependences in the input program:

$$\forall \delta \in \Delta : \theta(src(\delta)) \prec \theta(dst(\delta)), \tag{5}$$

where $\prec$ is the lexicographic order.

Definition 8: The (multi-dimensional) function π that maps every operation to its logical processor coordinate is called *placement*. This function has no correctness constraints, but influences heavily the number of communications.

These functions together define the parallel execution: all operations that are scheduled at a common time step t can be executed in parallel on the logical processors assigned by the placement function π.

In the polytope model, we restrict ourselves to (piecewise) affine functions for schedule and placement. Therefore, they can be written in matrix notation, either each individually or also both together.

Definition 9: The affine transformation that consists of the schedule and placement functions is called *space-time mapping*. It maps each operation to a logical time $\vec{t}$ and space coordinate $\vec{p}$:

$$STM : \begin{pmatrix} \vec{t} \\ \vec{p} \end{pmatrix} = M\vec{\imath}, \tag{6}$$

where the rows of M are composed of the rows of the matrices representing θ and π.

For each statement, we can now apply the space-time transformation to each iteration vector of its ISPC. The resulting set of vectors is called the *target index space* (TISPC).

D. A basic communication polytope

By applying the space-time transformation to the source and destination ISPC of each dependence δ, we obtain a dependence relation δ_{tr} in the TISPC. If the space coordinate of the source and the destination differ, a communication must be launched. Let us define a mathematical representation for these communications.

Definition 10: A *communication polytope* P_{comm} is constructed from δ_{tr} by concatenation of the time and space coordinates ($\vec{t}$, $\vec{p}$) of both, source and destination of the dependence relation:

$$P_{\mathrm{comm}} : \langle \vec{t}_{\mathrm{src}}, \vec{p}_{\mathrm{src}}, \vec{t}_{\mathrm{dst}}, \vec{p}_{\mathrm{dst}} \rangle \tag{7}$$

We can use this communication polytope to generate the required communication code. For each transformed true dependence δ, we construct two point-to-point communication statements, one for for sending and one for receiving the data element described by δ as follows:

- For the send statement, we project P_{comm} to $\langle \vec{t}_{\mathrm{src}}, \vec{p}_{\mathrm{src}} \rangle$, thus obtaining all index vectors in the target space for which a send operation has to be generated. Each send statement then has to send its data the logical space-time coordinates denoted by $(\vec{t}_{\mathrm{dst}}, \vec{p}_{\mathrm{dst}})$.
- For the receive statement, we just project P_{comm} to $\langle \vec{t}_{\mathrm{dst}}, \vec{p}_{\mathrm{dst}} \rangle$. Each receive statement then has to receive data that has been sent by send statements with the logical space-time coordinates $(\vec{t}_{\mathrm{src}}, \vec{p}_{\mathrm{src}})$.

At this point, we can automatically generate code for distributed memory architectures: we supply the polytopes modeling the space-time mapped computation statements and the send and receive statements to a tool that converts the polytope description to a loop nest, e.g., CLooG [4].

However, the target code obtained by application of the approach mentioned above is usually not efficient enough to produce a speed-up. The reason is the fine granularity of parallelism in the target program.

E. A refined communication polytope

In order to obtain coarser-grained parallelism, we apply tiling [1], [22]. We partition an n-dimensional space D into identical parallelepipeds (tiles), bounded by n families of hyperplanes. The coordinates of D are also called *virtual coordinates* from now on.

Since every tile has a lower and an upper bound in every dimension, a tile can be expressed by a $2n \times n$ matrix S, containing the normal vectors of the hyperplanes bounding the tile as rows, and a constant vector s of length $2n$:

$$S\vec{\imath} \leq \vec{s}, \tag{8}$$

where $\vec{\imath}$ is vector of D.

We use identical copies of the tile defined in (8) and shift them along the edges of the tile, i.e., the parallelepiped. This generates a partitioning of D. Thus, each point in D belongs to exactly one tile.

Every tile can be represented by a vector $\vec{\tau}$ in the *tile coordinate* system. For this purpose, let L be the matrix whose columns are the vectors of the shift just mentioned, i.e., the edges of the parallelepiped. L relates $\vec{\tau}$ with a point $\vec{\imath}_0 \in D$ as follows:

$$T \rightarrow D, L\vec{\tau} = \vec{\imath}_0 \tag{9}$$

Using this equation, every tile has a unique representative $\vec{\imath}_0$. All other points $\vec{\imath}$ inside the same tile have the property that their distance to $\vec{\imath}_0$ (often called the *offset*) lies within the tile described in (8). Combining this with (9), we obtain an inequality that expresses the relation between tile coordinates and virtual coordinates:

$$S(\vec{\imath} - L\vec{t}) \leq \vec{s} \tag{10}$$

Note that tiling doubles the dimensionality since the virtual coordinates are decomposed into tile coordinates and offsets.

In contrast to the traditional framework, we apply tiling to the transformed index space, i.e., after the application of the space-time mapping [12]. Thus, we can check separately the effect of tiling space and time dimensions.

- By applying tiling to spatial coordinates, communication is only performed between processor tile coordinates, thereby avoiding the overhead from unnecessary communication between the large number of virtual processors which are executed on the same real processor.
- Time tiling is used for aggregating virtual time steps into global time steps. During each global time step, data elements are not communicated immediately, but are aggregated in buffers, which are transmitted at the end of each global time step, thereby achieving message vectorization.
  Aside: it can be shown that only a single virtual time dimension can be tiled with a tile size greater than 1 and less than the maximal extent of the time dimension [11].

Let us discuss space and time tiling in more detail.

1) Tiling time dimensions: Since tiling time postpones the transmission of data, it also delays the receiver. This leads to a deadlock if two processors would like to exchange data between two virtual time steps: the sending is postponed, but none of the two processors can continue execution.

We can avoid this problem if we restrict ourselves to placements that satisfy an additional property.

Definition 11: A placement satisfies the *forward communications only* property (FCO), iff all communication vectors are component-wise non-negative in their spatial components, i.e.:

$$\forall \delta \in \Delta : dst(\delta) - src(\delta) \geq \vec{0}, \tag{11}$$

with Δ being the set of all true dependences, projected on their spatial dimensions.

If (11) also holds for anti and output dependences, we say that the placement satisfies the *strict FCO* property. Note that there exist placement algorithms that guarantee the (strict) FCO property [14].

If we have a placement satisfying the FCO property, we know that every communication targets a processor with a higher number than the source, thus avoiding communication cycles. We obtain a correct execution order of the tiles by delaying tiles executed on processors with high numbers longer than tiles executed on processors with small numbers [11]. More formally, we delay the execution of a given time tile t_{T} by the sum of entries in its processor tile coordinate ($\vec{p}_{\mathrm{T}}$), in order to get a modified time tile coordinate t'_{T}:

$$t'_{\mathrm{T}} := t_{\mathrm{T}} + \sum_{i=1}^{n} \vec{p}_{\mathrm{T}_i} \tag{12}$$

Note that restricting ourselves to placements that satisfy the strict FCO property allows our communication scheme to unpack all data elements from the receiver's buffer immediately after the corresponding communication, i.e., one global time step after the write access that lead to communication [13].

2) Tiling space dimensions: When tiling space dimensions, we can prevent unnecessary communication of data elements, if the relevant dependence exists only between virtual space coordinates that are belonging to the same processor tile coordinate. For this purpose, we add an additional constraint to our communication polytope. For each pair of source ($\vec{p}_{\mathrm{srcT}}$) and destination ($\vec{p}_{\mathrm{dstT}}$) processor tile coordinates, the following restriction must hold:

$$\vec{p}_{\mathrm{srcT}} \neq \vec{p}_{\mathrm{dstT}} \tag{13}$$

Unfortunately, the resulting index space is non-convex and can only be expressed by the union of polytopes for

$$\vec{p}_{\mathrm{srcT}} \prec \vec{p}_{\mathrm{dstT}} \tag{14}$$

and

$$\vec{p}_{\mathrm{dstT}} \prec \vec{p}_{\mathrm{srcT}} \tag{15}$$

(using the lexicographic order $\prec$ in all processor dimensions).

However, we can again exploit the FCO property and restrict the polytope representations to the case that Equation 14 holds.

3) The refined communication polytope: To summarize, tiling extends our communication polytope by additional tiling dimensions for time tile and processor tile coordinates for the source ($srcT$) and destination ($dstT$) of a transformed dependence relation (with skewing already applied):

$$P'_{\mathrm{comm}} : \langle t_{\mathrm{srcT}}, \vec{p}_{\mathrm{srcT}}, \vec{t}_{\mathrm{src}}, \vec{p}_{\mathrm{src}}, t_{\mathrm{dstT}}, \vec{p}_{\mathrm{dstT}}, \vec{t}_{\mathrm{dst}}, \vec{p}_{\mathrm{dst}} \rangle \tag{16}$$

F. Statement types for our communication scheme

For our refined communication scheme, we generate three new types of statements in addition to the transformed computation statements from the original input program. For every transformed dependence, two types of buffer management statements are generated. The third statement type is responsible for performing the communication.

1) Write-buffer statements: After each write access at the source of a true dependence, the computed data element is not sent directly by a point-to-point communication, but stored in a buffer by a write-buffer statement. This buffer holds all data destined for the corresponding destination processor.

2) Unpack-buffer statements: For each write-buffer statement, a corresponding unpack-buffer statement is generated, which is used to unpack data elements from the communicated buffers at the global time step immediately following the global time step of the write access.

For the generation of both buffer management statements, our extended communication polytope P'_{comm} from (16) can be used:

- To describe the source and destination processor of a given dependence, we now have to use the corresponding processor tile coordinates $\vec{p}_{\text{T}}$ instead of virtual processor coordinates $\vec{p}$ which we used in Section II-D.
- The polytopes for the write-buffer and the unpack-buffer statements contain both source and destination processor tile dimensions, but they differ in the ordering of these dimensions.
- It is important to use the same ordering of elements in the corresponding write and read buffers. For that purpose, we must use the same polytope description of logical space-time coordinates for both buffer management statements. It is irrelevant whether we use the coordinates of the source or the destination index space of a given dependence – we just have to use the same description for both statements.
- In our communication scheme, each unpacking operation of communicated data from a communication buffer is always performed at the global time step that follows the corresponding write operation (cf. Section II-E.1). Therefore, it is not required to enumerate all global time coordinates of the destination index space for each dependence. Instead, the global time coordinate for the unpack buffer statement is fully determined by the global time coordinate of the source index space of the corresponding dependence, to which one global time step is added.

The resulting enumeration order for write-buffer and unpack-buffer statements is given by projecting the communication polytope as follows:

$$P_{\text{write}} : \langle t_{\text{srcT}}, \vec{p}_{\text{srcT}}, \vec{t}_{\text{src}}, \vec{p}_{\text{src}}, \vec{p}_{\text{dstT}} \rangle$$

$$P_{\text{unpack}} : \langle t_{\text{srcT}} + 1, \vec{p}_{\text{dstT}}, \vec{t}_{\text{src}}, \vec{p}_{\text{src}}, \vec{p}_{\text{srcT}} \rangle$$

Note that by projecting away the global time dimension at the destination side of a given dependence relation δ, we prevent redundant communication of data. Consider the case that δ results from a single write access at the global time step t_n on processor p and multiple read accesses at the global time steps t_i (for $i \in \{n+1, \ldots\}$), all placed on processor q. Projecting away the global time dimension ($\vec{t}_{\text{dstT}}$) for the destination of δ yields a polytope description which enumerates only a single data element.

3) Communication statement: For each global time step that includes computation, a communication statement is generated, which performs the actual communication of the data elements stored in separate buffers for each destination processor.

For this communication statement, we re-use the polytope description of the transformed computation statements, where space-time mapping, tiling and skewing has already been applied. In order to enumerate all global time iterations that execute computations, we project on the global time coordinate:

$$P_{\text{comm}} : \langle t_{\text{T}} \rangle$$

Here, no processor tile coordinates are required, because the communication is performed by a collective operation that is executed on all real processors.

At this point we have polytope descriptions for all computation and communication-relevant statements of our refined communication scheme. The final step is to merge them correctly and to generate a program from them.

G. From the polytope representation to target code

For the generation of the target program's loop nest, all polytopes of the four different statement types have to be merged and scanned correctly. For this task, we use CLooG, an improved implementation by Bastoul [4] of Quilleré's algorithm [21].

To describe the execution domain for each statement in the target program, we use our polytope descriptions. However, we also have to take care of the scheduling of the four different statement types relative to each other:

1) For each virtual time step within a global time step, our refined communication scheme can lead to at most three different types of statements being executed in the following order:
    a) *Unpack-buffer* statement (reads the received value from the receive-buffer)
    b) *Compute* statement (computes the new value)
    c) *Write-buffer* statement (writes the new value to the send-buffer)
2) At the end of each global time step, the *communication* statement is executed on each processor.

We use additional constant scheduling dimensions that are inserted to guarantee the desired ordering [16]. CLooG provides a mechanism for inserting and reordering additional dimensions for the target loop nest. Affine functions (so-called *scatter functions*) are used to define equations between variables from the domain descriptions and the additionally inserted scatter dimensions. We also use these scatter functions to change the enumeration order of processor tiles for the write-buffer and unpack-buffer statement, allowing to use the same polytopes for the domain description of both statement types.

H. Post-processing

The generated loop nest is post-processed automatically, in order to insert if-statements that restrict the execution of processor tile iterations to processors with the corresponding *MPI* process number. If the number of available MPI processes is known at compile time, the tile sizes for processor tiles can be adjusted in order to obtain a mapping of each processor tile on exactly one MPI process. In this case, we know the number of processor tile coordinates in each dimension at compile time and can compute a one-dimensional MPI process number by an appropriate function `getRank`.

Inside the innermost loop for processor tiles $\vec{p}_T$, an if-statement is inserted, performing the mapping from $\vec{p}_T$ to a one-dimensional MPI process number. E.g., for $\vec{p}_T = (p_1, p_2)$, we obtain

```
if(getRank(p1,p2) == mpi_rank) {
  ... // execute inner loops
}
```

As an additional step in post-processing, the code for buffer management and performing the actual communication is inserted for the statement placeholders that are generated by CLooG within the loop bodies.

I. Mapping to real processors

If processor tiling is used for mapping the processor tile coordinates to one-dimensional MPI process numbers (as mentioned in Section II-H), the program has to be re-generated with adjusted tile sizes, if another number of available MPI processes is used.

In order to avoid this costly code generation, we use a dynamic approach for mapping to real processors:

- A map is created by enumerating all processor tile coordinates that contain computation statements and mapping them to a one-dimensional coordinate, using a cyclic or block distribution. For this purpose, the polytope representations of all computation statements are projected to their processor tile dimensions.
- The function `getRank` is replaced by a function `lookupRP`, that performs a lookup at run time in the generated processor map:

```
if(lookupRP(p1,p2) == mpi_rank) {
  ... // execute inner loops
}
```

Especially when using the cyclic distribution strategy, this approach yields very good load balance.

III. Experiments

A. Input program

For our experiments, we used one-dimensional successive over-relaxation (SOR) [10]:

```
DO K=1,M
  DO I=2,N-1
    A(I)=(A(I-1)+A(I+1))/2.0
  END DO
END DO
```

For the space-time transformation, the following schedule θ and placement π where chosen: $\theta(K, I) = 2 * K + I$ and $\pi(K, I) = K$. Tiling was applied to the resulting transformed target program, using a tile size of 40 and 400 for the time and processor dimension respectively. In this case, we only have one-dimensional time and processor dimensions, so we have to use a rectangular tile shape.

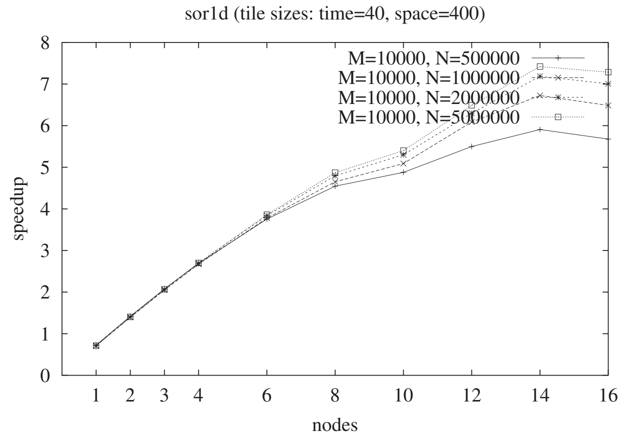

Fig. 1. Speedup

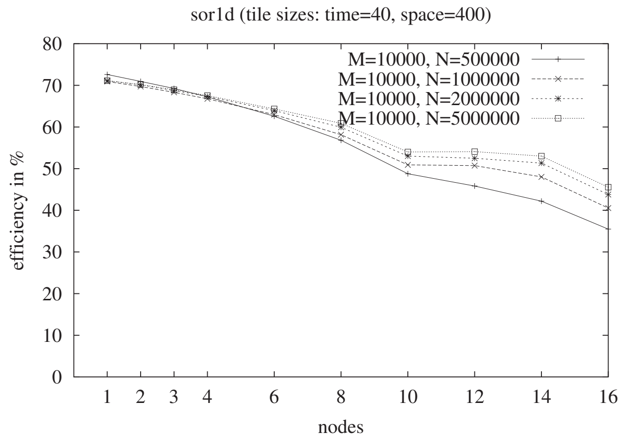

Fig. 2. Efficiency

B. Target architecture

We used a local cluster for our benchmarks that consists of 32 nodes of dual-Pentium III 1 GHz CPUs with 512 MB memory. These nodes are interconnected by an SCI network. Of the 32 nodes, we used between one and 16 nodes for our experiments, while using only one CPU core for each node.

C. Results

The resulting speedup is displayed in Figure 1 for different parameter settings. Figure 2 shows the corresponding efficiency. We chosen a larger number of iterations ($M := 10000$) for achieving more accurate timings.

Our experiments have shown that our refined communication scheme produces high speedups. For the problem sizes used in our examples, the generated code scales up to to 14 processors with quite good efficiency (between 51% and 71%).

The described code generation using CLooG for enumerating the target loop nest from the polytope descriptions usually yields very large code sizes for the target program (e.g. more than 12000 lines of code in the example used in our experiments). However, the overhead from buffer management, synchronization and the complexity of the loop bounds is reasonably low, as documented by the high efficiency in the case that the parallel target program is run on only one processor.

IV. Related work

There are other projects that also address automatic code generation for distributed memory architectures. Faber [6], [7] describes an approach that is also based on the polytope model. He generates *HPF* target code that includes annotations of how data is distributed among processors. The *HPF* compiler then generates the required communication from this distribution annotations. However, the application of this method is restricted because common *HPF* compilers often fail to generate the corresponding communication code for cases that require irregular communications. The code generation method described in this paper, by contrast, enables the generation of communication code for arbitrary affine communication patterns, as long as the applied placement satisfies the strict FCO property.

Ferner [9] also describes a method of automatically deriving communication code for distributed memory architectures. His approach is based on a parallelization technique of Lim and Lam [19] that generates so-called *partitions* of the index spaces of all statements in the input program. These partitions can be executed in parallel on logical processors. Ferner proposes a mapping algorithm that uses affine mappings from logical processors (partitions) to real processors and illustrates how the corresponding communication code loop nests can be obtained.

However, as opposed to the original parallelization technique described by Lim and Lam, he restricts the partitions to be one-dimensional, which limits the degree of parallelism obtained in the target program, but simplifies the code generation. He also restricts the target programs to asynchronous parallel programs, whereas Lim and Lam's method often results in synchronous parallel target programs that require an outer sequential loop for enumerating time steps.

Athanasaki et al. describe another approach that also uses tiling to reduce communication for distributed memory based clusters [2]. In their approach, an additional tiling transformation is used for aggregating processor tiles along certain hyperplanes into so-called *groups*, which can be executed efficiently by exploiting the availability of communication-free shared memory processors on each node in the cluster. They also use overlapping of computation and communication operations to achieve a form of pipeline parallelism. For the implementation of the communication statements, they make use of low level adaptations, e.g. using zero-copy network protocols [20] for SCI networks.

However, their approach is restricted to perfectly nested for-loops with uniform dependences. Also, the number of nodes and the number of shared memory processors per node have to be known at code generation time for this method, because both numbers are used statically for determining the tile and group sizes, respectively.

V. Conclusion

We proposed a method for automatic code generation for distributed memory architectures based on the polytope model. We illustrated a refined communication scheme, where tiling is used for aggregating data in buffers, which are transmitted at the end of global time steps. Thereby, message vectorization is achieved.

We also presented results of an experiment on a local cluster, which showed good speedup and efficiency, that also scales well for larger number of processors.

Acknowledgment

The authors would like to thank the German Research Foundation (DFG), the Procope project for supporting the research cooperation with our colleagues in France and also the CoreGRID Network of Excellence. We also would like to thank Cédric Bastoul for his great support with CLooG.

References

[1] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming (PPoPP'91).* ACM Press, 1991, pp. 39–50.

[2] M. Athanasaki, A. Sotiropoulos, G. Tsoukalas, N. Koziris, and P. Tsanakas, "Hyperplane Grouping and Pipelined Schedules: How to Execute Tiled Loops Fast on Clusters of SMPs," *The Journal of Supercomputing*, vol. 33, no. 3, pp. 197–226, Sep 2005.

[3] U. Banerjee, *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer Academic Publishers, 1992.

[4] C. Bastoul, "Efficient code generation for automatic parallelization and optimization," in *ISPDC'2 IEEE International Symposium on Parallel and Distributed Computing*, Ljubjana, Oct. 2003, pp. 23–30.

[5] A. Darte and F. Vivien, "Automatic parallelization based on multidimensional scheduling," 1994. [Online]. Available: citeseer.ist.psu.edu/darte94automatic.html

[6] P. Faber, *Transformation von Shared-Memory-Programmen zu Distributed-Memory-Programmen*. University of Passau, 1997, diploma thesis.

[7] P. Faber, M. Griebl, and C. Lengauer, "Issues of the automatic generation of HPF loop programs," in *13th Workshop on Languages and Compilers for Parallel Computing (LCPC 2000)*, ser. Lecture Notes in Computer Science 2017, S. P. Midkiff, J. E. Moreira, M. Gupta, S. Chatterjee, J. Ferrante, J. Prins, W. Pugh, and C.-W. Tseng, Eds. Springer-Verlag, 2001, pp. 359–362.

[8] P. Feautrier, "Some efficient solution to the affine scheduling problem, part I, one dimensional time," *Int. J. of Parallel Programming*, vol. 21, no. 5, pp. 313–348, Oct. 1992.

[9] C. Ferner, "Revisiting communication code generation algorithms for message-passing systems," *Int. J. of Parallel, Emergent and Distributed Systems*, submitted.

[10] C.-E. Fröberg, *Numerical Mathematics – Theory and Computer Applications*. Benjamin/Cummings, 1985.

[11] M. Griebl, "The mimimal number of communication startups when tiling space-time mapped programs," in *Ninth International Workshop on Compilers for Parallel Computers (CPC 2001)*, jun 2001, pp. 117–126.

[12] ——, "On tiling space-time mapped loop nests," in *Thirteenth annual ACM symposium on parallel algorithms and architectures (SPAA 2001)*, July 2001, pp. 322–323.

[13] ——, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004, habilitation thesis. [Online]. Available: http://www.uni-passau.de/~griebl/habilitation.html

[14] M. Griebl, P. Feautrier, and A. Größlinger, "Forward communication only placements and their use for parallel program construction," in *Languages and Compilers for Parallel Computing, 15th International Workshop, LCPC'02*, ser. Lecture Notes in Computer Science 2481. Springer-Verlag, 2002, to Appear.

[15] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, pp. 563–590, July 1967.

[16] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," Dept. of Computer Science, Univ. of Maryland, Tech. Rep. CS-TR-3193, Apr. 1993.

[17] L. Lamport, "The parallel execution of DO loops," *Comm. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[18] C. Lengauer, "Loop parallelization in the polytope model," in *CONCUR'93*, ser. Lecture Notes in Computer Science 715, E. Best, Ed. Springer-Verlag, 1993, pp. 398–416.

[19] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *Proceedings of the Twenty-fourth Annual ACM Symposium on the Principles of Programming Languages*, Paris, France, 1997. [Online]. Available: citeseer.csail.mit.edu/lim98maximizing.html

[20] F. O'Carroll, H. Tezuka, A. Hori, and Y. Ishikawa, "The design and implementation of zero copy mpi using commodity hardware with a high performance network," in *ICS '98: Proceedings of the 12th international conference on Supercomputing*. New York, NY, USA: ACM Press, 1998, pp. 243–250.

[21] F. Quilleré, S. Rajopadhye, and D. Wilde, "Generation of efficient nested loops from polyhedra," *International Journal of Parallel Programming*, vol. 28, no. 5, pp. 469–498, Oct. 2000.

[22] J. Xue, "Communication-minimal tiling of uniform dependence loops," *J. Parallel and Distributed Computing*, vol. 42, no. 1, pp. 42–59, Apr. 1997.