Effective Automatic Computation Placement and Data Allocation for Parallelization of Regular Programs

A THESIS

SUBMITTED FOR THE DEGREE OF

Master of Science (Engineering)

IN THE COMPUTER SCIENCE AND ENGINEERING

by

Chandan G

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

April 2014

Acknowledgements

Firstly, I would like to thank my research supervisor Prof. Uday Bondhugula for being a supportive guide and mentor. He has always been patient with me, and has encouraged me to think independently and allowed me to work on topics of my interest at my own pace. I have learnt a great deal from him and I will always remain grateful to him.

A big thanks to all my fellow members of the Multi-core Computing Lab – Irshad, Roshan, Ravi, Thejas, Vinayaka, Somashekaracharya, Narayan, Aravind and Vinay for making the lab a fun and wonderful workplace. Being able to interact with such smart people on a daily basis is a big part of my learning at IISc. I have learnt a lot from various discussions with them both technical and otherwise. A special thanks to Irshad for his help towards reviews and suggestions.

I am grateful to the institute for providing all the necessary facilities and making my stay in the campus comfortable. I would like to thank the staff of CSA office for taking care of the administrative tasks. I want to thank the Ministry of Human Resource Development for scholarship to support me during this course. I would also thank AMD for their research grant which was used to fund my conference travel.

I have been fortunate to be in the company of wonderful friends at IISc. Thanks to – Irshad, Roshan, Prasanna, Vinayaka, Aravind, Ninad, Apporva, Lakshmi, Ranjani, Malavika, Prachi, Pallavi, Thejas, Anirudh, Narayan, Jay, Siddharth, Ravi, Satya, Vasanta

and Vinay for ensuring that there was never a dull moment at the institute. I will always cherish the memories of birthday parties, treats, dinner outings, and travel adventures to Nandi hills, Sharavathi, Sarpass, Spiti, Scotland, London. Thanks for making my life at the institute, a joyful and memorable experience.

Finally, I would like to express my profound gratitude to my parents, and sister for their constant support and encouragement. They have given me the independence to take my own decisions and are always supportive of whatever decisions I have taken.

Publications based on this Thesis

1. Chandan Reddy, Uday Bondhugula.
   Effective Automatic Computation Placement and Data allocation for Parallelization of Regular Programs.
   In ACM International Conference on Supercomputing (ACM ICS), June 2014, Munich, Germany.

2. Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula.
   Generating efficient data movement code for heterogeneous architectures with distributed-memory.
   In the 22nd International Conference on Parallel Architectures and Compilation Techniques (ACM/IEEE PACT), September 2013, Edinburgh, Scotland.

Abstract

Scientific applications that operate on large data sets require huge amount of computation power and memory. These applications are typically run on High Performance Computing (HPC) systems that consist of multiple compute nodes, connected over an network interconnect such as InfiniBand. Each compute node has its own memory and does not share the address space with other nodes. A significant amount of work has been done in past two decades on parallelizing for distributed-memory architectures. A majority of this work was done in developing compiler technologies such as high performance Fortran (HPF) and partitioned global address space (PGAS). However, several steps involved in achieving good performance remained manual. Hence, the approach currently used to obtain the best performance is to rely on highly tuned libraries such as ScaLAPACK. The objective of this work is to improve automatic compiler and runtime support for distributed-memory clusters for regular programs. Regular programs typically use arrays as their main data structure and array accesses are affine functions of outer loop indices and program parameters. A lot of scientific applications such as linear-algebra kernels, stencils, partial differential equation solvers, data-mining applications and dynamic programming codes fall in this category.

In this work, we propose techniques for finding computation mapping and data allocation when compiling regular programs for distributed-memory clusters. Techniques for

transformation and detection of parallelism, relying on the polyhedral framework already exist. We propose automatic techniques to determine computation placements for identified parallelism and allocation of data. We model the problem of finding good computation placement as a graph partitioning problem with the constraints to minimize both communication volume and load imbalance for entire program. We show that our approach for computation mapping is more effective than those that can be developed using vendor-supplied libraries. Our approach for data allocation is driven by tiling of data spaces along with a compiler assisted runtime scheme to allocate and deallocate tiles on-demand and reuse them. Experimental results on some sequences of BLAS calls demonstrate a mean speedup of 1.82× over versions written with ScaLAPACK. Besides enabling weak scaling for distributed memory, data tiling also improves locality for shared-memory parallelization. Experimental results on a 32-core shared-memory SMP system shows a mean speedup of 2.67× over code that is not data tiled.

Contents

List of Tables

List of Figures

Keywords

Distributed-memory, Automatic Parallelization, Data-distribution, Polyhedral Model, Computation Placement

Chapter 1

Introduction

1.1 High Performance Computing (HPC) systems

Scientific applications that operate on large data sets require huge amount of computation power and memory. These applications are run on High Performance Computing (HPC) systems. A typical HPC setup consists of multiple compute nodes that are connected over a network interconnect such as InfiniBand. Each node consists of a set of processors that have shared memory; typically, an SMP system of general-purpose multi-cores. Each node could also consist of specialized accelerators such as GPUs, Xeon Phis connected to multi-core processors through PCIex bus. Each node has its own memory and does not share their address space with other nodes. Due to the high latency of interconnects, it is not practical to automatically maintain coherency across different nodes. Hence, it is the responsibility of the programmer to maintain coherency.

1.2 Programming HPC systems

To run scientific applications with large datasets on HPC systems and to achieve scalable performance, one has to perform following steps:

1. Extract maximum coarse grained parallelism by breaking large computation into smaller tasks and identifying the tasks that can be executed in parallel.
2. Distributing the parallel tasks across multiple nodes of cluster (computation placement).
3. Allocate the data required by each task on the node on which the task executes.
4. Schedule the tasks across multiple nodes without violating the dependences between them.
5. Once a task finishes its execution, perform data transfers required to maintain coherence between nodes.

The effort required to manually perform the above steps is enormous, error prone and time consuming. An approach for programming HPC systems that automatically performs all the above steps without any user input and achieves scalable performance for HPC systems is highly desirable.

1.3 Automatic parallelization for HPC systems

Automatic parallelization of arbitrary programs is a very hard problem. However, for a restricted class of programs called regular programs it is possible to develop compiler and runtime techniques required for automatic parallelization. Regular programs typically

use arrays as their main data structure and array accesses are affine functions of outer loop indices and program parameters. A lot of scientific applications such as linear-algebra kernels, stencils, partial differential equation solvers, data-mining applications and dynamic programming codes fall in this category of regular programs. In this thesis we develop efficient solutions for several steps (2,3 and 5 from section 1.2) required for automatic parallelization of regular programs.

1.4 Existing approaches

A significant amount of work has been done in past two decades on parallelizing for distributed-memory architectures. A majority of work was done in developing compiler technology for High Performance Fortran (HPF). However, even in domains where it was suitable, namely programs with regular accesses, there was limited success. Several steps involved in achieving good performance remained manual. The poor quality of communication code as well as the inability to automatically apply complex transformations was a big limitation. Hence, even for programs that involve regular accesses such as sequences of linear algebra kernels, the approach currently used to obtain the best performance is to either write manual MPI code or to rely on highly tuned libraries such as ScaLAPACK. In addition, none of the previous approaches on automatic distributed-memory parallelization and code generation have been directly employed so far even in domain-specific language compilation. MPI still happens to be the dominant and de facto programming model due to the lack of any compiler support. The objective of this thesis is to improve automatic compiler and runtime support for distributed-memory clusters of multi-cores with emphasis on exploiting locality.

Some of the limitations in parallelization and code generation for regular programs, in particular, affine loop nests, have been addressed in recent years [1, 2, 3]. These works

provide techniques for transformation and detection of parallelism, and generation of communication sets relying on the polyhedral framework. However, these works use a simple strategy to map identified parallelism – typically block or block-cyclic. Previous automatic data distribution works [4, 5, 6, 7] also employed only block or block-cyclic mappings for loop nests with the possibility to re-distribute in between.These strategies to map identified parallelism significantly impact communication volume and load balance. Some specialized mappings such as multi-partitioning [8, 9] were known and implemented in dHPF, but these works did not provide any automatic way to determine such mappings. In addition to this, there is significant room for improvement in the way data allocation is handled – to better exploit locality in conjunction with compute transformations. This thesis provides an effective solution to these missing steps.

1.5 Our approach

Although manual distributed-memory parallelization as seen by a programmer often starts with the step of data decomposition followed by computation decomposition, we show that this seemingly natural approach is not the efficient one when designing flexible and automatic compiler support. We argue that emphasis should first be placed on determining right computation transformation and a placement. If good computation placements are found, the initial data distribution only impacts "first-read" and "last write" communication. Determining a data distribution and then a compute distribution as done by some previous approaches may even prevent certain computation distributions where the owner of data has to change in order to exploit locality. Our approach neither has the notion of an owner for data, nor that of fixed distribution, nor re-distribution. Instead data that is accessed for a piece of computation is allocated on demand (if not already allocated), with communication data flowing from one node to another as dictated by computation

placement.

We develop an automatic technique to find good computation placements for the entire program. We model the problem of finding good computation placements as a graph partitioning problem with constraints to minimize both communication volume and load imbalance. For a given program, we build inter Tile Communication Graph (TCG) in which each vertex represents a compute tile. An edge is added between two vertices if and only if there is communication between the corresponding two tiles when they are executed on different nodes. The weight of the edge will be equal to the communication volume between the two tiles. Finding the optimal computation mappings is equivalent to partitioning the TCG into p (number of nodes) equal size partitions with the objective to minimize the sum of those edge weights that straddle partitions. We also identify compute tiles that belong to a parallel phase and add constraints to minimize load imbalance within each parallel phase. This approach encompassing traditional mappings such as block, block-cyclic and other specialized mappings such as multi-partitioning and any other arbitrary mappings. We also find the optimal dimensionality of these mappings.

Our approach for data allocation works by tiling of data spaces. A *data tile* is the granularity at which data is allocated and is itself contiguous in main memory. Data local to a node as well as that which is received from remote nodes is accessed by first addressing a data tile and then indexing into it. A compiler-based approach with light-weight runtime helper functions handles on-demand allocation and deallocation of tiles, and their reuse. The approach can work in conjunction with either static or dynamic scheduling of compute tiles. Besides enabling weak scaling for distributed memory, data tiling improves locality for shared-memory parallelization – by reducing cache conflict misses, data TLB misses, and false sharing, and allowing better prefetching.

1.6 Contributions

The contributions of this work are:

- developing a technique to map identified parallelism after transformation which will minimize both communication volume and load imbalance. This approach encompassing block, block-cyclic and other specialized and arbitrary mappings.
- devising a data allocation technique based on data tiling to provide improved locality and enable weak scaling for distributed memory parallelization.
- present a dynamic, schedule independent data tile buffer reuse techniques.
- develop an efficient data movement scheme based on inter tile dependences that will minimize the communication volume.
- demonstrating through experiments that our approach is significantly better than previous approaches and the code generated outperforms that which can be written even using vendor supplied BLAS libraries.

More specifically, for sequences of BLAS calls, the code we automatically generate out performs code manually written using Intel ScaLAPACK library by a mean factor of 1.82× while running on a 32-node InfiniBand cluster of multicores. Shared-memory parallelization results obtained on a 32-core shared-memory NUMA SMP system show a mean speedup of 2.67× over code that is not data tiled.

1.7 Thesis organization

The rest of this thesis is organized as follows. Chapter 2 presents a brief introduction to the polyhedral model. Chapter 3 describes motivation behind our computation distribution

strategy. Chapter 4 describes our approach to find computation placements. Chapter 5 describes how data tilings are found and how allocation is performed based on a tiled view of data spaces. Experimental results are presented in Chapter 7. Related work and conclusions are presented in Chapter 8 and Chapter 9 respectively.

Chapter 2

Background

Polyhedral model is a mathematical framework that is used to perform various loop transformations such as loop reversal, skewing, interchange, peeling, shifting, fusion, distribution and tiling. In this chapter, we present a brief overview of the polyhedral model and introduce the notation that will be used throughout the thesis.

All row vectors will be typeset in bold lowercase, while regular vectors are typeset with an overhead arrow. The set of all integers is represented by $\mathbb{Z}$.

2.1 Hyperplanes and polyhedra

Definition 1. ***Affine function*** *A k-dimensional function f is called affine function if it can be expressed in following form:*

$$f(\vec{v}) = M_f \vec{v} + \vec{f_0} \tag{2.1}$$

here $\vec{v} = \begin{pmatrix} v_1 \\ \vdots \\ v_d \end{pmatrix}$ *is a* d*-dimensional vector of integers and* $M_f \in \mathbb{Z}^{k \times d}$ *is an integer matrix with* k *rows and* d *columns and represents the linear transformation,* $f_0 \in \mathbb{Z}^k$ *is a* k*-dimensional vector that represents a constant offset.*

Definition 2. ***Affine sub-spaces*** *A set of vectors forms an affine sub-space if it is closed under affine combination. i.e., if* $\vec{x}, \vec{y}$ *are in the space, then all the vector which are an affine combination of* $\vec{x}, \vec{y}$ *also belong to space.*

Definition 3. ***Hyperplane*** *An affine hyperplane is an* $n-1$ *dimensional affine sub-space of an* n *dimensional space. An affine hyperplane can be viewed as a one-dimensional affine function* $\phi(\vec{v})$ *that maps an n-dimensional space onto a one-dimensional space.*

$$\phi(\vec{v}) = \mathbf{h} \cdot \vec{v} + c \tag{2.2}$$

The row vector $\mathbf{h}$ represents the normal to the hyperplane. All the hyperplanes which have same value of $\mathbf{h}$ are parallel to each other. Throughout this thesis, a hyperplane is often referred to by its row vector, $\mathbf{h}$. A hyperplane $\mathbf{h} \cdot \vec{v} = k$ divides the affine space into two half-spaces, the non-negative half-space $\mathbf{h} \cdot \vec{v} \geq k$, and the non-positive half-space, $\mathbf{h} \cdot \vec{v} \leq k$.

Definition 4. ***Polyhedron*** *A polyhedron is an intersection of a finite number of half-spaces.*

The set of affine inequalities, each representing a face, is used to represent the polyhedron. A polyhedron with m inequalities is represented as

$$\{\vec{x} \in \mathbb{Z}^n \mid A\vec{x} + \vec{b} \geq \vec{0}\}. \tag{2.3}$$

Definition 5. ***Affine loop nest*** *A sequence of arbitrarily nested loops with loop bounds and array accesses that are affine functions of program parameters and outer loop variables are called as affine loop nests.*

Program parameters, denoted by $\vec{p}$, represent problem sizes and other symbolic parameters that are loop invariant. Programs that only consist of affine loop nests are called as regular programs.

2.2 Polyhedral model

Polyhedral model is a mathematical intermediate representation that captures the dynamic instances of affine loop nests and dependences between them.

Let $S_1, S_2, \ldots, S_n$ be the statements of affine loop nests of a program. The iteration vector of a statement S, denoted by $\vec{i_S}$, represents a dynamic instance of statement appearing in the loop nest. The set of all iteration vectors for a given statement is called as the iteration domain of S and is denoted by D_S. In polyhedral model, the iteration domain of a statement is modeled as a set of integer points inside polyhedra. For the example shown in the Figure 2.1, (i, j) is the iteration vector and the iteration domain is a polyhedron formed by loop bounds $i \geq 0, i < N, j \geq 1, j < N$, as shown in Figure 2.3. Iteration domain is concisely represented as a matrix as shown in Figure 2.2.

```
for (i=0; i<N; i++)
 for (j=1; j<N; j++)
  X[i][j] = X[i][j] - X[i][j-1] * A[i][j] / B[i][j-1]; //S1
```

Figure 2.1: Example affine program

$$D_{S_1} : \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ -1 & 0 & 1 & -1 \\ 0 & -1 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ N \\ 1 \end{pmatrix} \geq 0$$

Figure 2.2: Matrix representation of iteration domain for S_1 from Figure 2.1

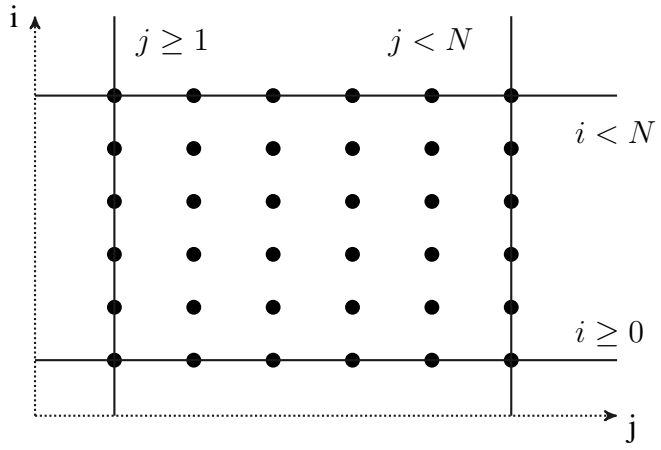


Figure 2.3: Iteration domain for S_1 from Figure 2.1

2.3 Array access functions

An array access function captures the data locations accessed by a statement. All array accesses in a regular program are affine functions of outer loop iterators and program parameters. In the example shown in Figure 2.1 array access $X[i][j-1]$ is represented by affine function X_{S_1}.

$$X_{S_1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \\ 1 \end{pmatrix} \tag{2.4}$$

2.4 Schedules

A schedule is an affine function that describes order in which each dynamic instance of the statement will be executed.

Definition 6. ***Affine schedule*** *An n-dimensional schedule of statement a S is an affine function of the outer loop iterations $\vec{i_S}$ and program parameters $\vec{p}$, denoted by Θ_S.*

$$\Theta_S(\vec{i_S}) = T_S \begin{pmatrix} \vec{i_S} \\ \vec{p} \\ 1 \end{pmatrix}, T_S \in \mathbb{Z}^{n \times (dim(\vec{i_S}) + dim(\vec{p}) + 1)} \tag{2.5}$$

A scheduling function associates a logical execution date (or a timestamp) to each dynamic instance of a given statement. This date can either be a scalar (one-dimensional schedule) or a vector (multi-dimensional schedule). The execution order of the instances is given by *lexicographic ordering* of the associated timestamps. Two instances with the same timestamp can be executed in parallel.

2.5 Polyhedral transformations

In the polyhedral model, loop transformations such as loop reversal, skewing, interchange, peeling, shifting, fusion, distribution and tiling are performed by changing the scheduling function Θ_S. A multi-dimensional scheduling function captures a sequence of affine loop transformations. A loop interchange transformation is to be performed by choosing $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ as scheduling function, for the example shown in Figure 2.1. Not all transformations preserve program semantics. The notation of dependences is used to reason about the valid program transformations.

2.6 Polyhedral dependences

Two iterations are said to be dependent if they access the same memory location and one of them is a write. Dependences are classified based on the order of read and write accesses. If the source access is a write and target is a read, it is called as RAW (Read after Write) or flow dependence. If a write precedes a read to same location, then the dependence is called a WAR (Write after Read) or anti-dependence. If both source and target accesses are writes then it is called a WAW (Write after Write) or output dependence. Read-after-read or RAR are not actual dependences, but they can be used in characterizing reuse. A program transformation is valid, only if it respects all dependences in the original program.

2.7 Data dependence graph

A Data Dependence Graph (DDG) is a directed multi-graph G = (V, E) with each vertex representing a statement and an edge, $e \in E$, from node S_i to S_j represents a dependence between source and target accesses in S_i and S_j respectively.

2.8 Dependence polyhedron

In polyhedral model, program data dependences are captured by *dependence polyhedron,* P_e which corresponds to an edge in the DDG. The dependence polyhedron is a subset of the cartesian product of the iteration domains of source and target access statements. It precisely captures the exact data dependences between the dynamic instance of S_i and S_j.

$$\langle \vec{s}, \vec{t} \rangle \in P_e \implies \vec{s} \in D_{S_i}, \vec{t} \in D_{S_j},\ \vec{s}\, and\, \vec{t}\, access\ same\ data \tag{2.6}$$

Dependence between access $X[i][j]$ and $X[i'][j'-1]$ in the example from Figure 2.1

can be represented by a polyhedron with the constraints shown in Equation(2.7).

$$\begin{aligned} 0 \leq i \leq N-1 \\ 1 \leq j \leq N-1 \\ 0 \leq i' \leq N-1 \\ 1 \leq j' \leq N-1 \\ i = i' \\ j = j'-1 \end{aligned} \tag{2.7}$$

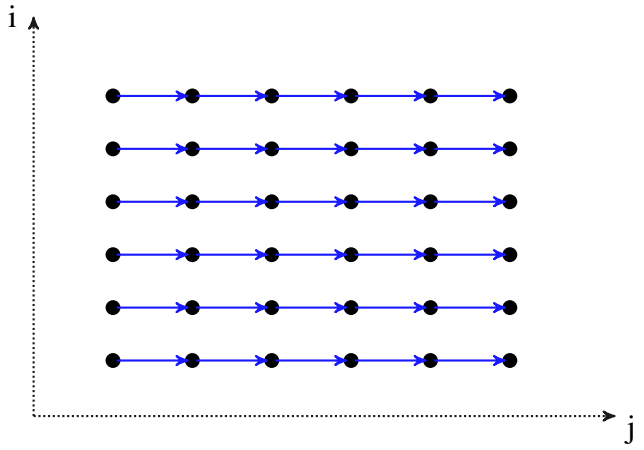


Figure 2.4: Dependence between access $X[i][j]$ and $X[i][j-1]$ from Figure 2.1

The dependence polyhedron is the most important structure that is used to find good program and data transformations, to compute communication sets and to determine valid execution order.

2.9 Operations on polyhedra

We can perform various operations such as union, intersection, difference on the polyhedra. There are libraries such as Piplib [10] and isl [11] which can be used to perform these operations. Another important operation is projection where we can project out different dimensions of a polyhedron using Fourier-Motzkin elimination.

Chapter 3

Parallelization Strategy

In this chapter we compare two different strategies for programming distributed memory architectures, namely owner computes rule and computer owns rule.

3.1 Owner computes rule

In owner computes rule, every data element has a fixed owner and all iterations that compute its new value should be executed on it. In this scheme, data distributions are determined first and computation placement are derived from the data distributions according to owner computes rule. This approach is used in most of the previous distributed memory compilation systems such as HPF (High Performance Fortran). Consider the Jacobi style stencil code shown in the Figure 3.1. All iterations of t for a particular value of i will write to the same location, hence, with owner computes rule all of them needs be executed on the same node. Figure 3.2 shows execution of the stencil with owner computes rule. Stencils have near-neighbor dependence patterns and hence, there is a need to communicate boundary values after every time step. Owner computes rule may prevent certain

```
for (t=0; t<T; t++)
 for (i=1; i<N-2; i++)
  S1:A[(t+1)%2][i]=(A[t%2][i-1]+A[t%2][i]+A[t%2][i+1])/3.0;
```

Figure 3.1: Jacobi style stencil

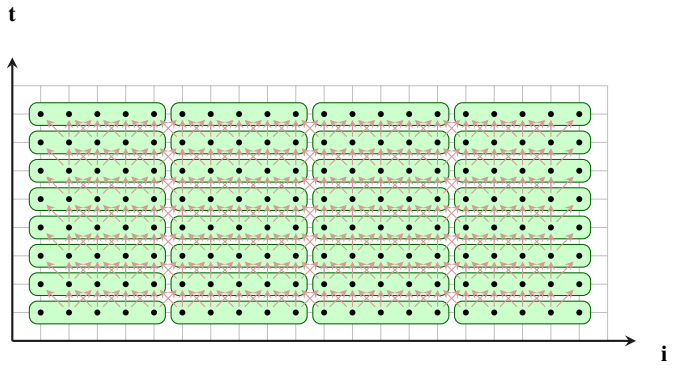


Figure 3.2: Stencil execution with owner computes rule

computation placements where the owner of data has to change in order to exploit locality like time tiling shown in Figure 3.3.

3.2 Computer owns rule

In computer owns rule, we first determine a computation placement and then data is allocated according to the computation placement. In this scheme, an array location does not have a single fixed owner; rather, its value could be recomputed by different compute nodes. The node which last writes to an array element will be its current owner. Due to this relaxation, computer owns rule is more flexible and generic than the owner computes rule. It can support any arbitrary computation placements such as time tiling of stencils [12] shown in Figure 3.3. Figure 3.4 shows the performance of `heat-2d` with and

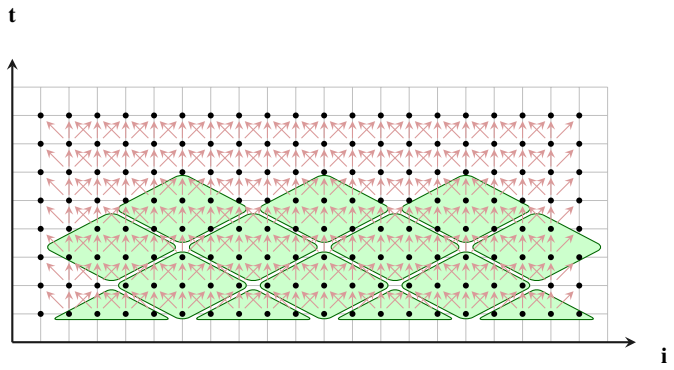


Figure 3.3: Stencil execution with time tiling

without time tiling. Time tiling improves the single thread performance on shared memory due to improved data locality. For distributed-memory, time tiling reduces the number of communication phases and leads to better communication coalescing. For the stencil example with owner computes rule (Figure 3.2), boundary values need to be communicated after every time step whereas, for time tiled code we need to communicate only after entire tile is executed thus reducing the number of communication phases. Also, with time tiled code a bigger coalesced message will be communicated whereas, with owner compute rules boundary values will be communicated after every time step. As shown in Figure 3.4, with time tiling we see very good performance which is close to ideal.

3.3 Data remapping with owner computes rule

HPF also provides an option to specify different data distributions for different parts of the program (dynamic distributions). Programmer first needs to decompose the input program into regions (phases). For each phase, programmer needs to specify data distributions and array alignments for each array. HPF remaps data between different phases. Even with this dynamic distribution support, we cannot implement locality enhancing transformation

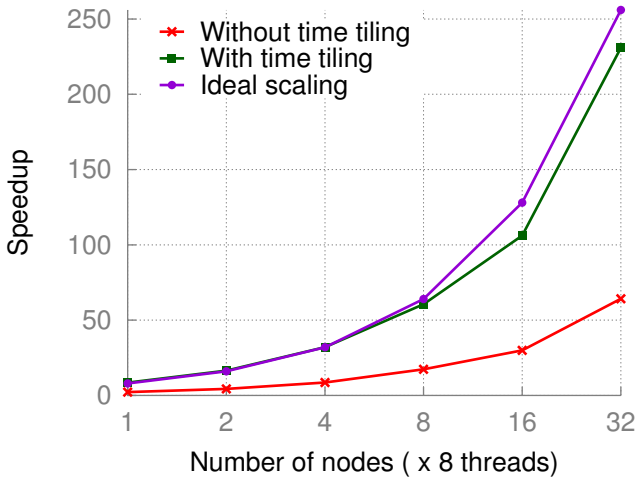


Figure 3.4: Performance of 2-d heat with and without time tiling

such as time tiling in HPF, as the owner of data keeps changing in every time step.

Hence, for our distributed memory compilation framework we choose the more flexible computer owns rule. We first determine a good computation placement and data is allocated according to the computation placement. In this approach, static, dynamic data distributions and array alignments are implicitly derived from the computation placement. Also, this approach enables us to model replication of data which is required to extract maximum parallelism.

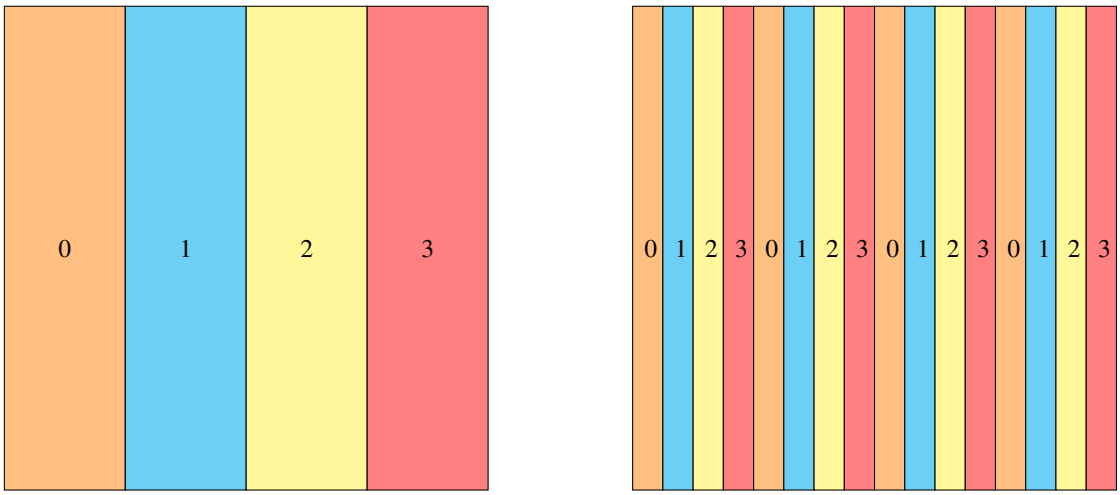
Chapter 4

Computation Placement

In this chapter, we describe how we find a suitable way of mapping available parallelism to a set of nodes. In particular, the presented strategy subsumes commonly used distributions like block, block-cyclic as well as more complex mapping schemes. The mappings are obtained for all parallel loop nests together.

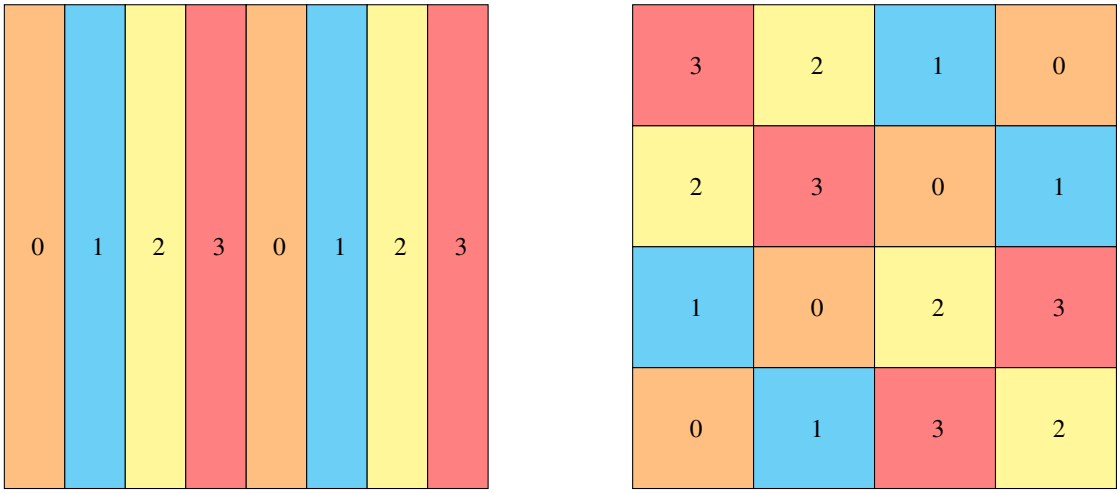
4.1 Common distribution patterns

In this section, we briefly describe commonly used distribution patterns – block, cyclic and block cyclic distributions. A *block* distribution (Figure 4.1a) distributes a set of iterations into equal or nearly equal contiguous chunks where the number of chunks is equal to the number of processors. A *cyclic* distribution (Figure 4.1b) distributes a set of iterations across processors in a round-robin manner at the granularity of a single iteration. When this granularity is changed to a contiguous chunk of some fixed size, the resulting mapping is a *block-cyclic* mapping (Figure 4.2a).



(a) Block distribution **(b)** Cyclic distribution

Figure 4.1: Distribution patterns



(a) Block cyclic distribution **(b)** Sudoku distribution

Figure 4.2: Distribution patterns

4.2 Sudoku mappings

We define another specialized mapping that we call a *sudoku* mapping due to its similarity with the popular number placement puzzle of the same name. A *sudoku* mapping assigns tiles from an n-dimensional view to processors in a way such that all tiles along any of the ($n-1$ dimensional) canonical hyperplanes are mapped on to distinct processors. One gets a perfect sudoku mapping when the iteration grid is a hypercube and a face has the same number of tiles as processors. We will see that such a mapping (Figure 4.2b) has interesting properties in minimizing communication if different pieces of computation require an array to be distributed in conflicting ways. Multipartitioning [8] implemented in dHPF is one possible perfect sudoku mapping and such a mapping was used in the manually parallelized versions of NAS BT and SP [13].

4.3 Motivating example

Consider sample ADI code shown in Figure 4.3 which is a representative version of NAS SP and NAS BT benchmarks. Figure 4.4 shows tiled and parallelized version of the representative ADI in Figure 4.4. The optimal computation mapping π_{S1} for the forward x sweep loop is the block distribution along ii loop iterations. This distribution leads to no communication and all nodes get equal number of iterations. Similarly, the optimal mapping π_{S2} for the y sweep is the block distribution of jj. However, these mappings are not optimal for the entire program as they demand a transpose of array X between the nests of $S1$ and $S2$, and thus a large amount of communication. Significantly better mappings exist and in this section, we will describe a technique to find such computation mappings automatically.

```
//forward x sweep
for (i=0; i<N; i++)
 for (j=1; j<N; j++)
  X[i][j] = X[i][j] - X[i][j-1] * A[i][j] / B[i][j-1]; //S1

//upward y sweep
for (j=0; j<N; j++)
 for (i=1; i<N; i++)
  X[i][j] = X[i][j] - X[i-1][j] * A[i][j] / B[i-1][j]; //S2
```

Figure 4.3: Sample ADI program with only forward x and y sweeps.

4.4 Finding computation placements

The computation mapping of a statement S_i, denoted by π_{S_i}, maps computation tiles to nodes. The chosen computation mappings have a significant impact on the execution time of a program. Two key factors to be considered while deciding computation mappings are communication volume and load balance. We call a computation mapping for a given program optimal if it leads to the lowest communication volume and perfect load balance. We model the problem of finding optimal computation mappings as a graph partitioning problem on the inter-tile communication graph (TCG). Each vertex in the TCG represents a computation tile of the program.

4.4.1 Communication volume minimization

An edge e is added between two vertices if and only if there is communication between the corresponding two tiles when they are executed on different nodes. The weight of the edge e_w will be equal to the communication volume between the two tiles. Finding the optimal computation mappings is equivalent to partitioning the TCG into p (number of nodes) equal size partitions with the objective to minimize the sum of those edge weights

that straddle partitions. Let E_b represent set of edges whose vertices are mapped onto different partitions.

$$obj = minimize \sum_{e \in E_b} e_w \tag{4.1}$$

This objective function represents the total communication volume for the entire program execution. The resulting computation mappings will thus have lower communication volume.

```
//forward x sweep
for (jj=0; jj<floord(N, 128); jj++) //serial loop
 for (ii=0; ii<floord(N, 128); ii++) //parallel loop
  for (i=max(1, ii*128); i<min(ii*128+127, N); i++)
   for (j=max(1,jj*128); j<min(jj*128+127, N); j++)
    X[i][j] = X[i][j] - X[i][j-1] * A[i][j] / B[i][j-1]; //S1

//upward y sweep
for (ii=0; ii<floord(N, 128); ii++) //serial loop
 for (jj=0; jj<floord(N, 128); jj++) //parallel loop
  for (j=max(1, jj*128); j<min(jj*128+127, N); j++)
   for (i=max(1, ii*128); i<min(ii*128+127, N); i++)
    X[i][j] = X[i][j] - X[i-1][j] * A[i][j] / B[i-1][j]; //S2
```

Figure 4.4: Tiled ADI program, tile size = 128

Figure 4.5 illustrates the tiled iteration domain along with RAW dependences for the tiled ADI example. A vertex is added to TCG for each of the tiles. Dependence edges that cross tile boundaries are used to determine the necessary communication sets and receiving tiles. For a given tile, an edge is added to each of its receiving tiles. The size of the communication set between sender and receiver tiles is set as the weight of edge between them. In the Figure 4.6a, from the tile $T0$ there are inter tile dependence to tiles $T3$ and $T9$. Hence, edges are added from $T0$ to its receivers $T3$ and $T9$. Since, there are 2 inter tile dependence edges from $T0$ to $T3$ edge weight is set to 2 for the edge between

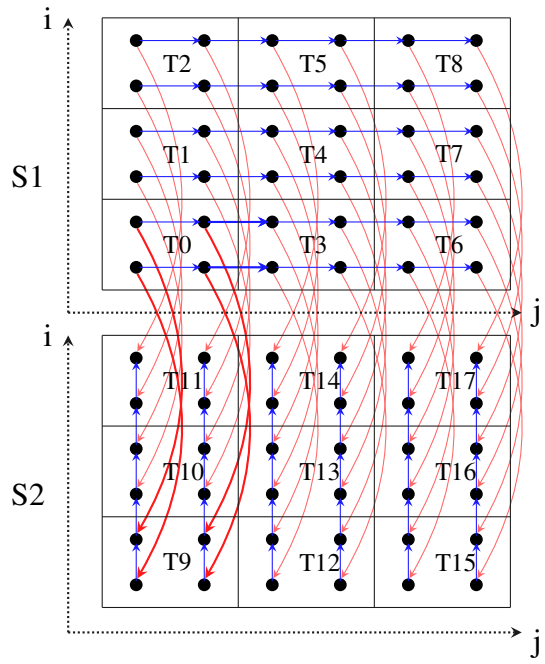
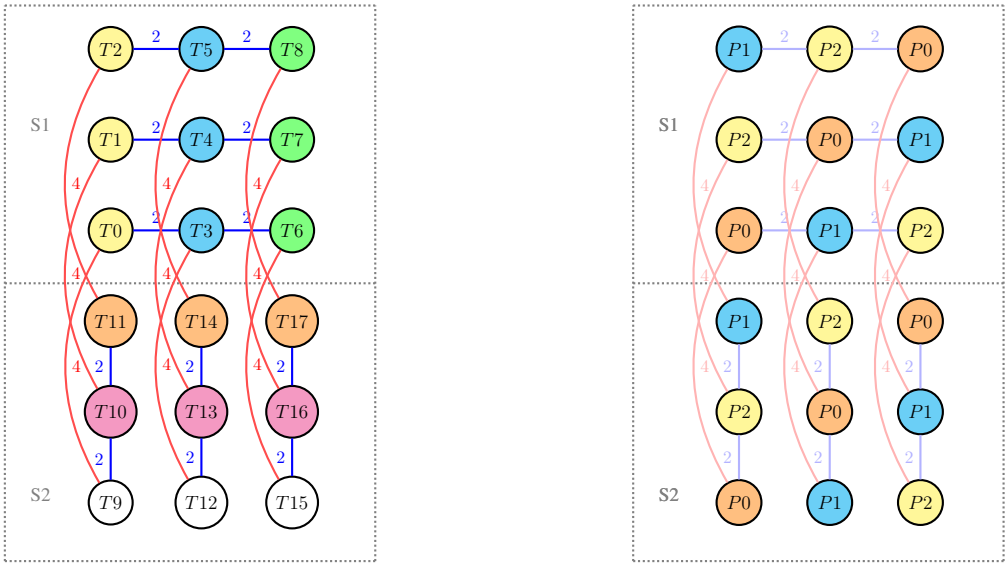


Figure 4.5: Tiled iteration space with dependences

them.

4.4.2 Load balancing constraints

A parallel phase is a contiguous band of parallel loops/dimensions that have been identified for potential extraction of parallelism. Programs often consist of multiple parallel phases. To achieve good load balance, it is essential that an equal or a nearly equal number of tiles are allocated to all nodes in each parallel phase. We identify the tiles that belong to a parallel phase and add constraints that will minimize load imbalance within each parallel phase. Vertex weights are used to distinguish between tiles that belong to different parallel phases. The vertex weight is a vector of size equal to the total number of parallel



(a) Inter-tile communication graph

(b) Computation distribution

Figure 4.6: TCG of ADI example

phases. All tiles which belong to the i^{th} parallel phase will have a vertex weight with the i^{th} component set and rest zero. Let S_i be the sum of the i^{th} vertex weight component of all vertices belonging to a single partition. For each vertex weight component, we add load balancing constraints that minimize the difference between S_i's of any two partitions. These constraints make sure that the resulting compute tile mapping will have an equal or a nearly equal number of tiles to each node in each parallel phase.

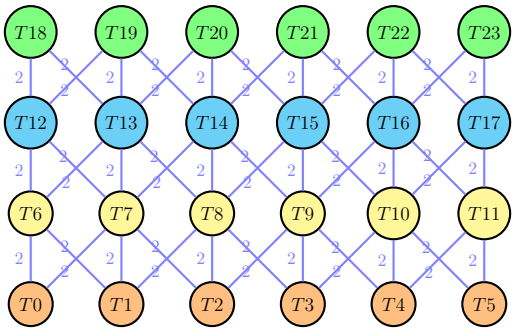
When performing static scheduling, all tiles belonging to a band of tiled parallel loops, for a given value of surrounding sequential loops, are said to belong to a single parallel phase. Figure 4.4 shows the tiled code for the ADI example. For the first loop, ii is the innermost tiled parallel loop. All the iterations of ii, for a particular value of outer sequential loop jj, belong to the same parallel phase. In case of dynamic scheduling, we do a topological sort of the TCG to identify tiles that belong to the same parallel phase – all

tiles at the same level belong to a single phase. For the ADI example in Figure 4.6a there are six parallel phases. All tiles belonging to a single parallel phase are marked with the same color. Tiles $T0$, $T1$ and $T2$ belong to same parallel phase and each will have vertex weights as $\langle 1, 0, 0, 0, 0, 0\rangle$. Similarly $T9$, $T12$ and $T15$ belong to same parallel phase and will have vertex weight as $\langle 0, 0, 0, 1, 0, 0\rangle$. The load balancing constraints ensure that Tiles $T0$, $T1$ and $T2$ are equally divided among all nodes.

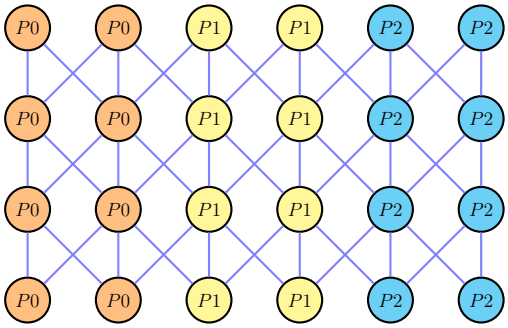
The above formulation finds partitions that have an equal number of tiles in each partition. If number of iterations in the tiles are not equal, assigning equal number of tiles to each partition could lead to load imbalance. To overcome this issue, we set number of iteration in the tile as the vertex weight component. Load balancing constraints on vertex weights ensure that each partition will have an equal number of iterations, in each parallel phase.

Figure 4.6b shows one of the optimal solutions for graph partitioning of the ADI example which is a 2-dimensional sudoku mapping. In each parallel phase, equal number of tiles are assigned to all the nodes, which ensures perfect load balance. Computation mapping is identical for both $S1$ and $S2$. Computation tile $\langle ii, jj\rangle$ of $S1$ and $\langle ii, jj\rangle$ of $S2$ are mapped to the same node, thus eliminating communication between $S1$ and $S2$. Figure 4.7b shows the computation mappings for stencil programs with near-neighbor communication, which is exactly the block distribution. Figure 4.8b shows the obtained computation mappings for tapered iteration spaces. This mapping is slightly different from block-cyclic mapping. In this mappings, first and last columns are mapped to node $P0$, whereas in block-cyclic mappings first and fourth columns will be mapped to node $P0$. The obtained mapping has slightly better load balance than block-cyclic distribution, as equal number of tiles are allocated to all the nodes.

The graph partitioning solution of the TCG also determines the optimal dimensionality of the computation mapping. Note that a higher dimensional mapping is not necessarily



(a) TCG with near-neighbor communication



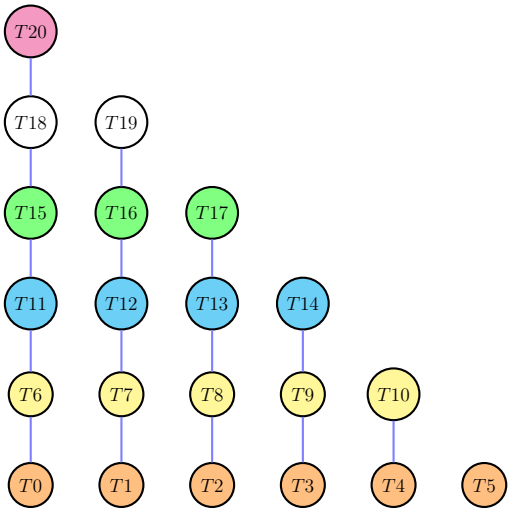
(b) computation placement with near-neighbor communication

Figure 4.7: Stencil with near-neighbor communication

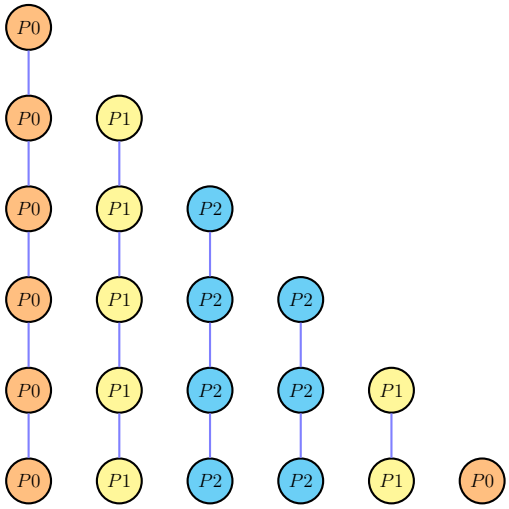
optimal for an entire sequence of loop nests being optimized. Consider a program with the first loop nest (forward x sweep) of ADI. The TCG of this program contains only the upper half of Figure 4.6a with just nodes of $S1$. The optimal computation mapping for this graph is a 1-d block distribution of ii loop. Similarly, for the lower half with the second loop nest (upward y sweep), the optimal computation mapping is a 1-d block distribution of the jj loop. Our graph partitioning approach is able to find these solutions.

4.5 Scalability of the solution

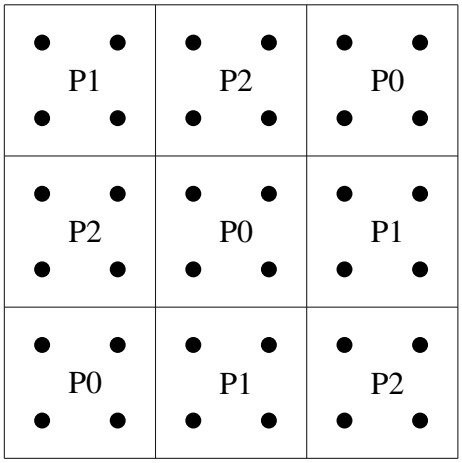
Graph partitioning with load balancing constraints is an NP-hard problem. Solutions to these problems are generally derived using heuristics and approximation algorithms. Open-source software packages such as METIS [14] and SCOTCH [15] can be used to solve graph partitioning problems. As the problem size increases, the number of vertices and edges in the graph also increase. The number of load balancing constraints also increases as we add load balancing constraints to each parallel phase, and this depends on the problem size. Even state-of-the-art graph partitioning software such as these do not



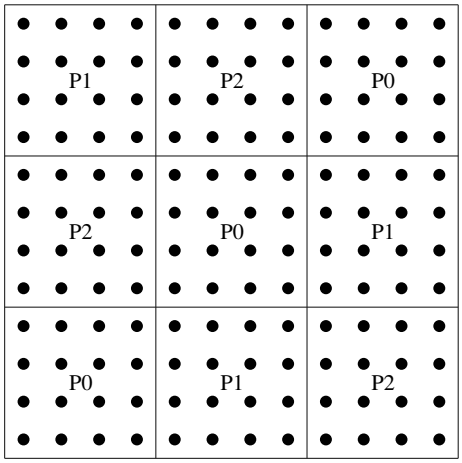
(a) TCG for unbalanced computation



(b) Computation placement for unbalanced computation



(a) Representative mapping



(b) Scaled mapping (dot is tile)

Figure 4.9: Scaling a computation mapping for ADI

scale as the problem size increases. For example, to partition a TCG of ADI with 64 vertices into 4 partitions, METIS takes around 240s. If the problem size is increased further, both the time taken and the memory required for partitioning increases drastically. Another major problem is that the quality of the obtained solution degrades as the problem size increases. For the ADI example, we observe that perfect "sudoku" mappings are not obtained for more than 32 vertices.

4.5.1 Graph partitioning algorithms

In most of these packages [16, 17, 18], graph partitioning is done in three phases, namely, graph coarsening, initial partitioning and uncoarsening/refining. In the graph coarsening phase, a smaller graph is derived from the input graph by collapsing together a set of adjacent vertices. This process is continued until the size of the graph is reduced to just a few hundred vertices. Now, in the initial partitioning phase, this coarsened graph is partitioned using brute force or relatively simple approaches. This step is very fast since the size of the graph is very small. Finally, in uncoarsening/refining phase, the partitioning of smaller graph is projected to successively larger graphs. The projected partitioning is refined using various heuristics, which will iteratively move the vertices between partitions, as long as such moves improve the quality of partitioning. This process is continued until the partitioning solution is projected and refined to the original graph.

4.5.2 Our approximation

In order to make our approach scalable for larger problem sizes, we use an approximation to partition the TCG. Note that the dependence patterns typically do not change as the problem size is increased beyond a certain point. Hence, the optimal mappings for a larger problem size can often be obtained by scaling the optimal mappings for a smaller one. The

computation mappings for larger problem sizes and the actual number of processors are derived from the computation mappings for a smaller problem size and number of processors. At compile time, we build the TCG for a smaller problem size. Problem sizes are chosen such that the number of vertices in each parallel phase is a particular number that is sufficient to distinguish the nature of the obtained mapping. The number of processors is fixed at four which we found to be sufficient in practice, and the problem size is set so that we have at least two times the number of processor tiles along each parallel dimension. This allows us to distinguish between block and block-cyclic mappings. The edges weights and vertex weights are computed for this smaller graph, and this is partitioned using METIS. This step is very fast owing to a very small graph. At runtime, when the actual problem sizes and number of nodes are known, the partitioning solution is derived from the solution of the smaller representative graph. The mapping obtained is first classified as either being block, block-cyclic, sudoku or an arbitrary one. If a mapping is identified as block, sudoku, or arbitrary, then we perform a "block" scaling of the mapping for the right problem size and the number of processors. This is illustrated in Figure 4.9 which shows the computation mapping obtained for larger problems sizes for the ADI example. We then generate a function that returns the correct mapping for any given number of nodes.

For block and sudoku mappings, block scaling ensures that the respective property continues to hold. For arbitrary mappings, we find the block scaling to be a reasonable approach though we have not seen cases where we found arbitrary mappings. More precisely, let N_{in_i} be the total number of tiles in the i^{th} distributed dimension of the input problem and N_{rep_i} the number of tiles in that of the smaller representative problem. Then, block scaling for π for a larger problem size is done as follows:

$$\pi_{in}(i,j) = \pi_{rep_i}(i/(N_{in_i}/N_{rep_i}), j/(N_{in_j}/N_{rep_j})). \tag{4.2}$$

When the solution obtained through graph partitioning corresponds to a cyclic or a block-cyclic mapping, we perform a cyclic scaling analogous to the block scaling described above. Overall, this approximate approach of using a representative graph and then scaling the solution analytically based on an identified template mapping does not take more than a second on any of the examples considered for evaluation. Actual communication costs finally depend on network topology – finding computation mappings that are optimal for a given network topology is beyond the scope of this work and is left for future.

Chapter 5

Data Allocation and Management

In this chapter, we present techniques to allocate data according to already found computation mappings. Recall from the previous chapter that the complete computation mapping itself is determined partly at compile time and partly at runtime, i.e., just before start of computation. Since a compute tile is our unit of mapping, we determine the data required to execute a particular compute tile, and allocate just that data at the node executing it.

Data accessed by a computation tile depends on the array access functions in the loop statement. It could be a contiguous block of array for ADI example 5.1, a row and a column for `floyd-warshall` as shown in Figure 5.2 or a diagonal if the array access is $X[i][i]$. There are different techniques to allocate only the required data on a node. In PGAS model data is allocated at the granularity of pages. This scheme leads to lot of redundant data allocation when the data that needs to be allocated, is not contiguous (Figure 5.2). Another approach is to allocate the bounding box (enclosed rectangle) of the required data. Even this scheme leads to inefficient allocation in case of diagonal data region (Figure 5.3).

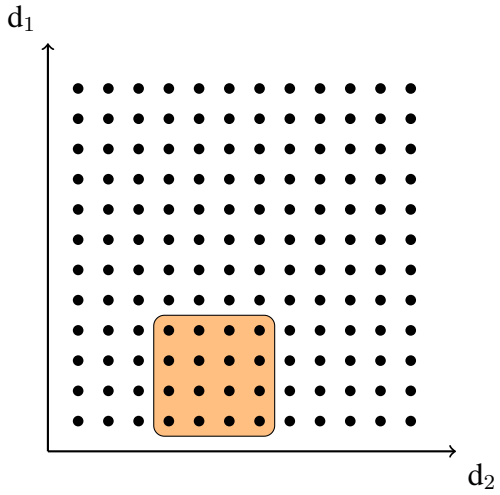


Figure 5.1: Data accessed by compute tile (1,0) due to array access $X[i][j-1]$ in ADI example

5.1 Data tiling

Our approach for data allocation is to tile the data space, similar to tiling an iteration space, compute the data required by a compute tile at the granularity of data tiles, and allocate only these required data tiles on-demand on the node executing it. This scheme leads to an efficient data allocation for all the above cases.

5.1.1 Data tiling hyperplanes

In order to tile the data space, we need to find the data tiling hyperplane compatible with the compute tiling hyperplanes. In this section, we describe techniques to find the shape of the computation and data tiles. We determine the shape of computation and data tiles such that the data accessed by a computation tile is packed into as few data tiles as possible.

Let $S_1, S_2, \ldots, S_n$ be the statements of a program. Let D_{S_j} be the domain of S_j. Let

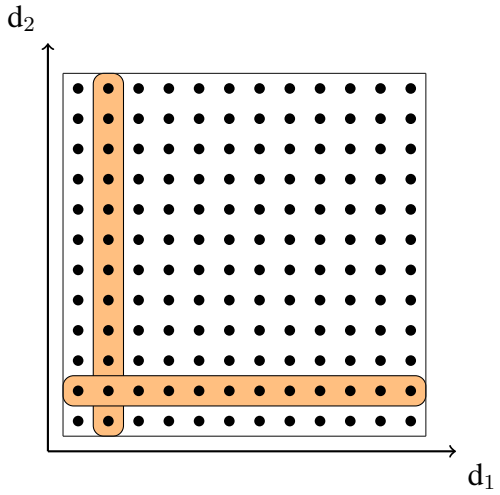


Figure 5.2: Row and column data access

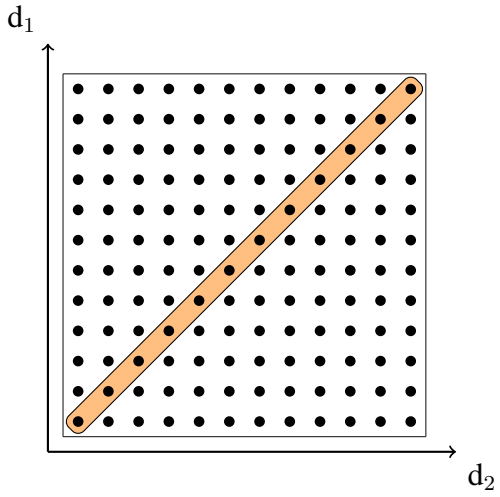


Figure 5.3: Diagonal data access due to $X[i][i]$

E be the set of dependence edges with each edge $e \in E$ characterized by a dependence polyhedron P_e. P_e is a set of linear constraints that relate source iterators and target iterators that are in dependence. A one-dimensional affine transformation for S_j, denoted by ϕ_{S_j}, is defined as

$$\phi_{S_j}(\vec{i}) = \vec{h}^{S_j} \cdot \vec{i} + h_0^{S_j}. \tag{5.1}$$

ϕ_{S_j} can be viewed as a function, mapping iterations of S_j to numbers that represent virtual processor ids. For example, $\phi_{S_j}(\vec{i}) = (1,0)^T.\vec{i} + 0$ maps all iterations (i,j) to virtual processor i. ϕ_{S_j} partitions the iteration space of S_j, and $\vec{h} = (1,0)$ represents the orientation of the hyperplane that partitions it. When ϕ_{S_j} satisfies certain properties, we call it a tiling hyperplane.

Similarly, for arrays we define an array mapping function ψ_{A_k} that maps array elements to virtual processors represented by

$$\psi_{A_k}(\vec{a}) = \vec{d}^{A_k} \cdot \vec{a} + d_0^{A_k}, \tag{5.2}$$

where $\vec{d}$ represents the orientation of the hyperplane that partitions the array space, d_0 is the constant offset, and $\vec{a}$ is a data element in A_k. We call these mappings data tiling hyperplanes if they are found to satisfy certain properties that we will describe later in this section.

Consider the first loop nest of the ADI example shown in Figure 4.3. Assume that the computation mapping for $S1$ is $\phi_{S_1}(\vec{i}) = (1,0)^T \cdot \vec{i}$. Different iterations of the i loop will be mapped to different virtual processors. For the array access $X[i][j]$, different iterations of i loop access different rows of array X. Hence, the first dimension of X has to be partitioned. This corresponds to the array mapping $\psi_X(\vec{a}) = (1,0)^T \cdot \vec{a}$, which in turn corresponds to a row distribution of X. If the array access had been $X[j][i]$, then for $\phi_{S_1}(\vec{i}) = (1,0)^T \cdot \vec{i}$, the corresponding data mapping would be $\psi_X(\vec{a}) = (0,1)^T \cdot \vec{a}$, i.e., a

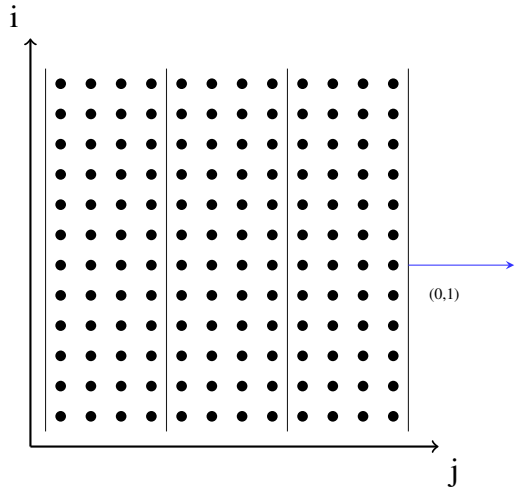


Figure 5.4: Partitioning the iteration space with (0,1) hyperplane

column-wise distribution of X. Hence, the choice of data partitioning hyperplanes, for a given computation mapping, is driven by the array accesses.

Let $F^i_{s_j,A_k}$ be the i^{th} access function of array A_k in statement S_j. In our model, F is an affine function of loop iterators and program parameters. $\phi_{s_j}(\vec{i})$ is the virtual processor to which iteration $\vec{i}$ will be mapped. $F^i_{s_j,A_k}(\vec{i})$ represents data accessed by $\vec{i}$. $\psi_{A_k}(F^i_{s_j,A_k}(\vec{i}))$ is the virtual processor to which the data accessed by $\vec{i}$ will be mapped. Now, we require that the data accessed by $\vec{i}$ be mapped to the same virtual processor as the one $\vec{i}$ is mapped to, i.e.,

$$\phi_{s_j}(\vec{i}) = \psi_{A_k}(F^i_{s_j,A_k}(\vec{i})). \tag{5.3}$$

The above condition is conceptually the same as that used by Anderson and Lam [19], but it was in a form that worked only for perfect loop nests and uniform dependences, and hence the subsequent approach relying on it was also different. It is not always possible to ensure condition (5.3) since multiple iterations can access the same data. Hence, what

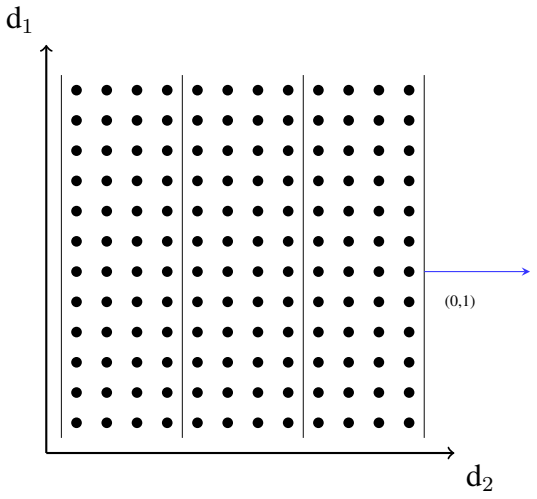


Figure 5.5: Partitioning the data space of X with (0,1) hyperplane for access $X[i][j]$

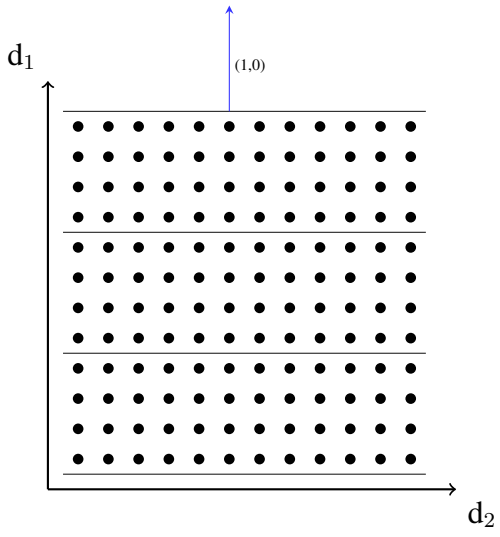


Figure 5.6: Partitioning the data space of X with (1,0) hyperplane with array access $X[j][i]$

we try to capture is the difference between the LHS and RHS of (5.3) as follows and try to minimize it:

$$\gamma^{i}_{s_j,A_k}(\vec{i}) = |\phi_{s_j}(\vec{i}) - \psi_{A_k}(F^{i}_{s_j,A_k}(\vec{i}))| \;\; \vec{i} \in D_{s_j}. \tag{5.4}$$

The above definition is thus used to connect compute and data tiling hyperplanes.

We first describe the existing technique to characterize and choose from valid compute tiling hyperplanes, and then show how data tiling constraints and objectives are integrated into it to minimize γs. Previous work [2] provided an automatic approach to find compute tiling hyperplanes that exposed maximal course-grained parallelism and locality based on an Integer Linear Programming formulation. To enforce validity for tiling for an edge e with dependence polyhedron P_e, the following constraint ensures non-negative dependence components that is sufficient for tiling:

$$\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \geq 0, \;\; \langle \vec{s}, \vec{t} \rangle \in P_e. \tag{5.5}$$

Then, the following cost function has been used to select the best hyperplane among the set of valid tiling hyperplanes.

$$\delta_e(\vec{t}, \vec{s}) = \phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}), \;\; \langle \vec{s}, \vec{t} \rangle \in P_e \tag{5.6}$$

This cost function is a measure of communication volume or reuse distance – and the ILP is solved to minimize it.

We now add data tiling hyperplane coefficients to the ILP described above and use (5.4) to relate compute and data tiling hyperplane coefficients. Equation (5.4) cannot be directly added to ILP formulation as it leads to non-linear constraints between hyperplane coefficients and loop iterators. The bounding function technique [20] can again be used here to obtain constraints in a linear form. When the iteration spaces are bounded, one

can obtain an upper bound on $\gamma^i_{s_j,A_k}(\vec{i})$. The maximum mismatch quantified by γ occurs when the iterations and the entire data accessed by them are mapped onto different virtual processors. This mismatch can be bounded by an affine function of program parameters $\vec{p}$, i.e., there exists $v_{A_k}(\vec{p}) = u_{A_k}.\vec{p} + w$ such that

$$v_{A_k}(\vec{p}) - \gamma^i_{s_j,A_k}(\vec{i}) \geq 0, \;\; \vec{i} \in D_{s_j} \tag{5.7}$$

By minimizing the bounding function coefficients $\vec{u}_{A_k}$, we indirectly minimize (5.4). Now the affine form of the Farkas lemma can be applied to (5.7).

$$v_{A_k}(\vec{p}) - \gamma^i_{s_j,A_k}(\vec{i}) \equiv \lambda_{s_j,A^i_{k0}} + \Sigma_t \lambda_{s_j,A^i_{kt}}(a_t\vec{i} + b_t), \tag{5.8}$$

where $\lambda_{S_j,A^i_{kt}} \geq 0$ are the Farkas multipliers and $a_t\vec{i} + b_t \geq 0$ are the faces of D_{s_j}. The coefficients of $\vec{i}$ and $\vec{p}$ in the resulting equations are eliminated using Fourier-Motzkin elimination to obtain constraints in a linear form.

The resulting ILP system with tile validity conditions (5.5), cost function constraints (5.6) and data hyperplane constraints (5.7) is solved using PIP [21] to get both compute and data tiling hyperplanes. PIP computes the lexicographical minimal solution for the ILP. Hence, the order of variables is important. We add separate bounding function coefficients for each of the arrays to ensure that non-zero bounding coefficients of one array do not affect the choice of data hyperplanes for the other arrays. If there is a mismatch between computation and data tiling hyperplanes, then the bounding function coefficients of (5.7) will be non-zero. If same bounding function coefficients are used for all arrays, then the mismatch between computation and data hyperplanes for a single array may lead to sub-optimal data tiling hyperplane for other arrays. Let u_s, w_s be the bounding function coefficients from the Equation (5.6), u_{A_k}, w_{A_k} be the bounding function coefficients for

array A_j resulting from (5.7), and h_{s_j} be the vector of compute hyperplane coefficients, and d_{A_k} that of the data tiling hyperplane coefficients. Equation (5.9) shows the order of variables used for the lexicographic minimal solution:

$$min_{\prec}(u_s, u_{A_1}, \ldots, w_s, w_{A_1}, \ldots, h_{s_1}, \ldots, d_{A_1}, \ldots) \tag{5.9}$$

5.1.2 Validity of data tiling hyperplanes

Consider the example shown in Figure 5.7. For compute tiling hyperplane (1,0), there is no need to tile arrays $v1$ and $v2$, as all iterations of i loop access entire $v1$ and $v2$ arrays. Once we solve the ILP, we obtain a compute tiling hyperplane for each of the statements, ϕ_{S_j}, and a data tiling hyperplane for each of the arrays, ψ_{A_k}. A data tiling hyperplane ψ_{A_k} is considered to be invalid for all $F^i_{S_j,A_k}$ if the ϕ_{S_j} lies in the union of the null spaces of all array access functions, $F^i_{S_j,A_k}$. If this condition is satisfied, then all iterations of S_j access the entire array A_k. Hence, it is not necessary to tile the array space even if the iteration space of S_j is tiled with ϕ_{S_j}. It is necessary to tile the data space only when different partitions of the iteration space, due to ϕ_{S_j}, access different parts of array A_k.

```
for (i=0; i<N; i++)
  for (j=1; j<N; j++)
    B[i][j] = A[i][j] + u1[i]*v1[j] + u2[i]*v2[j];
```

Figure 5.7: First loop nest of gemver

Algorithm 1: Finding compute and data hyperplanes

Input: Data dependences (E), array access functions ($\mathbf{F}$)

Output: $\phi_{S_j} \ \forall S_j$, $\psi_{A_k} \ \forall A_k$

$C \leftarrow \emptyset$;

valid_hyperplanes_$S_j \leftarrow \emptyset \ \ \forall S_j$;

valid_hyperplanes_$A_k \leftarrow \emptyset \ \ \forall A_k$;

num_valid_hyperplanes_$S_j \leftarrow 0 \ \ \forall S_j$;

num_valid_hyperplanes_$A_k \leftarrow 0 \ \ \forall A_k$;

max_num_hyperplanes $\leftarrow \max(dim(S_j)) \ \forall S_j$;

for each $e \in E$ *within fused loops* **do**

    Add validity constraints resulting from $\phi_{s_j}(\vec{t}) - \phi_{s_i}(\vec{s}) \geq 0$ to C

for each $e \in E$ **do**

    Add the bounding function constraints resulting from $|v(\vec{p}) - \delta_e(\vec{t}, \vec{s})| \geq 0$ to C;

for each $F^i_{s_j,A_k} \in \mathbf{F}$ **do**

    Obtain constraints resulting from $\phi_{s_j}(\vec{i}) - \psi_{A_k}(F^i_{s_j,A_k}(\vec{i})) \geq 0$ and $\psi_{A_k}(F^i_{s_j,A_k}(\vec{i})) - \phi_{s_j}(\vec{i}) \geq 0$ to C;

while *max_num_hyperplanes* $\geq$ *0* **do**

    Solve the ILP with constraints in C;

    max_num_hyperplanes $\leftarrow$ max_num_hyperplanes - 1;

    for each ϕ_{S_j} *found* **do**

        if *num_valid_hyperplanes_*$S_j <$ *dim(*S_j*)* **then**

            Add ϕ_{S_j} to valid_hyperplanes_S_j;

            num_valid_hyperplanes_S_j++;

            Add constraints to exclude hyperplanes linearly dependent on ϕ_{S_j};

    for each ψ_{A_k} *found* **do**

        if ψ_{A_k} *is a valid data tiling hyperplane* **then**

            if *num_valid_hyperplanes_*$A_k <$*dim(*A_k*)* **then**

                Add ψ_{A_k} to valid_hyperplanes_A_k;

                num_valid_hyperplanes_A_k ++;

                Add constraints to exclude hyperplanes linearly dependent on ψ_{A_k};

5.1.3 Optimal hyperplanes across non fused loops

Previous work [2] finds the compute tiling hyperplane by applying the validity and bounding constraints for all dependences within a band of fused loops. Compute tiling hyperplane of each of the fused loop bands are computed independent of each other. The computation and data tiling hyperplanes found this way may be suboptimal for a program that consists of multiple fused loop bands. In the distributed memory context, since data is distributed across multiple compute nodes, the compute and data hyperplanes found this way may lead to excess communication across non fused loops. This can happen when two non fused loops access the same array, and compute and data tiling hyperplanes of each fused loop enforce a different data tile mapping. In order to find the hyperplane that leads to minimum communication volume across entire program, we add the bounding function constraints, specified in equation (5.6), even for the dependences across non fused loops. These dependences capture the resulting communication volume across non fused loops, hence, by bounding and minimizing these dependences, we are finding the hyperplanes that leads to minimum communication volume for the entire program.

5.1.4 Iteratively finding all hyperplanes

A statement can have as many compute tiling hyperplanes as the dimensionality of its iteration space. Similarly, an array can have as many data tiling hyperplanes as its dimensionality. Solving the ILP formulation described in the previous section gives us a single compute tiling hyperplane for all statements and a single data tiling hyperplane for all arrays. We add new constraints to the ILP to ensure that subsequent compute tiling hyperplanes are linearly independent of ones already found. However, for data tiling hyperplanes, linear independence constraints are only added with respect to those that were found to be valid. Algorithm 1 describes the complete procedure to find all compute and

data tiling hyperplanes. For the ADI example 4.3, the first compute hyperplane for $S1$ and $S2$ is (1,0), corresponding data tiling hyperplane for arrays X, A and B is (1,0). The second compute hyperplane for $S1$ and $S2$ is (0,1), corresponding data tiling hyperplane for arrays X, A and B is (0,1). Since, each of the data tiling hyperplanes are linearly independent of each other, they form a full rank matrix.

5.2 Data allocation

In this section, we describe how data is indexed and managed once data tiling hyperplanes for each array have been determined. The data accessed for array A_k through access function F in S_j can be computed by taking the image of the D_{S_j} under F. However, we are interested in computing data accessed by a particular compute tile. This enables us to allocate only the data required for the tile on the node it executes on.

5.2.1 Data accessed by a compute tile

Data accessed by a compute tile is determined by computing the image of the access function while treating dimensions outer to the tile, that we call inter-tile iterators, as parameters. The resulting image will be a set, parametric in the inter tile iterators. By plugging in a particular value for these inter tile iterators, precise data accessed by that particular compute tile can be obtained. The shaded rectangle in Figure 5.8 shows the parametric data region of compute tile (1,0) due to the array access $A[i][j-1]$.

When the data tiling hyperplanes are used, the parametric data regions obtained above end up getting tiled as well in the same way iteration spaces are tiled. Same tile sizes are used when tiling the iteration space using compute tiling hyperplanes and the parametric data region using the corresponding data tiling hyperplane. After tiling, the dimensions of

the parametric data region include the newly added inter data tile iterators, intra data tile iterators and the inter compute tile iterators. For a particular value of compute tile iterators, inter data tile iterators enumerate all data tiles accessed, and intra data tile iterators scan data points inside a data tile.

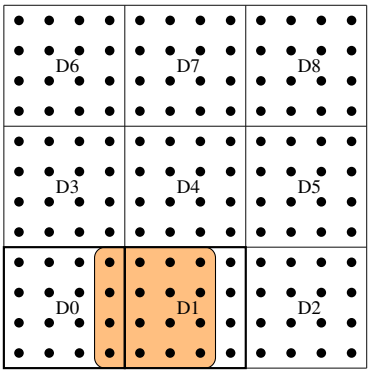


Figure 5.8: Accessed data for compute tile (0,1) for $A[i][j-1]$ in ADI

5.2.2 On-demand data tile memory allocation

Projecting out the inter data tile iterators, we get parametric polyhedron that can be used to enumerate all the data tiles that a compute tile accesses. We use this polyhedron to generate a function that will allocate memory for the data tiles required by a given compute tile. This function will be called just before the execution of a compute tile. This will ensure that data required by a compute tile is allocated only on that node which will execute the compute tile. For the ADI example generated function returns data tiles $D0$ and $D1$ for the compute tile (0,1), and only $D0$ for the compute tile (0,0).

Algorithm 2: Determine accessed data tiles for array A_k

Input: Parametric data region D_{A_k} of array A_k, data tiling hyperplanes ψ_{A_k}, tile sizes τ_k

Output: Parametric data tiles accessed

1 **for each** *data tiling hyperplane* $\psi_{A_k}(\vec{a}) = \vec{d_k} \cdot \vec{a} + d_{k0}$*, tile size* τ_k **do**

2 add an inter data tile dimension $\vec{a_T}$, corresponding to $\vec{a}$

add the following two constraints to D_{A_k}

$\tau_k * (\vec{d_k} \cdot \vec{a_T}) \leq \vec{d_k} \cdot \vec{a} + d_{k0} \leq \tau_k * (\vec{d_k} \cdot \vec{a_T}) + \tau_k - 1$

3 Project out the intra-tile data dimensions $\vec{a}$ in D_{A_k}

return D_{A_k}

5.2.3 Allocation of first-read data

Input data of the program being compiled has to be initialized and distributed before start of program execution. We thus also need to allocate that part of the input data that is "live in" to the compute tiles to be executed on that node. We call this the first-read data. First-read data of an array A is the set of all array elements whose values are first read by a compute tile before a write is performed if at all. This set is identified by computing the data accessed by all read accesses, and then subtracting out data accessed by target iterations of RAW dependences entering the tile.

5.3 Data tile buffer reuse

Programs access different parts of the array during the entire execution of the program. Consider the `floyd-warshall` kernel shown in Figure 5.9. For a particular outer k loop iterator, we need to allocate k^{th} row (due to $X[k][i]$) and k^{th} column (due to $X[j][k]$) of array X on a particular node. Hence, for all the iterations of k loop we would end up allocating entire array X on a single node. Often, we can reuse the data tile buffers, rather

```
for(k=0;k<N;k++)
 for(i=0;i<N;i++)
  for(j=0;j<N;j++)
X[i][j]=f(X[i][k],X[k][j]);
```

Figure 5.9: Floyd-Warshall kernel

than allocating new buffers for every new data tile. For the `floyd-warshall` example shown in Figure 5.10, we can reuse the data tile buffers $D0$, $D3$ and $D6$ after $k = 3$ iteration for data tile buffers $D1$, $D4$ and $D7$ which will be allocated in $k = 4$ iteration.

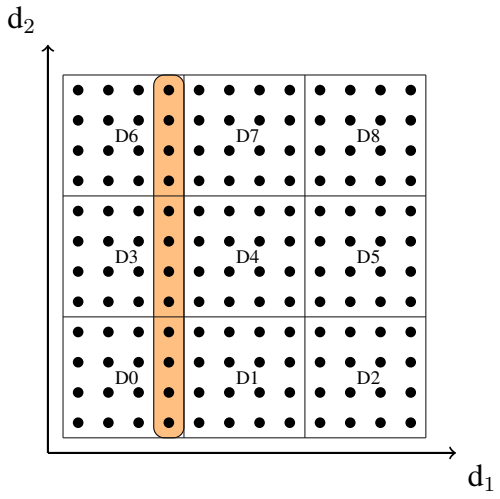


Figure 5.10: Data tiles allocated due to $X[k][j]$ for k = 3

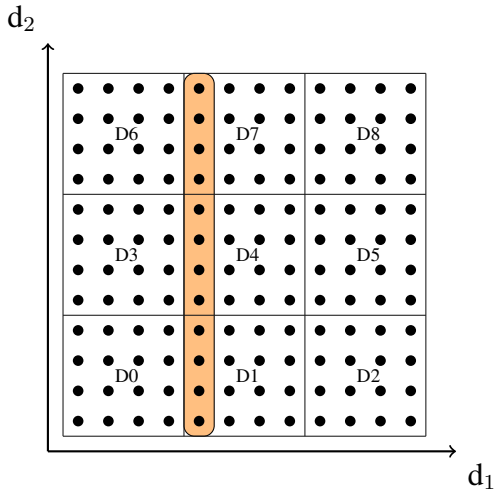


Figure 5.11: Data tiles allocated due to $X[k][j]$ for k = 5

A data tile buffer can be safely reused when all the compute tiles that require this data tile have finished their execution. We can precisely count the number of compute tiles that require a given data tile. Per data tile ref-count is used to capture number of compute tiles that need this data tile. We generate a function that will enumerate all the compute tiles executed by the given processor and use Algorithm 2 to get all the data tiles accessed by tile, and increment their ref-count. This function is invoked at the start of the program. Once the compute tile has finished its execution and data required by other tiles is packed, we decrement the ref-count of all the data tiles accessed by this compute tile. If the ref-count becomes zero we add these data tile buffer pointers to free-buffers queue. When we want to allocate a new buffer for another data tile, the free-buffer queue is checked fist, if it is non empty, one of its buffers is returned. A new allocation is done only when free-buffer queue is empty. Per data tile ref-count is used to track the liveness information of data tiles. Since, we use dynamic scheduling, actual schedule will be decided at runtime. Above techniques provide an efficient, dynamic and schedule independent mechanism for

data tile buffer reuse.

Algorithm 3: initialize_ ref_counts ()

Input: Set of all compute tiles

Output: data tile ref-counts

```
1 for each data tile d⃗ do
2     ref_count_ d⃗ ← 0;
3 for each compute tile t⃗ do
4     if π(t⃗) == node_id) then
5         required_data_tiles ← determine required data tiles (t⃗);
          for each data tile d⃗ ∈ required_data_tiles do
6             ref_count_d⃗ ++;
```

Algorithm 4: decrement ref-counts ()

Input: compute tile $\vec{t}$

Output: data tile ref-counts

```
1 required-data-tiles ← compute required data tiles (t⃗)
  for each data tile d⃗ of required-data-tiles do
2     decrement ref-count of d⃗
3     if ref-count of d⃗ equal to 0 then
4         add data tile buffer of d⃗ to free-buffer-queue
```

Algorithm 5: buffer allocate ()

Input: data tile $\vec{d}$

Output: data tile buffers

```
1 if free-buffer-queue is non-empty then
2     new-buffer ← dequeue from free-buffer-queue
3 else
4     new-buffer = allocate new buffer
5 return new-buffer
```

5.3.1 Ensuring thread safety

Multiple threads could simultaneously allocate data tile buffers, increment or decrement data tile ref-counts and reuse data tile buffers. We need to ensure that all the above operations are thread safe. We use a concurrent queue to maintain a list of free data tile buffers. Atomic increment and decrement are used to modify data tile ref-counts. Atomic compare-and-swap operations are used when the data tile buffer pointers are updated. Thus, the whole implementation is thread-safe and lock-free. We use lock free concurrent queue and atomic compare-and-swap from Intel TBB library [22] for our implementation.

5.4 Data tiling with dynamic scheduling

We have integrated data tiling with a dynamic scheduling framework which schedules at the granularity of compute tiles. First, we determine the computation mappings π_S for a given problem size and the number of nodes. The dynamic scheduling runtime distributes compute tiles according to π_S. The read-in data is allocated and data tile reference counts are initialized as per π_S mappings. All compute tiles that are mapped to a single node

are dynamically scheduled. Before start of tile execution, we call the on-demand allocate function (Algorithm 2), which will allocate all required data tiles if they had not been already allocated. After the tile has finished execution, we call the pack function which packs data required by other nodes from the current node. Packed data is sent to its receive nodes using asynchronous MPI primitives. Once the pack function is finished, we decrement the ref-counts of the data tiles used. Besides being called at schedule time, the on-demand allocate function is also called before data received from other nodes is unpacked.

5.5 Re-indexing data spaces

After we perform data tiling transformation, the memory layout of the arrays is changed. Array elements within the data tile are now packed in contiguous memory locations. So we need to modify the original array access functions such that they access correct elements in the new memory layout. The dimensionality of array A is double because of the new tile dimensions that are added. To ensure correctness, there should be a one-to-one mapping between original array dimensions and new tiled dimensions. We use following equation to obtain new array accesses from original array accesses.

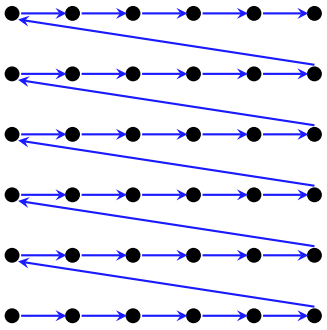


Figure 5.12: Original memory layout

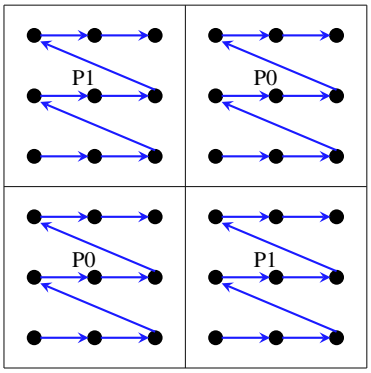


Figure 5.13: Tiled memory layout

$$\begin{aligned} \overrightarrow{T} &= (\psi_{A_k} \cdot \vec{a})/\tau_k \\ \vec{t} &= (\psi_{A_k} \cdot \vec{a})\%\tau_k \end{aligned} \tag{5.10}$$

where $\overrightarrow{T}$ is the access vector corresponding to inter-tile dimensions, $\vec{t}$ to intra-tile dimensions, ψ_A is the data tiling hyperplane, and τ is the tile size. Since array tiling hyperplanes form a full-ranked matrix, a one-to-one mapping exists between original and new array accesses. $\vec{T}$ represents inter data tile dimensions which enumerates data tiles and $\vec{t}$ scans points inside a data tile. Array access $X[i][j]$ will be transformed into $X[ii_t][jj_t][i_t][j_t]$. If the data tiling hyperplanes used are (1,0) and (0,1) and tile sizes τ_1, τ_2, the new array access will be
$A[i/\tau_1][j/\tau_2][i\%\tau_1][j\%\tau_2]$.
The size of the data tile that is to be allocated is $\tau_1 * \tau_2$. This mapping is exact for data hyperplanes which have only one non-zero component, i.e., all points in the new array layout will have corresponding points in the original layout. If the data tiling hyperplanes used are (1,1) and (0,1) and the tile size is τ, the new array access as per Equation (5.10) is $A[(i+j)/\tau_1][j/\tau_2][(i+j)\%\tau_1][j\%\tau_2]$. The size of the data tile that needs to be allocated is

$(\tau_1 + \tau_2) * \tau_2$. Hence, even if the data tiling hyperplane has more than one non-zero component, for example (1,1), we still obtain a correct one-to-one mapping but, we would end up allocating more memory. We choose this mappings in spite of it not being exact due to the simplicity of resulting new access expressions. With this mapping of Equation (5.10), we always have either a mod or a divide of an entire expression – this enables us to simplify access expressions (Section 5.5.1). There are more accurate mappings in the literature that provide the exact data remappings, but with a complicated access expressions. We cannot apply these optimizations (Section 5.5.1) for complex mappings and will incur significant runtime overhead. Hence, for the hyperplanes with more than one non zero component, we trade off memory for performance.

To selectively allocate only the required data tiles, we split the transformed array access $A_t[ii_t][jj_t][i_t][j_t]$ into two parts, (i) $ptr = A_t[ii_t][jj_t]$ which returns the pointer to data tile (ii_t, jj_t), and (ii) $ptr[i_t][j_t]$ which indexes the array element within a data tile. A_t is used as an array of pointers to data tiles. Only when a particular data tile (ii_t, jj_t) is required by a node, a new data tile buffer is allocated and stored in $A_t[ii_i][jj_t]$.

5.5.1 Simplification of access expressions

Modified access functions obtained after transformation have the additional cost of a divide, a mod, and an additional memory access (to obtain the data tile pointer) for each array access. This could lead to significant overhead and may prohibit other optimizations such as vectorization. We hoist the divide, mod and array dereference operations out of the innermost loop by splitting it.

Consider the computational and data tiled code shown in Figure 5.14. The innermost loop variable j always starts from a multiple of $tilesize(128)$ and executes $tilesize - 1$ times. For array access $A[i][j]$, the value of $j/128 = jj$ is constant all iterations of the

innermost loop j, i.e., all accesses are to a single data tile and hence can be hoisted out. The value of $j\%128$ goes from zero through $tilesize-1$. It can thus be replaced by $j-128*jj$ thereby eliminating the mod operation for every access. For the array access $X[i][j-1]$, only the first iteration of j refers to a different data tile, the remaining iterations access elements within the same data tile. Hence, after peeling the first iteration, we eliminate the mod, divide and array dereference operations from the innermost loop (Figure 5.15). If the array access was $X[i][j+1]$, we peel the last iteration. Thus, by splitting the innermost loop, we can hoist divide, mod and array dereference operations out of the innermost loop, reducing the overhead of modified array accesses.

```
//forward x sweep
  for (jj=0; jj<floord(N, 128); jj++)
   for (ii=0; ii<floord(N, 128); ii++)
    for (i=max(1,ii*128); i<min(ii*128+127, N); i++)
     for (j=max(1,jj*128); j<min(jj*128+127, N); j++)
      X[i/128][j/128][i%128][j%128] =
         X[i/128][j/128][i%128][j%128] -
         X[i/128][(j-1)/128][i%128][(j-1)%128] *
         A[i/128][j/128][i%128][j%128] /
         B[i/128][(j-1)/128][i%128][(j-1)%128]; //S1
```

Figure 5.14: Data tiled ADI example

```
//forward x sweep
  for (jj=0; jj<floord(N, 128); jj++)
   for (ii=0; ii<floord(N, 128); ii++)
    for (i=max(1, ii*128); i<min(ii*128+127, N); i++){
      j = max(1, jj*128);
      //peeled iteration
      X[i/128][j/128][i%128][j%128] =
              X[i/128][j/128][i%128][j%128] -
               X[i/128][(j-1)/128][i%128][(j-1)%128] *
               A[i/128][j/128][i%128][j%128] /
               B[i/128][(j-1)/128][i%128][(j-1)%128]; //S1
     j++;
     X_ptr = X[i/128][j/128];
     A_ptr = A[i/128][j/128];
     B_ptr = B[i/128][j/128];
     i_mod = i%128;
     lb = max(1, jj*128+1);
     for (j=max(1,jj*128+1); j<min(jj*128+127, N); j++)
      X_ptr[i_mod][j-lb] =  X_ptr[i_mod][j-lb] -
                X_ptr[i_mod][j-lb-1] * A_ptr[i_mod][j-lb] /
                X_ptr[i_mod][j-lb-1]; //S1
}
```

Figure 5.15: Optimized data tiled ADI example

Chapter 6

Data Movement Code Generation

In this chapter, we describe the techniques that are used to generate the data movement code. Once a compute tile finished its execution, we need to communicate all the values that are produced by the current compute tile and are required by other compute tiles. The flow (RAW) dependences that cross the compute tile boundaries are used to determine the communication set. This communication set is parameterized on a given compute tile. The generated data movement code is valid for any computation placement, problem size and number of nodes. We use Jacobi-style stencil example shown in 6.1 to illustrate the working of data movement schemes.

```
for ( t=1; t<=T-1; t++)
 for ( i=1; i<=N-2; i++)
   a[ t ][ i]=a[ t-1][i-1]+a[t-1][i]+a[t-1][i+1];
```

Figure 6.1: Jacobi-style stencil code

6.1 Flow-out (FO) scheme

Flow-out scheme, which is proposed by Bondhugula [3], computes a single communication set per compute tile that needs to be transferred to other compute tiles. For each tile $\vec{i}$ (inter tile iteration vector) and a data variable x, the flow-out set and receiving tiles are determined parameterized on $\vec{i}$. Per-dependence flow-out set $DFO_x(\vec{i}, D)$ is the set of all values which flow from a write in current tile to a read outside the tile due to a RAW dependences D. The complete flow-out set of tile is union of all the per-dependence flow-out sets.

$$FO_x(\vec{i}) = \bigcup_{\forall D} DFO_x(\vec{i}, D) \tag{6.1}$$

The receiving tiles $RI_x(\vec{i})$ is determined by projecting out the dimensions inner to $\vec{i}$ in D. This projected D is used to generate a $receivers_x(\vec{i})$ function which will return all the receive tiles that require at least one element of $FO_x(\vec{i})$. The flow-out set of a tile could be discontinuous in memory. Hence, flow-out set of tile is packed into single contiguous buffer and is sent to the receivers returned by $receivers_x(\vec{i})$. Once the data is received from other compute node, the flow-out data is unpacked only if $receivers_x(\vec{i})$ is non-empty and some data has been received from the $\pi(\vec{i})$.

Figure 6.2 illustrates the FO scheme, single flow-out set is sent to RT_1, RT_2 and RT_3. If RT_1 and RT_3 are mapped to different nodes, then unnecessary data is communicated. Thus, this scheme could communicate large volume of unnecessary data since every element in the packed buffer need not be communicated to every receiver compute node.

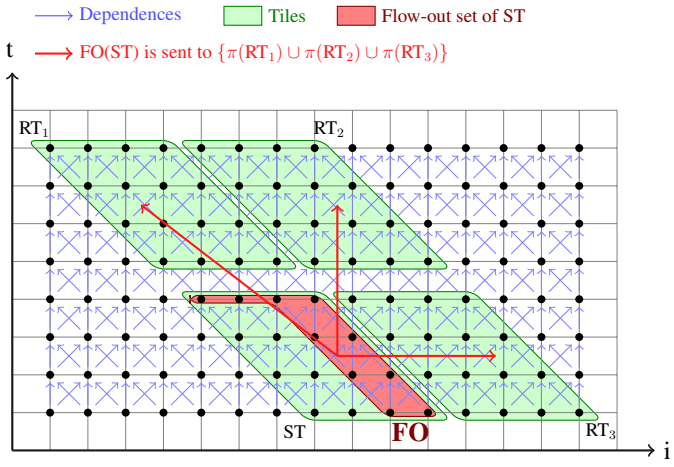


Figure 6.2: FO scheme for stencil

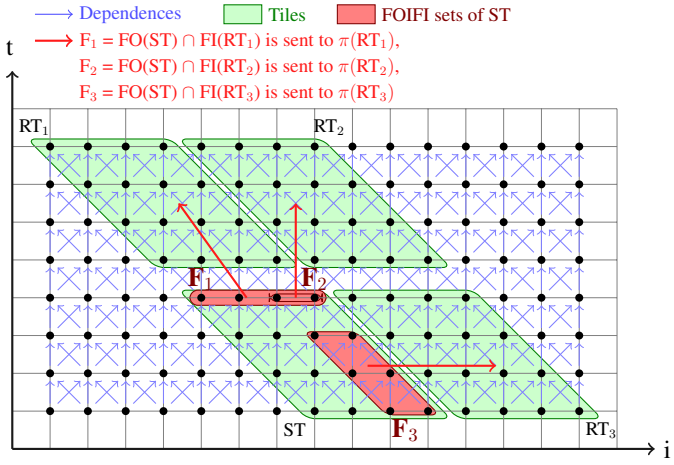


Figure 6.3: FOIFI scheme for stencil

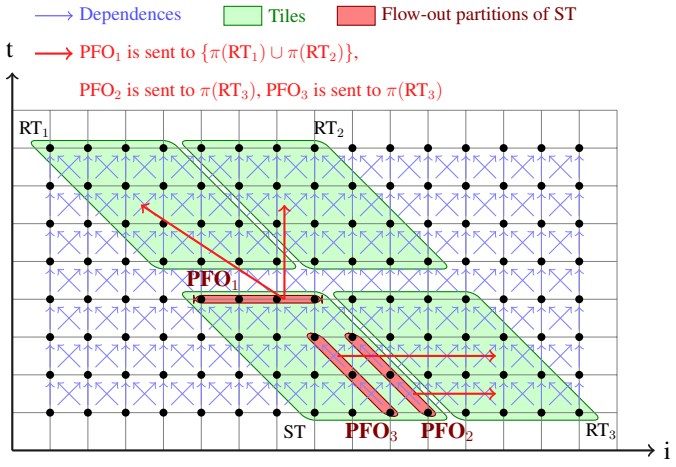


Figure 6.4: FOP scheme with multicast packing for stencil

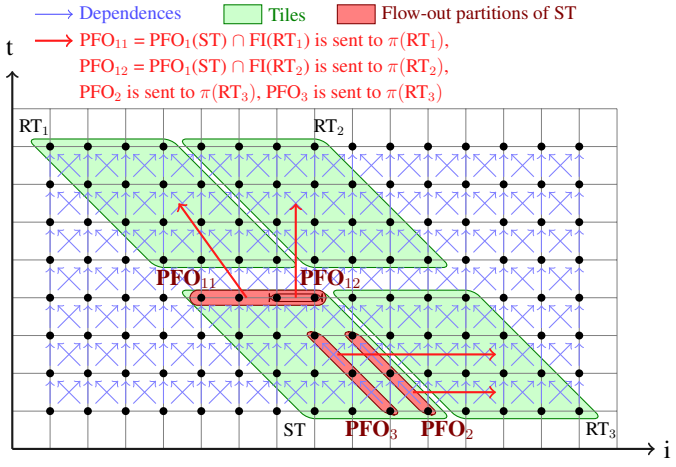


Figure 6.5: FOP scheme with unicast packing for stencil

6.2 Flow-out intersection flow-in (FOIFI) scheme

FOIFI scheme tries to avoid sending unnecessary data by computing the communication set that is parameterized on both sender and receiver tiles. It computes the exact data that needs to be send between a sender and a receiver tile.

Flow-in set: The set of all the values which flow to a read in a tile from a write outside a tile due to RAW dependence D is termed as the per-dependence flow-in $DFI_x(\vec{i}, D)$. The entire flow-in set $FI_x(\vec{i})$ is union of all the per-dependence flow-in sets.

$$FI_x(\vec{i}) = \bigcup_{\forall D} DFI_x(\vec{i}, D) \tag{6.2}$$

Flow set: The flow set from a source tile $\vec{i}$ to a target tile $\vec{i}$ is the set of all values written by $\vec{i}$, and then read by $\vec{i'}$. The flow set is determined by intersecting the flow-out set of $\vec{i}$ and flow-in set of $\vec{i'}$:

$$F_x(\vec{i} \rightarrow \vec{i'}) = FO_x(\vec{i}) \cap FI_x(\vec{i'}) \tag{6.3}$$

FOIFI scheme eliminates the redundant data of FO scheme and when each receiving tile is mapped to different compute node it ensures optimal communication volume. However, when multiple receiver tiles are mapped onto same compute node, this scheme leads to duplication of data since it accumulates the flow sets. For example, in Figure 6.3 if RT_1 and RT_2 are executed by the same compute node, then F_2 is sent twice.

6.3 Flow-out partitioning (FOP) scheme

FOP scheme tries to avoid both redundant data (FO) and duplicate data (FOIFI) by partitioning the communication set in a particular non-trivial way, and sending each partition

to only its receivers. Ideally, we want to partition the communication set such that all elements within each partition are required by all receivers of that partition. Since, the RAW dependences determine the communication sets and the receiving tiles, at compile time we partition these dependences into source-distinct partitions. Two sets of dependences are said to be source-distinct if the region of data that flow due to the dependences in different sets are disjoint. If two *source-identical* sets of RAW dependence polyhedra S_D^1 and S_D^2 of an iteration $\vec{i}$ are *source-distinct*, then:

$$DFO_x(\vec{i}, D_1) \cap DFO_x(\vec{i}, D_2) = \emptyset \quad \forall D_1 \in S_D^1, D_2 \in S_D^2 \tag{6.4}$$

A set of dependences is said to be **source-identical** if the region of data that flows due to each dependence in the set is the same. If S_D is *source-identical*, then:

$$DFO_x(\vec{i}, D_1) = DFO_x(\vec{i}, D_2) \quad \forall D_1, D_2 \in S_D \tag{6.5}$$

Algorithm 6: *source-distinct* partitioning of dependences

Input: RAW dependence polyhedra D_i and D_j

$(I_S, A_S) \leftarrow$ source (iterations, access) of D_i

$(I_T, A_T) \leftarrow$ source (iterations, access) of D_j

$D \leftarrow$ dependence from (I_S, A_S) to (I_T, A_T)

if D *is empty* **then**

    $D_S \leftarrow D_T \leftarrow$ empty

    return

$(I'_S, I'_T) \leftarrow$ (source, target) iterations of D

$D_S \leftarrow$ source I'_S and target unconstrained

$D_T \leftarrow$ source I'_T and target unconstrained

Output: *source-distinct* partitions $\{D_i - D_S\}, \{D_j - D_T\}, \{D_i \cap D_S, D_j \cap D_T\}$

In order to partition dependences, it is necessary to determine whether the regions of data that flow due to two dependences overlap, i.e., whether the region of data written by the source iterations of one dependence overlaps with that of the other. This can be determined by an explicit dependence test between the source iterations of one dependence and the source iterations of another dependence. A virtual dependence between two dependences, that captures the overlap in the regions of data that flow due to those dependence. If a virtual dependence does not exist between the two dependences, then they are *source-distinct*. Otherwise, the virtual dependence polyhedron contains the source iterations of each dependence polyhedron that access the same region of data. A new dependence polyhedron is formed from each dependence polyhedron by restricting the source iterations to their corresponding source iterations in the virtual dependence polyhedron. These two new dependences are *source-identical*. From the original dependence polyhedra, their corresponding source iterations in the virtual dependence polyhedron are subtracted out. These modified original dependences and the *source-identical* set of the new dependences are *source-distinct*.

After the dependences are partitioned, communication sets and receivers are determined for each partition. Since the flow-out partitions are disjoint, this scheme reduces the duplication of data. Also, for each partition we choose between *unicast-pack* and *multicast-pack* based on following non redundancy conditions. We choose *unicast-pack* at runtime if all the receiving tiles are mapped into different nodes. We choose *multicast-pack* if all the receiving tiles are mapped to same node. FOP scheme minimizes both redundant communication and minimizes data duplication and hence performs better than FO and FOIFI.

Chapter 7

Evaluation

In this chapter we present experiments demonstrating improvement over existing techniques. Our framework is implemented as a part of a publicly available source to source polyhedral tool chain. The input for our framework is sequential C code which can be arbitrarily nested affine loop nests. Compilable code to find computation placements and to distribute data is automatically generated. Program dependences are computed using ISL [11]. Polylib [23] is used to perform the polyhedral operations such as projection and union used in Chapter 5. Cloog-isl [1] is used to generate code from the polyhedral representation. METIS [14] is used to partition the initial graph and to determine computation placement.

We first determine the compute and data tiling hyperplanes using techniques described in Chapter 4. Computation hyperplanes are used to transform and tile the sequential code [24]. Techniques described in [3, 25] are used to construct communication sets and generate MPI code for distributed memory. The communication and distributed-memory code works for any arbitrary computation placement. Data spaces are tiled and array access expressions are modified using the data tiling hyperplanes. We generate functions to

perform on-demand allocation and buffer management as explained in Chapter 5. These functions are called at runtime for buffer allocation and management. All of these steps work in an end-to-end automatic manner taking unmodified sequential affine loop nests in C to parallelized code.

7.1 Benchmarks

We present results for Floyd-Warshall (`floyd-warshall`), LU Decomposition (`lu`), Cholesky Factorization (`cholesky`), Alternating Direction Implicit solver (`adi`), 2mm (`2mm`), and 3mm (`3mm`) benchmarks. All these benchmarks are chosen from the publicly available Polybench/C 3.2 suite [26]. For comparing against ScaLAPACK programs, we use `atax`, BiCG Sub Kernel (`bicg`), `gemver`, `gesummv`, and matrix vector product and transpose (`mvt`) benchmarks, also from the Polybench/C 3.2 suite [26]. All benchmarks use double-precision floating-point operations. The compiler used for all experiments is ICC 13.0.1 with options -O3 -ansi-alias -fp-model precise. *pluto-data-tile-gp* refers to our code. Where applicable, we compare or comment on solutions that would have been found by previous approaches [7, 8, 3], and we also mention the specific mapping found by the graph partitioning approach. Problem sizes used are listed in Table 7.1 and 7.2.

7.2 Distributed memory

7.2.1 Setup

The experiments were run on a 32-node InfiniBand cluster of dual-SMP Xeon servers. Each node on the cluster consists of two quad-core Intel Xeon E5430 2.66 GHz processors with 12 MB L2 cache and 16 GB RAM. The InfiniBand host adapter is a Mellanox

Benchmark	Problem size
`floyd-warshall`	4096 x 4096
`cholesky`	4096 x 4096
`lu`	8192 x 8192
`2mm`	2048 x 2048
`3mm`	2048 x 2048

Table 7.1: Problem sizes for shared memory evaluation

MT25204 (InfiniHost III Lx HCA). All nodes run 64-bit Linux kernel version 2.6.18. The cluster uses MVAPICH2-1.8.1 as the MPI implementation. We measured a point-to-point latency of 3.36 μs, unidirectional and bidirectional bandwidths of 1.5 GB/s and 2.56 GB/s respectively.

7.2.2 ScaLAPACK comparison

We developed ScaLAPACK versions of the benchmarks using multi-thread ScaLAPACK routines of Intel MKL 11.0.1 library. All experiments are run with 8 threads per node. Figures 7.1, 7.2, 7.3, 7.4 and 7.5 show the weak scaling performance for both ScaLAPACK code and our framework. ScaLAPACK internally uses 2-d block cyclic distributions for all routines. Our framework computes the optimal computation placements for each of the benchmarks. For `gemver`, our framework finds the *sudoku* distribution that significantly outperforms 2-d block cyclic distribution. As we are able to fuse the first two loop nests in `gemver` and perform data tiling, our single thread performance is improved by about 3x. For `mvt`, `bicg` and `gesummv` benchmarks, transformations applied result in an outer parallel loop. The output of our framework is a 1-d block distribution with no communication, and this results in near ideal scaling. The optimal computation placements are

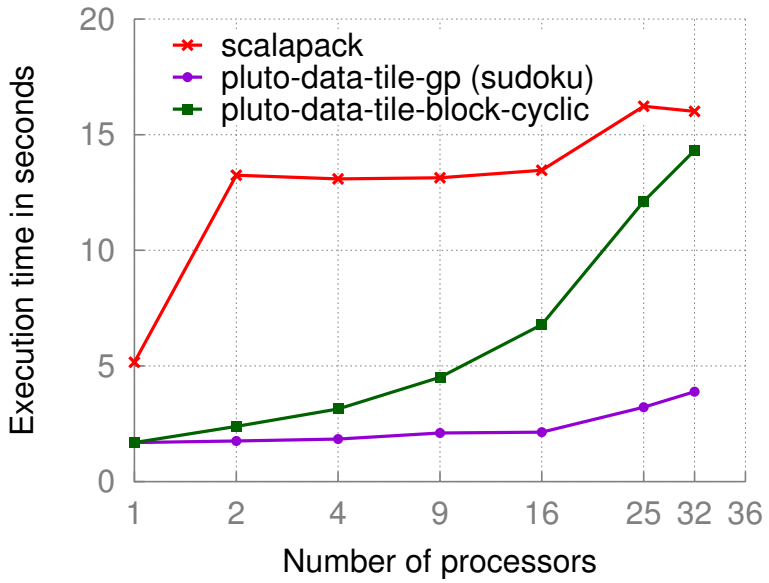


Figure 7.1: Weak scaling performance of gemver

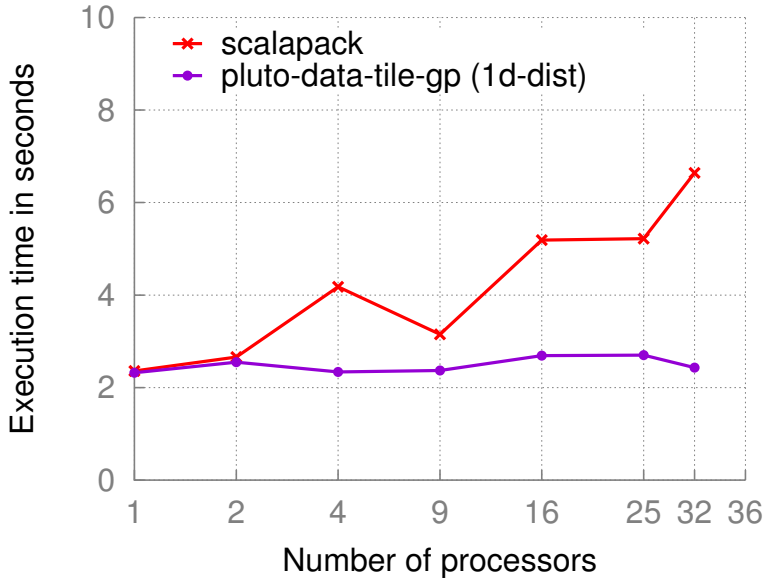


Figure 7.2: Weak scaling performance of bicg

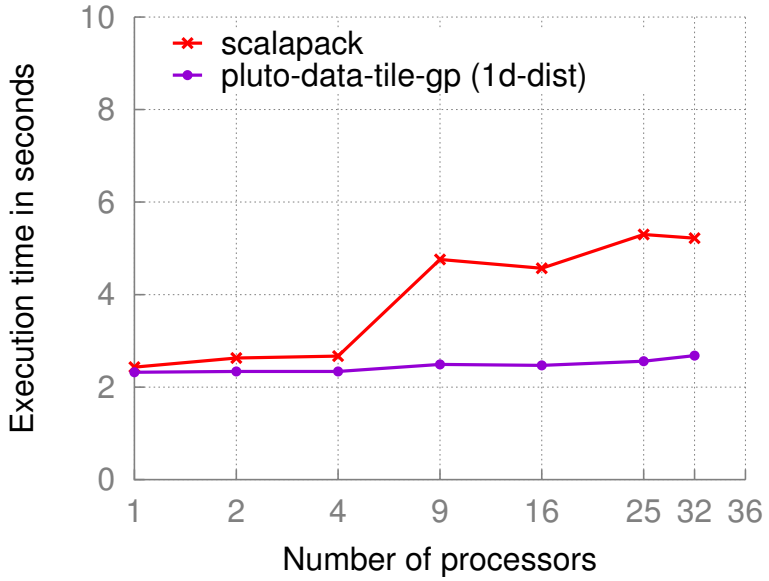


Figure 7.3: Weak scaling performance of mvt

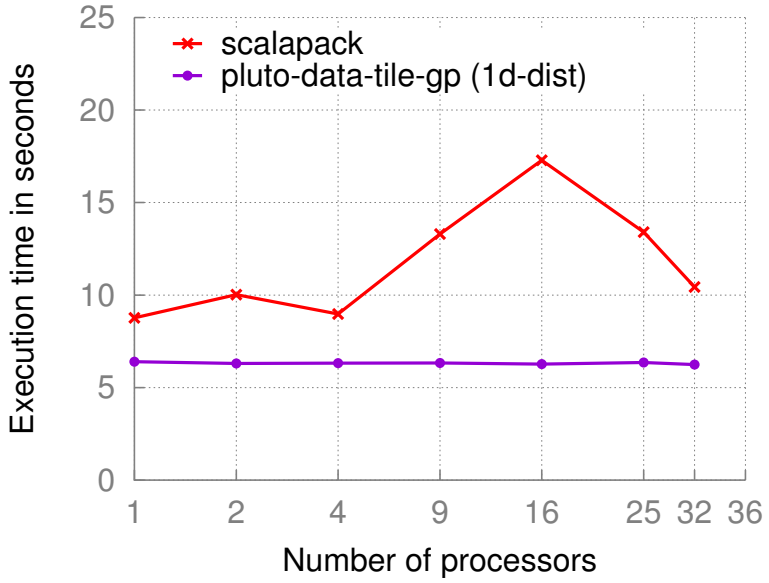


Figure 7.4: Weak scaling performance of gesummv

Benchmark	Problem size per processor
`gemver`	20000 x 20000
`bicg`	40000 x 40000
`gesummv`	30000 x 30000
`mvt`	30000 x 30000
`atax`	30000 x 30000
`floyd-warshall`	2048 x 2048
`lu`	4096 x 4096
`adi`	128 x 4096 x 4096

Table 7.2: Problem size (per proc) for distributed-memory evaluation

dependent on input program. Our framework was able to find computation placements and data allocations that are optimal for a given sequence of ScaLAPACK routines. For `atax` benchmark ScaLAPACK code performs slightly better than our code because there was no benefit with loop fusion and obtained computation mapping led to same communication volume as that of two separate ScaLAPACK library calls.

7.2.3 UPC comparison

Unified Parallel C (UPC) [27, 28] codes were compiled with Berkeley Unified Parallel C compiler version 2.16.0. All benchmarks were manually ported to UPC, while sharing data only if it may be accessed remotely and incorporating UPC- specific optimizations like localized array accesses, block copy, one-sided communication, where applicable.

Figures 7.6, 7.7 and 7.8 show the weak scaling performance for `floyd-warshall`, `lu` and `adi`. Previous schemes [3, 7] would have chosen 1-d block distribution for `adi`, that leads to $O(n^2)$ communication (nxn being the data size), and does not scale. On the

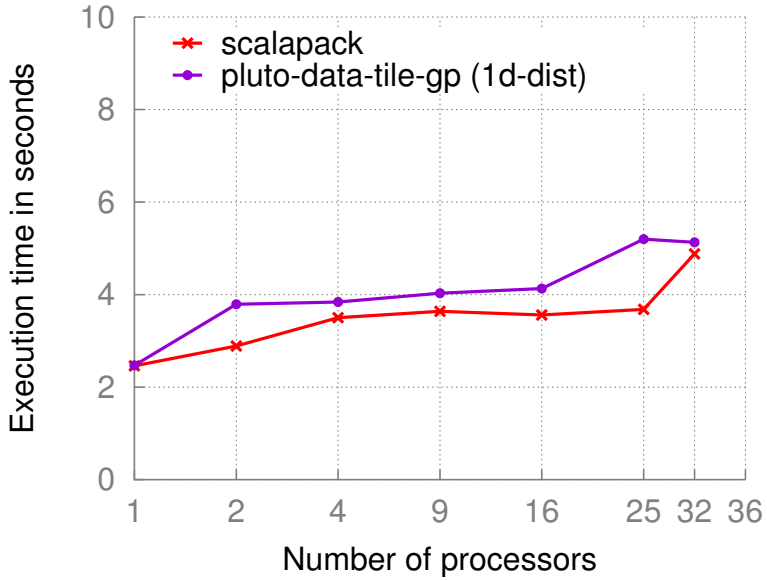


Figure 7.5: Weak scaling performance of atax

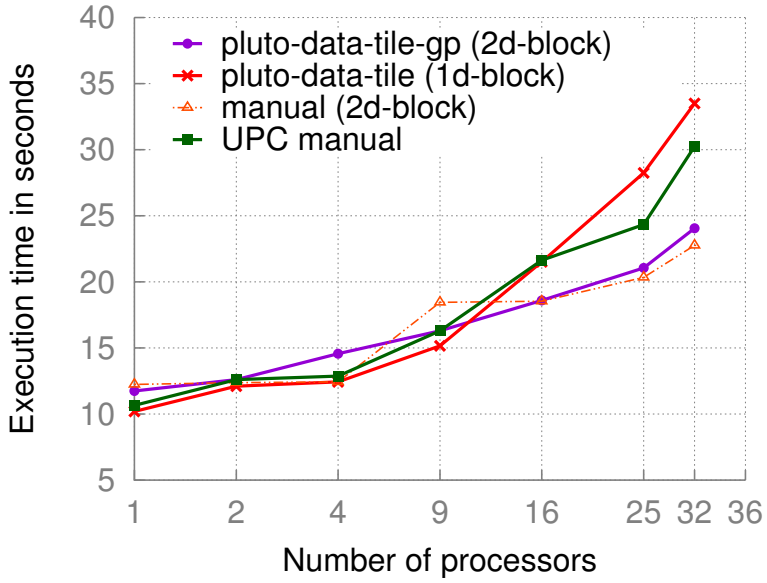


Figure 7.6: Weak scaling performance of floyd

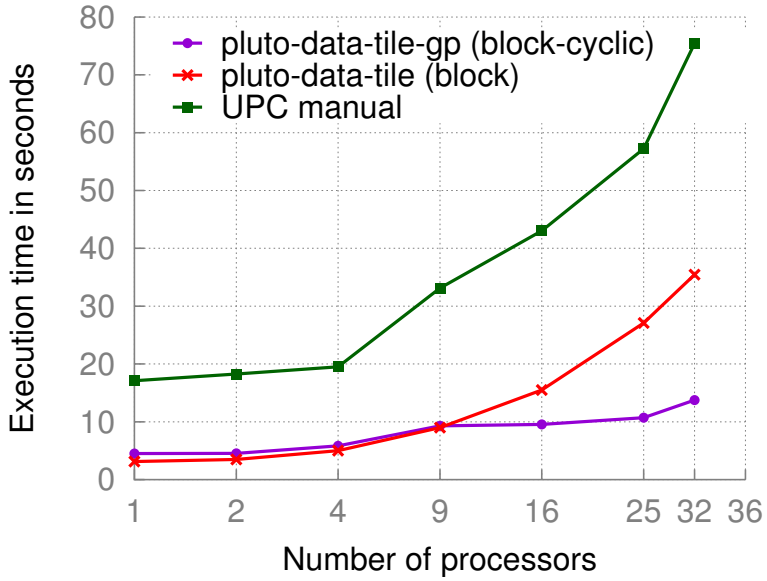


Figure 7.7: Weak scaling performance of lu

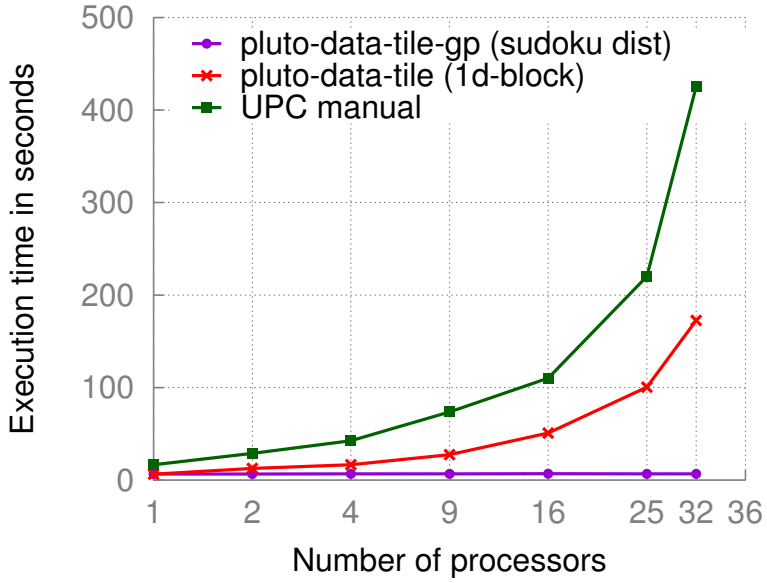


Figure 7.8: Weak scaling performance of adi

other hand, our framework finds the sudoku-like placement that has $O(n)$ communication only. Manually written UPC code also used 1-d block distribution, hence performs very poorly, Figure 7.6 shows the weak scaling performance for `floyd-warshall`. The performance of *pluto-data-tile-gp* is very close to manually written 2-d blocked floyd-warshall. 2-d block distribution performs better than a 1-d block one due to a higher ratio of computation to communication – in this case, it leads to a $3\times$ reduction in communication volume for 32 nodes. Our framework also implicitly finds the optimal dimensionality of the distribution leading to the minimum communication volume. Note that a higher dimensional mapping may not be necessarily optimal for an entire sequence of loop nests being optimized. Manually developed UPC code is used with 1-d block distributed, hence UPC performance is close to that of *pluto-data-tile-gp* with 1-d block distribution. For `lu` benchmark, our framework applies a complex 3-d tiling transformation. Writing UPC code incorporating the 3-d transformation is not trivial. Due to this UPC performs poorly.

7.2.4 Impact of data tile buffer reuse

Buffer reuse is an essential optimization to enable weak scaling in certain benchmarks such as `floyd-warshall`. In our framework, both local and remote data is allocated in similar fashion. For `floyd-warshall`, remote data received could span entire array, for all iterations of outer sequential loop. Without buffer reuse we would have allocated the entire array on each node, hindering weak scaling.

7.3 Shared memory

7.3.1 Setup

The experiments were run on two shared memory setups. First one is a four socket machine with AMD Opteron 6136 CPUs (2.4 GHz, 128 KB L1, 512 KB L2 and 6 MB L3 cache, 8 cores per socket) with 64 GB DDR3 RAM running 64-bit linux kernel version 2.6.35. Second machine is a two socket Intel Xeon E5645 CPUs clocked at 2.4 GHz, 6 cores per socket, 32 KB L1 cache, 512 KB L2 cache, 12 MB L3 cache and 24 GB DDR3 RAM running 64-bit linux kernel version 2.6.32. The shared memory has a NUMA architecture and we used `numactl` option to bind threads and pages appropriately for all our experiments. When not performing data tiling (for comparison), we did a simple interleaving of pages across all NUMA nodes. All the benchmarks are compiled with intel icc (version 12.1.3) with -O3, -fp-model-precise options.

7.3.2 Impact of data tiling

Figure 7.14 shows that data tiling leads to a significant improvement in single thread performance, and hence benefits shared-memory parallelization as well. Data tiling enhances the spatial locality of space tiled loops. After data tiling, data accessed by a compute tile is contiguous in memory. There will only be cold caches for all accesses to a data tile, i.e., conflict misses are eliminated. It also reduces TLB misses and false sharing. Due to simplification of the modified access functions, we completely eliminated associated overhead from the innermost loop. This results in a geometric mean speedup of 2.67$\times$ over code with no data tiling for 8 threads. Table 7.3 lists the hardware performance counter with and without data tiling for `floyd-warshall` on Intel Xeon machine. There is a significant reduction in the number of L2 load misses, L2 prefetch misses and offcore

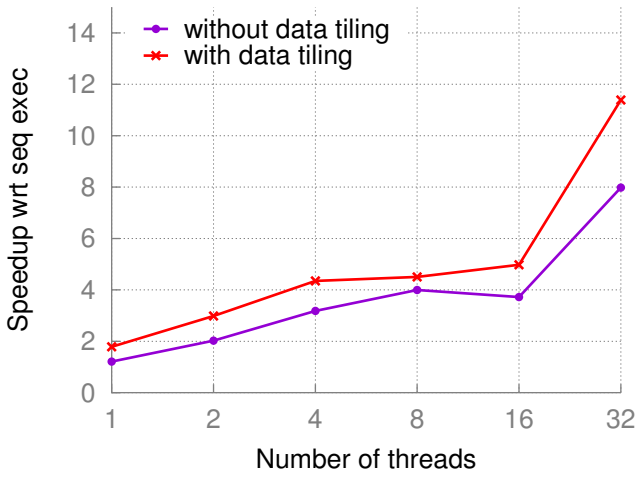


Figure 7.9: Speedup for floyd-warshall benchmark on AMD multicore machine: with and without data tiling, seq time is 231.87s

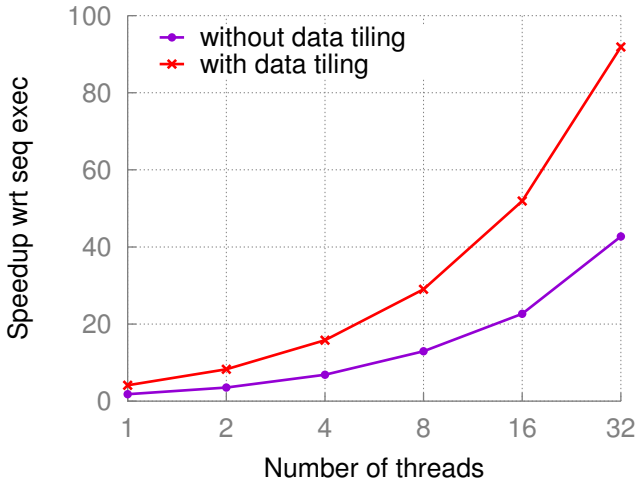


Figure 7.10: Speedup for lu benchmark on AMD multicore machine: with and without data tiling, seq time is 796.55s

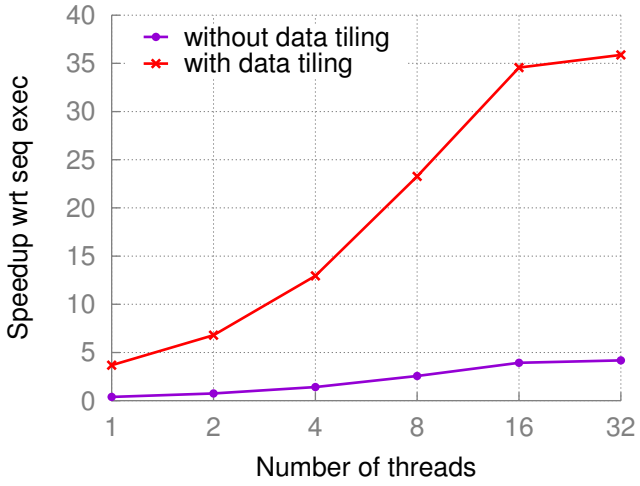


Figure 7.11: Speedup for cholesky benchmark on AMD multicore machine: with and without data tiling, seq time is 295.6s

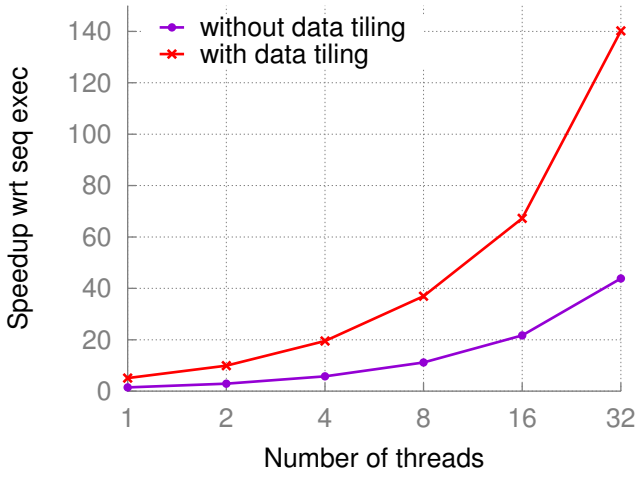


Figure 7.12: Speedup for 2mm benchmark on AMD multicore: with and without data tiling, seq time is 675s

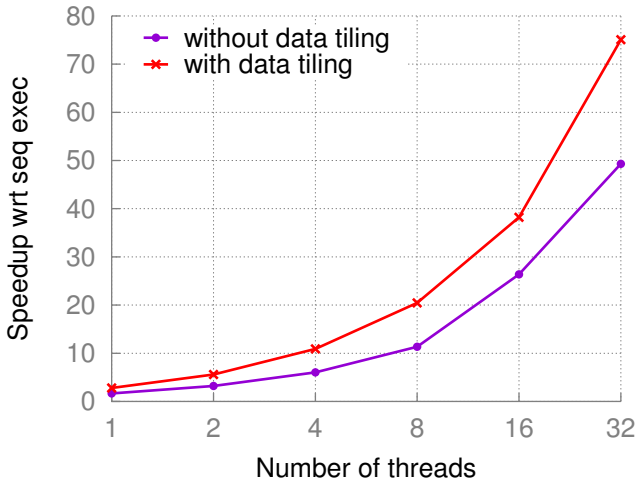


Figure 7.13: Speedup for 3mm benchmark on AMD multicore: with and without data tiling, seq time is 1200s

requests with data tiling. For `cholesky` we see a very high speedup of 5.42x. This is also due to data tiling enabling vectorization. `cholesky` kernel had spatially conflicting accesses in a single statement. This kernel is not readily vectorizable by icc as the memory accesses are not contiguous. If j is the innermost loop, then consecutive access of $A[j][i]$ are array size apart. However, after data tiling, accesses due to $A[j][i]$ are tile size apart, and icc can vectorize the code. So, in addition to enhancing locality, data tiling also enables vectorization.

Performance counter	Without data tiling (billion)	With data tiling (billion)
`L2 Load Misses`	5.46	3.19
`L2 Prefetch Misses`	10.6	5.6
`Offcore Requests`	34.1	28.7

Table 7.3: Performance counters for `floyd-warshall` on Intel Xeon machine

7.3.3 Impact of access function simplification

Simplification of access functions is a very important step. It not only eliminated the overhead of modified access functions (a mod, a divide and an array dereference per access) but also enabled optimizations such as vectorization. ICC compiler was not able to vectorize the code with modified array access expression because of the mod and divide operations. After simplification of modified access expressions, mod and divide operations were moved out of the innermost loop and ICC was able to vectorize it. This led to a large performance improvement of up-to 4x.

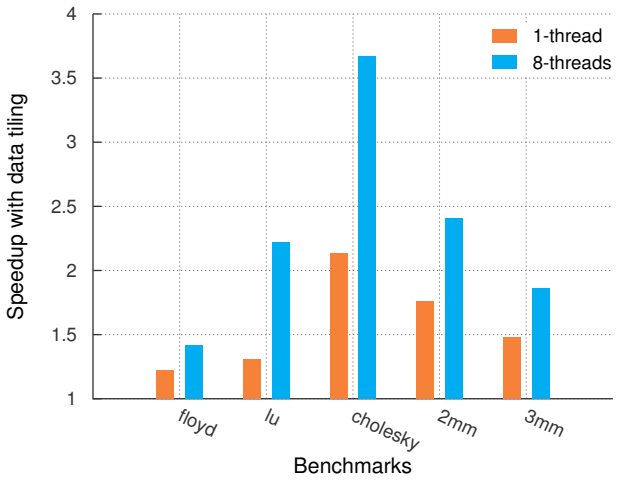


Figure 7.14: Speedup with data tiling over no data tiling on Intel shared memory multicore. Sequential execution times for floyd-warshall, lu, cholesky, 2mm, 3mm are 225s, 494s, 297s, 93s and 109s respectively.

Chapter 8

Related Work

In this chapter we describe some of the past works related to compiler support for programming distributed-memory architectures. We also present past works related to data tiling for shared memory systems.

8.1 Distributed memory

There are several previous works such as Kennedy and Kremer [7], Chapman et al [5], Garcia et al [6], Gupta and Banerjee [4], Lee and Kedem [29], Qi Ning et al [30], Peizong Lee [31], Couvertier et al [32] that have addressed the problem of finding automatic data distributions for distributed memory architectures in the context of regular programs. They first decompose the input program into regions (phases) at the granularity of loop nests. An array will have a single distribution throughout a phase and data is remapped between phases (dynamic distributions). Within each phase, they find data distributions and array alignments that lead to the least communication volume. Their solution space only includes data distributions supported by HPF (High Performance Fortran).

Framework	**Distribution strategy**	**Computation placements**	**Data distribution**	**Data movement**	**Program transformation support**	**Arbitrary mapping support**	**Minimize load imbalance**	**Buffer reuse**
Kennedy and Kremer [7] + HPF	Owner computes	derived from data distribution	automatic	automatic	no	no	no	no
Banerjee [4] + HPF	Owner computes	derived from data distribution	automatic	automatic	no	no	no	no
Anderson and Lam [19, 33]	Computer owns	heuristics	derived from computation distribution	automatic	yes	no	no	no
Garcia et al [6] + HPF	Owner computes	derived from data distribution	automatic	automatic	no	no	no	no
Manual UPC [6]	Owner computes	user specified	user specified	automatic	no	yes	no	no
Intel CnC [34]	user specified	user specified	user specified	manual	no	yes	no	yes
StarPU [35]	user specified	user specified	user specified	manual	no	yes	no	yes
Claßen and Griebl [36]	Computer owns	heuristics	not distributed	automatic	yes	yes	no	no
Bondhugula [3]	Computer owns	heuristics	not distributed	automatic	yes	yes	no	no
Our approach	Computer owns	graph partitioning based	on-demand data tiling based	automatic	yes	yes	yes	yes

Table 8.1: Related work comparison

Ramanujam and Sadayappan [37] use matrix notation to represent a communication free static distributions, and solve the problem with linear algebra techniques. However, there techniques can only find communication-free distributions.

Kennedy and Kremer [7] developed a framework for automatic data layout that builds and examines explicit search spaces of candidate data layouts. A candidate layout is an efficient layout for some part of the input program. The relevant program parts are called phases. A phase identifies operations on arrays between which remapping may be profitable. After generation of candidate layout search spaces for each phase, a single candidate layout is selected from each search space. The framework considers dynamic remapping only between phases. They use 0-1 integer programming to compute the optimal solutions to data layout selection problem and the inter dimensional alignment problem.

PARADIGM is a research tool developed by Palermo and Banerjee [4], that can automatically select dynamic data distributions starting from static distributions generated using a constraint-based algorithm and compile-time cost estimations based on empirically measured parameters. The technique proposed for automatic selection of dynamic data mappings can be broken down into two main steps. First, the program is recursively decomposed into a hierarchy of candidate phases. Then, taking into account the cost of redistributing data between different phases, the most efficient sequence of phases and transitions is selected.

Garcia et al. [6] propose a framework to automatically determine the data distributions and computation mappings. They try to find an optimal solution for the data mapping problem, given some characteristics of target architecture. They construct the Communication-Parallelism Graph (CPG), which contains information about possible data movement and parallelism within each phases, and remapping between phases. The data mapping problem is modeled as a minimal path problem with additional constraints to ensure the correctness of the solution. They also use 0-1 inter programming techniques to solve the

minimal path problem.

The work of Anderson and Lam [19] combined with distributed-memory code generation [33] deals with finding computation and data distributions in a unified manner. They first find affine computation and data mappings to virtual processors. They try to find these distributions such that communication is minimized and parallelism is maximized. The data and computation alignments are determined by computing the null space of the linear functions that represent these mappings. The null space computation is based on an iterative method that successively matches the mutual constraints of the data and computation mappings. The result of the null space computation is the set of all communication-free decompositions. Their algorithm for dynamic data layout and computation partitioning uses a greedy heuristic. Their algorithm tries to join loop nodes that are connected by a remapping edge in order to eliminate possible remapping costs. Edges are visited in the order of decreasing weights. The algorithm works in a bottom-up fashion, starting with innermost loops first. Two candidate loop nodes at the current level are merged into a single component if the performance of the joined nodes is higher than the performance of the individual nodes including the remapping costs. Once two nodes have been joined at a level, they are considered a single component for all subsequent levels. After the greedy algorithm terminates, the entire graph is partitioned into components, data and computation mappings for each component have been determined. In last step they use simple heuristics to map virtual processors onto physical processor. Their technique can only handle regular programs with perfectly nested loops and uniform dependences.

Compared to all previous techniques described, our technique first determines computation placements and the data distributions are then derived from it. Hence, it automatically captures array alignments, static and dynamic distributions, and array replications modeled in the previous approaches. To provide this flexibility, our approach includes an

elaborate data allocation scheme. To summarize, our approach has the following advantages over all previous works: (i) our solution space includes arbitrary mappings including multipartitioning-style, not just block and block cyclic, (ii) our framework has flexibility to apply locality enhancing transformations such as time tiling, since, we do not adhere to owner computes rule, and (iii) we minimize both communication volume and load imbalance.

Multipartitioning [8] and generalized multipartitioning [9] were specialized computation mapping schemes implemented in dHPF that provided excellent scaling for SP and BT from NAS parallel benchmarks. They are also suitable for the smaller *gemver* and *adi* codes we used for evaluation. However, a general mapping strategy that automatically deduced multipartitioning as a suitable mapping while also incorporating block, block-cyclic, and other arbitrary mappings for affine loop nests did not exist prior to this work.

Partitioned Global Address space (PGAS) [28] is a parallel programming model proposed to ease the effort needed to program distributed-memory architectures. It assumes a global memory address space that is logically partitioned and a portion of it is local to each process or thread. The PGAS model is the basis of Unified Parallel C [27], Coarray Fortran [38], Chapel [39], X10 [40], and Global Arrays [41]. PGAS attempts to combine the advantages of a SPMD programming style for distributed memory systems (as employed by MPI) with the data referencing semantics of shared memory systems. The RSTREAM compiler provides some support for distributed memory execution [42]. However, crucial steps of finding good computation placements is left to programmer. Data is allocated at the granularity of pages hence, could lead to inefficient data allocations. It also leads to suboptimal communication when the data that needs to be communicated is discontinuous in memory.

Griebl [43] provides a discussion on distributed-memory auto-parallelization using polyhedral framework. The work proposes a technique for scheduling and allocation

keeping distributed memory architectures in mind. They generate code for both block and block-cyclic distributions and it is upto the users to choose between them. They do not deal with the problem of data distribution. Entire data is allocated in all the compute nodes.

Kwon et al [44] propose a framework to translate OpenMP to MPI programs for a subset of affine loop nests that transfer same set of data for every iteration of outer sequential loop. They do not address problem of finding good computation placements and data distribution. They use only block computation distribution and entire data is allocate in all nodes.

Another set of works – Intel Concurrent Collections (CnC) [34] and StarPU [35] focus on providing high-level programming models which enable easy expression of parallelism for distributed-memory architectures. However, in both these models programmer has to specify computation placements and precise communication sets.

Baskaran et al [45] propose a data allocation scheme for GPUs. Their algorithm identifies convex union (convex hull) of all data accessed by a compute tile. The entire single convex bounding box is allocated on GPU memory. Depending on array accesses, the bounding box of a compute tile could be very large. Hence, this approach could allocate very large data region even though the actual accessed area is much smaller. For `floyd-warshall` benchmark, this scheme would allocate entire array instead of a single row and column.

Ramashekar et al. [46] provides a Bounding Box based Memory Manager (BBMM) when parallelizing for multi-GPU machines which distributes and manages data across multiple GPUs to enable weak scaling. Their approach is to refine the initial bounding boxes by performing set operations at runtime. Bounding boxes are split if there is any overlap between them. However, splitting of bounding boxes introduces conditionals in modified access functions, which may disable vectorization. Also, this refined approach

could lead to inefficient allocations for certain cases such as, a diagonal array access pattern.

8.2 Shared memory

Many recent works such as Lu et al [47] and Zhang et al [48] addressed the problem of optimizing data layouts for shared-memory architectures. Lu et al [47] proposed a data layout framework to enhance locality on NUCA-based chip multiprocessors. They find a single "localizable" data and computation partitioning, and the data is tiled along only one dimension. On the other hand, we find a full-ranked computation and data mapping in a unified manner, and the data is thus tiled along multiple dimensions. This approach is required due to our problem being very different from that of [47]. Zhang et al [48] proposed techniques to determine data tiling hyperplanes and computation-to-core mappings. They too formulate a graph partitioning problem to find the computation-to-core-mapping to minimize communication volume. However, they do not consider load balance and use simple heuristics to partition the graph. Both the approaches do not address on-demand allocation and buffer reuse and their techniques do not provide the flexibility to support any arbitrary computation mapping.

Jeremy et al [49] propose techniques to optimize sequences of BLAS kernels calls for shared-memory architectures. They develop a domain-specific language to express linear algebra operations and their BTO (Build to Order) compiler performs optimizations such as loop fusion, tiling etc. across sequences of BLAS calls. Our approach fits well in such domain-specific compilers as well to enable targeting distributed memory.

Our data tiling scheme also achieves the benefits of explicit data copying that is typically done manually for improved cache and prefetching performance, although not to the same extent – since explicit copying performs an exact allocation of only the accessed data

while data tiling allocates the set of all data tiles that include any accessed data. Blocked and block recursive data layouts for matrices for BLAS routines have also been studied in depth [50]. However, the approaches were manual and compiler support for them was missing.

Chapter 9

Conclusions

9.1 Summary

Programming High Performance System with distributed-memory is a tedious, time consuming and error-prone task. Programmer has to extract parallelism, choose good computation placements, accordingly distribute and manage data, and generate precise communication code. An approach for programming HPC systems that automatically performs all the above steps and achieves scalable performance for HPC systems is highly desirable. In this thesis, we have provided efficient and automatic techniques to find good computation mappings, efficiently distribute and manage data and to generate precise communication code.

We developed an automatic technique to find good computation placements for the entire program. We modeled the problem of finding good computation placements as a graph partitioning problem with the constraints to minimize both communication volume and load imbalance. Our approach encompasses traditional mappings such as block, block-cyclic and other specialized mappings such as sudoku mappings and any other arbitrary

mappings.

We proposed a data allocation technique based on data tiling to provide improved locality and to enable weak scaling for distributed memory parallelization. Data local to a node as well as that which is received from remote nodes was allocated on demand at the granularity of data tiles. Besides enabling weak scaling for distributed memory, data tiling also improves locality for shared-memory parallelization.

We presented a dynamic, schedule independent data tile buffer reuse techniques. We used a compiler-based approach with light-weight runtime helper functions to track the liveness of data tiles. This technique handles on-demand allocation and reuse of data tiles.

We also develop an efficient data movement scheme based on inter tile dependences that will minimize the communication volume which was valid for any computation placement, problem size and number of nodes.

We showed through experimental results, how our approach for computation mapping is able to come up with more effective mappings than those that can be used with vendor-supplied BLAS libraries. These mappings that were automatically determined also subsume mappings with similar properties that were implemented and used manually in previous works. Experimental results on sequences of BLAS calls demonstrated a mean speedup of 1.82$\times$ over versions written with ScaLAPACK and a maximum speedup on 4$\times$ while running on a 32-node cluster. Besides enabling weak scaling for distributed memory, data tiling also improves locality for shared-memory parallelization. Experimental results on a 32-core shared-memory NUMA SMP system showed a mean speedup of 2.67$\times$ over code that is not data tiled.

9.2 Future work

- In our current approach to find computation mappings, we try to minimize the overall communication volume. But, the actual communication cost depends on the network topology of a HPC system. Finding computation mappings that are optimal for a given network topology is left as a future work.

- We could also like to explore different dynamic scheduling strategies based on data tiling. For example, on a system with constraints on memory size, it is essential to reduce the memory footprint of a program. This can be achieved by maximizing the data tile buffer reuse. A scheduling policy which will give priority to a compute tile that will free maximum number of data tile buffers will enhance the data tile buffer reuse.

- Another possible scheduling policy could be to maximize inter tile data reuse. This can be achieved by choosing a compute tile that has maximum number of data tiles in common with the current compute tile that has just finished its execution. This scheme will enhance data locality since it enables reuse of data across different compute tiles.

Bibliography

[1] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, (Washington, DC, USA), pp. 7–16, IEEE Computer Society, 2004.

[2] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model," in *International conference on Compiler Construction (ETAPS CC)*, 2008.

[3] U. Bondhugula, "Compiling affine loop nests for distributed-memory parallel architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, (New York, NY, USA), pp. 33:1–33:12, ACM, 2013.

[4] M. Gupta and P. Banerjee, "Paradigm: A compiler for automatic data distribution on multicomputers," in *Proceedings of the International Conference on Supercomputing*, ICS '93, pp. 87–96, ACM, 1993.

[5] B. M. Chapman, T. Fahringer, and H. P. Zima, "Automatic support for data distribution on distributed memory multiprocessor systems," in *LCPC*, pp. 184–199, 1993.

[6] J. Garcia, E. Ayguade, and J. Labarta, "A novel approach towards automatic data distribution," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, pp. 78–78, IEEE, 1995.

[7] K. Kennedy and U. Kremer, "Automatic data layout for distributed-memory machines," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 4, pp. 869–916, 1998.

[8] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarria-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi, "Advanced optimization strategies in the rice dHPF compiler," *Concurrency: Practice and Experience*, pp. 741–767, 2002.

[9] A. Darte, J. Mellor-Crummey, R. Fowler, and D. Chavarría-Miranda, "Generalized multipartitioning of multi-dimensional arrays for parallelizing line-sweep computations," *Journal of Parallel and Distributed Computing*, vol. 63, pp. 887–911, Sep 2003.

[10] "PIP: The Parametric Integer Programming Library." http://www.piplib.org.

[11] S. Verdoolaege, "Integer Set Library." an integer set library for program analysis.

[12] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 40, IEEE Computer Society Press, 2012.

[13] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The nas parallel benchmarks—summary and preliminary results," in *Proceedings of the 1991*

ACM/IEEE Conference on Supercomputing, Supercomputing '91, (New York, NY, USA), pp. 158–165, ACM, 1991.

[14] "METIS -Family of Graph and Hypergraph Partitioning Softwares." http://glaros.dtc.umn.edu/gkhome/views/metis.

[15] "Scotch - Sequential and parallel graph partitioning software." http://www.labri.fr/perso/pelegrin/scotch.

[16] G. Karypis and V. Kumar, "Metis-unstructured graph partitioning and sparse matrix ordering system, version 2.0," 1995.

[17] F. Pellegrini and J. Roman, "Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs," in *High-Performance Computing and Networking*, pp. 493–498, Springer, 1996.

[18] D. Lasalle and G. Karypis, "Multi-threaded graph partitioning," in *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, (Washington, DC, USA), pp. 225–236, IEEE Computer Society, 2013.

[19] J. M. Anderson and M. S. Lam, "Global optimizations for parallelism and locality on scalable parallel machines," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pp. 112–125, 1993.

[20] P. Feautrier, "Some efficient solutions to the affine scheduling problem: Part I, one-dimensional time," *International Journal of Parallel Programming*, vol. 21, no. 5, pp. 313–348, 1992.

[21] P. Feautrier, "Parametric integer programming," *RAIRO Recherche Opérationnelle*, vol. 22, no. 3, pp. 243–268, 1988.

[22] "Intel - Thread Building Blocks."
https://www.threadingbuildingblocks.org/.

[23] "PolyLib - A library of polyhedral functions."
http://icps.u-strasbg.fr/polylib/.

[24] "PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores."
http://pluto-compiler.sourceforge.net.

[25] R. Dathathri, C. Reddy, T. Ramashekar, and U. Bondhugula, "Generating efficient data movement code for heterogeneous architectures with distributed-memory," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[26] "Polybench." http://polybench.sourceforge.net.

[27] "UPC - Unified Parallel C."
www.http://upc.lbl.gov/.

[28] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, *et al.*, "Productivity and performance using partitioned global address space languages," in *Proceedings of the 2007 international workshop on Parallel symbolic computation*, pp. 24–32, ACM, 2007.

[29] P. Lee and Z. M. Kedem, "Automatic data and computation decomposition on distributed memory parallel computers," *ACM Trans. Programming Languages and Systems*, vol. 24, p. 2002, 2002.

[30] Q. Ning, V. V. Dongen, G. R. Gao, V. Van, D. Guang, and R. Gao, "Automatic data

and computation decomposition for distributed memory machines," in *In Proceedings of the 28th Annual Hawaii International Conference on System Sciences, Maui*, 1995.

[31] P. Lee, "Efficient algorithms for data distribution on distributed memory parallel computers," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, pp. 825–839, 1997.

[32] I. Couvertier-Reyes, *Automatic Data and Computation Mapping for Distributed-memory Machines*. PhD thesis, 1996. AAI9637768.

[33] S. P. Amarasinghe and M. S. Lam, "Communication optimization and code generation for distributed memory machines," in *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pp. 126–138, 1993.

[34] Z. Budimlic, A. Chandramowlishwaran, K. Knobe, G. Lowney, V. Sarkar, and L. Treggiari, "Multi-core implementations of the concurrent collections programming model," *CPC'09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[35] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, 2011.

[36] M. Claßen and M. Griebl, "Automatic code generation for distributed memory architectures in the polytope model," in *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments, Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International.*, 2006.

[37] J. Ramanujam and P. Sadayappan, "A methodology for parallelizing programs for multicomputers and complex memory multiprocessors," in *IN PROCEEDINGS OF SUPERCOMPUTING '89*, pp. 637–646, 1989.

[38] "Co-Array Fortran."
http://www.co-array.org/.

[39] "The Chapel Parallel Programming Language."
http://chapel.cray.com/.

[40] "X10 Programming Language."
http://x10-lang.org/.

[41] "Global Arrays Toolkit."
http://hpc.pnl.gov/globalarrays/.

[42] B. Meister, A. Leung, N. Vasilache, D. Wohlford, C. Bastoul, and R. Lethin, "Productivity via automatic code generation for pgas platforms with the r-stream compiler," in *Workshop on Asynchrony in the PGAS Programming Model*, 2009.

[43] M. Griebl, *Automatic Parallelization of Loop Programs for Distributed Memory Architectures*. University of Passau, 2004. Habilitation thesis.

[44] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff, "A hybrid approach of openmp for clusters," in *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, (New York, NY, USA), pp. 75–84, ACM, 2012.

[45] M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and

P. Sadayappan, "Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories," in *ACM SIGPLAN PPoPP*, Feb. 2008.

[46] T. Ramashekar and U. Bondhugula, "Automatic data allocation and buffer management for multi-gpu machines," *ACM Trans. Archit. Code Optim.*, vol. 10, pp. 60:1–60:26, Dec. 2013.

[47] Q. Lu, C. Alias, U. Bondhugula, *et al.*, "Data layout transformation for enhancing data locality on nuca chip multiprocessors," in *International Conference on Parallel Architectures and Compilation Techniques*, pp. 348–357, 2009.

[48] Y. Zhang, W. Ding, M. Kandemir, J. Liu, and O. Jang, "A data layout optimization framework for nuca-based multicores," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 489–500, ACM, 2011.

[49] J. G. Siek, I. Karlin, and E. R. Jessup, "Build to order linear algebra kernels," in *International Symposium on Parallel and Distributed Processing 2008 (IPDPS 2008*, pp. 1–8, 2008.

[50] F. G. Gustavson, I. Jonsson, B. Kågström, and P. Ling, "Towards peak performance on hierarchical smp memory architectures - new recursive blocked data formats and blas," in *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, 1999.