Computer Science
Technical Report

Colorado State University

Canonic Multi-Projection: Memory Allocation for Distributed Memory Parallelization

Tomofumi Yuki and Sanjay Rajopadhye

September 20, 2011

Colorado State University Technical Report CS11-106

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

Canonic Multi-Projection: Memory Allocation for Distributed Memory Parallelization

Tomofumi Yuki and Sanjay Rajopadhye

***Abstract*—The Polyhedral model is now the accepted technology for automatic parallelization of affine control loop programs. It has been successful in automatically generating tiled shared memory parallel programs for shared memory platforms (plus vectorization). We address the challenges arising when we move toward distributed memory parallelization, based on wavefront execution of parameterized tiles.**

Contrary to the shared memory case, the memory allocation of the sequential program must be reconsidered for distributed memory parallelization. Most polyhedral parallelizers for shared memory architecture bypass this problem by retaining the memory allocation of the original program. The memory problem is critical in distributed memory parallelization, since each processor must be allocated its own memory, and retaining the original memory allocation is no longer an option.

For polyhedral programs with uniform dependences, we present a memory allocation scheme that enables efficient communication with minimal storage use and access overhead. The key ideas in our memory allocation are (i) using multiple storage for values produced by a statement, and (ii) redundantly storing the "halo" regions of tiles to homogenize the memory accesses.

I. Introduction

In the recent years, we have seen multi-core processors becoming more and more main stream, leading to active research in many areas related to parallel computing. One of the successes in automatic parallelization using the polyhedral model is automatic generation of shared memory parallel programs. PLuTo [1] is a now well-known automatic parallelizer using polyhedral techniques that parallelizes sequential C programs for shared memory using OpenMP. PLuTo uses tiling [2] combined with wave-front parallel execution of the tiles to achieve efficient parallelization.

PLuTo requires the tile sizes to be fixed at code generation time, but recent work by Hartano et al. [3], [4], and by Kim and Rajopadhye [5] presented new algorithms for generation of shared memory code with parameterized tile sizes. We may say that shared memory code generation for polyhedral programs is a largely solved problem, although many issues about the choice of good transformations for various target architectures remain open.

However, a large number of scientists today still use distributed memory parallelization of some form, MPI being the most commonly used programming model. It is unclear how far shared memory architecture can scale, and on-chip manycore machines are evolving towards a distributed memory architecture. Therefore, developing methods for distributed memory architecture is an important next step.

One of the main challenges in distributed memory parallelization beyond shared memory is memory allocation. Since the memory is now split among multiple processors, the original memory allocation in the sequential program can no longer be used. Furthermore, additional memory for buffered and aggregated communication is essential for performance reasons. Finding memory allocations that achieve efficient communication, storage, and access of data is an important challenge.

This challenge has been partially addressed for polyhedral programs with fixed sized tiling [6] (or no tiling [7]). However, these techniques do not carry over to parameterized tiles, since programs after tiling with parameterized tile sizes no longer fit the polyhedral model. Tiling based parallelization is known to be efficient, and there are a number of important reasons for generating parameterized tiled code [8]. In distributed memory architectures, tile sizes have close relations to load balancing and communication frequency, further emphasizing its importance.

Previously developed memory allocation strategies are not suitable for distributed memory due to correctness and/or performance reasons. Memory allocations for polyhedral representation of programs [9], [10], [11], [12] are not applicable for parameterized tiles. Schedule independent memory allocation by Strout et al. [13] can provide legal memory allocations for tiled programs with parametric tile sizes. However, when the memory allocations are not along canonic axis of the iteration space, memory accesses require modulo operations involving the tile sizes.

For example, consider tiling Smith-Waterman, a sequence alignment algorithm as shown in Figure 1. We repeatedly use Smith-Waterman as examples throughout this paper, mainly due to its relatively simple dependencies that nevertheless sufficient to illustrate the intuition of our approach. Moreover, Smith-Waterman is both an interesting and important target for parallelization and is a topic of active research in high performance

```
for (i=0; i<=N; i++)
   for (j=0; j<=M; j++)
      H[i][j] = max(
          H[i-1][j-1] + costA(i,j),
          H[i][j-1]   + costB(i,j),
          H[i-1][j]   + costC(i,j));
```

Fig. 1: Smith-Waterman is a dynamic programming algorithm updating table H using its three neighbors, and cost associated by taking different paths.

computing. For instance, huge sequences, such as the entire human genome [14], can benefit from distributed memory parallelization.

For the set of dependencies in Smith-Waterman, an efficient memory allocation that do not preclude tiling would be the diagonal projection shown in Figure 2. Each processor only needs memory for a tile, which is much smaller than the memory requirement for sequential execution of the same program. Thus the memory allocated for each processor is proportional to the tile sizes using this memory allocation.

However, when a processor reuses the memory allocated across tiles, along with the values stored, modulos by tile sizes are required when accessing memory. As seen in Figure 2, projections from each tile do not completely overlap with each other. The overlapping region must be mapped to the same physical location to preserve legality, while its position with respect to other points in its tile are different (one is the left-most 4 and the other is the right-most), and thus requiring modulo operations.

Because modulo operations are very expensive, and cannot be optimized when the divisor is not known at compile time, this memory allocation suffers from high performance penalty. For Smith-Waterman in the example, we observed that that using schedule independent memory allocation incurred a performance hit, by factor of about 8, in sequential execution. Moreover, at lower-left corner of the tiles, a tile requires values from 3 other tiles, further complicating communication of data across processors. Thus, there is a strong need for memory allocations that avoid these challenges.

In this paper, we make a first step towards distributed memory code generation in the polyhedral model with parameterized tile sizes. We address the challenge of memory allocation when dependencies on computed values (those that are not inputs to the polyhedral section) to be uniform. A number of dense linear algebra kernels and stencil computations fit this restriction, and many affine dependencies can be transformed as a sequence of uniform dependencies.

The key ideas presented in this paper are:

- Multiple allocations for a single statement
- Canonic allocations along the axes
- Redundant allocation for tile boundaries ("halo" regions)

These ideas combined achieves simplified communication across processors and homogenized accesses to memory.

The rest of the paper is organized as follows. We discuss related work in Section II, and introduce necessary background an notations in Section III. We illustrate our memory allocation strategy in Section IV, and describe an implementation of the technique in Section V. We further extend the proposed memory allocation for efficient distributed memory parallelization in Section VI. We evaluate the overhead of our memory allocation in comparison with other memory allocation schemes in Section VII, and conclude by discussing a number of future directions we seek to pursue in Section VIII.

II. Related Work

Amarasinghe and Lam [7] presented one of the earliest approaches for generating MPI style code from polyhedral representations without tiling. Claßen and Griebl [6] later presented their approach for fix sized tiles.

The important distinction between these work and ours is that we target distributed memory code with parameterized tiling. Parameterized tiling cannot be expressed affine transformations, complicating analyses and code generation.

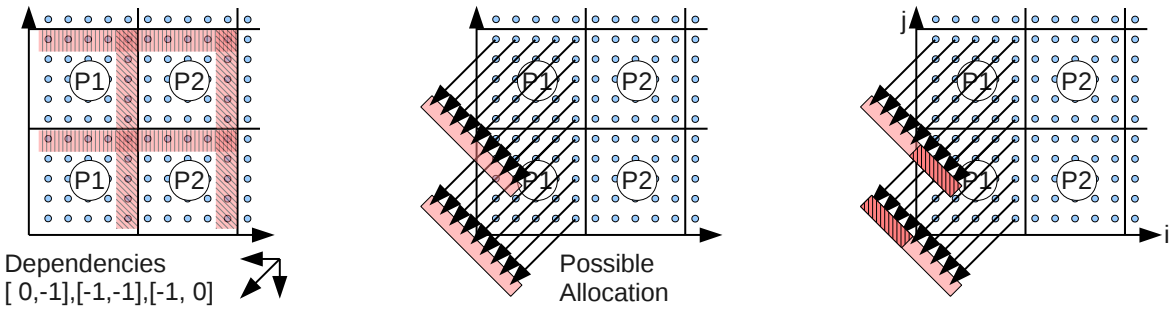
In this paper, our focus is in memory allocation, where most previous approaches in the polyhedral model are no longer applicable due to the non-affine nature of the parameterization of tile sizes [9], [10], [11], [12].

III. Background

In this section we introduce polyhedral background used in the paper. The polyhedral model builds on a dependence analysis technique called the exact array data-flow analysis [15]. For a restricted class of programs, known as Static Control Parts (SCoPs), precise dependence analysis can be performed. SCoPs are loop nests with loop bounds and array accesses expressed as affine functions of the surrounding loop indices and program parameters.

Array data-flow analysis answers the question, which *instance* of which statement produced the value used by a given *instance* of a statement. After this analysis, programs can be viewed in polyhedral representation, where each statement is associated with its *domain*; the set of points where the statement is executed; and dependencies[1] between statements expressed as affine functions.

[1]In our notation a dependence is from consumer to the producer of values.



(a) Values used by other tiles are highlighted.

(b) Possible memory allocation that retains all necessary values after sequential execution of each tile.

(c) Highlighted regions are values produced and used across the tiles allocated to the same processor.

Fig. 2: Memory allocation for tiled Smith-Waterman. Circles show a possible processor allocation. Top row and rightmost column are the values used by other tiles. Schedule independent memory allocation [13] would give a memory allocation as shown in (b) for this program. The memory allocated for the oblique projection should be shared across tiles for a same processor to reduce memory usage. However, as illustrated in (c), the view of the same memory is different (shifted) across tiles. Use of moudulo operations by tile width and height to avoid this issue results in high overhead with parametric tile sizes.

a) Domains and Functions: Polyhedral domain is a finite union of integer polyhedra, where an integer polyhedron is the set of points that satisfy a finite number of linear or affine inequality constraints. The linear/affine constraints involve the surrounding loop indices and program parameters.

Affine functions are mapping of set of points to another set of points, where the destination is expressed as affine expressions of the source. Given a list of indices z, the source points, an affine function is denoted as $(z \rightarrow Az + b)$ where A is a constant matrix and b is a constant vector.

Uniform function is a special case when A is the identity matrix. When a function is uniform, we refer to the vector b as the dependence vector, which is sufficient to describe uniform dependencies.

b) Dependence Level: For uniform dependencies, we refer to the first (outermost) dimension where the corresponding element of the dependence vector is non-zero, as the *level* of the dependence.

c) Lexicographic Order: In this paper, we assume such scheduling has been performed as a pre-processing, and use lexicographic order as the scanning order of polyhedra. Thus the lexicographical ordering between the set of points in the domain directly corresponds to the execution order. Symbols $\prec$ and $\preceq$ are used to denote lexicographical precedence, strict and non-strict, respectively, for two sets of points.

d) Lifetime of Values: Lifetime of a computed value is defined as the distance between the time (i.e., iteration) instant when a value is produced and that of its last use. Since the lexicographic ordering involves multiple dimensions, the distance is also expressed as a multi-dimensional vector.

Similarly, we refer to the distance between the producer and the consumer of a dependence as lifetime with respect to a dependence. The lifetime imposed by a uniform dependence with dependence vector b, is simply $-b$.

e) Canonic Projection: Canonic projections, are projections along one of the canonical axes in the domain. We denote canonic projections along the d-th dimension as p_d.

f) Pseudoprojective Mapping: Pseudoprojection is an extension to affine mapping that adds modulo factors to each dimension. Modulo operations are performed by the given modulo factors to each dimension of the result of the image by an affine mapping. For an affine function from $\mathcal{Z}^n$ to $\mathcal{Z}^m$, the modulo factors are expressed as a vector of length m. Our convention is that no modulo operations are performed for dimensions with modulo factor 0.

In prior work on memory allocations for polyhedral model, an extension to affine functions, pseudoprojections were used to express memory allocations [10], [12]. Accesses to alternating locations are expressed as modulo operations and as long as the modulo factor is a small compile-time constant, the overhead is small.

g) Other Notations: We use the following two vectors to simplify our presentation in the paper. u^d is a vector where every component is 0 except for the d-th element that has the value of 1. v^d is a vector where every component is 0 from 0 to $d-1$-th element, and 1 from d to the last element. For $n = 3$, $u^1 = [0, 1, 0]$ and $v^1 = [0, 1, 1]$.

IV. Canonic Multi-Projection

In this section, we illustrate one of our key contributions, that allows us to avoid the use of modulos with runtime parameters in access functions, in the context of sequential parameterized tiled code with uniform dependencies. We start by introducing components that together form our memory allocation scheme.

We assume that the program is after affine transformations to make tiling legal. One of such transformations and scheduling strategies are used in PLuTo [1]. After such transformation, all dependence vectors in the program are in the non-positive orthant (0 or negative in all dimensions).

Our memory allocation targets a tile of tiled programs, where tiles are executed with wave-front parallelization, and the execution order within a tile is sequential.

A. Canonic Projection

We have two main motivations for using canonic projections as memory mapping. One is that the image of a rectilinear tile by a canonic projection of tiles is one of its facets, and for uniform dependencies, values produced at tile facets are the values used by neighboring tiles, and thus are what need to be transfered across processors.

The other is to avoid modulos by runtime parameters in the access functions. Since tiling hyperplanes are also along the canonic axes, neighboring tiles along an axis can share the same array without using modulo operations, avoiding the problem of oblique projections illustrated in Figure 2.

Any canonic projection p_d preserves a value written into a location for exactly one iteration of the d-th dimension. Thus, a canonic projection can preserve the values produced to satisfy dependencies with unit dependence vectors along the projected dimension. Extending its lifetime to more than one iteration of the d-th dimension can be trivially achieved by using pseudoprojections and adding a modulo factor for the d-th dimension. With pseudoprojections, dependencies with any multiple of unit dependence vectors along the projected dimension can be satisfied.

B. Multi-Projection

Multi-projection is when the values produced by a statement is stored in multiple arrays with different projections. Because any single canonic projection is not sufficient to preserve the values produced for all dependencies, we use multiple projections. For example, if a point (i, j) in two dimensional space depends on $(i-1, j)$ and $(i, j-1)$, these two dependencies cannot be satisfied by memory allocation projecting along the i axis or the j axis.

We could satisfy the dependencies by using modulo factors, two for the above example, but we still need the horizontal tile face to be stored separately leading to increased memory usage. Thus, we use multiple canonic projections to satisfy all the dependencies and preserve the tile facets.

C. Which Projection Satisfies Which Dependence

With multi-projections, we must answer the question, which projection satisfies which dependence. In other words, different dependencies to the same statement need to access different arrays in the generated code, and which array to access must be statically determined for each dependence.

We answer this question by a simple mapping using the dependence level; a dependence with dependence level d is satisfied by p_d. The set of dependencies where the first (outermost) non-zero entry is the d-th dimension is satisfied by the canonic projection along the d-th dimension.

D. Delayed Write

However, canonic projections by itself is only sufficient for dependencies along the axes. We propose *delaying* the writes to satisfy other uniform dependencies. The *delay* is also multi-dimensional in our context, expressed as a vector of length n, the number of dimensions. We leave the discussion of how a delay is implemented out of this section, and present an efficient way to implement delays in Section V.

For example, a dependence $(i, j \rightarrow i-1, j-1)$ cannot be satisfied because the projection p_i is responsible for this dependence, which has a mapping of the form $(i, j \rightarrow j)$, and the value computed at $(i, j-1)$ overwrites the value computed at $(i-1, j-1)$ before the value at (i, j) is computed. One way to preserve the value is to use modulo by two for p_i, but it doubles the memory usage. This problem can be avoided without increasing memory usage by *delaying* the write by one time step of the j dimension.

The required delay δ^d for a projection along d-th dimension is computed by analyzing the lifetime of dependencies to be satisfied by p_d. Given the lifetime with respect to a dependence satisfied by p_d, λ, the required delay for this dependence is $\lambda - \lambda_d$ (i.e., d-th element is set to 0). We take the max of such required delays for all dependencies to be satisfied by p_d to find δ^d.

We use modulo factors to preserve the value, instead of delaying, for the d-th dimension, and thus the d-th element of the lifetime is ignored, but the required delay is identical to the lifetime otherwise. We use modulo factors for the d-th dimension, because delaying the write

for some iteration of the d-th dimension may violate legality.

E. Canonic Multi-Projection

We have introduced our motivations and approach in an incremental fashion. All of the above combined forms our proposed memory allocation scheme, named canonic multi-projection, summarized below.

- Given a statement with the set of dependencies that depends on it, Y, we first construct the sets Y_d for each dimension in the domain of the statement. Y_d consists of dependencies in Y with dependence level d.
- For each dimension d, we allocate memory using p_d, the canonic projection along the d-th dimension. We first find the required modulo factor m_d by taking the max of the d-th element of the require lifetimes for each dependence. Then the memory mapping function is of the form $I \bmod (m_d u^d)$, where I is the identity function.
- Then for each mapping, we find the delay factor δ^d, computed as the max of the lifetimes with respect to the dependencies in Y_d, and the d-th element set to 0.

When the modulo factor m_d is 1, the d-th dimension may be dropped from the right hand side of the mapping function, since modulo by 1 always returns 0.

1) Example: Given a statement with the set of dependencies, $Y = \{[1,0,0],[0,1,1],[1,0,1],[1,1,1],[0,0,1],[0,0,2]\}$ we first create three sets of dependencies:

$$Y_0 = \{[1,0,0],[1,0,1],[1,1,1]\}$$
$$Y_1 = \{[0,1,1]\}$$
$$Y_2 = \{[0,0,1],[0,0,2]\}$$

For each projection p_d, we find the mod factors m_d:

$$m_0 = 1, m_1 = 1, m_2 = 2$$

The resulting mapping functions are:

$$p_0 = (i,j,k \rightarrow j,k)$$
$$p_1 = (i,j,k \rightarrow i,k)$$
$$p_2 = (i,j,k \rightarrow i,j) \bmod [0,0,2]$$

Finally, the delay lengths required for each mapping are:

$$\delta^0 = [0,1,1];\ \delta^1 = [0,0,1];\ \delta^2 = [0,0,0]$$

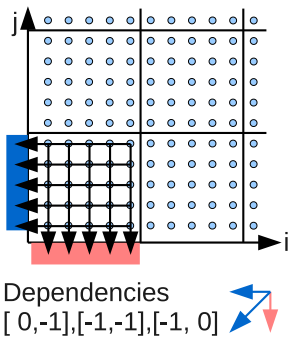


Fig. 3: Canonic Multi-Projection for Smith-Waterman dependencies. Computed values are stored in to vertically projected array as well as the other projected horizontally.

F. Example : Smith-Waterman

We illustrate the memory allocation for our running example, Smith-Waterman, in Figure 3. The horizontal and diagonal dependencies are satisfied using the horizontal projection, and the vertical dependence is satisfied using the vertical projection. The write into the vertical array, corresponding to p_i, is delayed by one iteration of the j dimension.

G. Correctness of the Allocation

Let us first assume that the dependence vector of all dependencies consists of either 0 or -1. We later show how arbitrary uniform dependence vectors are handled.

The set of all possible dependencies that must be satisfied by p_d, denoted as S_d, consists of vectors with no non-zero entry in its dependence vectors before the d-th dimension, due to how dependencies are mapped to projections described above. In practice, the set of dependencies to a statement may be a subset of S_d.

Given an iteration point, $z = (z_1, z_2, ..., z_n)$, in n dimensional space, $\mathcal{Z}^n$, its value stored with p_d is live from z to $z' = z + u_d$; until another point along d is visited. When the writes into a projection is delayed by δ, the duration that the values are preserved is shifted to start from $z + \delta$ and end at $z' + \delta$. There are two conditions that a δ must satisfy :

- $z + \delta \prec z^1$; z^1 is the *first* use of value produced by z
- $z^* \preceq z' + \delta$; z^* is the *last* use of value produced by z

The first condition is to guarantee that the values is "ready" before its first use, and the second condition is to guarantee is that the value is not over-written before its last use. It is not strict precedence in this case assuming all reads happen before any write. We show that there

always exists a δ such that the above conditions are satisfied.

Given the set S_d, the first use z^1 is z plus the lexical minimum of the lifetimes with respect to each dependence in S_d, and likewise, z^* is z plus the lexical maximum of the lifetimes. Lifetimes with respect to the dependencies in S_d all have the following form: $u_d + \sum_{k=d+1}^{n-1} a_k u_k$ where values a_k may be either 0 or 1. Then the lexical minimum is when a is all 0, which is u_d, and maximum is when a is all 1, which is v_d. Thus, $z^1 = z + u_d$ and $z^* = z + v_d$.

There is a trivial solution for δ to satisfy the second condition: $\delta = z^* - z' = z + v_d - (z + u_d) = v_{d+1}$

When $\delta = v_{d+1}$, it can be shown that the first condition is also met by re-writing the first condition; $z + \delta \prec z + u_d \Rightarrow \delta \prec u_d$. $v_{d+1} \prec u_d$ by the lexicographical ordering.

1) Correctness for Arbitrary Uniform Dependence: In the above proof, we restricted the components of the dependence vectors to be either 0 or -1. The strategy can be extended to any uniform dependencies by adding a modulo factors along the projected axes. The set of dependencies S_d is extended to contain all possible dependencies of the form: $m_d u_d + \sum_{k=d+1}^{n-1} a_k m_k u_k$ where m is the mod factor.

H. Access Function

The access functions to the allocated arrays require two slight modifications in the generated code. For a projection p_d, the final access function is composition of three functions: $p_d \circ (z \rightarrow z + o) \circ (z \rightarrow z - ti)$, where o is a vector that aligns the domain of the statement to the origin, and ti is the tile origins; the lexicographically minimum point of a tile. Compositions with these two functions ensure that the tile origins are mapped to the origin, so that (i) no point in a tile is mapped to negative indices, and (ii) no memory is wasted since C arrays are indexed starting from 0.

V. Implementing Delayed Writes

In the previous section, we introduced a new memory allocation scheme that only uses canonic projections. It required writes to the memory to be delayed by some number of iterations. One way of achieving such delayed writes would be using extra memory to operate as shift registers, but it would increase memory usage. In this section, we show how the delaying of writes can be achieved without any extra memory.

A. Propagating Values from Inner Projections

The key intuition is in reusing the values stored by other projections. We claim that the value to be written to p_d at $z+\delta$ is already written to, and still live at another projection along $d+1$-th dimension. Thus, the value to be written to p_d after some delay δ becomes a copy from p_{d+1}.

B. Composition of Delays

When both p_d and p_{d+1} are delayed, the delay for p_{d+1} is also reflected to p_d through the sequence of copying. Thus, at each dimension d, the write is delayed only for the $d+1$-th dimension of the required delay lengths. Delaying of the remaining dimensions are reflected by delaying the write to corresponding inner projections.

This can potentially lead to delaying p_d for longer than necessary in the $d+1$-th dimension. However, because of lexicographical ordering, additional delays in inner dimensions do not violate the legality of the allocation. Similarly, modulo factor for p_d must be greater than or equal to the d-th element of the delay lengths of p_{d-1}.

We present the refined algorithm to find the delay lengths and modulo factors for each dimension, specific to this implementation in Algorithm 1. Algorithms previously described in Section IV *is* legal if the delay is implemented by other means that do not depend on p_d.

The algorithm takes the delay lengths of $d-1$-th dimension as input, and thus executed from the outermost to innermost in sequence. In addition to the delay lengths required by the dependencies to be satisfied by p_d, it also takes into account the delay lengths for compositional delays.

Input:
I_d : set of dependencies to be satisfied by p_d
Λ_d : set of lifetimes with respect to each dependencies in I_d
δ^{d-1} : delay of the previous dimension (0 vector when $d = 0$)
Output:
δ^d : delay length for p_d
m_d : modulo factor for p_d
begin
  $\delta^d = \delta^{d-1}$
  $m_d = \delta^{d-1}_d$
  foreach $\lambda \in \Lambda_d$ **do**
    $\delta^d = \max(\lambda, \delta^d)$
    $m_d = \max(m_d, \lambda_d)$
  end
  `// No delay in the d-th dimension`
  $\delta^d_d = 0$
end

Algorithm 1: Find delay lengths and modulo factor for p_d

C. Copy Statements

For a statement with n dimensional domain, total of $n+1$ statements are generated. The first statement

```
for (i = LBi; i <= UBi; i++)
  for (j = LBj; j <= UBj; j++)
    for (k = LBk; k <= UBk; k++)
      tmp = S1(i,j,k);       //eval
      Pi[j-1][k] = Pj[i][k]; //write_i
      Pj[i][k-1] = Pk[i][j]; //write_j
      Pk[i][j] = tmp;        //write_k
```

Fig. 4: Implementation of delayed writes. `LBx` and `UBx` denote the lower and upper bound of the respective loops. `Px` is the array corresponding to p_x, with no modulo factor.

evaluates the original statement and assigns the result to a temporarily variable, say `tmp`. The subsequent statements stores the appropriate value to projections along each dimension.

When the delay lengths δ^d for a projection along d is the zero vector, the value stored in `tmp` is directly stored. Otherwise the value in memory corresponding to the iteration point $z - \delta^d_{d+1} u^{d+1}$ is copied from p_{d+1} to p_d. Only the $d+1$-th element of the delay lengths δ^d is respected due to composition of delays. These statements are ordered from writes to the projections along outermost dimensions to the inenr ones, so that reads happen before writes.

D. Example

Figure 4 shows an example of how statements are generated to achieve delays without extra memory. The statement use all three projections without any modulo factors, and the writes to p_i are delayed by $[0,1,1]$, and the writes to p_j are delayed by $[0,0,1]$.

Since p_k is not delayed, `tmp` is written to the array `Pk`. p_i is delayed by $[0,1,1]$, but since only the d+1-th dimension is delayed, the iteration z writes to $z - [0,1,0]$, projected by $(i,j,k \to j,k)$. The mapping can be expressed as composition of two functions $(i,j,k \to j,k) \circ (i,j,k \to i,j-1,k)$, $(i,j,k \to j-1,k)$. Similarly, the read from p_j is $(i,j,k \to i,k) \circ (i,j,k \to i,j-1,k)$, $(i,j,k \to i,k)$.

Applying the same procedure as above to p_j, delayed by $[0,0,1]$, gives $(i,j,k \to i,k-1)$ as the write to p_j and $(i,j,k \to i,j)$ as the read from p_k.

Figures 5a and 5b pictorially illustrates another example with two dimensional iteration space with Smith-Waterman.

E. Correctness of the Propagation

It can be guaranteed that the value to be written to p_d with some delay δ is still live in the projection for a inner dimension, p_{d+1}. Let us again restrict all the component of the dependence vectors to be either 0 or -1 for simplicity. The same generalization as in Section IV-G is applicable.

Recall from Section IV-G that the values stored in p_d is preserved from $z+\delta^d$ to $z+u^d+\delta^d$. For a projection p_d, with delay δ^d, the value produced at z must be available at $z+\delta^d$ in p_{d+1}. Thus, we must show that the following conditions are satisfied for selected δs.

- $z+\delta^{d+1} \prec z+\delta^d$: The write to p_d is after the value produced at z is written to p_{d+1}.
- $z+\delta^d \preceq z+u^{d+1}+\delta^{d+1}$: The write to p_d is before the value produced at z stored in p_{d+1} is over-written.

Given the restriction on the dependence vectors, δ^d is bounded as follows: $u^{d+1} \preceq \delta^d \preceq v^{d+1}$. Take the first condition: $z+\delta^{d+1} \prec z+\delta^d$ and replace δ^{d+1} with its upper bound and δ^d with its lower bound, and remove z from both hands to yield: $v^{d+2} \prec u^{d+1}$ By the definition of lexicographical ordering, we verify that the first condition is satisfied for any possible δ.

Take the second condition: $z+\delta^d \preceq z+u^{d+1}+\delta^{d+1}$ subtract $z+u^{d+1}$ from both hands to obtain: $\delta^d - u^{d+1} \preceq \delta^{d+1}$ Because δ^d always has 1 in the $d+1$-th component and δ^{d+1} always has 0 in the $d+1$-th component, the condition is satisfied iff: $\delta^d_x \leq \delta^{d+1}_x$ is met for all x such that $d+2 \leq x < n$. The above is satisfied when δ is computed using Algorithm 1. When computing δ^{d+1}, δ^{d+1}_{d+1} is set to 0, and for inner dimensions x, δ^{d+1}_x is at least δ^d_x, because it is initialized to δ^d_x and the only operation performed is $\max$.

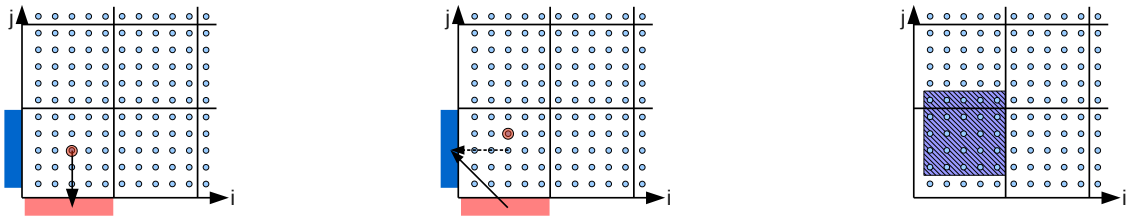
1) Flushing of Delays: In order to maintain the atomicity of tiles, the delayed writes should complete within the tile. For space reasons, we omit the detail of the flushing, but it can be implemented as a simple post processing of the loops in a tile.

VI. Halo Regions

In the previous sections, we described our memory allocation for preserving tile faces and avoid modulo by run-time parameters. Now we present additional techniques to apply canonic multi-projection for distributed memory parallelization. We show how canonic multi-projection combined with the use of "halo regions"; the set of points outside of, but used by, a tile; can simplify communication in distributed memory. The key idea is to include the halo regions when allocating memory for canonic multi-projection.

A. Need for Homogenized Access

In Figure 6a, we illustrate the halo region of a tile in Smith-Waterman. These shaded regions are values from other tiles, and if these accesses were to be handled in communication buffers, the diagonal dependence must be special cased in three different ways at the boundaries.



(a) Compute and store the value into p_j. The write into p_i does not take place yet.

(b) At the next iteration point, copy the value stored in the p_j over to the p_i.

(c) Delayed writes take place in the highlighted iteration points.

Fig. 5: Illustration of how delayed write is implemented with out extra memory.

This complicates the code generation, and also leads to inefficient code because of conditionals in the innermost loops. With polyhedral analysis, finding the domain to special case can be done, but such approach cannot be applied to parameterized tiled codes. We would like all dependencies to access arrays in a homogenized manner to avoid these issues.

B. Memory Allocation for Halo Regions

We homogenize the access by allocating extra memory for storing the values corresponding to the data read from halos. This is equivalent to allocating memory for slightly larger tiles, by expanding each dimension of the tiles.

How the tiles are expanded is different for each canonic projection. For projection p_d with δ^d, the tile is expanded by δ^d. The expansion is in negative direction, since all dependencies in a tilable program is non-positive and therefore, halo regions are in the negative direction. In addition, the memory access function is shifted to the positive direction by the expansion factor so that it is still aligned to the origin.

The expansion factor corresponds to the maximum length of the dependence, including those for compositional delays[2]. This ensures that all accesses from a tile have corresponding memory locations, allowing homogenized access.

C. Values at the Halo Regions

However, we must ensure that the extra memory allocated for the halo regions to have the "correct" values, such that canonic multi-projection is still legal. This is achieved by applying delayed writes, presented in Sections IV and V, to values computed in previously executed tiles. Because the extra memory is redundantly allocated, the delayed writes must be performed for each redundant copy.

[2]Recall that the δ^d is computed based on the maximum lifetime based on dependencies to be satisfied by p_d, and δ^{d-1}

Let us first assume that the values from previously executed tiles are correctly transfered to the memory allocated for a tile, when execution of a tile starts. Although all required values are given to the current tile, it is still not sufficient for the memory allocation to be legal.

Consider the case for Smith-Waterman illustrated in Figure 6. At the beginning of a tile, some of the values to be read by the diagonal dependence are stored in p_j. Because the diagonal dependence is satisfied by p_i, values in p_j must be copied for homogenized access to be legal.

The correctness comes from the same proof in the previous sections, with a slight modification to the length of the values preserved. The values from other tiles are preserved until the first write; lexical minimum among the set of points mapped to the same location.

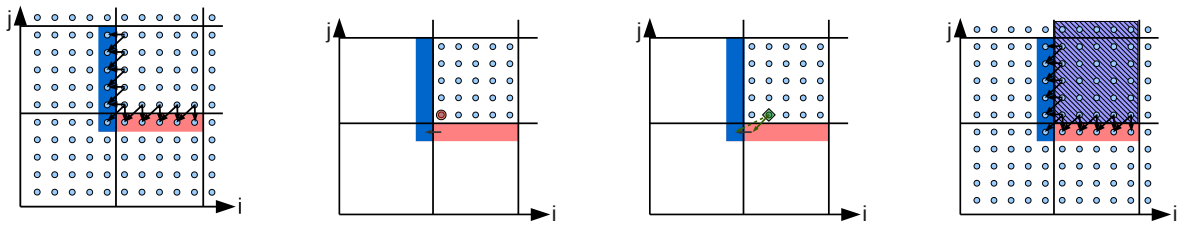
D. Simplified Communication

Maintaining the values of halo regions as well as other points in a homogeneous fashion also contributes to simplified communication. In the Smith-Waterman example, the diagonal dependence causes a tile to depend on two tiles from its neighboring column of tiles. Without redundant storage, this requires communicating with two tiles.

When a tile maintains values including the halo regions, the value for diagonal dependence is available in both tiles. Thus, only a single transfer is required for communicating required values. Moreover, because each canonic projection correspond to faces of tiles, the program can always send all content in a projection without requiring further analysis.

In Figure 6, notice that the delayed write at the lower-right corner of a tile would copy the value computed at the tile below into p_i, which is eventually sent to its horizontal neighbor and used to satisfy the diagonal dependence.

a) Memory Usage: The memory allocation presented in this paper has similar memory usage to schedule independent memory allocation by Strout et. al [13].



(a) Halo regions of Smith-Waterman (b) At the first iteration of a tile, the value in p_j is copied to p_i. (c) In the beginning of the next column, the diagonal dependence use the copied value. (d) Where copies corresponding to delays are executed.

Fig. 6: Delayed write of the halo regions. The value of the halos are communicated to a tile from another tiles. Treating those values exactly like computed values, and performing delayed writes gives correct values for the halo regions.

It could use significantly more memory when compared to other allocations optimizing for memory and possibly losing parallelism. For tiled programs (assuming dependencies cross all tile boundaries), all facets[3] of th tiles must be preserved for execution of subsequent tiles. Thus, the lower bound on required memory for a tile is the sum of all facets. For rectilinear domain, its facets can be obtained by projections along canonic axes, exactly matching the memory allocated by canonic multi-projection. There is a trivial optimization we use to avoid excessive storage (e.g., storing all three faces of matrix multiply), by making sure the allocated projection is used.

VII. Evaluation

The proposed memory allocation scheme introduces a large number of copies in the code. We evaluate the overhead through experiments using four polyhedral kernels with different memory allocations implemented in C with OpenMP.

A. Experimental Setup

We used two machines, a 8 core machine with two Intel Xeon E5450 processors sharing 16GB of memory, and a 8 core AMD Opetron in Cray XT6m with 32GB of memory for our experiments. We used ICC 12.0.3 on the Xeon, and CrayCC 7.3.3 on the Cray XT6m.

b) Kernels: In addition to Smith-Waterman, we used three kernels from PolyBench 2.0 [16], matrix multiply (`gemm`), 1D Jacobi Stencil (`jacobi-1d-imper`), and 2D Jacobi Stencil (`jacobi-2d-imper`) for experimentation.

[3]We count the two facets that are parallel to each other as one, so that a cube has *three* faces, to simplify our presentation.

c) PGAS-Style OpenMP: For canonic multi-projection, we generated codes that mimics code for distributed memory structure by allocating memory separately for each processor. All local variables are accessed with the first dimension of the access function being the thread ID. This is similar to the Partitioned Global Address Space, used in a number of recent parallel programming languages.

We omit the detail of our code generator due to space limitations. The main motivation for such code is to ensure that the proposed allocation works in such environment. Thus, the code only writes to memory locations assigned to other threads only at the end of each tile in a separate block of code for communication.

d) Comparison Targets: In addition to the canonic multi-projection, we generate code with Universal Occupancy Vector (UOV) [13] based allocation, and other commonly used allocation for the benchmark where applicable. For matrix multiply, commonly used allocation matched UOV-based allocation. For other benchmarks, a typical memory allocation is to use two copies of 1D, or 2D array depending on the benchmark, and use them in turn. This is implemented using pseudo projective allocation with modulo factor of 2.

For these allocations, we generate code that are tiled and paralellized for shared memory using existing techniques [4], [5]. Note that these memory allocations are not suitable for distributed memory parallelization (except for matrix multiply), unlike canonic multi-projection.

e) PLuTo: The above two code generators are implemented in a same framework developed in our group, and thus it is more appropriate for comparing effect of different memory allocations. We also provide data from PLuTo [1] as an external point of reference. However, pre-vectorization in PLuTo was disabled since our code generator do not perform this optimization. Our goal is to compare differences due to memory, and thus we would

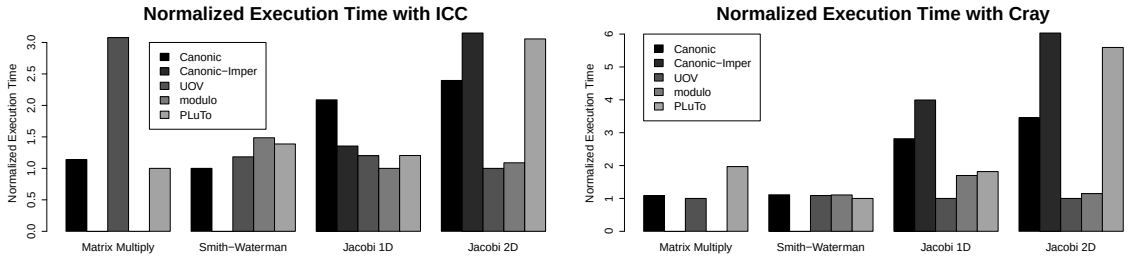


Fig. 7: Sequential performance of benchmarks with different memory allocations. The execution times are normalized to the best for each benchmark.

like to keep the generated codes to have similar level of other optimizations.

f) Imperfect Nesting in Jacobi: For Jacobi stencils, the versions in PolyBench are implemented with explicit copies to avoid pointer swaps for modulo accesses. This is due to the initial choice of memory allocation, and the corresponding polyhedral representation is also influenced. Therefore, for our target of comparison (UOV-based and modulo allocation), we use "cleaner" polyhedral representation coming from single assignment versions of the program. For canonic multi-projection, we experimented with both versions.

B. Experimental Results

Figure 7 shows sequential performance with different memory allocations. We achieve reasonable performance with matrix multiply and Smith-Waterman, but the performance of Jacobi stencils are poor in some cases. In the following, we discuss several observations of our result.

1) Performance Degradation in Jacobi Stencils: Canonic multi-projection applied on polyhedral representations extracted from the imperfectly nested versions achieve similar level of performance as PLuTo. However, we observed significant overhead compared to UOV-based allocation. With profiling, we found that there were significant difference in the number of instructions being executed.

Our hypothesis is that the additional copy instructions for canonic multi-projection are being significant. We experimented with variations of the 2D Jacobi of PolyBench, by adding additional floating point operations to the kernel. Figure 8 illustrates the effect of adding additional operations. The original kernel have 5 operations (4 adds, 1 multiply), and we show results up to 15 operations per iteration point.

For ICC, additional 10 operations are sufficient for canonic multi-projection to reach comparable performance with UOV-based allocation. With Cray, 10 operations were still not enough, but the relative performance

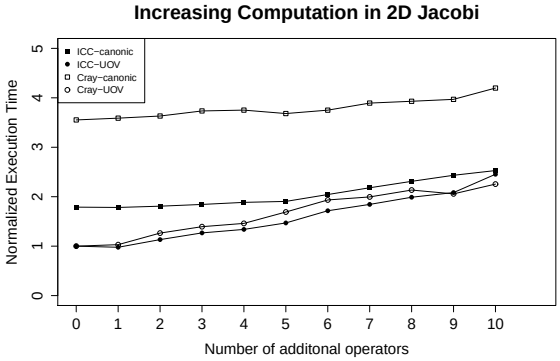


Fig. 8: Effect of increasing the amount of computation in 2D Stencil. The execution times are normalized to the original program with UOV-based allocation on each machine.

is improving as more operations are introduced. We did confirm that making the kernel much more compute intensive eventually makes the performance comparable on Cray as well.

This result suggests that if the application is compute intensive, the overhead of our memory allocation may be small enough. We continue to seek for optimizing our approach to increase the range of applicable kernels.

2) Scaling: Figure 9 summarizes the parallel scaling of each implementation by reporting the parallel efficiency with 8 cores. Overall, our approach have similar parallel efficiency.

3) Performance Variation Across Compilers: In different occasions, we have observed significant performance difference across compilers. For instance, the results for matrix multiply varied significantly even though the code generated by our code generator and by PLuTo were quite similar. We speculate that dynamically allocating memory with *runtime* parameters makes it more difficult for back-end compilers to perform low-level optimizations. However, we do not have definitive

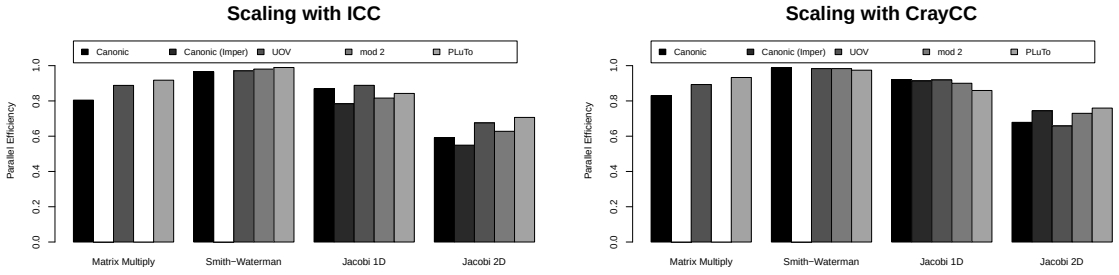


Fig. 9: Parallel efficiency with 8 cores with respect to the running time with 1 core. The parallel efficiency is comparable to other implementations in all benchmarks.

answers to the observed variations.

VIII. Conclusion and Future Work

In this paper, we have presented a memory allocation scheme for distributed memory parallelization, and evaluated its overhead in a shared memory environment with PGAS-Style code. The evaluation suggests that our approach would have acceptable overhead for compute intensive programs.

Although currently limited to uniform dependencies, we believe that our memory allocation significantly simplifies code generation for programming models that require explicit data transfer. We have started implementing a code generator with C and MPI, and are also working on extensions to handle non-uniform dependencies.

There are a number of interesting directions to extend beyond uniform dependencies. For instance, with a one-dimensional processor allocation, there are $n - 1$ dimensions where communications are local to a physical processor. Thus, it may be sufficient to restrict the dependencies such that there is at least one dimension where the only dependencies that cross are uniform. Since tilable programs are also fully permutable, this dimension can be made the inter-processor communication dimension, avoiding communications with affine dependence.

References

[1] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*.

[2] M. Wolfe, "More iteration space tiling," in *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pp. 655–664, ACM, 1989.

[3] M. Baskaran, A. Hartono, S. Tavarageri, T. Henretty, J. Ramanujam, and P. Sadayappan, "Parameterized tiling revisited," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pp. 200–209, ACM, 2010.

[4] A. Hartono, M. Baskaran, J. Ramanujam, and P. Sadayappan, "Dyntile: Parametric tiled loop generation for parallel execution on multicore processors," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, IEEE, 2010.

[5] D. Kim, *Parameterized and multi-level tiled loop generation*. PhD thesis, COLORADO STATE UNIVERSITY, 2011.

[6] M. Claßen and M. Griebl, "Automatic code generation for distributed memory architectures in the polytope model," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, p. 243, IEEE, 2006.

[7] S. Amarasinghe and M. Lam, "Communication optimization and code generation for distributed memory machines," in *ACM SIGPLAN Notices*, vol. 28, pp. 126–138, ACM, 1993.

[8] L. Renganarayanan, D. Kim, S. Rajopadhye, and M. Strout, "Parameterized tiled loops for free," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pp. 405–414, ACM, 2007.

[9] V. Lefebvre and P. Feautrier, "Automatic storage management for parallel programs," *Parallel Computing*, vol. 24, no. 3-4, pp. 649–671, 1998.

[10] F. Quilleré and S. Rajopadhye, "Optimizing memory usage in the polyhedral model," *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 5, pp. 773–815, 2000.

[11] W. Thies, F. Vivien, S. Amarasinghe, and U. L. Pasteur, "A unified framework for schedule and storage optimization," pp. 232–242, 2001.

[12] A. Darte, R. Schreiber, and G. Villard, "Lattice-based memory allocation," *IEEE Transactions on Computers*, pp. 1242–1257, 2005.

[13] M. Strout, L. Carter, J. Ferrante, and B. Simon, "Schedule-independent storage mapping for loops," *ACM SIGOPS Operating Systems Review*, vol. 32, no. 5, pp. 24–33, 1998.

[14] A. C. M. A. d. M. Edans Flavius de Oliveira Sandes, "Smith-waterman alignment of huge sequences with gpu in linear space," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE, 2011.

[15] P. Feautrier, "Dataflow analysis of array and scalar references," *International Journal of Parallel Programming*, vol. 20, no. 1, pp. 23–53, 1991.

[16] L.-N. Pouchet, "Polybench." http://www.cse.ohio-state.edu/ pouchet/software/polybench/.