Computer Science
Technical Report

Colorado State University

Parametrically Tiled Distributed Memory Parallelization of Polyhedral Programs

Tomofumi Yuki Sanjay Rajopadhye

June 10, 2013

Colorado State University Technical Report CS13-105

Computer Science Department
Colorado State University
Fort Collins, CO 80523-1873

Phone: (970) 491-5792 Fax: (970) 491-2466
WWW: http://www.cs.colostate.edu

Abstract

We present a method for parallelizing polyhedral programs targeting distributed memory platforms. We use wave-front of tiles as the parallelization strategy, and uniformize affine dependences to localize communication to neighbor-to-neighbor send-recv, and also to enable parametric tiling.

We evaluate our approach on a collection of polyhedral kernels from the PolyBench suite, and show that our approach scales to as many as 96 cores, well beyond the shared memory limit, with performance comparable to PLuTo, a state-of-the-art shared memory automatic parallelizer using the polyhedral model.

1 Introduction

Parallel processing has been a topic of study in computer science for a number of decades. However, it is only in the past decade that coarse grained parallelism, such as loop level parallelism, become more than a small niche. Until the processor manufacturers encountered the "walls" of power and memory [6, 23, 51], the main source of increase in compute power came from increasing the processor frequency; through an easy ride on the Moore's Law. Now the trend has completely changed, and multi-core processors are the norm. Compute power of a single core has stayed flat, or even decreased for energy efficiency, in the past 10 years. Suddenly, parallel programming became a topic of high interest in order to provide continuous increase in performance for the mass.

One successful approach for automatically taking advantage of multi-core architectures is tiling [19, 50]. Tiling improves data locality, and exposes coarse grained parallelism, leading to efficient parallelizations. In particular, for programs that fit the polyhedral model, fixed size tiling is a very well studied problem [5, 19].

However, the performance of tiled code is hugely influenced by the choice of tile sizes [9, 26, 39, 42]. This led to the development of techniques for parameterized tiled code generation [17, 20].

These advances in automatic parallelization based on tiling are currently largely limited to shared memory programs. However, shared memory architectures cannot be expected to scale to large number of cores. One of the reasons being cache coherency; as a number of cores that share a memory increases, maintaining cache coherency becomes increasingly expensive. It is then a natural flow to target distributed memory code generation.

Distributed memory parallelization introduces a number of new problems. In addition to the partitioning of computation, we must address two additional problems: data partitioning and communication. These two problems are not visible for shared memory parallelization, since the shared memory conveniently hides these issues from the software layer. Moreover, the problem of tile size selection is even more critical in distributed memory parallelization. In addition to locality and load balancing concerns, which are also faced by shared memory parallelization, the volume and frequency of communication is also influenced by tile sizes.

In this paper, we present a code generation technique for efficiently parallelizing polyhedral applications using MPI. How we address the problems described above is summarized below:

- Computations are partitioned using wave-front of tiles. The partitioning of computation is identical to that used for shared memory by PLuTo [5]. However, unlike PLuTo or other related work [1, 4, 8, 25], we allow parametric tiling.

- Data Partitioning requires legal memory allocation for parametric tiling. We use extensions to Schedule-Independent Storage Mapping [44], but any other legal mappings can be used. The storage mapping we use is described in a separate article [54].

- Communications are greatly simplified by our assumption that the dependences crossing processor boundaries are uniform. We justify this restriction by showing most of `PolyBench` can be completely uniformized with simple techniques in Section 3.

  With uniform dependences, many questions related to communication; how to find processors that need to communicate, how to pack/unpack buffers, how to overlap communication with computation; can easily be answered.

Our work utilize polyhedral representations of programs and is limited to polyhedral programs.

The rest of this document is organized as follows. We first introduce necessary concepts from the polyhedral model, and the key transformation, tiling, in Section 2. In Section 3, we present a study of "uniform-ness" of polyhedral programs using `PolyBench` to motivate our assumption. We briefly describe the partitioning of computation and data in Section 4. Section 5 presents the core of our parallelization, communication placement and code generation. We evaluate our work against PLuTo, an automatic parallelizer using the same parallelization strategy for shared memory, in Section 6. We discuss other related work in Secion 7 and conclude in Section 8.

2 Background

In this section, we introduce necessary background to our paper. We first introduce necessary notations and concepts from the polyhedral model. We then review tiling, a well-known loop transformation, which we use to expose coarse grained parallelism.

2.1 The Polyhedral Model

The strength of the polyhedral model as a framework for program analysis and transformation are its mathematical foundations for two aspects that should be (but are often not) viewed separately: program *representation/transformation* and *analysis*. Feautrier [13] showed that a class of loop nests called Affine Control Loops (or Static Control Parts) can be represented in the polyhedral model. This allows compilers to extract regions of the program that are amenable to analyses and transformations in the polyhedral model, and to optimize these regions. Such code sections are often found in kernels of scientific programs, such as dense linear algebra, stencil computations, or dynamic programming. These computations are used in wide range of applications; climate modeling, engineering, signal/image processing, bio-informatics, and so on.

In the model, each instance of each statement in a program is represented as an *iteration point*, in a space called *iteration domain* of the statement. Each such point is hence, an *operation*. The iteration domain is described by a set of linear inequalities forming a convex polyhedron denoted as $\{z|\langle\text{constraints on } z\rangle\}$.

Dependences are modeled as pairs of affine function and domains, where the function represents the dependence between two iteration points, and the domain represents the set of points where the dependence exists. Affine functions are expressed as $(z \rightarrow z')$, where z' consists of affine expressions of z. Alternatively, dependences may be expressed as relations, sometimes called the dependence polyhedra, where functions and domains are merged into a single object. As a shorthand to write statement S depends on statement T, we also write $S[z] \rightarrow T[z']$.

Uniform and Affine Dependences A dependence is said to be uniform if it is a translation by some constant vector. In other words, it can be expressed as $f(x) = Ix + b$ in matrix form. Since the constant vectors are sufficient to characterize uniform dependences, they are referred as dependence vectors in this document.

Affine dependences are more general in that it can be any affine expression involving the indices. Its matrix form is more general and is expressed as $f(x) = Ax + b$.

2.2 Tiling

Tiling is a well known loop transformation for locality [19, 50], and extracting coarser grained parallelism [19, 38], by partitioning the iteration space to tiles (blocks) of computation, some of which may run in parallel

Another important notion related to tiling is the categorization of tiles into three types:

- Full Tile: All points in the tile are valid iterations.
- Partial Tile: Only a subset of the points in the tile are valid iterations.
- Empty Tile: No point in the tile is a valid iteration.

When a program is tiled, only a subset of its points are valid iterations. When all or none of the iterations of a tile are valid, it is called partial or empty tile respectively. To reduce control overhead, one of

the goals in tile loop generation is to avoid visiting empty tiles. Renganarayanan et al. [41] proposed what is called the *OutSet* that tightly over-approximates the set of non-empty tile origins, constructed as a syntactic manipulation of loop bounds. Similarly, they have presented *InSet* that exactly captures the set of full-tiles, tiles where all points are valid, origins.

2.3 Parametric Tiling

The choice of tile sizes significantly impacts performance [9, 26, 39, 42] (see Renganarayana's doctoral dissertation [40] for a comprehensive study of analytical models.)

However, analytical models are difficult to create, and can lose effectiveness due to various reasons, such as new architecture, new compilers, and so on. As an alternative method for predicting "good" tile sizes, more recent approaches employ some form of machine learning [36, 46, 53]. In these methods, machine learning is used to create models as platform evolves, and to avoid the need for creating analytical models to keep up with the evolution.

A complementary technique to the tile size selection problem is parameterization of tile sizes as run-time constants. If the tile sizes are run-time specified constants, code generation and compilation time can be avoided when exploring tile sizes. Tiling with fixed tile sizes; a parameter of the transformation that determines the size of tiles, can fit the polyhedral model. However, when tile sizes are parameterized, non-affine constraints are introduced and this falls out of the polyhedral formalism.

This led to the development of a series of techniques, beyond the polyhedral model, for parameterized tiled code generation [16, 17, 20, 21, 22, 41]. Initially, parameterized tiling was limited to perfectly nested loops and sequential execution of the tiles [22, 41]. These techniques were then extended to handle imperfectly nested loops [16, 21], and finally to parallel execution of the wave-front of tiles [17, 20].

DynTile [17] by Hartono et al., and D-Tiling [20] by Kim et al. are the current state-of-the-art of parameterized tiling for shared memory programs. These approaches both handle parameterized tiling of imperfectly nested loops, and its wave-front parallelization. Both of them manage the non-polyhedral nature of parameterized tiles by applying tiling as a syntactic manipulation of loops.

Our approach for distributed memory parallelization extends the ideas used in these approaches to handle parametric tiling with distributed memory.

2.4 Legality of Tiling

Legality of tiling is a well established concept defined over contiguous subsets of the schedule dimensions (in the RHS; scheduled space,) also called *bands* [5]. These dimensions of the schedules are tilable, and are also known to be fully permutable. We call a subset of dimensions in an iteration space to be tilable, if the identity schedule is tilable for the corresponding subset.

The RHS of the schedules given to statements in a program all refers to the common schedule space, and have the same number of dimensions. Among these dimensions, a dimension is tilable if all dependences are not violated (i.e., the producer is not scheduled after the consumer, but possibly be scheduled to the same time stamp,) with a one-dimensional schedule using only the dimension in question. Then any contiguous subset of such dimensions forms a legal tilable band.

2.5 Parametric Tiling for Shared Memory Parallelization

Our approach extends upon the state-of-the-art techniques for parameterized tiled code generation [17, 20]. The two techniques are nearly identical, and differs only by minor implementation details.

Since parametrically tiled programs do not fit the polyhedral model, current techniques *syntactically* modify the loop nests. The code generation works differently for sequential and parallel execution of the tiles. This is because parallel execution of tiles require *skewing* of the tiles, which can be represented as affine transformations if the tile sizes are fixed, that are no longer polyhedral transformations.

For sequential execution of the tiles, the input is a d-dimensional loop nest, and the output is two loop nests called *tile loops* and *point loops* that are d-dimensional each. Tile loops are loop nests that visit all the *tile origins*, the lexicographical minima of a tile. Points loops are loop nests, parameterized by tile origins,

```
//computation of wave-front time start/end
start = ...; end = ...;
//time loop enumerating wave-front hyper-planes
for (time = start; time <= end; time++)
 //loops to visit tile origins for a time step
 forall (ti1 = LB1; ti1 <= UB2; ti1+=ts1) {
   ...
    // compute last tile origin as a function of
    // current time step and other tile origins
    tid = getLastTI(time, ti1, ti2, ...)
    //guard against outset
    if (LBd <= tid && tid <= UBd)
      //point loops
      ...
 }
 #synchronization
```

Figure 1: Structure of loop nests for wave-front execution of parametric tiles [17, 20]. Bounds on `time`; `start` and `end` are computed using Equation 1. The last tile origin, ti_d is computed using ti_1 through ti_{d-1}, and ti_d is not a loop iterator. The function `getLastTI` computes ti_d using Equation 2. There is a guard to check if the computed ti_d is not an empty tile, and then the point loop. All loops up to ti_{d-1} can be executed in parallel.

that visit all points in a tile. The final output is simply the two loops composed; point loops as the body of tile loops.

For parallel execution, the tile loops are generated using a different algorithm. After applying tiling, the wave-front of tiles that are executed in parallel are usually those on a hyper-plane characterized by normal vector $\vec{\mathbf{1}} = [1, 1, ..., 1]$. The structure of the tile loops produced for parallel execution is illustrated in Figure 1.

This is necessary for parametric tiles, because parametric tiling falls outside the polyhedral model. The first iterator *time* corresponds to the global time step of wave-front execution, controlling which wave-front hyper-plane should be executed. Usually, wave-front hyper-planes characterized by normal vector $\vec{\mathbf{1}} = [1, 1, ..., 1]$ is used. This can be viewed as the hyper-plane with 45 degrees slope in all dimensions. This hyper-plane is always legal to be executed in parallel, and can be further optimized when some dimensions do not carry any dependences by replacing 1 with 0 for dimensions where no dependences are carried across tiles. In the following, we refer to the *time* as the wave-front time.

Given such a vector v, the wave-front time step for each tile can be computed as: $time = \sum_{k=1}^{d} v_k \left\lfloor \frac{ti_k}{ts_k} \right\rfloor$ where ti is the vector of tile origins, and ts is the vector of tile sizes. To simply our presentation, we assume that $v = \vec{\mathbf{1}}$, and use the following:

$$time = \sum_{k=1}^{d} \left\lfloor \frac{ti_k}{ts_k} \right\rfloor \tag{1}$$

The above formula will also give the lower and upper bounds of time, *start* and *end*, by substituting ti_k with its corresponding lower bound and upper bound respectively.

Similarly, if the *time* and ti_1 through ti_{d-1} is given, ti_d can be computed as:

$$ti_d = (time - \sum_{k=1}^{d-1} \frac{ti_k}{ts_k}) ts_d \tag{2}$$

Using Equations 1 and 2, loops that visit tile origins of all tiles to be executed in parallel, parameterized by *time*, are produced.

3 "Uniform-ness" of Affine Control Programs

In this section, we discuss the "uniform-ness" of affine control programs. The polyhedral model can handle affine dependences and most techniques for multi-core processors are formalized for programs with affine dependences. If the target architecture has shared memory, affine accesses to shared memory have little or no difference in cost compared to uniform accesses. However, affine dependences are significantly more complex, and as soon as we reach beyond shared memory, handling affine dependences again becomes a problem. When generating distributed memory parallel code, affine dependences can result in broadcast or broadcast to complicated subsets of the processors. The pattern of communication can be affected by the problem size and by tile sizes if tiling is used.

In hardware synthesis, a class of techniques called uniformization or localization, which replaces affine dependences with chains of uniform dependences, is a well studied topic as the length of communication directly impacts the area cost [7, 37, 43, 48, 52]. When parallelization was a small niche; before the rise of multi-cores, a class of techniques called uniformization or localization, which replaces affine dependences with uniform dependences was of great interest [7, 37, 43, 48]. There is also a related approach for converting affine dependences to uniform dependences that "fold" the iteration space using piece-wise affine transformations [32, 52].

When the target architecture is hardware, such as FPGAs or ASICs, implementing communications induced by affine dependences are very expensive. For example, communications among 1D processors over $\{i|0 \leq i \leq 10\}$, that corresponds to an affine dependence $(i \rightarrow 0)$ require all processors to be connected to processor 0. Instead, if we replace the affine dependence with a uniform dependence $(i \rightarrow i-1)$, and propagate the value, only neighboring processors need to be connected. The process of replacing an affine dependence with sequences of uniform dependences is called uniformization or localization, and has a significant impact on the performance of hardware, especially on the die area consumed by implementing communications. Since shared memory hides these issues from the software layer, techniques for uniformization are rarely used in recent work toward automatic parallelization.

In this section, we ask ourselves the question: "How affine are affine control programs?", and re-evaluate the affine-ness of affine dependences in realistic applications. We use `PolyBench/C 3.2` as our benchmark, which consists of 30 benchmarks, and show that most of `PolyBench` can be easily uniformized.

3.1 Uniformization

We restrict ourselves to a simple uniformization technique called nullspace pipelining [7, 37, 48]. Roychowdhury [43] has shown that all affine dependences can be uniformized, provided an extra dimension can be added to the iteration space. We only use pipelining since adding a dimension is often not desirable.

In addition to uniformization, we require a pre-processing step, called *embedding* or alignment, that transforms all statement domains to have the same number of dimensions. Since two statements must be in the same dimension for a dependence to even have a chance to be uniform, this pre-processing is essential.

We use simple heuristics similar to those used in data alignment to find legal embeddings [15, 27, 28]. The problem of automatically determining uniformizable or optimal embeddings is still open.

3.2 "Uniform-ness" of `PolyBench`

We evaluate the "uniform-ness" of `PolyBench` to by measuring how many of the benchmarks can be uniformized by applying the methods described above. The program is said to be uniform if all dependences, excluding input dependences, are. Table 1 shows if dependences in a benchmark is completely uniform in three stages:

- Uniform at Start: We perform constant propagation to inline constants and scalar variables in the polyhedral representation and check if it is uniform. We take the polyhedral representation extracted from C programs, and perform a form of constant propagation to inline constants and scalar variables, and check if it is uniform.
- Uniform after Embedding: The program after applying the embedding is analyzed.
- Uniform after Pipelining: The program after applying uniformization by pipelining is analyzed.

Benchmark	Uniform at Start	Uniform after Embedding	Uniform after Pipelining
correlation			
covariance			
2mm			✔
3mm			✔
atax			✔
bicg	✔	✔	✔
cholesky			?[2]
doitgen		✔	✔
gemm	✔	✔	✔
gemver			✔
gesummv		✔	✔
mvt	✔	✔	✔
symm			?[3]
syr2k	✔	✔	✔
syrk	✔	✔	✔
trisolv			✔
trmm			?[3]
durbin			
dynprog			?[4]
gramschmit			✔
lu			✔
ludcmp[1]			
floyd-warshall			✔
reg_detect		✔	✔
adi			?[5]
fdtd-2d		✔	✔
fdtd-apml		✔	✔
jacobi-1d-imper	✔	✔	✔
jacobi-2d-imper	✔	✔	✔
seidel-2d	✔	✔	✔

Table 1: Uniform-ness of `PolyBench/C 3.2` [33]. Excluding the four benchmarks with bugs and questionable implementations, 21 out of 25 (84%) can be fully uniformized.

[1] `ludcmp` is LU decomposition followed by forward substitution.
[2] `cholesky` is not uniformized due to a specific implementation choice. The computation is known to be uniformizable.
[3] `symm` and `trmm` do not correctly implement their BLAS equivalent. Correct implementations are uniform.
[4] `dynprog` is uniformizable, but only due to a bug in its implementation. Optimal String Parenthesization is not uniformizable.
[5] `adi` is uniformizable, but contains a bug. Correct implementation is still uniformizable.

We found 5 benchmarks with bugs or questionable implementations that make a computation known to be uniformizable not, or the other way around. Excluding these benchmarks, 21 out of 25 benchmarks are completely uniform after pipelining.

Of the remaining four, three benchmarks can technically be handled. These three benchmarks `correlation`, `covariance`, and `ludcmp` share a common property that some matrix is computed in the first phase of the kernel, and then used in the next phase with transpose-like dependences. Transpose-like dependences are dependences where the linear part is some permutation of the indices, and are not handled by pipelining. However, it is possible to decompose the program into phases such that the values computed in the former can be used as inputs to the latter. Then the problematic dependences become input dependences, and the program remains uniform. This is obviously not an ideal solution, as it significantly limits the design space.

This leaves `durbin` as the only remaining benchmark, which is known to be difficult to parallelize. In fact, PLuTo will not parallelize this kernel, and it can only tile one of the dimensions.

In conclusion, we found a significant portion of `PolyBench` to be uniformizable, albeit with heuristic heavy embedding, and one of the simplest techniques for uniformization. Although no general claims can be made from the study of `PolyBench`, we believe that 84% of `PolyBench` is significant.

3.3 Retaining Tilability after Pipelining

Let us first introduce the notion of *dependence cones*:

- f^* denotes the dependence cone; the smallest polyhedral cone of a dependence f, defined over domain $\mathcal{D}$, containing its range $f(\mathcal{D})$.
- A dependence cone f^* is said to be "pointed" if there exists a vector π such that $\forall x \in f^*, \pi \cdot x > 0$.
- Dependence cone for a set of dependence is the smallest polyhedral cone that contains the union of all ranges.

Van Dongen and Quinton [48] show that if the dependence cone formed by the set of dependences in the program is pointed, then the extremal rays of this cone can be used as uniformization vectors to ensure that the resulting program also has a pointed dependence cone. This simple result going back more than 25 years is, in hindsight, all we need to ensure tilability after uniformization.

If the dependences before uniformization is such that the program is tilable, then it is tilable after uniformization if the dependences are uniformized with non-negative uniformization vectors. Also following from the same assumption, the dependence cone for all dependences in the starting program is at most the entire non-negative orthant. Thus, it immediately follows that the extremal rays of this cone are non-negative vectors, and thus satisfying the condition.

In the original article, published before the invention of tiling, the dependence cone was used to ensure that the choice of uniformization vectors maintain "schedulability" of system of affine recurrences by affine schedules. We have shown that this result carries over to "tilability", and this is rather obvious in hindsight.

4 Partitioning of Computation and Data

We now present three necessary elements for distributed memory parallelization; computation partitioning, data partitioning, and communication.

The basic source of parallelism is the wave-front of the tiles. After tiling, the wave-fronts of the tiles are always legal to run in parallel. This approach has been shown effective in shared memory, by combining this parallelism with data locality improvements from tiling [5]. The polyhedral representation of loop programs is first transformed according to the PLuTo schedule (we use a variation in Integer Set Library by Verdoolaege [49]). Then we tile, and parallelize the wave-front of the tiles.

The code to implement the partitioning is slightly more complicated than annotating with OpenMP pragmas in the shared memory case. We implement a cyclic distribution of processors among the outermost parallel loop using the processor ID, `pid`, and the number of processors, `numP`, as follows:

```
for (ti1=LB1+pid*ts1; ti1<=UB1; ti+=ts1*numP)
```

The partitioning of computation is then followed by memory allocation. In our approach memory is allocated per "slices of tiles". A slice of a tile is the set of all tiles that share a common ti_1, the first tile origin. Since this is the unit of distribution of work among processors, separate memory is allocated for each slice. For simplifying the control structure of the generated code, we allocate memory in each processor assuming every tile is a full tile. Each processor allocates memory for the above slice, multiplied by the number of blocks assigned.

We also extend the slice by the longest dependence crossing processor boundaries to accommodate for values communicated from neighboring tiles. This corresponds to what are often called "halo" regions or ghost cells. The values received from other processors are first unpacked from the buffers to arrays allocated for tiles. Then the computation does not have to be special cased for tile boundaries, since even at the bounds, same array accesses can be used.

When the tile sizes are parameterized, most techniques for memory allocations [11, 29, 34, 47] cannot be used due to the non-affine nature of parameterized tiling. Since tile sizes are not known at compile time, it is essential that we find a memory allocation that is legal for all legal tile sizes. We have previously shown that such an allocation is possible for uniform dependence programs [54]. Our distributed memory code generation can work with any other memory allocation, provided that they are legal.

5 Communication

When we can assume all dependences that cross processor boundaries are uniform, communication is greatly simplified. We first assume that the length of uniform dependences in each dimension are strictly less than the tile size of each dimension. Then we are sure that the dependences are always to a neighboring tile, including diagonal neighbors.

5.1 Communicated Values

The values to be communicated are always values produced at the tile facets. The values produced at the tile facet touching another tile must be communicated if the two tiles are mapped to different processors. The facets that need to be communicated may be of a constant thickness if the length of dependence is greater than 1 in the dimension that crosses processor boundaries. We call this thickness the communication depth, as it corresponds to how deep into the neighboring tiles the dependences reach.

The depth is defined by the maximum length of a dependence in the first tiled dimension. For example, dependences $[-2, 0]$ and $[-2, -2]$ are both considered to have length 2 in the first dimension. The memory allocated are extended exactly by this depth, and the received values are stored in the "halo" region.

If the memory allocation is legal, then it follows that the values stored in the "halo" regions stay live until their use, even though some of the values may not be immediately used by the receiving tile. Moreover, with the possible exception at the iteration space boundaries, the values communicated are also guaranteed to be used in the next wave-front time step. This follows from the assumption that no dependences are longer than the tile size in each dimension. Therefore, we conclude that the volume of communication is optimal.

The participants of a communication can be identified by the following functions that take the vector of tile origins ti as an input:

$$\texttt{sender}(ti) = [ti_1 - ts_1, ti_2, \cdots, ti_d] \tag{3}$$

$$\texttt{receiver}(ti) = [ti_1 + ts_1, ti_2, \cdots, ti_d] \tag{4}$$

The above is the solution for one of the most complicated problems related to communication; finding the communicating partners. For programs with affine dependences, much more sophisticated analysis is required [4, 8, 24].

The packing and unpacking of buffers for communicating these variables are implemented as loops after the execution of a tile. The bounds of these loops, and the size of the buffer can be trivially computed, as they are always tile facets of some constant thickness. As an optimization, we allocate a single buffer for multiple statements, if the data types match.

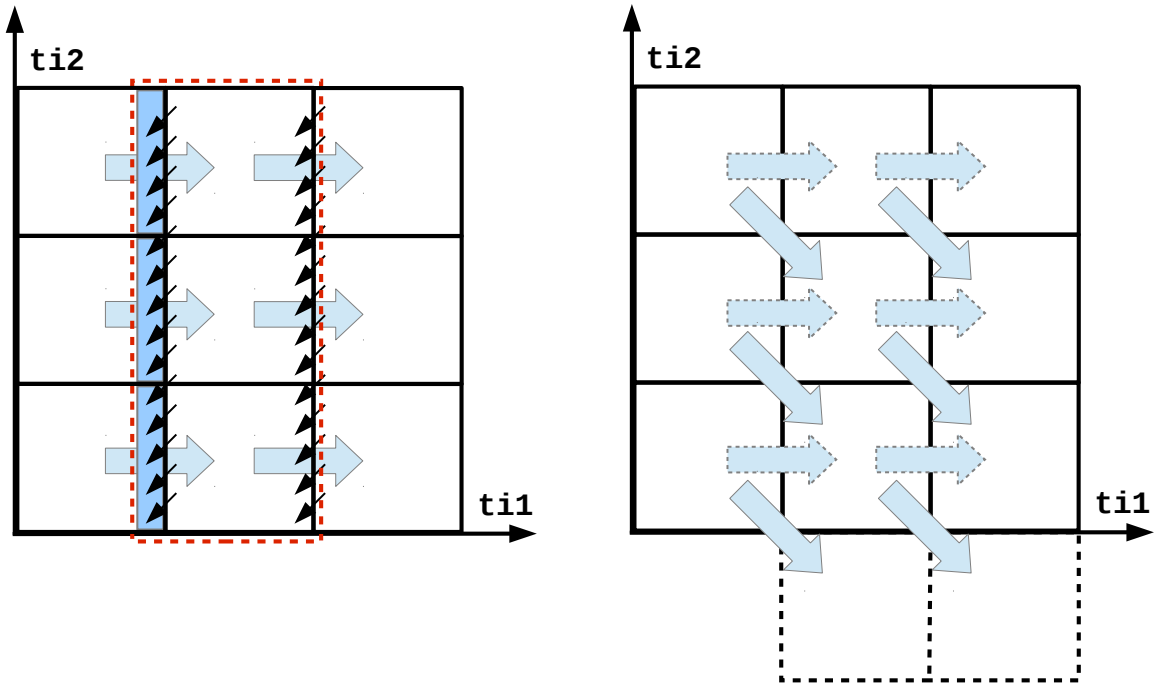
Figure 2a illustrates communication for 2D iteration space.

5.2 Need for Asynchronous Communication

Conceptually, we may place MPI send and receive calls at the end and the beginning of each tile. With the exception of those tiles at the initial boundary, a tile receives values from neighboring processors before the start of the computation. Similarly, after a tile is computed, values are sent to its neighbor, with the exception of those tiles at the terminating end.

MPI specifies that calls to `MPI_Send` blocks until the buffer for communication has been secured. In most cases, data is copied to system buffer and the control almost immediately returns. However, it should be assumed, for the sake of correctness, that `MPI_Send` is blocking until the corresponding `MPI_Recv` is called.

From the time and receiver functions (Equation 1 and 4), and also from Figure 2a, it is clear that the receiver tile is one wave-front time step later than the sender. This means that with naïve placement of MPI



(a) The shadowed column (blue) is the tile face being communicated. With greater depth, more columns will be included in the communication in 2D case. The dashed rectangle (red) denotes a slice for which memory is allocated.

(b) Illustration of when the values should be received. The horizontal arrows show the naïve placement, and the diagonal arrows show the optimized placement.

Figure 2: Illustration of communications for 2D iteration space with $[-1,-1]$ dependence. The big arrows show the communication, from sender to receiver. Note that the upper right corner of a tile, the top-most value of each face, is sent to the horizontal neighbor, but the dependence is from the diagonal neighbor. Legal memory allocation guarantees that this value is never overwritten until its use, and therefore it is safe to send early.

calls, the corresponding receive is not reached until all sends in a wave-front time complete. This clearly can cause a dead lock, when one of the calls to `MPI_Send` actually behaves as a blocking operation.

Therefore, in order to ensure correctness, we must manage buffers ourselves and use asynchronous versions of MPI functions. Even if no dead locks occur, notice that the number of communications "in flight" will increase as the number of tiles increase. The amount of buffer required for the naïve placement of communications corresponds to the maximum number of tiles in a wave-front time step. This is obviously wasteful, and should be avoided.

In addition, we may benefit from overlapping of communication with computation, sometimes called double buffering, by using asynchronous calls. Since data transfers in distributed memory environment typically have significant latency, it is a well-known optimization to hide the latency by computing another set of values as a data transfer is in progress (e.g., [10, 12, 45].) Tiling based parallelization can achieve this overlap if the number of parallel tiles is more than the number of physical processors. After a processor computes a tile, there are more tiles to be computed within the same wave-front time step, allowing overlap.

5.3 Placement of Communication

We now address the problem of where the asynchronous communications should be placed to ensure correctness, to reduce the number of in flight communications, and to overlap communication with computation. Let us ignore the problem of code generation for the moment, and assume that communication code can be inserted at the beginning or the end of any tile. We will deal with the problem of code generation separately

in Section 5.4.

Let us consider a send by a tile identified by its tile origin s, and its receiver, $r = s + [ts_1, 0, \cdots, 0] =$ `receiver`(s), where `receiver` is the receiver in the naïve placement as defined in Equation 4. Instead of this naïve placement, we propose to receive the data at $c = r - [0, \cdots, 0, ts_d]$, or in relation to the sender, $c = s + [ts_1, 0, \cdots, 0, -ts_d]$. The new communication pattern is illustrated in Figure 2b. Note that even though the tile that receives the values is now different, the memory locations where the received buffer is unpacked remain the same.

The tile c is the tile with the same tile origins up to $d-1$ as r, but one wave-front time earlier. Let $v[x : y]$ denote a sub-vector of v consisting of elements x through y. Since c and r only differ in the d-th dimension, $c[1 : d-1] = r[1 : d-1]$. Let the common part be denoted as a vector x, and r be a tile executed at time t, then we observe the following using Equation 2, $r_d =$ `getLastTI`(t, x) and $c_d =$ `getLastTI`$(t-1, x)$. Since r was one wave-front time later than s, it follows that c is in the same wave-front time as s.

Furthermore, due to the loop structure of parameterized tiling, and how computation is partitioned, we can say that if s is the n-th tile visited by a virtual processor p, then c is the n-th tile visited by the neighboring virtual processor $p+1$, provided that the receive happens also in empty-tiles at the boundaries. Recall that c and s only differ in the first and the last tile origins, and that the difference $s - c$ is $[ts_1, 0, \cdots, 0, -ts_d]$. The tile loops to iterate ti_2 through ti_{d-1} are common across all processors, and thus the same values are visited by all processors in the same order. Also recall that ti_d is uniquely computed by Equation 2. The values of ti_1 for p and $p+1$ differs by $ts1$, and if ts_2 through $ts_d - 1$ are equal, the tile visited by the two processors always differ by $[ts_1, 0, \cdots, 0 - ts_d]$, and therefore, the number of tiles visited by each virtual processor is always the same. The neighboring slices differ in the first tile origin by ts_1, and the last ti_d is uniquely computed by Equation 2 for each wave-front time step.

Thus, if the communication by s is to be received by c, even a single buffer is sufficient for correct execution. However, this leads to a globally sequentialized execution, unless a processor is only responsible for a single slice at each wave-front. This is because only the first (boundary) processor without any data to receive can proceed to send before receive, and all other processors must wait to receive data before sending. This wait on receive before send propagates to all processors, completely sequentializing the execution.

By increasing the number of buffers to 2 per processor, one for send and one fore receive, each processor can perform both send and receive simultaneously, allowing for parallel execution of tiles. In addition, when the program requires multiple datatypes to be communicated the number of required buffers are multiplied.

However, with 3D or higher iteration spaces, where a slice of tiles contains multiple tiles to be executed in a wave-front time step, 2 buffers are not sufficient for the last physical processor. This is because the send by the last physical processor is not received until the first processor moves to its next slice. Thus, only for the last processor, the size of the send buffer must be multiplied by the number of tiles executed by a slice in one wave-front time step.

Finally, we must also change the placement of receive within a tile to overlap communication with computation. Let us define two functions `prev` and `next` that respectively give the tile executed in previous and next wave-front time step for a given tile origin. These equations can be easily derived from Equation 1.

$$\texttt{prev}(ti) = ti - [0, \cdots, 0, ts_d] \tag{5}$$

$$\texttt{next}(ti) = ti + [0, \cdots, 0, ts_d] \tag{6}$$

The order of operations in a tile ti is as follows:

1. Issue receive (`MPI_Irecv`) from `sender`(`next`(ti)).
2. Perform computations for the current tile ti.
3. Wait (`MPI_Wait`) for the send of the previous tile to complete.
4. Send (`MPI_Isend`) outputs of ti to `prev`(`receiver`(ti)).
5. Wait (`MPI_Wait`) for receive to complete, and unpack buffer.

The key is to receive and unpack after the computation, instead of the beginning of the tile, as in the naïve approach. Also note that the send always has one tile worth of computation in between its issue and its corresponding wait. Thus, the communication is overlapped with a tile of computation.

5.3.1 Legality

The legality of the above placement can be easily established. The order in which tiles in a wave-front hyper-plane is executed should not matter since they are parallel. Therefore, although some of the parallel tiles may be executed sequentially in the final code due to resource limitations, we may think of virtual processors running at most one tile at each wave-front time step. Then the beginning of the $n+1$-th tile, and the end of n-th tile are equivalent in the final order within a virtual processor. The former case is that of the naïve placement, and the latter is our proposed placement.

5.3.2 Beyond Double Buffering

Increasing the number of buffers beyond two will only allow the receive (`MPI_Irecv`) to start early by making buffers available earlier. This does not necessarily translate to performance, if maximum communication and computation overlap is achieved with double buffering. Denoting the time to compute a tile as α, time to transfer outputs of a tile as β, we claim that double buffering sufficient if $\alpha \geq \beta$. When $\beta > \alpha$, one alternative to increasing the number of buffers is to increase α, i.e., increase the tile volume relative to its surface area. Since only the tile size corresponding to the communicated facet affects β, such balance can be controlled for tiled programs. Although changing the tile size in favor of such balance may influence locality, multi-level tiling can separate these concerns.

For tiled programs, α and β are both influenced by tile sizes. Since a facet of tiles are communicated, β will increases as tile sizes ts_2 through ts_d increases. ts_1 does not affect communication, since the outermost tile dimension is the dimension distributed among processors. On the other hand, increasing any tile size (ts) will increase the amount of computation per tile. Therefore, we expect that by appropriately selecting tile sizes, buffering factor greater than 2 is not necessary. It is also known that the improvement due to overlapping communication with computation is at most a factor of 2 [35].

Changing the tile size may alter other behaviors of the program, such as locality. However, the block cyclic distribution we use can control locality. Increasing the block size increases the number of tiles performed between communication are increased, and hence increases the amount of computation without affecting locality. Because block size does not affect the size of tile facet being communicated, the volume of communication stays unchanged. Hence, block sizes can be used to provide enough overlap with communication and computation without destroying locality. This does not change the legality of our approach, as multiplying ts_1 by block size gives equivalent behavior aside from the order of computation within a tile (or tile blocks.)

5.4 Code Generation

Now, let us discuss the code generation to realize the above placement of communications. Placement of the send is unchanged from the naïve placement, where values are sent after computing values of a tile. The placement of the receive seems more complicated, since $\texttt{prev}(ti)$ may be an empty tile that we would like to avoid visiting. Since the wave-front hyper-plane may be growing (or shrinking) in size as the wave-front time step proceeds, both $\texttt{prev}(ti)$ and $\texttt{next}(ti)$ have a chance of not being in the outset. However, we must visit $\texttt{prev}(ti)$ even if there is no computation to be performed just to receive data for next time step.

We take advantage of the structure of loop nests after parametric tiling (Recall Figure 1) to efficiently iterate the necessary tiles. The outermost d loops visit tile origins. The outermost loop iterates over time, and then the tile origins up to $d-1$-th dimension are visited. Then the final tile origin is computed using Equation 2. Furthermore, the check against the outset to avoid unnecessary point loops are after computing the last tile origin. Therefore, it is easy to visit some of the empty tiles by modifying the guards.

Let $\texttt{InOutSet}(ti)$ be a function that returns true if the tile origin, ti is within the outset. Using the above, and previously defined function `sender`, `receiver`, and `next`, operations in a tile are guarded by the following:

- Computation: $\texttt{InOutSet}(ti)$.
- Send: $\texttt{InOutSet}(ti) \wedge \texttt{InOutSet}(\texttt{receiver}(ti))$.
- Recv: $\texttt{InOutSet}(\texttt{next}(ti)) \wedge \texttt{InOutSet}(\texttt{sender}(\texttt{next}(ti)))$.

```
for (time = start; time <= end; time++)
 for(ti1=LB1+pid*ts1; ti1<=UB1; ti1+=ts1*numP){
  ...
   tid = getLastTI(time, ti1, ti2, ...)
   //issue receive for the next tile
   tidNext = getLastTI(time+1, ti1, ti2, ...)
   if(SenderInOutSet(ti1, ti2, ..., tidNext)
         && InOutSet(ti1, ti2, ..., tidNext)) {
     MPI_Irecv(&recvBuffer, ..., &recvReq];
   }
   //compute if in outset
   if(InOutSet(ti1, ti2, ..., tid)) {
    //point loops
    ...
    //send values
    if(ReceiverInOutSet(ti1, ti2, ..., tid)) {
      MPI_Wait(&sendReq);
      //copy outputs to buffer
      ...
      MPI_Isend(&sendBuffer, ..., &sendReq);
    }
   }
   //values should have arrived while computing
   if(SenderInOutSet(ti1, ti2, ..., tidNext)
         && InOutSet(ti1, ti2, ..., tidNext)) {
    MPI_Wait(&recvReq);
    //copy received values to local memory
    ...
   }
 }
```

Figure 3: Structure of generated code. The function `getLastTI` is called for *time* and *time* + 1 to find `next`(ti). Then the issue of receive, computation, sends, and completion of receive is performed within the corresponding guards. Variables `recvReq` and `sendReq` are handles given by asynchronous communication calls that are later waited on.

Note that for two tiles ti, and ti', the following equations hold:

$$ti = \texttt{sender}(\texttt{next}(ti'))$$

$$\texttt{receiver}(ti) = \texttt{next}(ti')$$

Thus, every send has a corresponding receive.

If a tile ti is in the OutSet, tile origins $[ti_1, ti_2, ..., ti_{d-1}]$ are visited *every* wave-front time step. When the ti_d is not in the outset for the current wave-front, the point-loops are skipped by the guard on OutSet. Therefore, if ti requires data at time t, then it is guaranteed that $[ti_1, ti_2, ..., ti_{d-1}]$ is visited at $t-1$ (or any other time step). Thus, we can check if `next`(ti) requires data in the next time step, and receive even if the tile is not in the OutSet. The tile origins for `next`(ti) can be computed by computing ti_d with $time+1$, and it does not matter if ti is not in the outset if the code is inserted outside the guard on OutSet for ti.

Figure 3 illustrates the structure of code we generate using the above strategy.

6 Evaluation

We evaluate our approach by applying our code generator to `PolyBench/C 3.2` [33]. Our evaluation is two-fold, we first show that despite the constraint on uniform dependences in one of the dimensions, most of

PolyBench can be handled by our approach.

We then compare the performance of our code with shared memory parallelization using PLuTo, but we expect to scale well beyond the number of cores on a single shared memory node. Our goal is to show comparable performance to PLuTo. We do not expect our code to scale any better than PLuTo as we use the same parallelization strategy based on tiling.

6.1 Applicability to PolyBench

Out of the 30 benchmarks in PolyBench, 23 satisfies the condition that at least one dimension of its tilable space is uniform. Among the remaining 7, 4 can be handled by splitting the kernel into multiple phases as discussed in Section 3. These benchmarks are correlation, covariance, 3mm, and ludcmp. Although phase detection of these kernels is trivial, the general problem is difficult, and we do not address this problem. We handle these benchmarks by manually splitting into phases, and then combining the codes generated individually.

The three benchmarks that cannot be handled are the following:

- cholesky in PolyBench cannot be uniformized. Cholesky decomposition can be handled if coded differently.
- durbin, and gramschmidt do not have large enough tilable bands to benefit from tiling based parallelism. These two kernels are not parallelized by PLuTo.

Even though we require uniform dependence on a dimension, we can handle virtually all of the PolyBench, since those that we cannot handle are not tilable either. We also note that all of the 23 benchmarks, as well as the different phases of the 4 that require phase detection, are all fully uniform after our uniformization step.

6.2 Performance Evaluation

We evaluate our approach on a subset of kernels in PolyBench. We exclude a number of kernels for the following reasons:

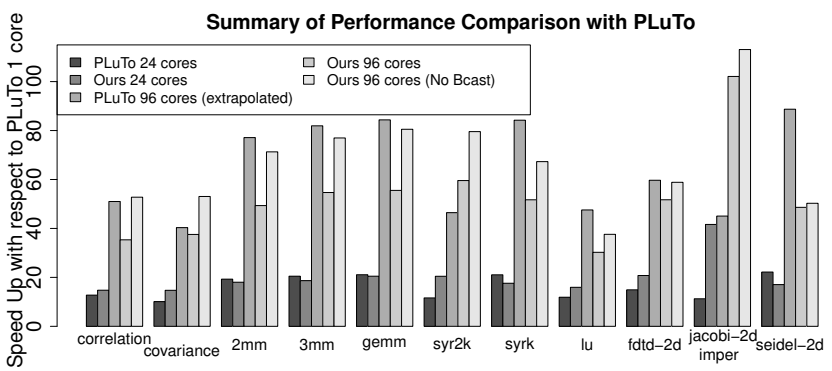
- trmm, dynprog, and adi have bugs and do not correctly implement their computation.
- doitgen, reg_detect, and fdtd-apml use single assignment memory, and thus no meaningful comparison can be made with other tools that retain original memory allocation.
- symm, and ludcmp use scalars for accumulation preventing PLuTo from parallelizing the computation.
- atax, bicg, gemver, gesummv, mvt, trisolv, and jacobi-1d-imper are small computations (only 2D loop,) and runs only for a few seconds sequentially.
- floyd-warshall has very limited parallelism, and shows poor parallel performance (about 2x) with 24 cores when parallelized for shared memory via PLuTo.

This leaves us with the following benchmarks: correlation, covariance, 2mm, 3mm, gemm, syr2k, syrk, lu, fdtd-2d, jacobi-2d-imper, and seidel-2d.

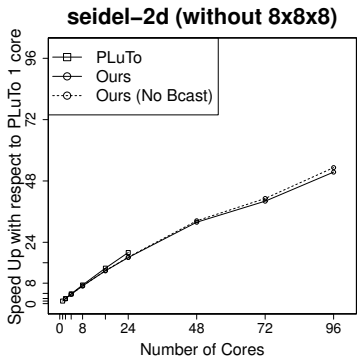
We measured the performance using Cray XT6m. A node in the Cray XT6m has two 12 core Opteron processors, and 32GB of memory. We used CrayCC/5.04 with -O3 option on the Cray. PLuTo was used with options --tile --parallel --noprevector. The problem sizes were chosen such that PLuTo parallelized code with 24 cores run for around 20 seconds.

The tile sizes were selected through non-exhaustive search guided by educated guesses. The tile sizes explored for PLuTo (and for our code up to 24 cores) were limited to square/cubic tiles, where the same tile size is used in each dimension. For our code executing on larger core counts, tile sizes were more carefully chosen. In most benchmarks, we found that the best tile sizes are different depending on the number of cores used. Note that PLuTo can only generate tiled code with fixed tile sizes.

The comparison is summarized in Figure 4a. The results show that the MPI code produced by our code generator show comparable scaling as the shared memory parallelization by PLuTo. We require that the cost



(a) Summary of performance of our code generator in comparison with PLuTo.



(b) Performance of `seidel-2d` excluding $8 \times 8 \times 8$ tiles.

Figure 4: Numbers for PLuTo with 96 cores are extrapolated (multiplied by 4) from the speed up with 24 cores. Ours (No Bcast) are number with the time to broadcast inputs removed. Broadcast of inputs takes no more than 2.5 seconds, but have strong impact on performance for the problem sizes used. This is a manifestation of the well known Amdahl's Law, and is irrelevant to weak scaling that we focus on. With the cost of broadcast removed, our code generator matches or exceeds the scaling of shared memory parallelization by PLuTo.

of initial broadcast to be removed for comparable scaling, which is not necessary when much larger problem sizes are used. Some of the kernels with large differences are explained in more detail in the following.

We perform better than PLuTo in `syr2k` due to a difference in the fusing decisions made by PLuTo and ISL. PLuTo does not fuse the loop nests, leading to two parallel loops.

In `jacobi-2d-imper`, values computed a time step is explicitly copied to another array before the next time step. Jacobi stencil in `PolyBench` uses explicit copying over other methods (e.g., pointer swaps) so that the computation can be analyzed via polyhedral analysis. UOV-based allocation after PLuTo scheduling allocates a scalar for a statement that is immediately copied to an array allocated for the other statement. Thus, our code uses only one array as opposed to two used by shared memory parallelization using PLuTo.

We are significantly outperformed by PLuTo with `seidel-2d`. This is due to a combination of multiple factors. The best tile size we found for PLuTo in this benchmark was $8 \times 8 \times 8$. This is a surprisingly small tile considering the cache size and memory foot print. We found that the Cray compiler uses optimization that are otherwise unused when the tile sizes are small, compile time constant for this code. For instance, turning off loop unrolling has no influence on other tile sizes, but slows down the 8 cubic tiled program by more than 15%. We do still have linear scaling, and the scaling is very similar to PLuTo once small tile sizes that enable additional compiler optimizations are removed from the data set, as illustrated in Figure 4b.

In addition to the lost compiler optimization due to parametric tiling, the dependence patterns in Gauss Seidel stencil computation negatively impact the performance of our code. The Gauss Seidel in `PolyBench` does not update boundary values of the 2D data array, and only updates the interior points at each time step. Therefore, the program has only one statement, and boundary cases are all handled implicitly through loop bounds and memory. However, when memory based dependences are ignored, a number of boundary conditions arise as different statements. The number of statements in our code reaches 14, where 13 of them are boundary cases, and comes with non-negligible control overhead.

7 Related Work

Although parallelism was not commonly exploited until the rise of multi-core architectures, it was used in the High Performance Computing field much before multi-cores. In HPC, more computational power is always demanded, and in many of the applications, such as simulating the climate, ample parallelism exists. As a natural consequence, distributed memory parallelization has been a topic of study for a number of decades.

When programming for distributed memory architectures, a number of problems that were not encountered in shared memory cases must be addressed. The two key issues that were not encountered in shared memory parallelization are *data partitioning* and *communication*.

Data Partitioning: When generating distributed memory programs starting from sequential programs, memory re-allocation must be re-considered. With shared memory, the same memory allocation was legal and efficient. However, with distributed memory, reusing the same memory allocation as the original program on all nodes multiplies the memory consumed by the number of nodes involved. Thus, it is necessary to re-allocate memory such that the total memory consumed is comparable to the original usage to provide scalable performance.

Communication: Since data are now local to each node, communication becomes necessary unless the computations are completely independent. In shared memory, communication is all taken care by the hardware or the run-time system. The only thing that is visible at the software are synchronization points, indicating points at which values written by a processor become available to others.

Note the above two problems, and also the partitioning of computation, which also arise in shared memory, are inter-related. The choice of data/computation partitioning can change what values are communicated and vice versa.

We distinguish our work from others in the following aspects:

- We apply uniformization and tiling, and make use of its properties to simplify and optimize communication.
- We support parametric tiling. None of the existing approaches handle parametric tiling for distributed memory parallelization.
- We explicitly manage re-allocation of memory. None of the existing polyhedral parallelizers for distributed memory even mention data partitioning. Instead, they use the same memory allocation as the original sequential program on all nodes.
- In contrast to non-polyhedral approaches, we use the polyhedral machinery to:
  - apply loop transformations to expose coarse grained parallelism,
  - apply tiling, not performed by most approaches, and
  - in contrast to those that perform tiling, we handle imperfectly nested affine loops.
- We require at least one dimension to be uniform, or can be made uniform. This restriction
  - does not prevent us from handling most of `PolyBench` [33], and
  - simplifies communication and enables optimization of buffer usage, as well as overlap of communication with computation.

7.1 Tiling-Based Approaches

There are few approaches that use or handle tiling based parallelization. Tiling is a critical transformation for efficient execution of scientific applications. Some are based on polyhedral analysis [1, 4, 8]. Bondhugula [4] present an approach using polyhedral analysis that builds on previous ideas using the polyhedral formalism to compute values to be communicated. The proposed approach is more general than ours in that it handles arbitrarily affine dependences. However, the tile sizes must be compile-time constants.

Goumas et al. [14] proposed a system for generating distributed memory parallel code that also apply tiling. Their approach is limited to perfectly nested loops with uniform dependences. They use non-rectangular, non-parameterized, tiling instead of skewing followed by rectangular tiling.

7.2 Loop Parallelism Approaches

There are a number of work that either detect parallelizable loops, or start with shared memory parallelization with such information. We first tile the iteration space, and use specific properties from tiled iteration spaces in our distributed memory parallelization.

The Paradigm compiler by Banerjee et al. [3] is a system for distributed memory parallelization. For regular programs, they apply static analysis to detect and parallelizable loops, and then insert necessary communications.

Li and Chen [30] make a case that once computation and data partitioning is done, it is not difficult to insert communications that correctly parallelize the program in distributed memory. They focus on finding reference patterns that can be implemented as aggregated communications. Similarly, Kwon et al. [25] present an approach for translating OpenMP programs to MPI programs. They handle a subset of affine loop nests where the set of values communicated do not change depending on the values of loop iterators surrounding the communication.

Pandore [2] is a compiler that take HPF(-like) programs as inputs, and produces distributed memory parallel code. Pandore uses a combination of static analysis and a run-time to efficiently manage pages of distributed arrays. Instead of finding out which values should be communicated as a block, the communication is always at the granularity of pages. Similarly, data partitioning is achieved by not allocating memory for pages not accessed by a node.

As part of the `dHPF` compiler developed for High Performance Fortran [18], Mellor-Crummey et al. [31] use analysis on integer sets to optimize computation partitioning. In HPF, data partitioning is provided by the programmer, and it is the job of the compiler to find efficient parallelization based on the data partitioning. Although their approach is not completely automatic, they are capable of handling complex data and computation decompositions such as replicating computations.

8 Conclusions and Future Work

We have presented our approach for generating distributed memory parallelization. The key idea in our approach is to only allow uniform dependences to cross processor boundaries. Although this may seem restrictive, effectively all benchmarks in `PolyBench` that is amenable to tiling can be handled. Because of many simplifications that can be made thanks to uniform dependences, communication becomes extremely simple, and resulting codes have similar scaling as shared memory parallelization.

Direct extensions of our work include use of multi-level tiling to support hybrid parallelization, and improvement of uniformization such that only a dimension is uniformized, rather than trying to completely uniformize.

We also feel that it is important to look for other classes of dependences between affine and uniform, that are more expressive than uniform, but easier to analyze than affine.

References

[1] Saman P. Amarasinghe and Monica S. Lam. “Communication optimization and code generation for distributed memory machines”. In: *Proceedings of the 14th ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI ’93. 1993, pp. 126–138.

[2] Françoise André et al. “The Pandore data-parallel compiler and its portable runtime”. In: *High-Performance Computing and Networking*. Ed. by Bob Hertzberger and Giuseppe Serazzi. Vol. 919. Lecture Notes in Computer Science. 1995, pp. 176–183.

[3] P. Banerjee et al. “The Paradigm compiler for distributed-memory multicomputers”. In: *Computer* 28.10 (1995), pp. 37–47.

[4] Uday Bondhugula. *Automatic Distributed-Memory Parallelization and Code Generation using the Polyhedral Framework*. Tech. rep. IISc Research Report, IISc-CSA-TR-2011-3, 2011.

[5] Uday Bondhugula et al. “A practical automatic polyhedral parallelizer and locality optimizer”. In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. 2008, pp. 101–113.

[6] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. “Low-power CMOS digital design”. In: *IEICE Transactions on Electronics* 75.4 (1992), pp. 371–382.

[7] Z. Chen and W. Shang. “On uniformization of affine dependence algorithms”. In: *Proceedings of the 4th IEEE Symposium on Parallel and Distributed Processing*. 1992, pp. 128–137.

[8] M. Claßen and M. Griebl. “Automatic code generation for distributed memory Architectures in the polytope model”. In: *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*. IPDPS ’06. 2006, p. 243.

[9] S. Coleman and K.S. McKinley. “Tile size selection using cache organization and data layout”. In: *Proceedings of the ACM SIGPLAN 1995 Conference on Programming language Design and Implementation*. PLDI ’95. 1995, pp. 279–290.

[10] D. Culler et al. *LogP: Towards a realistic model of parallel computation*. Vol. 28. 7. 1993.

[11] Alain Darte, Robert Schreiber, and Gilles Villard. “Lattice-Based Memory Allocation”. In: *IEEE Transactions on Computers* 54.10 (2005), pp. 1242–1257.

[12] F. Desprez. “Procdures de base pour le calcul scientifique sur machines parallles mmoire distribue”. LIP ENS-Lyon. PhD thesis. Institut National Polytechnique de Grenoble, 1994.

[13] Paul Feautrier. “Dataflow Analysis of Array and Scalar references”. In: *International Journal of Parallel Programming* 20.1 (1991), pp. 23–53.

[14] G. Goumas et al. “Message-passing code generation for non-rectangular tiling transformations”. In: *Parallel Computing* 32.10 (2006), pp. 711–732.

[15] Manish Gupta and Prithviraj Banerjee. “Demonstration of Automatic Data Partitioning Techniques for Parallelizing Compilers on Multicomputers”. In: *IEEE Transactions on Parallel and Distributed Systems* 3 (1992), pp. 179–193.

[16] Albert Hartono et al. “Parametric multi-level tiling of imperfectly nested loops”. In: *Proceedings of the 23rd International Conference on Supercomputing*. ICS ’09. 2009, pp. 147–157.

[17] Albert Hartono et al. “DynTile: Parametric tiled loop generation for parallel execution on multicore processors”. In: *Proceedings of the 24th IEEE International Symposium on Parallel and Distributed Processing*. 2010, pp. 1–12.

[18] High Performance Fortran Forum. “High Performance Fortran Language Specification”. In: (1993).

[19] F. Irigoin and R. Triolet. “Supernode partitioning”. In: *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. PoPL ’88. 1988, pp. 319–329.

[20] DaeGon Kim. “Parameterized and Multi-level Tiled Loop Generation”. PhD thesis. Fort Collins, CO, USA: Colorado State University, 2010.

[21] DaeGon Kim and Sanjay Rajopadhye. “Efficient Tiled Loop Generation: D-tiling”. In: *The 22nd International Workshop on Languages and Compilers for Parallel Computing*. LCPC ’09. 2009.

[22] D.G. Kim et al. “Multi-level tiling: M for the price of one”. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC ’07. 2007, p. 51.

[23] N.S. Kim et al. “Leakage current: Moore’s law meets static power”. In: *Computer* 36.12 (2003), pp. 68–75.

[24] D. Kranz et al. *Integrating message-passing and shared-memory: early experience*. Vol. 28. 7. 1993.

[25] O. Kwon et al. “Automatic Scaling of OpenMP Beyond Shared Memory”. In: *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing*. LCPC ’11. 2011.

[26] M.D. Lam, E.E. Rothberg, and M.E. Wolf. “The cache performance and optimizations of blocked algorithms”. In: vol. 25. PLDI ’91. 1991, pp. 63–74.

[27] Peizong Lee. "Efficient Algorithms for Data Distribution on Distributed Memory Parallel Computers". In: *IEEE Transactions on Parallel and Distributed Systems* 8 (1997), pp. 825–839.

[28] Peizong Lee and Zvi M. Kedem. "Automatic Data and Computation Decomposition on Distributed Memory Parallel Computers". In: *ACM Transaction on Programming Languages and Systems* 24 (1999), p. 20.

[29] V. Lefebvre and P. Feautrier. "Automatic storage management for parallel programs". In: *Parallel Computing* 24.3-4 (1998), pp. 649–671.

[30] J. Li and M. Chen. "Generating explicit communication from shared-memory program references". In: *Proceedings of the 1990 ACM/IEEE Conference on Supercomputing*. SC '90. 1990, pp. 865–876.

[31] J. Mellor-Crummey et al. "Advanced Optimization Strategies in the Rice dHPF Compiler". In: *Concurrency and Computation: Practice and Experience* 14.8-9 (2002), pp. 741–767.

[32] N. Osheim et al. "Smashing: Folding Space to Tile through Time". In: *Proceedings of the 21th International Workshop on Languages and Compilers for Parallel Computing*. LCPC '07. 2008, pp. 80–93.

[33] Louis-Noël Pouchet. *PolyBench*. `www.cs.ucla.edu/~pouchet/software/polybench/`.

[34] Fabien Quilleré and Sanjay Rajopadhye. "Optimizing memory usage in the polyhedral model". In: *ACM Transactions on Programming Languages and Systems* 22.5 (2000), pp. 773–815.

[35] M.J. Quinn and P.J. Hatcher. "On the utility of communication-computation overlap in data-parallel programs". In: *Journal of Parallel and Distributed Computing* 33.2 (1996), pp. 197–204.

[36] M. Rahman, L.N. Pouchet, and P. Sadayappan. "Neural network assisted tile size selection". In: *Proceedings of the International Workshop on Automatic Performance Tuning*. 2010.

[37] S V Rajopadhye, S Purushothaman, and R M Fujimoto. "On synthesizing systolic arrays from recurrence equations with linear dependencies". In: *Proceedings of the 6th Conference on Foundations of software technology and theoretical computer science*. 1986, pp. 488–503.

[38] J. Ramanujam and P. Sadayappan. "Tiling of iteration spaces for multicomputers". In: *Proceedings of the 1990 International Conference on Parallel Processing*. Vol. 2. ICPP '90. 1990, pp. 179–186.

[39] Lakshminarayanan Renganarayana and Sanjay Rajopadhye. "Positivity, posynomials and tile size selection". In: *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*. SC '08. 2008, pp. 1–12.

[40] Lakshminarayanan Renganarayanan. "Scalable and Efficient Tools for Multi-level Tiling". PhD thesis. Fort Collins, CO, USA: Colorado State University, 2008.

[41] Lakshminarayanan Renganarayanan et al. "Parameterized tiled loops for free". In: *Proceedings of the 28th ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI '07. 2007, pp. 405–414.

[42] Gabriel Rivera and Chau wen Tseng. "A Comparison of Compiler Tiling Algorithms". In: *Proceedings of the 8th International Conference on Compiler Construction*. 1999, pp. 168–182.

[43] Vwani Prasad Roychowdhury. "Derivation, extensions and parallel implementation of regular iterative algorithms". PhD thesis. Stanford, CA, USA: Stanford University, 1989.

[44] M.M. Strout et al. "Schedule-independent storage mapping for loops". In: *ACM SIGOPS Operating Systems Review* 32.5 (1998), pp. 24–33.

[45] A. Sussman. "Model-driven mapping onto distributed memory parallel computers". In: *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*. SC '92. 1992, pp. 818–829.

[46] S. Tavarageri et al. "Dynamic selection of tile sizes". In: *Proceedings of the 18th International Conference on High Performance Computing*. 2011, pp. 1–10.

[47] William Thies et al. "A Unified Framework for Schedule and Storage Optimization". In: *Proceedings of the 22nd International Conference on Programming Language Design and Implementation*. PLDI '01. 2001, pp. 232–242.

[48] V. Van Dongen and P. Quinton. "Uniformization of linear recurrence equations: a step toward the automatic synthesis of systolic arrays". In: *Proceedings of the International Conference on Systolic Arrays*. 1988, pp. 473–482.

[49] S. Verdoolaege. "isl: An integer set library for the polyhedral model". In: *Mathematical Software–ICMS 2010* (2010), pp. 299–302.

[50] M. Wolfe. "Iteration space tiling for memory hierarchies". In: *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*. 1987, pp. 357–361.

[51] W.A. Wulf and S.A. McKee. "Hitting the memory wall: Implications of the obvious". In: *ACM SIGARCH Computer Architecture News* 23.1 (1995), pp. 20–24.

[52] Yoav Yaacoby and Peter Cappello. "Converting affine recurrence equations to quasi-uniform recurrence equations". In: *VLSI Algorithms and Architectures*. Ed. by John Reif. Vol. 319. Lecture Notes in Computer Science. 1988, pp. 319–328.

[53] T. Yuki et al. "Automatic creation of tile size selection models". In: *Proceedings of the 8th IEEE ACM International Symposium on Code Generation and Optimization*. CGO '10. 2010, pp. 190–199.

[54] Tomofumi Yuki and Sanjay Rajopadhye. "Memory Allocations for Tiled Uniform Dependence Programs". In: *3rd International Workshop on Polyhedral Compilation Techniques*. 2013.