Compiling for a Dataflow Runtime on Distributed-Memory Parallel Architectures

A Thesis

Submitted For the Degree of

Master of Science (Engineering)

in the Computer Science and Engineering

by

Roshan Dathathri

Computer Science and Automation

Indian Institute of Science

BANGALORE – 560 012

April 2014

Acknowledgements

First and foremost, I would like to thank my advisor, Dr. Uday Bondhugula, for his invaluable guidance. I am grateful for the enormous amount of freedom he provided in pursuing research. He has always been enthusiastic to discuss new ideas and provide feedback. In addition to him, I collaborated with Chandan Reddy, Thejas Ramashekar, and Ravi Teja Mullapudi on problems that are a part of this thesis. Our technical discussions have been very fruitful. I am thankful to my collaborators for helping me in improving the quality of work presented in this thesis. I have enjoyed working with all of them.

I would like to specially thank all the members of the Multicore Computing Lab for maintaining a culture that is refreshing and relaxing. It has been a pleasure to work in such an environment.

I extend my gratitude to the department for providing a conducive environment for research. I thank Dr. R. Govindarajan and Dr. Uday Bondhugula for offering courses on computer architecture and advanced compilers respectively. Reading and discussing research papers in these courses taught me to analyze, summarize, and critique research ideas. The courses laid the foundation for my research. I would also like to thank all the non-technical staff in the department, including Mrs. Lalitha, Mrs. Suguna, and Mrs. Meenakshi, for ensuring that the time I spent in the department was hassle free.

It has been a privilege to study and live in the Indian Institute of Science (IISc). I enjoyed living in such a beautiful and peaceful campus. The hostel and mess facilities made my stay very comfortable. The swimming pool and gymkhana made my stay fulfilling. I am grateful to the institute for all the facilities it provides to students.

I have been fortunate to have a close group of friends in the institute. I thank my friends Anirudh, Apoorva, Aravind, Chandan, Irshad, Jay, Lakshmi, Malavika, Narayan, Ninad, Pallavi, Prachi, Prasanna, Ranjani, Satya, Siddharth, Thejas, Vasantha and Vinayaka. Their company made my stay a delightful one. I will cherish my experiences with them.

There have been many people who have motivated me to pursue higher studies and research. I am thankful to all of them. Some of them deserve a special mention. My friend Vineeth introduced me to research and PhD programs. My friend Rajath joined IISc to do research and influenced me to join IISc the following year. I found an interest in research around the same time as my friends Somashekar and Madhusudan, and we helped each other to pursue that interest. My managers at National Instruments (NI), Arjun and Praveen, encouraged me to pursue higher studies. It was through an initiative of NI that I first met Dr. Uday Bondhugula, which planted the seed for my research.

Finally, I would like to thank my parents for their support and encouragement. They have given me far more than I could ever give back.

Publications based on this Thesis

1. Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. Generating Efficient Data Movement Code for Heterogeneous Architectures with Distributed-memory. In *Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 375–386, 2013. IEEE Press.

2. Roshan Dathathri, Ravi Teja Mullapudi, and Uday Bondhugula. Compiling Affine Loop Nests for a Dataflow Runtime on Shared and Distributed Memory. Submitted (undergoing peer-review).

Abstract

Programming for parallel architectures that do not have a shared address space is tedious due to the need for explicit communication between memories of different compute devices. A heterogeneous system with CPUs and multiple GPUs, or a distributed-memory cluster are examples of such systems. Past works that try to automate data movement for such architectures can lead to excessive redundant communication. In addition, current de-facto parallel programming models like MPI make it difficult to extract *task-level dataflow parallelism* as opposed to *bulk-synchronous parallelism*. Thus, task parallel approaches that use point-to-point synchronization between dependent tasks in conjunction with dynamic scheduling dataflow runtimes are becoming attractive. Although good performance can be extracted for both shared and distributed memory using these approaches, there is very little compiler support for them. In this thesis, we propose a fully automatic compiler-assisted runtime framework that takes sequential code containing affine loop nests as input, extracts coarse-grained dataflow parallelism, statically analyzes data to be communicated, and generates the components of the dynamic scheduling dataflow runtime along with efficient data movement code for distributed-memory architectures.

Firstly, we describe an automatic data movement scheme that minimizes the volume of communication between compute devices in heterogeneous and distributed-memory systems. We show that by partitioning data dependences in a particular non-trivial way, one can generate data movement code that results in the minimum volume for a vast majority of cases. The techniques are applicable to any sequence of affine loop nests and work on top of any choice of loop transformations, parallelization, and computation placement. The data movement code generated minimizes the volume of communication for a particular configuration of these. We use a combination of powerful static analyses relying on the polyhedral compiler framework and lightweight runtime routines they generate, to build a source-to-source transformation tool that automatically generates communication code. We demonstrate that the tool is scalable and leads to substantial gains in efficiency. On a heterogeneous system, the communication volume is reduced by a factor of 11× to 83× over state-of-the-art, translating into a mean execution time speedup of 1.53×. On a distributed-memory cluster, our scheme reduces the communication volume by a factor of 1.4× to 63.5× over state-of-the-art, resulting in a mean speedup of 1.55×. In addition, our scheme yields a mean speedup of 2.19× over hand-optimized Unified Parallel C (UPC) codes.

Secondly, we describe the design of compiler-runtime interaction to automatically extract coarse-grained dataflow parallelism in affine loop nests on both shared and distributed memory. We use techniques from the polyhedral compiler framework to extract tasks and generate components of the runtime which are used to dynamically schedule the generated tasks. The runtime includes a distributed decentralized scheduler that dynamically schedules tasks on a node. The schedulers on different nodes cooperate with each other through asynchronous point-to-point communication of data required to preserve program semantics – all of this is achieved automatically by the compiler. While running on 32 nodes with 8 threads each, our compiler-assisted runtime yields a mean speedup of 143.6× over the sequential version, and a mean speedup of 1.6× over the state-of-the-art automatic parallelization approach that uses *bulk-synchronous parallelism* while using our own efficient data movement scheme. We also compare our system with past work that addresses some of these challenges on shared-memory, and an emerging runtime (Intel Concurrent Collections) that demands higher programmer input and effort in parallelizing. To the best of our knowledge, ours is also the first automatic scheme that allows for dynamic scheduling of affine loop nests on a cluster of multicores.

Keywords

communication optimization, data movement, task parallelism, dynamic scheduling, dataflow runtime, compiler-runtime framework, polyhedral model, distributed memory, heterogeneous architectures

Contents

Acknowledgements **i**

Publications based on this Thesis **iii**

Abstract **iv**

Keywords **vi**

List of Figures **xi**

List of Tables **xiii**

List of Algorithms **xiv**

1 Introduction **1**

1.1 Distributed-memory parallel architectures 1
1.2 Productivity and performance 2
1.3 Existing approaches 3
1.4 Automatic data movement and compilation for a dataflow runtime 4
1.5 Contributions 6

2 Polyhedral Model **9**

3 Generating Efficient Data Movement Code **11**

3.1 Illustrative examples 11
3.2 Background and motivation 12
3.2.1 Flow-out (FO) scheme 12
Overview 13
Flow-out set 14
Receiving iterations 14
Packing and unpacking 15
Communication volume 16
3.3 Flow-out intersection flow-in (FOIFI) scheme 16
3.3.1 Overview 19
3.3.2 Flow-in set 20

3.3.3 Flow set 20
3.3.4 Communication volume 21
3.4 Flow-out partitioning (FOP) scheme 22
3.4.1 Overview 23
3.4.2 Partitioning of dependences 25
3.4.3 Partitioned communication sets 26
3.4.4 Receiving iterations of the partition 27
3.4.5 Packing and unpacking 27
3.4.6 Communication volume 28
3.5 Implementation 30
3.6 Experimental evaluation 32
3.6.1 Distributed-memory architectures 32
Setup 32
Benchmarks 32
Evaluation 33
Analysis 34
3.6.2 Heterogeneous architectures 38
Intel-NVIDIA system setup 38
AMD system setup 38
Benchmarks 39
Evaluation 39
Analysis 41

4 Targeting a Dataflow Runtime 43
4.1 Motivation and design challenges 43
4.1.1 Dataflow and memory-based dependences 43
4.1.2 Terminology 44
Tasks 44
Scheduling tasks 45
4.1.3 Synchronization and communication code 46
4.1.4 Objectives 47
4.2 Compiler-assisted dataflow runtime 47
4.2.1 Overview 47
4.2.2 Synthesized Runtime Interface (SRI) 50
Inter-task dependences 52
Constraints on scheduling 55
Communication and placement 55
Computation 56
Thread-safety 56
4.2.3 Distributed Function-based Dynamic Scheduling (DFDS) 57
Priority 61
Dynamic a priori placement 62
4.3 Implementation 62
4.4 Experimental evaluation 64

Benchmarks 64
Intel Concurrent Collections (CnC) implementations 65
4.4.1 Shared-memory architectures 66
Setup 66
Evaluation 66
Analysis 68
4.4.2 Distributed-memory architectures 70
Setup 70
Evaluation 70
Analysis 72

5 Related Work **75**
5.1 Data movement for distributed-memory architectures 75
5.2 Data movement for heterogeneous architectures 78
5.3 Automatic parallelization frameworks 79
5.4 Dataflow runtime frameworks 81
5.4.1 Frameworks for linear algebra computations 81
5.4.2 General-purpose frameworks 83

6 Conclusions and Future Work **87**
6.1 Conclusions 87
6.2 Future work 89

References **90**

List of Figures

3.1 Jacobi-style stencil code 12

3.2 Floyd-Warshall code 12

3.3 Illustration of data movement schemes for Jacobi-style stencil example . 17

3.4 Illustration of data movement schemes for Floyd-Warshall example (CS_i sets are used only for illustration; communication sets are determined as described in text) 18

3.5 FOP – strong scaling on the distributed-memory cluster 37

3.6 `floyd` – speedup of FOP, FOIFI, FO and hand-optimized UPC code over `seq` on the distributed-memory cluster 38

3.7 FOP – strong scaling on the Intel-NVIDIA system 41

4.1 Inter-task dependences example 45

4.2 Overview of the scheduler on each node 48

4.3 Illustration of inter-task dependences for an example 53

4.4 Overview of our tool 63

4.5 Speedup of auto-DFDS, auto-static, and manual-CnC over seq on a shared-memory multicore (Note that performance of auto-DFDS and auto-static on a single thread is different from that of seq due to automatic transformations) 67

4.6 Maximum idle time across 32 threads on a shared-memory multicore . . 69

4.7 Speedup of auto-DFDS over manual-CnC on a shared-memory multicore . 70

4.8 Speedup of auto-DFDS, auto-static, and manual-CnC over seq on a cluster of multicores (Note that performance of auto-DFDS and auto-static on a single thread is different from that of seq due to automatic transformations) 71

4.9 Maximum computation time and maximum communication time in auto-static across all threads on 32 nodes (multicores) of a cluster 73

4.10 Non-overlapped communication time reduction: auto-DFDS over auto-static on 32 nodes (multicores) of a cluster 73

4.11 Speedup of auto-DFDS over manual-CnC on a cluster of multicores 74

List of Tables

3.1 Problem and tile sizes - distributed-memory cluster 33
3.2 Total communication volume on the distributed-memory cluster – FO and FOIFI normalized to FOP 34
3.3 Total execution time on the distributed-memory cluster – FOIFI, FO, OMPD and UPC normalized to FOP 35
3.4 Problem and tile sizes - Intel-NVIDIA system 39
3.5 Problem and tile sizes - AMD system 39
3.6 Total communication volume and execution time of FO and FOP on the Intel-NVIDIA system 40
3.7 Total communication volume and execution time of FO and FOP on the AMD system 40

4.1 Synthesized Runtime Interface (SRI) 51
4.2 Synthesized Runtime Interface (SRI) that assists dynamic scheduling: generated by analyzing inter-task dependences (decrementDependentOfRemote() should be called for remote tasks while the rest should be called for local tasks) 54
4.3 Problem and tile sizes - shared-memory multicore (Note that computation tiling transformations for auto and manual-CnC may differ; data is tiled in manual-CnC but not in auto) 65

4.4 Problem and tile sizes - cluster of multicores (Note that computation tiling transformations for auto and manual-CnC may differ; data is tiled in manual-CnC but not in auto) 65
4.5 Standard-deviation over mean of computation times of all threads in % on 32 threads of a shared-memory multicore: lower value indicates better load balance 68
4.6 Standard-deviation over mean of computation times of all threads in % on 32 nodes (multicores) of a cluster: lower value indicates better load balance 72

List of Algorithms

1 *source-distinct* partitioning of dependences 25

2 Distributed Function-based Dynamic Scheduling (DFDS) 57

3 initTasks() 58

4 computeTasks() 59

5 sendDataOfTask() 59

6 receiveDataFromTasks() 60

Chapter 1

Introduction

In this chapter, we discuss the problems being addressed in this thesis, the drawbacks of the existing approaches that try to solve them, and our contributions in solving them.

1.1 Distributed-memory parallel architectures

Clusters of multicore processors have emerged as the parallel architecture of choice both for medium and high-end high-performance computing. Different processors or nodes in such a cluster do not typically share a global address space. Programming for such distributed-memory architectures is difficult due to the need for explicit communication between memories of different nodes. In addition, extraordinary amount of programming effort is required to extract high parallel performance on such distributed-memory architectures. Key issues are the inability to extract parallelism beyond plain loop parallelism and independent tasks: this impacts load balance and synchronization overheads.

The design of new languages, compilers, and runtime systems are crucial to provide productivity and high performance while programming parallel architectures. Integrating language and programming model design with compiler and runtime support is naturally a powerful approach owing to the amount of information available to the compiler and runtime in generating efficient code. Several systems [16,17,33,44,46] have been designed in an integrated manner to various degrees.

1.2 Productivity and performance

MPI is the current de-facto parallel programming model for distributed-memory architectures. MPI requires the programmer to explicitly move data between memories of different nodes, which makes it more tedious to program than programming models for shared-memory architectures like OpenMP. If the compiler has to deal with partitioning and scheduling computation on distributed-memory architectures, it has to address the problem of moving data between memories. Since the volume of data transferred significantly impacts performance, only those values should be moved that is required to be moved in order to preserve program semantics. The problem of generating efficient data movement code is common to all systems with distributed-memory architectures. A heterogeneous system with CPUs and multiple GPUs, or a distributed-memory cluster are examples of such systems. It is a key part of the larger parallelization problem for such systems, which involves other orthogonal sub-problems, such as computation and data transformations, computation placement and scheduling. A compiler that handles data movement automatically would provide high productivity while programming distributed-memory architectures.

Though OpenMP and MPI are the most commonly used programming models for shared-memory and distributed-memory architectures respectively, using them to extract *task-level dataflow parallelism* as opposed to *bulk-synchronous parallelism* is very difficult to almost infeasible. Bulk-synchronous parallelism is one in which all iterations of a parallel loop are synchronized in bulk after their execution, and before moving onto the next set of parallel iterations. Task-level dataflow parallelism is the parallelism that can be extracted from the dynamic dependence graph of tasks (a directed acyclic graph) – each task itself can correspond to a block of iterations of a parallel loop or more generally, a piece of computation that is to be executed in sequence atomically, i.e., synchronization or communication is performed only before and after execution of the task but not during it. *Asynchronous parallelization* enabled by explicit point-to-point synchronization between tasks that are actually dependent is known to provide better performance than *bulk-synchronous parallelization* [8, 20, 22, 48]. Compiler and runtime support for task-level dataflow parallelism would deliver high parallel performance on distributed-memory architectures.

1.3 Existing approaches

Affine loop nests are any sequence of arbitrarily nested loops where the loop bounds and array accesses are affine functions of surrounding loop iterators and program parameters. They form the compute-intensive core of scientific computations like stencil-style computations, linear algebra kernels, alternating direction implicit (ADI) integrations. Past works on distributed-memory compilation deal with input more restrictive than affine loop nests [2,34]. The schemes proposed in works which handle arbitrary affine loop nests while automating data movement for distributed-memory architectures [1, 13, 23, 24] or for multi-device heterogeneous architectures [32] can lead to excessive redundant communication. Hence, there is no precise and efficient automatic data movement scheme at present which handles affine loop nests for distributed-memory architectures.

Several programming models and runtimes have been proposed to support task-level dataflow parallelism. Some recent works that address this include that of Baskaran et al. [8], Song and Dongarra [46], DAGuE [17], DPLASMA [16], Concurrent Collections (CnC) [18], the EARTH model [48], the codelet model [52], and SWARM [35], though in varying contexts. The work of Baskaran et al. [8] is the only one which takes sequential code as input and requires no additional programming effort (i.e., it is fully automatic), but it is applicable for affine loop nests only on shared-memory architectures. Although good performance can be extracted for both shared and distributed memory using the other task-parallel approaches, it still requires considerable programming effort. The programmer is responsible for identifying the data to be moved in such approaches too. Moreover, one of the key issues in leveraging dynamic scheduling dataflow runtimes such as CnC is in determining the right decomposition of tasks; the granularity of tasks impacts load balance and synchronization overheads. Choosing the right decomposition can improve the performance by orders of magnitude. The decomposition into tasks and choice of granularity itself has a direct connection with loop transformations such as tiling, making a strong case for integration of compiler support. However, there is no existing framework that supports compiling affine loop nests automatically for a dynamic scheduling dataflow runtime on distributed-memory architectures.

1.4 Automatic data movement and compilation for a dataflow runtime

In this thesis, we propose a fully automatic compiler-assisted runtime framework that takes sequential code containing affine loop nests as input, extracts coarse-grained dataflow parallelism, statically analyzes data to be communicated, and generates the components of the dynamic scheduling dataflow runtime along with efficient data movement code for distributed-memory architectures. The techniques we present rely on a combination of powerful static analyses employing the polyhedral compiler framework and lightweight runtime routines generated using them. To this end, we propose a static analysis scheme that minimizes data movement, and a compiler-assisted runtime framework that automatically extracts coarse-grained dataflow parallelism for affine loop nests on distributed-memory architectures. Thus, our framework provides high productivity while delivering high parallel performance on distributed-memory architectures.

The static analysis techniques we develop determine data to be transferred between compute devices in heterogeneous and distributed-memory systems with a goal to move only those values that need to be moved in order to preserve program semantics. A compute device is typically a GPU or the CPUs in a heterogeneous system, and a node in a distributed-memory cluster. We show that by partitioning polyhedral data dependences in a particular non-trivial way, and determining communication sets and their receivers based on those partitions, one can determine communication data precisely, while avoiding both unnecessary and duplicate data from being communicated – a notoriously hard problem and a limitation of all previous polyhedral data movement works. Our scheme can handle any choice of loop transformations and parallelization. The code it generates is parametric in problem size symbols and number of processors, and valid for any computation placement (static or dynamic). The data movement code generated minimizes the volume of communication, given a particular choice of all these.

We build a source-to-source transformation tool that automatically generates communication code, and demonstrate that the tool is scalable and leads to substantial gains in efficiency. On a heterogeneous system, the communication volume is reduced by a factor of 11× to 83× over state-of-the-art, translating into a geometric mean execution time speedup of 1.53×. On a distributed-memory cluster, our scheme reduces the communication volume by a factor of 1.4× to 63.5× over state-of-the-art, resulting in a mean speedup of 1.55×. In addition, our scheme yields a mean speedup of 2.19× over hand-optimized Unified Parallel C (UPC) [49] codes; UPC is a unified programming model for shared and distributed memory systems.

While designing compiler support for dynamic scheduling dataflow runtimes, we also develop and use our own runtime. However, the focus of our work is in effectively exploiting runtime support and features through powerful compile-time analysis and transformation to provide a fully automatic solution. This is done so that efficient execution on shared as well as distributed memory is achieved with no programmer input. Hence, a distributed-memory cluster of multicores is a typical target. This work's objective is not to compare the efficiency of the developed runtime itself with other existing dynamic scheduling ones. Runtimes such as SWARM [35] and those employed in CnC [18] and DAGuE [17] share some of the same design principles and concepts as ours. The choice to develop our own runtime in conjunction with this work was driven by the need to allow sufficient customization and flexibility for current and future compiler support.

We describe the design of compiler-runtime interaction to automatically extract coarse-grained dataflow parallelism in affine loop nests on both shared and distributed memory. We use techniques from the polyhedral compiler framework to extract tasks and generate components of the runtime which are used to dynamically schedule the generated tasks. The runtime components are lightweight helper functions generated by the compiler. The task dependence graph is also encapsulated in such compiler-generated functions. This also allows the same generated code to execute in parallel on shared-memory, distributed-memory, or a combination of both.

The runtime includes a distributed decentralized scheduler that dynamically schedules tasks on a node. There is no coordination between the schedulers on each node, and there is no synchronization between nodes. The only asynchronous point-to-point communication messages between nodes is that of data required to preserve program semantics, embedded with meta-data. The distributed schedulers cooperate with each other using this meta-data. In this way, our system achieves cooperation without coordination in distributed dynamic schedulers. We are also able to thus automatically obtain overlap of computation and communication, and load-balanced execution. All of this is achieved automatically by the compiler.

We build a source-to-source transformation tool that automatically generates code targeting a dynamic scheduling dataflow runtime. While running on 32 nodes with 8 threads each, our compiler-assisted runtime yields a geometric mean speedup of 143.6× over the sequential version, and a mean speedup of 1.6× over the state-of-the-art automatic parallelization approach that uses *bulk-synchronization* while using our own efficient data movement scheme. We also compare our system with past work that addresses some of these challenges on shared-memory, and an emerging runtime (Intel Concurrent Collections) that demands higher programmer input and effort in parallelizing. To the best of our knowledge, ours is also the first automatic scheme that allows for dynamic scheduling of affine loop nests on a cluster of multicores.

1.5 Contributions

The main contributions of this thesis can be summarized as follows:

- We describe two new static analysis techniques to generate efficient data movement code between compute devices that do not share an address space.

- We implement our data movement techniques in a source-level transformer to allow automatic distribution of loop computations on multiple CPUs and GPUs of a heterogeneous system, or on a distributed-memory cluster.

- We experimentally evaluate our data movement techniques and compare it against existing schemes showing significant reduction in communication volume, translating into significantly better scaling and performance.
- We represent the dynamic dependence graph of tasks in a compact manner using helper functions generated by the compiler.
- We design the compiler-runtime interface as a set of compiler-generated functions that is required by the dataflow runtime.
- We design a novel compiler-assisted dataflow runtime framework that achieves cooperation without coordination in distributed dynamic schedulers.
- We implement our compiler-assisted dataflow runtime in a source-level transformer to allow for dynamic scheduling of affine loop nests on a cluster of multicores.
- We experimentally evaluate our compiler-assisted dataflow runtime and compare it against the state-of-the-art automatic parallelization approach that uses *bulk-synchronization* while using our own efficient data movement scheme, showing better load balance and communication-computation overlap, translating into significantly better scaling and performance.
- We also compare our fully automatic framework against manually optimized Intel Concurrent Collections (CnC) codes making a strong case for building compiler support for dataflow runtimes.

The rest of this thesis is organized as follows. Chapter 2 provides background on the polyhedral model. Chapter 3 describes our static analysis techniques to generate efficient data movement code along with its motivation, implementation and evaluation. Chapter 4 describes our compiler-assisted dataflow runtime along with its motivation, implementation and evaluation. Chapter 5 discusses related work and conclusions are presented in Chapter 6.

Chapter 2

Polyhedral Model

If the array accesses in loops are affine functions of surrounding loop iterators and program parameters, they are known as affine accesses. Affine loop nests are loop nests with affine accesses and loop bounds. The polyhedral compiler framework provides a representation that captures the execution of a sequence of arbitrarily nested affine loop nests in a mathematical form suitable for analysis and transformation using machinery from linear algebra and linear programming. We provide a brief description of the polyhedral model in this chapter. The reader is referred to the Clan user guide [9] for a more detailed introduction to the polyhedral representation.

The *iteration space*, domain or *index set* of every statement can be represented as the set of integer points inside a (convex) polyhedron. The polyhedron is typically represented by a conjunction of affine inequalities. The dimensions of the polyhedron correspond to surrounding loop iterators as well as *program parameters*. Program parameters are symbols that do not vary in the portion of the code we are representing; they are typically the problem sizes. Each integer point in the polyhedron, also called an *iteration vector*, contains values for induction variables of loops surrounding the statement from outermost to innermost.

The *data dependence graph*, is a directed multi-graph with each vertex representing a statement in the program and edge representing a polyhedral dependence from a dynamic instance of one statement to another. Every edge is characterized by a polyhedron, called *dependence polyhedron*, which precisely captures dependences between dynamic instances of the source statement and the target statement. The dependence polyhedron is specific to a source access (in the source statement) and a target access (in the target statement). The dependence polyhedron is also typically represented by a conjunction of affine inequalities, and is in the sum of the dimensionalities of the source and target iterations spaces, and the number of program parameters.

During analysis and transformation, one might end up with a union of convex polyhedra. Polylib [42], Omega [43], and ISL [50] are three libraries that provide operations to manipulate such a union of polyhedra. The latter two are precise and deal with integer points. The various operations provided by the libraries to manipulate polyhedra include union, intersection, and difference of polyhedra. The libraries also provide operations to project out the dimensions within a polyhedron, and to get the image of a polyhedron under an affine function treating certain dimensions as parameters. All of the sets that we describe and compute in this thesis are unions of convex polyhedra, and any of the above libraries can be used to manipulate them.

A code generator such as Cloog [12] is used to generate code that scans a union of convex polyhedra, i.e., iterates over all the integer points in it. The code generator can also be used to generate code that treats certain dimensions as parameters, and scans the other dimensions. To achieve this, the dimensions are permuted so that certain outermost dimensions are treated as parameters by the code generator while scanning the other inner dimensions. Such a code generator can be used to iterate over all the integer points in any set that we describe and compute in this thesis.

Chapter 3

Generating Efficient Data Movement Code

In this chapter, we present our work on data movement code generation for distributed-memory architectures. Sections 3.1 and 3.2 provide background on the existing communication schemes. Sections 3.3 and 3.4 describe the design of our static analysis techniques to address the data movement problem. Section 3.5 describes our implementation and experimental evaluation is presented in Section 3.6.

3.1 Illustrative examples

To illustrate the working of data movement schemes, we use two examples, which are tiled and parallelized:

- Jacobi-style stencil code in Figure 3.1: It contains uniform array access functions and uniform dependences. Since each iteration depends only on its neighboring iterations in each dimension, it exhibits near-neighbor communication patterns.
- Floyd-Warshall code in Figure 3.2: It contains non-uniform array access functions and non-uniform dependences. Since all iterations in a dimension depend on a common iteration, it exhibits broadcast communication patterns.

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

Figure 3.1: Jacobi-style stencil code

```
for (k=0; k<=N−1; k++)
  for (i=0; i<=N−1; i++)
    for (j=0; j<=N−1; j++)
      a[i][j]=((a[i][k]+a[k][j])<a[i][j])?(a[i][k]+a[k][j]): a[i][j];
```

Figure 3.2: Floyd-Warshall code

3.2 Background and motivation

It is known that anti (WAR) dependences and output (WAW) dependences do not necessitate communication; only the flow (RAW) dependences necessitate communication [13, 27]. Previous research [1] has also shown that it is essential for any communication scheme to perform optimizations like vectorization and coalescing.

- *communication vectorization* reduces communication frequency by aggregating the data to be communicated between each pair of processors for all parallel iterations of the distributed loop(s).
- *communication coalescing* reduces redundant communication by combining the data to be communicated due to multiple data accesses or dependences.

3.2.1 Flow-out (FO) scheme

We provide a brief description of an existing communication scheme proposed by Bondhugula [13], termed **flow-out (FO) scheme**, that our schemes build upon. Bondhugula describes static analysis techniques using the polyhedral compiler framework to determine data to be transferred between compute devices parametric in problem size symbols and number of processors, which is valid for any computation placement (static or dynamic). The key idea is that since code corresponding to a single iteration of the innermost distributed loop will always be executed atomically by one compute device, communication parameterized on that iteration can be determined statically.

The term *innermost distributed loop* is used to indicate the innermost among loops that have been identified for parallelization or distribution across compute devices. So, an iteration of the innermost distributed loop represents an atomic computation tile, on which communication is parameterized; the computation tile may or may not be a result of loop tiling. An iteration of the innermost distributed loop is uniquely identified by its iteration vector, i.e., the tuple of values for induction variables of loops surrounding it from outermost to innermost (including the innermost distributed loop). Hence, communication is parameterized on the iteration vector of an iteration of the innermost distributed loop.

Overview

For each innermost distributed loop, consider an iteration of it represented by iteration vector $\vec{i}$. For each data variable x, that can be a multidimensional array or a scalar, the following is determined at compile-time parameterized on $\vec{i}$:

- Flow-out set, $FO_x(\vec{i})$: the set of elements that need to be communicated from iteration $\vec{i}$.
- Receiving iterations, $RI_x(\vec{i})$: the set of iterations of the innermost distributed loop(s) that require some element in $FO_x(\vec{i})$.

Using these parameterized sets, code is generated to execute the following in each compute device c at runtime:

- `multicast-pack`: for each iteration $\vec{i}$ executed by c, if some $\vec{i'} \in RI_x(\vec{i})$ will be executed by another compute device c' ($c' \neq c$), pack $FO_x(\vec{i})$ into a local buffer,
- Send the packed buffer to the set of other compute devices c' ($c' \neq c$) which will execute some $\vec{i'} \in RI_x(\vec{i})$ for any $\vec{i}$ executed by c, and receive data from other compute devices,

- unpack corresponding to `multicast-pack`: for each iteration $\vec{i}$ executed by another compute device c' ($c' \neq c$), if some $\vec{i'} \in RI_x(\vec{i})$ will be executed by some other compute device c'' ($c'' \neq c'$), and if c received some data from c', unpack $FO_x(\vec{i})$ from the received buffer associated with c'.

Flow-out set

The flow-out set represents the data that needs to be sent from an iteration. The set of all values which flow from a write in an iteration to a read outside the iteration due to a RAW dependence is termed as the **per-dependence flow-out set** corresponding to that iteration and dependence. For a RAW dependence polyhedron D of data variable x whose source statement is in $\vec{i}$, the per-dependence flow-out set $DFO_x(\vec{i}, D)$ is determined by computing the region of data x written by those source iterations of D whose writes are read outside $\vec{i}$. The region of data written by the set of source iterations of D can be determined by computing an image of the set of source iterations of D under the source access affine function of D. Wherever we use the term *region of data* in the rest of this thesis, the region of data can be computed in a similar manner.

The flow-out set of an iteration is the set of all values written by that iteration, and then read outside the iteration. Therefore:

$$FO_x(\vec{i}) = \bigcup_{\forall D} DFO_x(\vec{i}, D) \tag{3.1}$$

Since the flow-out set combines the data to be communicated due to multiple dependences, *communication coalescing* is implicitly achieved. Code is generated to enumerate the elements in the flow-out set of the given iteration at runtime.

Receiving iterations

$RI_x(\vec{i})$ are the iterations $\vec{i'}$ of the innermost distributed loop(s) that read values written in $\vec{i}$ ($\vec{i'} \neq \vec{i}$). For each RAW dependence polyhedron D of data variable x whose source statement is in $\vec{i}$, $RI_x(\vec{i})$ is determined by projecting out dimensions inner to $\vec{i}$ in D and scanning the target iterators while treating the source iterators as parameters. Since the goal is to determine the compute devices to communicate with, code is generated for a pair of helper functions $\pi(\vec{i})$ and $receivers_x(\vec{i})$.

- $\pi(\vec{i})$ returns the compute device that executes $\vec{i}$.
- $receivers_x(\vec{i})$ returns the set of compute devices that require at least one element in $FO_x(\vec{i})$.

π is the placement function which maps an iteration of an innermost distributed loop to a compute device. It is the inverse of the computation distribution function which maps a compute device to a set of iterations of the innermost distributed loop(s) (which it executes). So, π can be easily determined from the given computation distribution function. Since π is evaluated only at runtime, the computation placement (or distribution) can be chosen dynamically. $receivers_x(\vec{i})$ enumerates the receiving iterations and makes use of π on each receiving iteration to aggregate the set of distinct receivers $(\pi(\vec{i}) \notin receivers_x(\vec{i}))$.

Packing and unpacking

The flow-out set of an iteration could be discontiguous in memory. So, at runtime, the generated code packs the flow-out set of each iteration executed by the compute device to a single buffer. The data is packed for an iteration $\vec{i}$ only if $receivers_x(\vec{i})$ is a non-empty set. The packed buffer is then sent to the set of receivers returned by $receivers_x(\vec{i})$ for all iterations $\vec{i}$ executed by it. Note that for each variable, there is a single send-buffer for all the receivers. Since the communication set for all iterations executed by a compute device is communicated to all receivers at once, `communication vectorization` is achieved. After receiving data from other compute devices, the generated code unpacks the flow-out set of each iteration executed by every compute device other than itself from the respective received buffer. The data is unpacked for an iteration $\vec{i}$ only if $receivers_x(\vec{i})$ is a non-empty set, and if some data has been received from the compute device that executed $\vec{i}$.

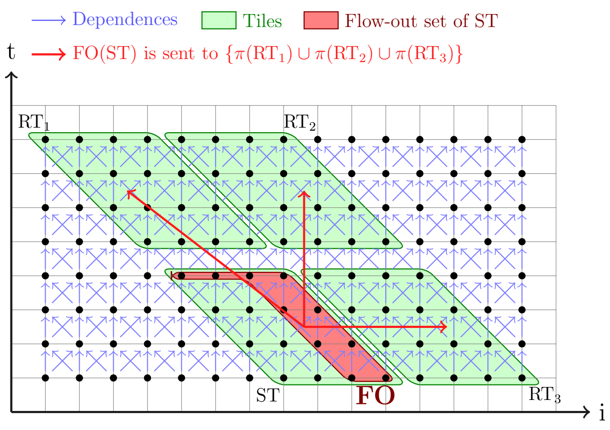
Both the packing code and the unpacking code traverse the iterations executed by a compute device, and the flow-out set of each iteration in the same order. Therefore, the offset of an element in the packed buffer of the sending compute device matches that in the received buffer of the receiving compute device. In order to allocate buffers for sending and receiving, reasonably tight upper bounds on the required size of buffers can be determined from the communication set constraints, but we do not present details on it due to space constraints.

Communication volume

A compute device might execute more than an iteration of the innermost distributed loop. Since all the iterations of the innermost distributed loop can be run in parallel, they cannot have any WAW dependences between them, and therefore, the writes in each iteration are unique. This implies that the flow-out set for each iteration is disjoint, and so, accumulating the flow-out set of each iteration does not lead to duplication of data. However, all the elements in the flow-out set of an iteration might not be required by all its receiving iterations. As illustrated in Figure 3.3a, if RT_1 and RT_3 are executed by different compute devices, then unnecessary data is communicated to those compute devices. Similarly, in Figure 3.4b, if RT_1 and RT_2 are executed by different compute devices, then unnecessary data is communicated to the compute device that executes RT_2. 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 device; different receivers might require different elements in the packed buffer.

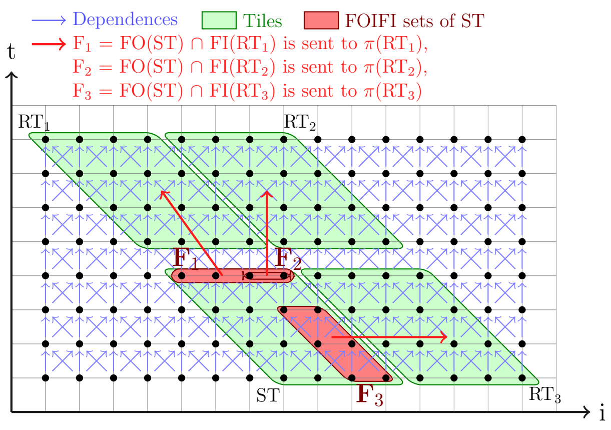
3.3 Flow-out intersection flow-in (FOIFI) scheme

FO scheme [13] could send unnecessary data since it only ensures that at least one element in the communicated data is required by the receiver. The goal, however, is that all elements in the data sent from one compute device to another compute device should be required by the receiver compute device. The problem in determining this



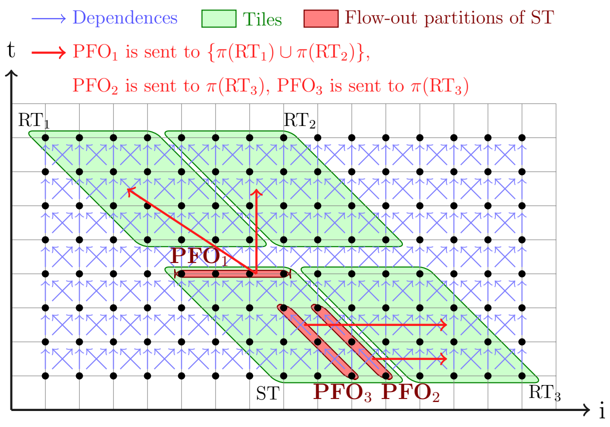
Dependences
Tiles
Flow-out set of ST
FO(ST) is sent to $\{\pi(\mathrm{RT}_1) \cup \pi(\mathrm{RT}_2) \cup \pi(\mathrm{RT}_3)\}$
t
i
RT_1
RT_2
RT_3
ST
FO

(a) FO scheme



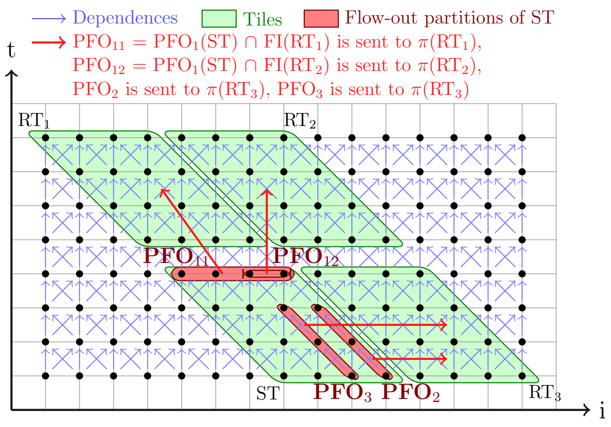
Dependences
Tiles
FOIFI sets of ST
$\mathrm{F}_1 = \mathrm{FO(ST)} \cap \mathrm{FI}(\mathrm{RT}_1)$ is sent to $\pi(\mathrm{RT}_1)$,
$\mathrm{F}_2 = \mathrm{FO(ST)} \cap \mathrm{FI}(\mathrm{RT}_2)$ is sent to $\pi(\mathrm{RT}_2)$,
$\mathrm{F}_3 = \mathrm{FO(ST)} \cap \mathrm{FI}(\mathrm{RT}_3)$ is sent to $\pi(\mathrm{RT}_3)$
t
i
RT_1
RT_2
RT_3
ST
F_1
F_2
F_3

(b) FOIFI scheme



Dependences
Tiles
Flow-out partitions of ST
PFO_1 is sent to $\{\pi(\mathrm{RT}_1) \cup \pi(\mathrm{RT}_2)\}$,
PFO_2 is sent to $\pi(\mathrm{RT}_3)$, PFO_3 is sent to $\pi(\mathrm{RT}_3)$
t
i
RT_1
RT_2
RT_3
ST
PFO_1
PFO_3
PFO_2

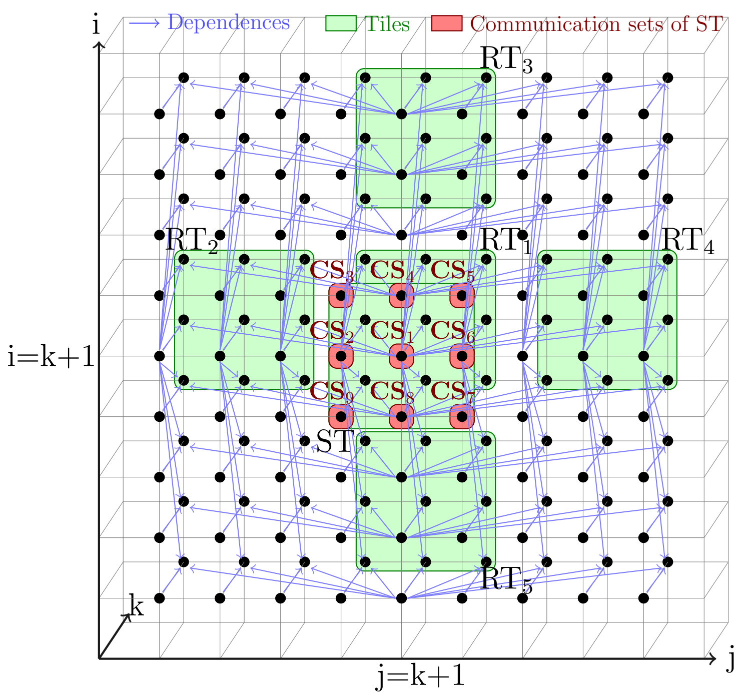
(c) FOP scheme using multicast-pack



Dependences
Tiles
Flow-out partitions of ST
$\mathrm{PFO}_{11} = \mathrm{PFO}_1(\mathrm{ST}) \cap \mathrm{FI}(\mathrm{RT}_1)$ is sent to $\pi(\mathrm{RT}_1)$,
$\mathrm{PFO}_{12} = \mathrm{PFO}_1(\mathrm{ST}) \cap \mathrm{FI}(\mathrm{RT}_2)$ is sent to $\pi(\mathrm{RT}_2)$,
PFO_2 is sent to $\pi(\mathrm{RT}_3)$, PFO_3 is sent to $\pi(\mathrm{RT}_3)$
t
i
RT_1
RT_2
RT_3
ST
PFO_{11}
PFO_{12}
PFO_3
PFO_2

(d) FOP scheme using unicast-pack

Figure 3.3: Illustration of data movement schemes for Jacobi-style stencil example



(a) Iteration space

$FO(ST) = CS_1 \cup CS_2 \cup CS_3 \cup CS_4 \cup CS_5 \cup CS_6 \cup CS_7 \cup CS_8 \cup CS_9$
FO(ST) is sent to $\{\pi(RT_1) \cup \pi(RT_2) \cup \pi(RT_3) \cup \pi(RT_4) \cup \pi(RT_5)\}$

(b) FO scheme

$F_1 = CS_1 \cup CS_2 \cup CS_3 \cup CS_4 \cup CS_5 \cup CS_6 \cup CS_7 \cup CS_8 \cup CS_9$
F_1 represents $FO(ST) \cap FI(RT_1)$
F_1 is sent to $\pi(RT_1)$
$F_2 = CS_1 \cup CS_4 \cup CS_8$
F_2 represents $FO(ST) \cap FI(RT_2)$ and $FO(ST) \cap FI(RT_4)$
F_2 is sent to $\pi(RT_2)$ and $\pi(RT_4)$
$F_3 = CS_1 \cup CS_2 \cup CS_6$
F_3 represents $FO(ST) \cap FI(RT_3)$ and $FO(ST) \cap FI(RT_5)$
F_3 is sent to $\pi(RT_3)$ and $\pi(RT_5)$

(c) FOIFI scheme

$PFO_1 = CS_1$
PFO_1 is sent to $\{\pi(RT_1) \cup \pi(RT_2) \cup \pi(RT_3) \cup \pi(RT_4) \cup \pi(RT_5)\}$
$PFO_2 = CS_4 \cup CS_8$
PFO_2 is sent to $\{\pi(RT_2) \cup \pi(RT_4)\}$
$PFO_3 = CS_2 \cup CS_6$
PFO_3 is sent to $\{\pi(RT_3) \cup \pi(RT_5)\}$
$PFO_4 = CS_3 \cup CS_5 \cup CS_7 \cup CS_9$
PFO_4 is sent to $\pi(RT_1)$

(d) FOP scheme using multicast-pack

Figure 3.4: Illustration of data movement schemes for Floyd-Warshall example (CS_i sets are used only for illustration; communication sets are determined as described in text)

at compile-time is that placement of iterations to compute devices is not known, even for a static computation distribution (like block-wise), since problem sizes and number of processes are not known. Nevertheless, data that needs to be sent from one iteration to another, parameterized on a sending iteration and a receiving iteration, can be determined precisely at compile-time.

3.3.1 Overview

For each innermost distributed loop, consider an iteration of it represented by iteration vector $\vec{i}$. For each data variable x, that can be a multidimensional array or a scalar, the following is determined at compile-time parameterized on $\vec{i}$:

- Flow set, $F_x(\vec{i} \rightarrow \vec{i'})$: the set of elements that need to be communicated from iteration $\vec{i}$ to iteration $\vec{i'}$ of an innermost distributed loop.
- Receiving iterations, $RI_x(\vec{i})$: the set of iterations of the innermost distributed loop(s) that require some element written in iteration $\vec{i}$.

Using these parameterized sets, code is generated to execute the following in each compute device c at runtime:

- `unicast-pack`: for each iteration $\vec{i}$ executed by c and iteration $\vec{i'} \in RI_x(\vec{i})$ that will be executed by another compute device $c' = \pi(\vec{i'})$ $(c' \neq c)$, pack $F_x(\vec{i} \rightarrow \vec{i'})$ into the local buffer associated with c',
- Send the packed buffers to the respective compute devices, and receive data from other compute devices,
- unpack corresponding to `unicast-pack`: for each iteration $\vec{i}$ executed by another compute device c' $(c' \neq c)$ and iteration $\vec{i'} \in RI_x(\vec{i})$ that will be executed by c, i.e., $\pi(\vec{i'}) = c$, unpack $F_x(\vec{i} \rightarrow \vec{i'})$ from the received buffer associated with c'.

Note that this scheme requires a distinct buffer for each receiving compute device. Packing is required since the flow set could be discontiguous in memory. Both the packing code and the unpacking code traverse the iterations $\vec{i}$ executed by a compute device, the receiving iterations $\vec{i'} \in RI_x(\vec{i})$, and the elements in $F_x(\vec{i} \rightarrow \vec{i'})$ in the same order. Therefore, the offset of an element in the packed buffer of the sending compute device matches that in the received buffer of the receiving compute device. The communication set of each receiver for all iterations executed by a compute device is communicated to that receiver at once, thereby achieving *communication vectorization*. Code generation for $RI_x(\vec{i})$ and $\pi(\vec{i})$ is same as that in FO scheme.

3.3.2 Flow-in set

The flow-in set represents the data that needs to be received by an iteration. The set of all values which flow to a read in an iteration from a write outside the iteration due to a RAW dependence is termed as the **per-dependence flow-in set** corresponding to that iteration and dependence. For a RAW dependence polyhedron D of data variable x whose target statement is in $\vec{i}$, the per-dependence flow-in set $DFI_x(\vec{i}, D)$ is determined by computing the region of data x read by those target iterations of D whose reads are written outside $\vec{i}$. The flow-in set of an iteration is the set of all values read by that iteration, and previously written outside the iteration. Therefore:

$$FI_x(\vec{i}) = \bigcup_{\forall D} DFI_x(\vec{i}, D) \tag{3.2}$$

3.3.3 Flow set

The data that needs to be sent from one iteration to another is represented by the flow set between the iterations. The flow set from an iteration $\vec{i}$ to an iteration $\vec{i'}$ ($\vec{i} \neq \vec{i'}$) is the set of all values written by $\vec{i}$, and then read by $\vec{i'}$. For each data variable x, the flow set F_x from an iteration $\vec{i}$ to an iteration $\vec{i'}$ is determined at compile-time by intersecting the flow-out set of $\vec{i}$ with the flow-in set of $\vec{i'}$:

$$F_x(\vec{i} \rightarrow \vec{i'}) = FO_x(\vec{i}) \cap FI_x(\vec{i'}) \tag{3.3}$$

Hence, this communication scheme is termed as the flow-out intersection flow-in (FOIFI) scheme. Since the flow set combines the data to be communicated due to multiple dependences, *communication coalescing* is implicitly achieved. Code is generated to enumerate the elements in the flow set between two given iterations at runtime.

3.3.4 Communication volume

The flow sets from different sending iterations are disjoint like the flow-out sets. In contrast, the flow-in sets of different iterations can overlap; different iterations can receive same data from the same sending iteration. This implies that the flow sets from a sending iteration to different receiving iterations need not be disjoint. So, when different receiving iterations of an iteration will be executed by the same compute device, a union of their flow sets is required to avoid duplication. The union cannot be performed at compile-time since the placement (or distribution) of iterations to compute devices is not known, while performing the union at runtime can be prohibitively expensive. This scheme could lead to duplication of data since it accumulates the flow sets to the buffer associated with the receiver compute device.

When each receiving iteration is executed by a different compute device, FOIFI scheme ensures that every element of the communicated data is required by the receiver, unlike FO scheme. The placement of iterations to compute devices, however, cannot be assumed. Different iterations can receive the same elements from the same sending iteration. So, when different receiving iterations of an iteration will be executed by the same compute device, this scheme could lead to duplication of data since it accumulates the flow sets to the buffer associated with the receiver compute device. For example, in Figure 3.3b, if RT_1 and RT_2 are executed by the same compute device, then F_2 is sent twice to that compute device. Similarly, in Figure 3.4c, if RT_1, RT_2 and RT_4 are executed by the same compute device, then F_2 is sent thrice to that compute device; the amount of duplication depends on the number of iterations a compute device executes in j dimension with the same i and k. Thus, this scheme could communicate a significantly large volume of duplicate data. The amount of redundancy cannot be theoretically bounded, and can be more than that of a naive scheme in the worst case.

3.4 Flow-out partitioning (FOP) scheme

FO scheme does not communicate duplicate data, but ignores whether a receiver requires most of the communication set or not. On the other hand, FOIFI scheme precisely computes the communication set required by a receiving iteration, but could lead to huge duplication when multiple receiving iterations are executed by the same compute device. A better approach is one that avoids communication of both duplicate and unnecessary elements. We show that this can be achieved by partitioning the communication set in a particular non-trivial way, and sending each partition to only its receivers.

The motivation behind partitioning the communication set is that different receivers could require different elements in the communication set. So ideally, the goal should be to partition the communication set such that all elements within each partition are required by all receivers of that partition. However, the receivers are not known at compile-time and partitioning at runtime is expensive. RAW dependences determine the receiving iterations, and ultimately, the receivers. Hence, we partition the communication set at compile-time, based on RAW dependences. To this end, we introduce new classifications for RAW dependences below.

Definition 1. *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.*

Consider a set of RAW dependence polyhedra S_D of an iteration $\vec{i}$. If S_D is *source-identical*, then:

$$DFO_x(\vec{i}, D_1) = DFO_x(\vec{i}, D_2) \quad \forall D_1, D_2 \in S_D \tag{3.4}$$

Definition 2. *Two source-identical sets of dependences are said to be* ***source-distinct*** *if the regions 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:

$$\begin{aligned} DFO_x(\vec{i}, D_1) \cap DFO_x(\vec{i}, D_2) = \emptyset \\ \forall D_1 \in S_D^1, D_2 \in S_D^2 \end{aligned} \tag{3.5}$$

DEFINITION 3. *A* ***source-distinct partitioning*** *of dependences partitions the dependences such that all dependences in a partition are source-identical and any two partitions are source-distinct.* (Note that a single dependence polyhedron might be partitioned into multiple dependence polyhedra).

A *source-identical* set of dependences determines a communication set identical for those dependences. Each such set in a *source-distinct* partitioning will therefore generate bookkeeping code to handle its own communication set. If the number of *source-identical* sets is more, then the overhead of executing the bookkeeping code might outweigh the benefits of reducing redundant communication. Hence, it is beneficial to reduce the number of *source-identical* sets of dependences, i.e., the *source-distinct* partitions. A source-distinct partitioning of dependences is said to be minimal if the number of partitions is minimum across all such partitioning of dependences.

3.4.1 Overview

For each innermost distributed loop, consider an iteration of it represented by iteration vector $\vec{i}$. For each data variable x, that can be a multidimensional array or a scalar, a minimal *source-distinct* partitioning of RAW dependence polyhedra, whose source statement is in $\vec{i}$, is determined at compile-time. For each *source-identical* set (partition) of RAW dependence polyhedra S_D, the following is determined parameterized on $\vec{i}$:

- Partitioned flow-out set, $PFO_x(\vec{i}, S_D)$: the set of elements that need to be communicated from iteration $\vec{i}$ due to S_D.

- Partitioned flow set, $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$: the set of elements that need to be communicated from iteration $\vec{i}$ to iteration $\vec{i'}$ due to S_D.

- Receiving iterations of the partition, $RI_x(\vec{i}, S_D)$: the set of iterations of the innermost distributed loop(s) that require some element in $PFO_x(\vec{i}, S_D)$.

Using these parameterized sets, code is generated to execute the following in each compute device c at runtime:

- For each *source-identical* set of RAW dependence polyhedra S_D and iteration $\vec{i}$ executed by c, execute one of these:
  - `multicast-pack`: for each other compute device c' ($c' \neq c$) that will execute some $\vec{i'} \in RI_x(\vec{i}, S_D)$, i.e., $c' = \pi(\vec{i'})$, pack $PFO_x(\vec{i}, S_D)$ into the local buffer associated with c',
  - `unicast-pack`: for each iteration $\vec{i'} \in RI_x(\vec{i}, S_D)$ that will be executed by another compute device $c' = \pi(\vec{i'})$ ($c' \neq c$), pack $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$ into the local buffer associated with c',
- Send the packed buffers to the respective compute devices, and receive data from other compute devices,
- For each *source-identical* set of RAW dependence polyhedra S_D and iteration $\vec{i}$ executed by another compute device c' ($c' \neq c$), execute one of these:
  - unpack corresponding to `multicast-pack`: if c will execute some $\vec{i'} \in RI_x(\vec{i}, S_D)$, i.e., $\pi(\vec{i'}) = c$, unpack $PFO_x(\vec{i}, S_D)$ from the received buffer associated with c',
  - unpack corresponding to `unicast-pack`: for each iteration $\vec{i'} \in RI_x(\vec{i}, S_D)$ that will be executed by c, i.e., $\pi(\vec{i'}) = c$, unpack $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$ from the received buffer associated with c'.

Note that this scheme requires a distinct buffer for each receiving compute device. Both the packing code and the unpacking code traverse the sets of RAW dependence polyhedra S_D, the iterations $\vec{i}$ executed by a compute device, the receiving iterations $\vec{i'} \in RI_x(\vec{i}, S_D)$, and the elements in $PFO_x(\vec{i}, S_D)$ or $PF_x(\vec{i} \rightarrow \vec{i'}, S_D)$ in the same order. Therefore, the offset of an element in the packed buffer of the sending compute device matches that in the received buffer of the receiving compute device. The communication set of each receiver for all iterations executed by a compute device is communicated to that receiver at once, thereby achieving `communication vectorization`. Code generation for $\pi(\vec{i})$ is same as that in FO scheme.

Algorithm 1: *source-distinct* partitioning of dependences

```
Input: RAW dependence polyhedra $D_i$ and $D_j$
1 $(I_S, A_S) \leftarrow$ source (iterations, access) of $D_i$
2 $(I_T, A_T) \leftarrow$ source (iterations, access) of $D_j$
3 $D \leftarrow$ dependence from $(I_S, A_S)$ to $(I_T, A_T)$
4 if $D$ is empty then
5     $D_S \leftarrow D_T \leftarrow$ empty
6     return
7 $(I'_S, I'_T) \leftarrow$ (source, target) iterations of $D$
8 $D_S \leftarrow$ source $I'_S$ and target unconstrained
9 $D_T \leftarrow$ source $I'_T$ and target unconstrained
Output: source-distinct partitions $\{D_i - D_S\}, \{D_j - D_T\}, \{D_i \cap D_S, D_j \cap D_T\}$
```

3.4.2 Partitioning of dependences

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. Such a dependence might not be semantically valid (e.g., when there is overlap in the regions of data that flow due to dependences with the same source statement). It is just a virtual dependence between two dependences, that captures the overlap in the regions of data that flow due to those dependences.

Algorithm 1 partitions two dependence polyhedra using this 'dependence between dependences' concept. 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*.

Before partitioning dependences whose source statement is in $\vec{i}$, each RAW dependence polyhedron D is restricted to those source iterations of D whose writes are read outside $\vec{i}$. Initially, each dependence is in a separate partition. For any two partitions, Algorithm 1 is used as a subroutine for each pair of dependences in different partitions; the *source-identical* set of new dependences, if any, formed by all of these pairs is a new partition. This is repeated until no new partitions can be formed, i.e., until all partitions are *source-distinct*. The number of dependences in each new partition is the sum of those in the two partitions, and cannot be more than the number of initial dependences. The source iterations in each new dependence polyhedron keep monotonically decreasing. So, the partitioning should terminate, and a *source-distinct* partitioning always exists for any set of dependences. Such a *source-distinct* partitioning is minimal since it creates a new partition only if it is necessary. This simple approach can be improved upon and optimized; it is presented as is for clarity of exposition.

3.4.3 Partitioned communication sets

If S_P is the set of *source-distinct* partitions of RAW dependence polyhedra whose source statement is in $\vec{i}$, then $\forall S_D \in S_P$:

$$\begin{aligned} PFO_x(\vec{i}, S_D) &= \bigcup_{\forall D \in S_D} DFO_x(\vec{i}, D) \\ &= DFO_x(\vec{i}, D) \;\; \forall D \in S_D \quad \text{(from (3.4))} \end{aligned} \tag{3.6}$$

$$PF_x(\vec{i} \to \vec{i'}, S_D) = PFO_x(\vec{i}, S_D) \cap FI_x(\vec{i'}) \tag{3.7}$$

From Equation (3.5) and (3.6):

$$\begin{gathered} PFO_x(\vec{i}, S_D^1) \cap PFO_x(\vec{i}, S_D^2) = \emptyset \\ \forall S_D^1, S_D^2 \in S_P \mid S_D^1 \neq S_D^2 \end{gathered} \tag{3.8}$$

From Equation (3.1) and Definition 3:

$$FO_x(\vec{i}) = \bigcup_{\forall S_D \in S_P} PFO_x(\vec{i}, S_D) \tag{3.9}$$

Hence, this communication scheme is termed as flow-out partitioning (FOP) scheme. Since the flow-out partitions are disjoint and each of them combines the data to be communicated due to multiple dependences, this scheme reduces duplication, thereby achieving *communication coalescing*.

3.4.4 Receiving iterations of the partition

For iteration $\vec{i}$ and each RAW dependence polyhedron D in the set of RAW dependence polyhedra S_D, $RI_x(\vec{i}, S_D)$ is determined by projecting out dimensions inner to $\vec{i}$ in D and scanning the target iterators while treating the source iterators as parameters. To determine the compute devices to communicate with, the generated code makes use of π on each receiving iteration of this partition to aggregate the set of distinct receivers that require at least one element in $PFO_x(\vec{i}, S_D)$.

3.4.5 Packing and unpacking

Code for `multicast-pack` and `unicast-pack`, the two methods of packing, is generated similar to that in FO and FOIFI schemes respectively. For each iteration and partition, either `multicast-pack` or `unicast-pack` is executed, the choice of which could be determined either at compile-time or at runtime. Since the method of packing determines the communication volume, the goal of choosing the method of packing is to minimize redundant communication. We choose:

- `unicast-pack` at *compile-time* if the set of dependence polyhedra determining the partition contains only one dependence polyhedron such that each source iteration in it has at most one target iteration dependent on it: each element in the partition is required by only one receiving iteration of the partition, and therefore, there is no duplication in the communicated data for any placement of iterations to compute devices.
- `multicast-pack` at *compile-time* if the flow-in of the partition is independent of the parallel dimension(s): each element in the partition is required by all receiving iterations of the partition, and therefore, unnecessary data is not communicated for any placement of iterations to compute devices.
- `unicast-pack` at *runtime* if each receiving iteration of the executed iteration and partition is allocated to a different compute device: there is no duplication in the communicated data since each element of the communicated data is required by only one iteration that will executed by the receiving compute device.
- `multicast-pack` at *runtime* if all the receiving iterations of the executed iteration and partition are allocated to the same compute device: unnecessary data is not communicated since each element of the communicated data is required by some iteration that will be executed by that compute device.

These conditions ensure no redundancy in communication. In the absence or failure of these non-redundancy conditions, we choose `multicast-pack` since `unicast-pack` could lead to more redundant communication in the worst case.

3.4.6 Communication volume

The flow (RAW) dependence polyhedra for the Floyd-Warshall example in Figure 3.2, where (k, i, j) is the source iteration and (k', i', j') the target iteration, are: (the bounds on i, j, k are omitted for brevity)

$$\begin{aligned} D_1 &= \{k' = k+1, i' = i, j' = j\} \\ D_2 &= \{k' = k+1, i' = i, j = k+1, 0 <= j' <= N-1\} \\ D_3 &= \{k' = k+1, i = k+1, j' = j, 0 <= i' <= N-1\} \end{aligned}$$

The *source-distinct* partitioning of these dependence polyhedra yields four partitions: P_1 containing subsets of D_1, D_2 and D_3; P_2 containing subsets of D_1 and D_2; P_3 containing subsets of D_1 and D_3; P_4 containing a subset of D_1. As shown in Figure 3.4d, PFO_1, PFO_2, PFO_3 and PFO_4 are the partitioned flow-out sets of P_1, P_2, P_3 and P_4 respectively, which are sent to their respective receivers using `multicast-pack`. There is no redundancy in communication for any placement of iterations to compute devices.

For the Jacobi-style stencil example in Figure 3.1, where (t, i) is the source iteration and (t', i') the target iteration, the flow (RAW) dependence polyhedra are:
(the bounds on t, i are omitted for brevity)

$$\begin{aligned} D_1 &= \{t' = t+1, i' = i+1\} \\ D_2 &= \{t' = t+1, i' = i\} \\ D_3 &= \{t' = t+1, i' = i-1\} \end{aligned}$$

The *source-distinct* partitioning of these dependence polyhedra yields three partitions: P_1 containing subsets of D_1, D_2 and D_3; P_2 containing subsets of D_1 and D_2; P_3 containing a subset of D_1. As shown in Figure 3.3c, PFO_1, PFO_2 and PFO_3 are the partitioned flow-out sets of P_1, P_2 and P_3 respectively. There could be some redundancy in communication for PFO_1, depending on the placement of iterations to compute devices. However, the non-redundancy conditions can be used to choose between `unicast-pack` and `multicast-pack` at runtime. If `unicast-pack` is chosen for PFO_1 as shown in Figure 3.3d when RT_1 and RT_2 are allocated to different compute devices, and `multicast-pack` is chosen otherwise (Figure 3.3c) (in this case, RT_1 and RT_2 are allocated to same compute device), then there is no redundancy in communication for any placement of iterations to compute devices.

In general, if `unicast-pack` is used for all partitions, then FOP scheme behaves similar to FOIFI scheme. If `multicast-pack` is used for all partitions, then the communication volume of FOP scheme cannot be more than that of FO scheme. Depending on the method of packing, FOP scheme is at least as good as FO and FOIFI schemes. FOP scheme is effective in minimizing redundant communication since the partitions of the communication set reduce the granularity at which receivers are determined. To further minimize redundant data movement, FOP schemes uses the non-redundancy conditions to choose between `unicast-pack` and `multicast-pack`; the communication set partitions also reduce the granularity at which these conditions are applied. Hence, FOP scheme minimizes communication volume better than FO and FOIFI schemes.

3.5 Implementation

Our framework is fully implemented as part of a publicly available source-to-source polyhedral tool chain. Clan [11], ISL [50], Pluto [14], and Cloog-isl [10] are used to perform polyhedral extraction, dependence testing, automatic transformation, and code generation, respectively. Polylib [42] is used to implement the polyhedral operations in Sections 3.2.1, 3.3 and 3.4. ISL [50] is used to eliminate transitive dependences and compute *last writers* or the *exact dataflow*. This ensures that when there are multiple writes to a location before a subsequent read due to transitively covered RAW dependences, only the *last write* to the location is communicated to the compute device that reads it.

The input to our framework is sequential code containing arbitrarily nested affine loop nests, which is tiled and parallelized using the Pluto algorithm [5, 15]; loop tiling helps reduce the bookkeeping overhead at runtime while being precise in communication. Communication sets are statically determined for a parametric tile. Our framework then automatically generates code for distributed-memory systems from this transformed code using techniques described in the work of Bondhugula [13]. The entire data is initially made available in every compute device, and the final result is collected at the master compute device, without violating sequential semantics. FO, FOIFI and FOP schemes take the parallelized code as input and insert data movement code soon after each parallelized loop nest. So, at runtime, data movement code is executed after each distributed phase. In FOP, partitions with the same receiving tile constraints are merged and partitions with constant volume are merged at compile-time so as to minimize the bookkeeping overhead at runtime without sacrificing communication volume. Non-blocking MPI primitives are used to communicate between nodes in the distributed-memory system.

For heterogeneous systems, the host CPU acts both as a compute device and as the orchestrator of data movement between compute devices, while the GPU acts only as a compute device. The data movement code is automatically generated in terms of OpenCL calls invoked from the host CPU. Each compute device has an associated management thread on the host CPU. This thread is responsible for launching computation kernels and handling data movement to and from the compute device it is managing. The computations are distributed onto each device at the granularity of tiles. Once all the tiles are executed on a compute device, the management thread for that device calls the `copyOut` function which copies the communication set for each computed tile from the source device onto the host CPU. Once `copyOut()` completes, the management thread issues the `copyIn` function which copies the communication set now residing on the host CPU onto the destination compute devices. On heterogeneous systems, packing functionally corresponds to `copyOut()` and unpacking corresponds to `copyIn()`. We notice that, for most of the cases, the data to be communicated is in a rectangular region of memory. OpenCL 1.1 provides `clEnqueueReadBufferRect()` and `clEnqueueWriteBufferRect()` to copy such rectangular regions of data in a single call. We make use of these functions everywhere to minimize the number of OpenCL calls and thereby minimize the associated call overheads.

3.6 Experimental evaluation

In this section, we evaluate our data movement techniques on a distributed-memory cluster, and on heterogeneous systems with CPUs and multiple GPUs.

We compare FOP, FOIFI and FO schemes using the same parallelizing transformation. This implies that the frequency of communication is the same for them. The schemes differ only in the communication volume, and in the way the data is packed and unpacked. Since everything else is the same, comparing the total execution times of these communication schemes directly compares their efficiency.

3.6.1 Distributed-memory architectures

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 MT25204 (InfiniHost III Lx HCA). All nodes run 64-bit Linux kernel version 2.6.18. The cluster uses MVAPICH2-1.8 as the MPI implementation. It provides a point-to-point latency of 3.36 μs, unidirectional and bidirectional bandwidths of 1.5 GB/s and 2.56 GB/s respectively. All codes were compiled with Intel C compiler (ICC) version 11.1 with flags '-O3 -fp-model precise'. All Unified Parallel C (UPC) codes were compiled with Berkeley Unified Parallel C compiler [19] version 2.16.0.

Benchmarks

We present results for Floyd Warshall (`floyd`), LU Decomposition (`lu`), Alternating Direction Implicit solver (`adi`), 2-D Finite Different Time Domain Kernel (`fdtd-2d`), Heat 2D equation (`heat-2d`) and Heat 3D equation (`heat-3d`) benchmarks. The first four are from the publicly available Polybench/C 3.2 suite [41]; `heat-2d` and `heat-3d` are widely used stencil computations [47]. 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. The outermost parallel loop in `heat-2d`, `heat-3d` and `adi` was marked as parallel using OpenMP, and given as input to OMPD [34]; since the data to be communicated in `floyd` and `lu` is dependent on the outer serial loop, OMPD does not handle them. In our tool which implements FOP, FOIFI and FO, the benchmark itself was the input and tile sizes were chosen such that the performance on a single node is optimized, as listed in Table 3.1. All benchmarks use double-precision floating-point operations. Problem sizes used are listed in Table 3.1.

Evaluation

Though our tool generates MPI+OpenMP code, we ran all the benchmarks with one OpenMP thread per process and one MPI process per node to focus on the distributed-memory part. We evaluate both the communication volume and execution time of FO, FOIFI and FOP. In addition, we intend to evaluate systems that provide abstractions to free the user from the burden of moving data on distributed-memory architectures, either fully automatically like ours or at least those that provided some runtime support. OMPD falls into the former category, while UPC falls into the latter category. Though the computation transformations or the underlying runtime may be different, we compare the execution time of OMPD and UPC with FOP with an intention of evaluating the efficiency of these approaches.

Benchmark	**Problem sizes**	**Tile sizes**
floyd	N=8192	64 (2d)
lu	N=4096	64 (3d)
fdtd-2d	N=4096, T=1024	16 (2d)
heat-2d	N=8192, T=1024	256 (3d)
heat-3d	N=512, T=256	16 (4d)
adi	N=8192, T=128	256 (2d)

Table 3.1: Problem and tile sizes - distributed-memory cluster

Benchmark	**4 nodes**			**8 nodes**		
	FOP	**FOIFI**	**FO**	**FOP**	**FOIFI**	**FO**
floyd	1.51GB	31.8×	63.5×	3.53GB	15.9×	63.5×
lu	0.45GB	5.3×	1.4×	0.99GB	3.0×	1.4×
fdtd-2d	0.21GB	1.0×	14.3×	0.47GB	1.0×	15.1×
heat-2d	0.75GB	1.0×	2.0×	1.74GB	1.0×	2.0×
heat-3d	5.61GB	1.0×	2.0×	13.09GB	1.0×	2.0×
adi	191.24GB	1.0×	4.0×	223.11GB	1.0×	8.0×

Benchmark	**16 nodes**			**32 nodes**		
	FOP	**FOIFI**	**FO**	**FOP**	**FOIFI**	**FO**
floyd	7.56GB	7.9×	63.5×	15.62GB	4.0×	63.5×
lu	1.88GB	1.9×	1.4×	2.59GB	1.5×	1.5×
fdtd-2d	0.97GB	1.0×	15.5×	1.97GB	1.0×	15.7×
heat-2d	3.73GB	1.0×	2.0×	7.72GB	1.0×	2.0×
heat-3d	28.07GB	1.0×	2.0×	58.01GB	1.0×	2.0×
adi	239.05GB	1.0×	16.0×	247.02GB	1.0×	32.0×

Table 3.2: Total communication volume on the distributed-memory cluster – FO and FOIFI normalized to FOP

Analysis

Table 3.2 compares the total communication volume of FO, FOIFI and FOP. Table 3.3 compares the total execution time of hand-optimized UPC codes with that of OMPD, FO, FOIFI and FOP. `seq` – sequential time, is the time taken to run the serial code compiled with ICC. Execution time of one node is different for each of the schemes, indicating the bookkeeping overhead when there is no data to be packed, unpacked or communicated. Execution time of FO, FOIFI and FOP on one node is different from `seq` time due to tiling and other loop transformations performed by our tool on the sequential code.

Across all benchmarks and number of nodes, FOP reduced communication volume by a factor of 1.4× to 63.5× over FO. This translates to a huge improvement in execution time, except for `heat-2d`, where the communication time is a minor component of the total execution time (less than 2% in most cases). For `adi` and `floyd` which are communication intensive, the reduction in communication volume for FOP gives up to 15.9× speedup over FO.

Nodes	FOP	FOIFI	FO	UPC
1	2065.2s	1.01×	1.00×	0.97×
4	521.4s	1.10×	1.20×	0.97×
8	263.9s	1.18×	1.75×	0.97×
16	137.6s	1.33×	3.93×	0.97×
32	81.1s	1.46×	11.18×	0.93×

(a) floyd – `seq` time is 2012s

Nodes	FOP	FOIFI	FO	OMPD	UPC
1	228.3s	1.00×	1.00×	3.42×	5.33×
4	59.8s	1.00×	1.01×	3.29×	5.11×
8	31.4s	1.00×	1.02×	3.92×	5.47×
16	17.3s	1.00×	1.03×	3.58×	5.00×
32	10.2s	1.00×	1.04×	3.06×	4.25×

(b) heat-2d – `seq` time is 796.4s

Nodes	FOP	FOIFI	FO	UPC
1	29.5s	1.00×	1.00×	2.86×
4	9.1s	1.42×	1.02×	2.42×
8	5.4s	1.70×	1.05×	2.30×
16	4.1s	1.84×	1.05×	1.50×
32	3.9s	1.58×	1.00×	1.25×

(c) lu – `seq` time is 82.9s

Nodes	FOP	FOIFI	FO	OMPD	UPC
1	235.5s	1.00×	1.00×	2.51×	2.68×
4	65.4s	1.00×	1.05×	2.39×	2.46×
8	36.1s	1.00×	1.15×	2.82×	2.54×
16	21.4s	1.00×	1.23×	2.58×	2.21×
32	14.1s	1.00×	1.33×	2.29×	1.78×

(d) heat-3d – `seq` time is 590.6s

Nodes	FOP	FOIFI	FO	UPC
1	359.5s	1.00×	1.00×	0.98×
4	90.8s	1.00×	1.03×	1.26×
8	66.9s	1.00×	1.04×	1.01×
16	33.8s	1.00×	1.09×	1.01×
32	16.8s	1.00×	1.24×	0.99×

(e) fdtd-2d – `seq` time is 351.7s

Nodes	FOP	FOIFI	FO	OMPD	UPC
1	422.7s	1.00×	0.95×	6.27×	7.90×
4	231.7s	1.00×	2.11×	3.55×	4.68×
8	143.6s	1.00×	4.00×	3.43×	4.29×
16	78.6s	1.00×	7.87×	2.88×	4.47×
32	41.0s	1.00×	15.9×	2.95×	5.22×

(f) adi – `seq` time is 2717s

Table 3.3: Total execution time on the distributed-memory cluster – FOIFI, FO, OMPD and UPC normalized to FOP

In `adi`, data written by a compute device should be distributed across all compute devices (matrix transpose) twice every outer serial loop iteration. FO just sends the entire data written by a compute device to all other compute devices, whereas FOP and FOIFI use `unicast-pack` (satisfies the non-redundancy condition at compile-time) to only send what is required by the receiver compute device, thereby eliminating redundant communication.

For `floyd`, FO communicates around 63.5× more than FOP. The redundancy in communication is closely related to the tile size, which is 64 in this case. As seen in Figure 3.4b, FO broadcasts the entire region of data written by the tile when it has some receivers. On the other hand, as seen in Figure 3.4d, FOP sends partitions of the data set to their own receivers. Since the same distribution is being used for every outer sequential iteration, four of those partitions (corner partitions) are not communicated, which significantly reduces the communication volume. Since communication time is a major component of the total execution time (more than 50% in some cases), the reduction in communication volume for FOP gives 1.53× to 5.74× speedup over FO for 4 or more nodes.

For `floyd` and `lu`, FOIFI communicates 1.5× to 31.8× more volume of data than FOP due to duplication in communicated data when multiple receiving iterations are placed (executed) on the same node. This yields 1.1× to 1.84× speedup of FOP over FOIFI. For `lu`, the redundancy in communication of FOIFI is much more than even that of FO and consequently, FOIFI performs much worse than FO. In summary, FOP gives a mean speedup of 1.55× and 1.11× over FO and FOIFI respectively across all benchmarks and number of nodes.

OMPD communicates the same volume of data as FOP for `adi`. However, OMPD incurs some additional overhead since it determines the communication sets at runtime. Moreover, our tool automatically tiles the parallel loop. For `heat-2d` and `heat-3d`, our tool transforms and tiles the code to yield both locality and load balance [5]. OMPD cannot handle such transformed code since the communication set is dependent on the outer serial loop. Even though OMPD is communicating the minimum volume of data for the non-transformed codes, FOP gives a mean speedup of 3.06× over OMPD across all benchmarks and number of nodes since it handles transformed codes with lesser runtime overhead.

Manually optimized UPC codes communicate the same volume of data for `fdtd-2d` and `floyd` as FOP. Since the data to be communicated is contiguous in global memory, UPC code has no additional overhead, and so, performs slightly better than FOP. On the other hand, the data to be communicated for `adi` is not contiguous in global memory. FOP packs and unpacks such non-contiguous data with very little runtime overhead while UPC incurs significant runtime overhead to handle such multiple shared memory requests to non-contiguous data. So, even though the data to be communicated is the same, FOP outperforms UPC code. For `lu`, `heat-2d` and `heat-3d`, manually writing

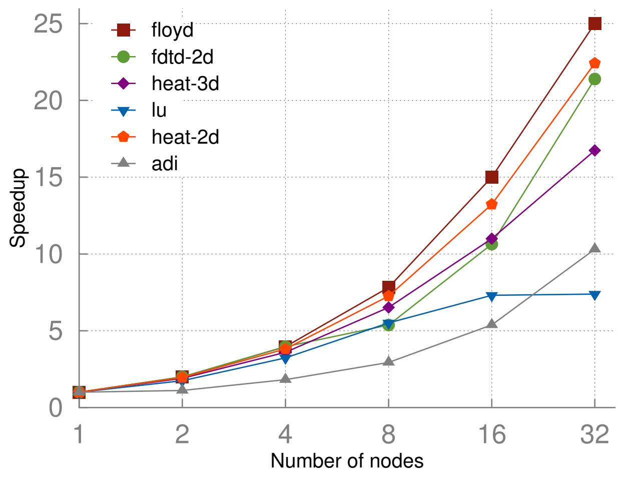


Figure 3.5: FOP – strong scaling on the distributed-memory cluster

code incorporating the transformations automatically used by our tool is not trivial. Moreover, the UPC model is not suitable when the same data element could be written by different nodes in different parallel phases, as is the case with these transformations; without such transformations, UPC code performs poorly. Due to these limitations of UPC, FOP gives a mean speedup of 2.19× over hand-optimized UPC codes across all benchmarks and number of nodes.

Manually determining communication sets for the tiled and transformed code is not trivial – this prohibits a fair comparison of our schemes with hand-written MPI codes. However, for the transformations and placement chosen in the benchmarks, we manually verified that FOP was achieving the minimum communication volume across different number of nodes, resulting in the best performance and facilitating the benchmarks to scale well. As shown in Figure 3.5, execution times of FOP decrease as the number of nodes are increased for all benchmarks, except for `lu` going from 16 to 32 nodes. In this case, performance does not improve by much due to the large volume of data that is required to be communicated. Nevertheless, `floyd` with the existing FO could not scale beyond 4 nodes, while FOP enables scaling similar to hand-optimized UPC codes as shown in Figure 3.6.

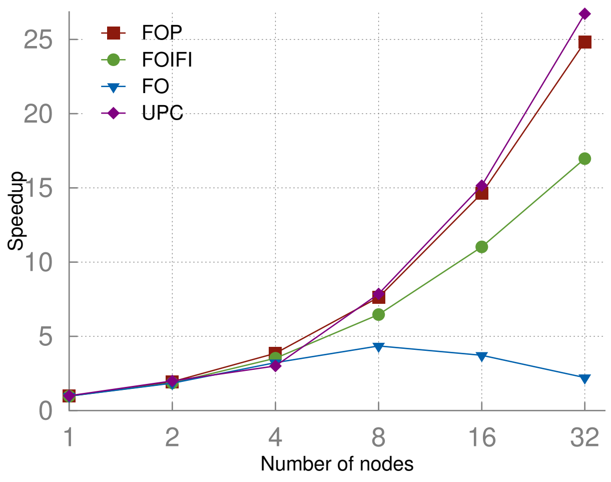


Figure 3.6: `floyd` – speedup of FOP, FOIFI, FO and hand-optimized UPC code over `seq` on the distributed-memory cluster

3.6.2 Heterogeneous architectures

Intel-NVIDIA system setup

The Intel-NVIDIA system consists of an Intel Xeon multicore server consisting of 12 Xeon E5645 cores running at 2.4 GHz. The server has 4 NVIDIA Tesla C2050 graphics processors connected on the PCI express bus, each having 2.5 GB of global memory. NVIDIA driver version 304.64 supporting OpenCL 1.1 was used as the OpenCL runtime. Double-precision floating-point operations were used in all benchmarks. The host codes were compiled with gcc version 4.4 with -O3. The problem sizes are chosen such that the entire array data fits within each GPU's global memory. Problem and tile sizes used are listed in Table 3.4.

AMD system setup

The AMD system consists of a AMD A8-3850 Fusion APU, consisting of 4 CPU cores running at 2.9 GHz and an integrated GPU based on the AMD Radeon HD 6550D architecture. The system has two ATI FirePro V4800 discrete graphics processors connected on the PCI express bus, each having 512 MB of global memory. Since these GPUs do not support double-precision floating-point operations, we use single-precision floating-point operations in all benchmarks. AMD driver version 9.82 supporting OpenCL 1.2 was used as the OpenCL runtime. The host codes were compiled with g++ version 4.6.1 with -O3. The problem sizes are chosen such that the entire array data fits within each GPU's global memory. Problem and tile sizes used are listed in Table 3.5.

Benchmarks

We evaluate FO and FOP for `floyd`, `lu`, `fdtd-2d`, `heat-2d` and `heat-3d` benchmarks. All these benchmarks have an outer serial loop containing a set of inner parallel loops. Wherever multiple nested parallel loops existed, the outermost among them was distributed across devices. The OpenCL kernels were manually written by mapping the parallel loops in a DOALL manner onto the OpenCL work groups and work items.

Benchmark	**Problem sizes**	**Tile sizes**
floyd	N=10240	32 (2d)
lu	N=11264	256 (2d)
fdtd-2d	N=10240, T=4096	32 (2d)
heat-2d	N=10240, T=4096	32 (2d)
heat-3d	N=512, T=4096	32 (3d)

Table 3.4: Problem and tile sizes - Intel-NVIDIA system

Benchmark	**Problem sizes**	**Tile sizes**
floyd	N=10240	32 (2d)
fdtd-2d	N=5120, T=4096	32 (2d)
heat-2d	N=8192, T=4096	32 (2d)

Table 3.5: Problem and tile sizes - AMD system

Evaluation

We consider the following combination of compute devices: (1) 1 CPU, (2) 1 GPU, (3) 1 CPU + 1 GPU, (4) 2 GPUs, (5) 4 GPUs. We evaluate FO and FOP on the Intel-NVIDIA system for all these cases. On the AMD system, we evaluate FO and FOP for (1), (2) and (4) cases, using only the discrete GPUs. In the first two cases, the devices

Benchmark	Device combination	Total execution time				Total communication volume		
		-	FOP	FO	Speedup	FOP	FO	Reduction
floyd	1 CPU (12 cores)	890s	–	–	–	–	–	–
	1 GPU	113s	–	–	–	–	–	–
	1 CPU + 1 GPU	–	148s	180s	1.22	0.8 GB	25.0 GB	32
	2 GPUs	–	65s	104s	1.60	1.6 GB	51.0 GB	32
	4 GPUs	–	43s	107s	2.49	3.1 GB	102.0 GB	32
lu	1 CPU (12 cores)	412s	–	–	–	–	–	–
	1 GPU	77s	–	–	–	–	–	–
	1 CPU + 1 GPU	–	92s	132s	1.43	0.9 GB	63 GB	70
	2 GPUs	–	64s	147s	2.30	0.7 GB	62.0 GB	83
	4 GPUs	–	60s	208s	3.47	1.2 GB	63.0 GB	51
fdtd-2d	1 CPU (12 cores)	1915s	–	–	–	–	–	–
	1 GPU	397s	–	–	–	–	–	–
	1 CPU + 1 GPU	–	580s	603s	1.03	0.9 GB	11.0 GB	11
	2 GPUs	–	207s	236s	1.14	0.9 GB	22.0 GB	22
	4 GPUs	–	117s	164s	1.40	2.2 GB	62.0 GB	28
heat-2d	1 CPU (12 cores)	1112s	–	–	–	–	–	–
	1 GPU	266s	–	–	–	–	–	–
	1 CPU + 1 GPU	–	242s	255s	1.05	0.6 GB	21.0 GB	32
	2 GPUs	–	138s	157s	1.14	0.6 GB	21.0 GB	32
	4 GPUs	–	80s	124s	1.55	1.9 GB	62.0 GB	32
heat-3d	1 CPU (12 cores)	3080s	–	–	–	–	–	–
	1 GPU	1932s	–	–	–	–	–	–
	1 CPU + 1 GPU	–	1718s	2018s	1.17	16.0 GB	512.0 GB	32
	2 GPUs	–	1086s	1379s	1.26	16.0 GB	512.0 GB	32
	4 GPUs	–	670s	1658s	2.47	49.0 GB	1535.4 GB	32

Table 3.6: Total communication volume and execution time of FO and FOP on the Intel-NVIDIA system

Benchmark	Device combination	Total execution time				Total communication volume		
		-	FOP	FO	Speedup	FOP	FO	Reduction
floyd	1 CPU (4 cores)	1084s	–	–	–	–	–	–
	1 GPU	512s	–	–	–	–	–	–
	2 GPUs	–	286s	305s	1.07	0.8 GB	25.0 GB	32
fdtd-2d	1 CPU (4 cores)	1529s	–	–	–	–	–	–
	1 GPU	241s	–	–	–	–	–	–
	2 GPUs	–	133s	242s	1.82	0.2 GB	2.15 GB	17
heat-2d	1 CPU (4 cores)	3654s	–	–	–	–	–	–
	1 GPU	256s	–	–	–	–	–	–
	2 GPUs	–	142s	353s	2.49	0.25 GB	8.0 GB	32

Table 3.7: Total communication volume and execution time of FO and FOP on the AMD system

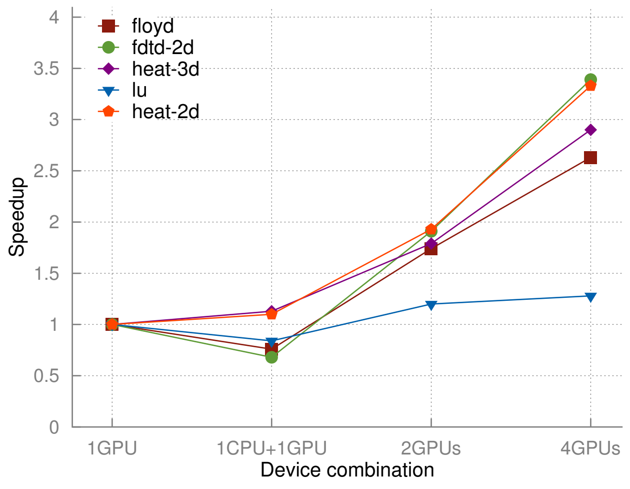


Figure 3.7: FOP – strong scaling on the Intel-NVIDIA system

run the entire OpenCL kernel. For cases (3), (4) and (5), kernel execution is partitioned across devices. For (4) and (5), the computation is equally distributed (block-wise). Since the CPU and GPUs have different compute powers, the computation distributions were chosen to be asymmetric for case (3). For all benchmarks, case (3) had 10% of computation distributed onto the CPU and 90% onto the GPU.

Analysis

Table 3.6 shows results obtained on the Intel-NVIDIA system. For all benchmarks, the running time on 1 GPU is much lower than that on the 12-core CPU. This running time is further improved by distributing the computation onto 2 and 4 GPUs. For all benchmarks, we see that FOP significantly reduces communication volume over FO. The computation tile sizes directly affects the communication volume (e.g., 32× for `floyd`). For the transformations and placement chosen for these benchmarks, we manually verified that FOP achieved the minimum communication volume across different device combinations. This reduction in communication volume results in a corresponding reduction in execution time facilitating strong scaling of these benchmarks, as shown in Figure 3.7 – this was not possible with the existing FO. For example, FO for `heat-3d` has very high communication overhead and does not scale beyond two GPUs. For `floyd` and `lu`, FO scales up to 2 GPUs, but not beyond it. However, FOP easily scales up to 4 GPUs for all benchmarks. For `floyd`, `lu` and `fdtd-2d`, CPU's performance becomes the bottleneck, even when it only executed 10% of the computation. Hence, we observe 1 CPU + 1 GPU performance to be worse than 1 GPU performance for these benchmarks. On the other hand, 1 CPU + 1 GPU gives 9% and 11% improvement over 1 GPU for `heat-2d` and `heat-3d` respectively. FOP gives a mean speedup of 1.53× over FO across all benchmarks and applicable device combinations.

Table 3.7 shows results obtained on the AMD system. The OpenCL functions used to transfer rectangular regions of memory are crucial for copying non-contiguous (strided) data efficiently. We found these functions to have a prohibitively high overhead on this system. This compelled us to use only those functions which could copy contiguous regions of memory. Hence, we present results only for `floyd`, `heat-2d` and `fdtd-2d` since the data to be moved for these benchmarks is contiguous. For all benchmarks, the running time on 1 GPU is much lower than that on the 4-core CPU. We could not evaluate them on 1 CPU + 1 GPU since the OpenCL data transfer functions crashed when CPU was used as an OpenCL device. Distributing computation of `floyd` onto 2 GPUs with FO performs better than 1 GPU, even though FO communicates large amounts of redundant data, because the compute-to-copy ratio is high in this case. However, FO does not perform well on 2 GPUs for `heat-2d` and `fdtd-2d` since these benchmarks have a low compute-to-copy ratio and the high volume of communication in FO leads to a slowdown. The FOP scheme, on the other hand, performs very well on 2 GPUs, yielding a near-ideal speedup of 1.8× over 1 GPU for all benchmarks.

Chapter 4

Targeting a Dataflow Runtime

In this chapter, we present our work on compiling for a dataflow runtime using our data movement techniques described in Chapter 3. Section 4.1 provides background on runtime design issues. Section 4.2 describes the design of our compiler-assisted dataflow runtime. Section 4.3 describes our implementation and experimental evaluation is presented in Section 4.4.

4.1 Motivation and design challenges

In this section, we discuss the motivation, challenges and objectives in designing a compiler-assisted dataflow runtime.

4.1.1 Dataflow and memory-based dependences

It is well known that flow dependences lead to communication when parallelizing across nodes with private address spaces. Previous work [13] has shown that when multiple writes to an element occur on different nodes before a read to it, only the *last write* value before the read can be communicated using non-transitive flow dependences. Previous work [13] has also shown that the *last write* value of an element across the iteration space can be determined independently (*write-out set*). Memory-based dependences, namely anti and output dependences, do not lead to communication but have to be preserved when parallelizing across multiple cores that share memory. We will see that a compiler that targets a runtime for a distributed-memory cluster of multicores should pay special attention to these.

4.1.2 Terminology

Tasks

Task is a part of a program that represents an atomic unit of computation. A task is to be atomically executed by a single thread, but multiple tasks can be simultaneously executed by different threads in different nodes. Each task can have multiple accesses to multiple shared data variables. A flow (RAW) data dependence from one task to another would require the data written by the former to be communicated to the latter, if they will be executed on different nodes. Even otherwise, it enforces a constraint on the order of execution of those tasks, i.e., the dependent task can only execute after the source task has executed. Anti (WAR) and output (WAW) data dependences between two tasks are memory-based, and do not determine communication. Since two tasks that will be executed on different nodes do not share an address space, memory-based data dependences between them do not enforce a constraint on their order of execution. On the other hand, for tasks that will be executed on the same node, memory-based data dependences do enforce a constraint on their order of execution, since they share the local memory.

There could be many data dependences between two tasks with source access in one task and target access in the other. All these data dependences can be encapsulated in one inter-task dependence to enforce that the dependent task executes after the source task. So, it is sufficient to have only one inter-task dependence from one task to another that represents all data dependences whose source access is in the former and target access is in the latter. In addition, it is necessary to differentiate between an inter-task dependence that is only due to memory-based dependences, and one that is also

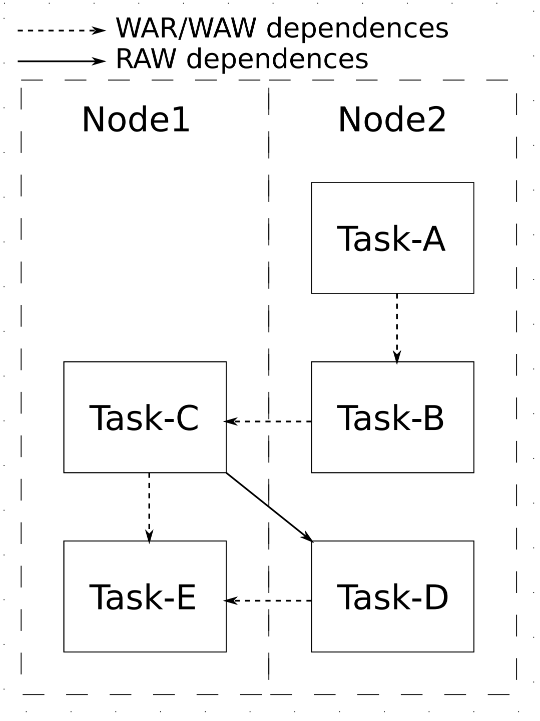


Figure 4.1: Inter-task dependences example

due to a flow dependence. If two tasks will be executed on different nodes, an inter-task dependence between them that is only due to memory-based dependences does not enforce a constraint on the order of execution. Finally, our notion of task here is same as that of a "codelet" in the codelet execution model [52].

Scheduling tasks

Consider the example shown in Figure 4.1, where there are 5 tasks Task-A, Task-B, Task-C, Task-D and Task-E. The inter-task dependences determine when a task can be scheduled for execution. For instance, the execution of Task-A, Task-B, Task-C, Task-D and Task-E in that order by a single thread on a single node is valid since it does not violate any inter-task dependence. Let Task-A, Task-B and Task-D be executed on Node2, while Task-C and Task-E be executed on Node1, as shown in Figure 4.1. On Node2, Task-A can be scheduled for execution since it does not depend on any task. Since Task-B depends on Task-A, it can only be scheduled for execution after Task-A has finished execution. Task-C in Node1 depends on Task-B in Node2, but the dependence is only due to WAR or WAW data dependences. So, Task-C can be scheduled for execution immediately. Similarly, Task-E in Node1 can ignore its WAR or WAW dependence on Task-D in Node2, but it has to wait for Task-C's completion before it can be scheduled for execution. On the other hand, Task-D in Node2 depends on Task-C in Node1, and it can only be scheduled for execution once it receives the required data from Task-C.

4.1.3 Synchronization and communication code

On shared-memory, threads use synchronization constructs to coordinate access to shared data. *Bulk synchronization* is a common technique used in conjunction with loop parallelism to ensure that all threads exiting it are able to see writes performed by others. For distributed-memory, data is shared typically through message passing communication code. Nodes in a distributed-memory cluster are typically shared-memory multicores. Bulk synchronization of threads running on these cores could lead to under-utilization of threads. Dynamically scheduling tasks on threads within each node eliminates bulk synchronization and balances the load among the threads better. Hence, dynamic scheduling would scale better than static scheduling as the number of threads per node increase.

Globally synchronized communication in a distributed cluster of nodes has significant runtime overhead. Asynchronous point-to-point communication not only reduces runtime overhead, but also allows overlapping computation with communication. Even with a single thread on each node, dynamically scheduling tasks within each node with asynchronous point-to-point communication would significantly outperform statically scheduled tasks with globally synchronized communication.

To dynamically schedule tasks, inter-task dependences are used at runtime. If the task dependence graph is built and maintained in shared-memory, then the performance might degrade as the number of tasks increase. So, the semantics of the task dependence graph (i.e., all tasks and dependences between tasks) should be maintained without building the graph in memory. In a distributed cluster of nodes, maintaining a consistent semantic view of the task dependence graph across nodes might add significant runtime overhead, thereby degrading performance as the number of tasks increase. To reduce this overhead, each node can maintain its own semantic view of the task dependence graph, and the required communication between nodes can help them to cooperatively maintain their semantics without any coordination.

4.1.4 Objectives

Our key objectives are:

1. extraction of coarse-grained dataflow parallelism,
2. allowing load-balanced execution on shared and distributed-memory parallel architectures,
3. overlap of computation and communication, and
4. exposing sufficient functionality that allows the compiler to exploit all of these features automatically including generation of communication code.

We leverage recent work [13] along with our techniques described in Chapter 3 for the application of loop transformations and parallelism detection, and subsequent generation of communication sets.

4.2 Compiler-assisted dataflow runtime

In this section, we first present an overview of the design of our compiler-assisted dataflow runtime. We then present the detailed design of our compiler-runtime interaction, followed by the detailed design of our dataflow runtime.

4.2.1 Overview

A task is a portion of computation that operates on a smaller portion of data than the entire iteration space. Tasks exhibit better data locality, and those that do not depend on one another can be executed in parallel. With compiler assistance, tasks can be automatically extracted from affine loop nests with precise dependence information. Given a distributed-memory cluster of multicores, a task is executed atomically by a thread on a core of a node. A single task's execution itself is sequential with synchronization or communication performed only before and after its execution but not during it. Our aim is to design a distributed decentralized dataflow runtime that dynamically schedules tasks on each node effectively.

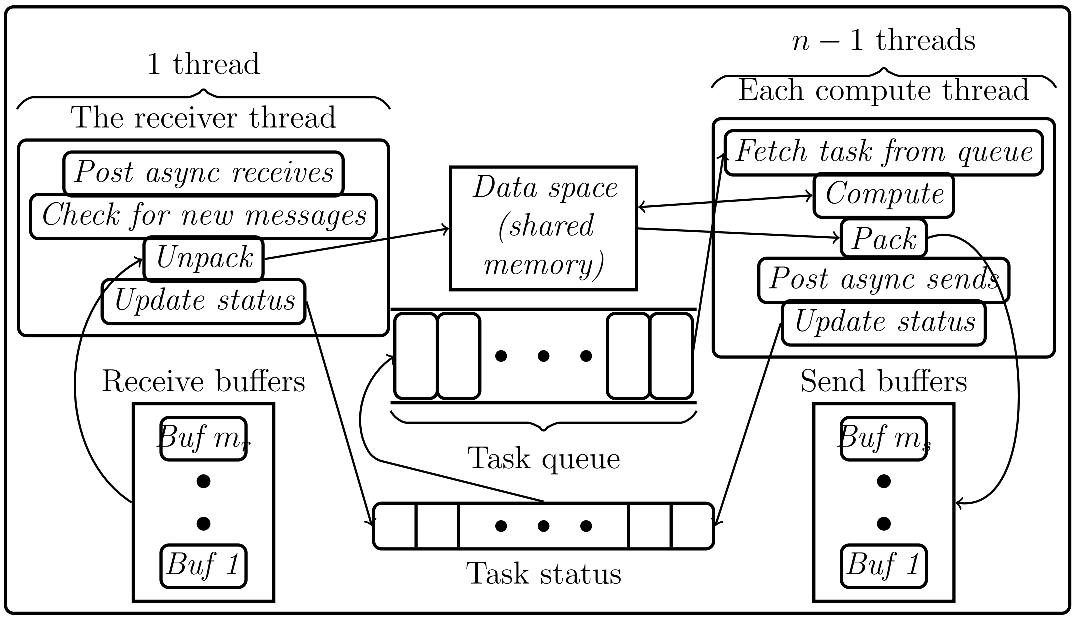


Figure 4.2: Overview of the scheduler on each node

Each node runs its own scheduler without centralized coordination. Figure 4.2 depicts the scheduler on each node. Each node maintains a status for each task, and a queue for the tasks which are ready to be scheduled for execution. There are multiple threads on each node, all of which can access and update these data structures. Each thread maintains its own pool of buffers that are reused for communication. It adds more buffers to this pool if all the buffers are busy in communication.

A single dedicated thread on each node receives data from other nodes. The rest of the threads on each node compute tasks that are ready to be scheduled for execution. The computation can update data variables in the local shared memory. After computing a task, for each node that requires some data produced by this task, the thread packs the data from the local shared memory to a buffer from its pool that is not being used, and asynchronously sends this buffer to the node that requires it. After packing the data, it updates the status of the tasks which are dependent on the task that completed execution. The receiver thread preemptively posts anonymous asynchronous receives using all the buffers in its pool, and continuously checks for new completion messages. Once it receives the data from another node, it unpacks the data from the buffer to the local shared memory. After unpacking the data, it preemptively posts another anonymous asynchronous receive using the same buffer, and updates the status of the tasks which are dependent on the task that sent the data. When the status of a task is updated, it is added to the queue if it is ready to be scheduled for execution.

Each compute thread fetches a task from the task queue and executes it. While updating the status of tasks, each thread could add a task to the task queue. A concurrent task queue is used so that the threads do not wait for each other (lock-free). Such dynamic scheduling of tasks by each compute thread on a node balances the load shared by the threads better than a static schedule, and improves resource utilization. In addition, each compute thread uses asynchronous point-to-point communication and does not wait for its completion. After posting the non-blocking send communication messages, the thread progresses to execute another task from the task queue (if it is available) while some communication may still be in progress. In this way, the communication is automatically overlapped with computation, thereby reducing the overall communication cost.

Each node asynchronously sends data without waiting for confirmation from the receiver. Each node receives data without prior coordination with the sender. There is no coordination between the nodes for sending or receiving data. The only messages between the nodes is that of the data which is required to be communicated to preserve program semantics. These communication messages are embedded with meta-data about the task sending the data. The meta-data is used to update the status of dependent tasks, and schedule them for execution. The schedulers on different nodes use the meta-data to cooperate with each other. In this way, the runtime is designed for cooperation without coordination.

4.2.2 Synthesized Runtime Interface (SRI)

The status of tasks are updated based on dependences between them. A task can be scheduled for execution only if all its dependent tasks have finished execution. Since building and maintaining the task dependence graph in memory could have excessive runtime overhead, our aim is to encapsulate the semantics of the task dependence graph to yield minimal runtime overhead. To achieve this, we rely on the observation that, for affine loop nests, the incoming or outgoing edges of a task in a task dependence graph can be captured as a function (code) of that task using dependence analysis. In other words, the semantics of the task dependence graph can be encapsulated at compile time in functions parametric on a task. These functions are called at runtime to dynamically schedule the tasks. The set of parameterized task functions (PTFs) generated for a program form the Synthesized Runtime Interface (SRI) for that program. We now define the SRI that is required, and show that it can be generated using static analysis techniques.

A task is an iteration of the *innermost parallelized loop* that should be executed atomically. The *innermost parallelized loop* is the innermost among loops that have been identified for parallelization, and we will use this term in the rest of this section. A task is uniquely identified using the iteration vector of the innermost parallelized loop, i.e., the tuple task_id of integer iterator values ordered from the outermost iterator to the innermost iterator. In addition to task_id, some of the PTFs are parameterized on a data variable and a node. A data variable is uniquely identified by an integer data_id, which is its index position in the symbol table. A node is uniquely identifiable by an integer node_id, which is typically the rank of the node in the global communicator.

The PTFs can access and update data structures which are local to the node, and are shared by the threads within the node. The PTFs we define can access and update these locally shared data structures:

1. **readyQueue** (task queue): a priority queue containing task_id of tasks which are ready to be scheduled for execution.

2. **numTasksToWait** (task status): a hash map from task_id of a task to a state or counter, indicating the number of tasks that the task has to wait before it is ready to be scheduled for execution.

The PTFs do not coordinate with other nodes to maintain these data structures, since maintaining a consistent view of data structures across nodes might add significant run-time overhead. So, all operations within a PTF are local and non-blocking.

Function call	**Category**	**Operation**
incrementForLocalDependent(task_id)	Scheduling	Increment numTasksToWait of the task task_id for each local task that it is dependent on
incrementForRemoteDependent(task_id)	Scheduling	Increment numTasksToWait of the task task_id for each remote task that it is dependent on
decrementDependentOfLocal(task_id)	Scheduling	Decrement numTasksToWait of the tasks that are dependent on the local task task_id
decrementDependentOfRemote(task_id)	Scheduling	Decrement numTasksToWait of the local tasks that are dependent on the remote task task_id
countLocalDependent(task_id)	Scheduling	Returns the number of local tasks that are dependent on the task task_id
countRemoteDependent(task_id)	Scheduling	Returns the number of remote tasks that are dependent on the task task_id
isReceiver(node_id,data_id,task_id)	Communication	Returns true if the node node_id is a receiver of elements of data variable data_id from the task task_id
pack(data_id,task_id, node_id, buffer)	Communication	Packs elements of data variable data_id from local shared-memory into the buffer, that should be communicated from the task task_id to the node node_id
unpack(data_id,task_id, node_id, buffer)	Communication	Unpacks elements of data variable data_id to local shared-memory from the buffer, that has been communicated from the task task_id to the node node_id
pi(task_id)	Placement	Returns the node node_id on which the task task_id will be executed
compute(task_id)	Computation	Executes the computation of the task task_id

Table 4.1: Synthesized Runtime Interface (SRI)

The name, arguments, and operation of the PTFs in the SRI are listed in Table 4.1. The PTFs are categorized into those that assist scheduling, communication, placement, and computation.

Inter-task dependences

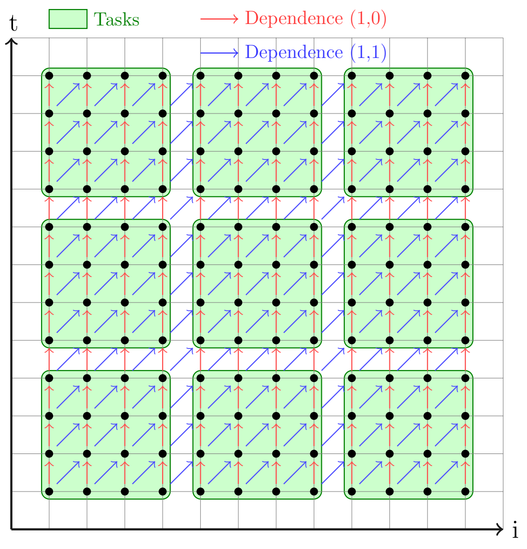
Baskaran et al. [8] describe a way to extract inter-tile dependences from data dependences between statements in the transformed iteration space. Inter-task dependences can be extracted in a similar way. Figure 4.3 illustrates the inter-task dependences for an example. Recall that a task is an iteration of the innermost parallelized loop. For each data dependence polyhedron in the transformed iteration space, all dimensions inner to the innermost parallelized loop in the source domain and the target domain are projected out to yield an inter-task dependence polyhedron corresponding to that data dependence. As noted in Section 4.1.2, it is sufficient to have only one inter-task dependence between two tasks for all data dependences between them. Therefore, a union of all inter-task dependence polyhedra corresponding to data dependences is taken to yield the inter-task dependence polyhedron.

Note that a single task can be associated with multiple statements in the polyhedral representation. In particular, all statements inside the innermost parallelized loop characterizing the task are the ones associated with the task. A task can also be created for a statement with no surrounding parallel loops, but is part of a sequence of loop nests with parallel loops elsewhere.

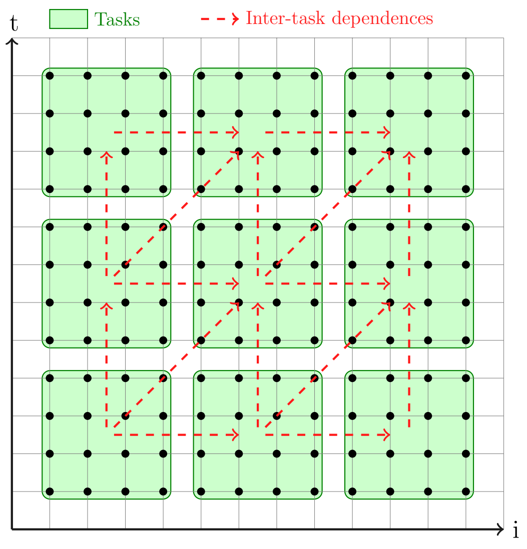
We now introduce notation corresponding to background presented on the polyhedral framework in Chapter 2. Let S_1, S_2, ..., S_m be the statements in the polyhedral representation of the program, m_S be the dimensionality of statement S, d_i and d_j be the depths of the innermost parallelized loops corresponding to tasks T_i and T_j respectively, $s(T)$ be the set of polyhedral statements in task T, and D_e be the dependence polyhedron for a dependence between S_p and S_q. Let $\mathsf{project\_out}(\mathsf{P}, \mathsf{i}, \mathsf{n})$ be the polyhedral library routine that projects out n dimensions from polyhedron P starting from dimension number i (0-indexed). Then, the inter-task dependence polyhedron for tasks T_i and T_j is computed as follows:

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

(a) Original code



(b) Dependences between iterations



(c) Dependences between tasks

Figure 4.3: Illustration of inter-task dependences for an example

$$
\begin{aligned}
D_e^{'} &= \text{project\_out}\left(D_e, m_{S_p}+d_j, m_{S_q}-d_j\right)\\
D_e^{T} &= \text{project\_out}\left(D_e^{'}, d_i, m_{S_p}-d_i\right)\\
D^T(T_i \rightarrow T_j) &= \bigcup_e \Big(\langle\vec{s},\vec{t}\rangle \in D_e^T,\ \forall e \in E, e=(S_p,S_q),\\
&\quad \forall S_p \in T_i, S_q \in T_j\Big)\,. \qquad (4.1)
\end{aligned}
$$

The inter-task dependence polyhedron is a key compile-time structure. All PTFs that assist in scheduling rely on it. A code generator such as Cloog [12] is used to generate code iterating over certain dimensions of $D^T(T_i \rightarrow T_j)$ while treating a certain number of outer ones as parameters. For example, if the target tasks need to be iterated over for a given source task, we treat the outer d_i dimensions in D^T as parameters and generate code scanning the next d_j dimensions. If the source tasks are to be iterated over given a target task, the dimensions are permuted before a similar step is performed.

Type of dependence	**Parame-terized on**	**Iterates over**	**Condition on enumerated task**	**Conditional action**
RAW, WAR or WAW	target task	source tasks	local task	numTasksToWait[target_task_id]++

(a) incrementForLocalDependent

Type of dependence	**Parame-terized on**	**Iterates over**	**Condition on enumerated task**	**Conditional action**
RAW	target task	source tasks	remote task	numTasksToWait[target_task_id]++

(b) incrementForRemoteDependent

Type of dependence	**Parame-terized on**	**Iterates over**	**Condition on enumerated task**	**Conditional action**
RAW, WAR or WAW	source task	target tasks	none	numTasksToWait[target_task_id]−− If target task is local AND numTasksToWait[target_task_id] == 0: readyQueue.push(target_task_id)

(c) decrementDependentOfLocal

Type of dependence	**Parame-terized on**	**Iterates over**	**Condition on enumerated task**	**Conditional action**
RAW	source task	target tasks	local task	numTasksToWait[target_task_id]−− If numTasksToWait[target_task_id] == 0: readyQueue.push(target_task_id)

(d) decrementDependentOfRemote

Type of dependence	**Parame-terized on**	**Iterates over**	**Condition on enumerated task**	**Conditional action**
RAW, WAR or WAW	source task	target tasks	local task	return_count++

(e) countLocalDependent

Type of dependence	**Parame-terized on**	**Iterates over**	**Condition on enumerated task**	**Conditional action**
RAW	source task	target tasks	remote task	return_count++

(f) countRemoteDependent

Table 4.2: Synthesized Runtime Interface (SRI) that assists dynamic scheduling: generated by analyzing inter-task dependences (decrementDependentOfRemote() should be called for remote tasks while the rest should be called for local tasks)

Constraints on scheduling

As illustrated in the example in Section 4.1.2, memory-based data dependences, i.e., WAR and WAW dependences, do not enforce a constraint on the order of execution of tasks on different nodes since those tasks will not share an address space at execution time. So, the inter-task dependence polyhedron between tasks placed on different nodes is extracted using RAW dependence polyhedra alone. On the other hand, memory-based data dependences do enforce a constraint on the order of execution of tasks on the same node. So, the inter-task dependence polyhedron between tasks on the same node is extracted using RAW, WAR and WAW dependence polyhedra. For a PTF that traverses the incoming edges of a task, the target task in the inter-task dependence polyhedron is treated as a parameter, and code is generated to enumerate the source tasks. For a PTF that traverses the outgoing edges of a task, the source task in the inter-task dependence polyhedron is treated as a parameter, and code is generated to enumerate the target tasks. Each PTF can check if the enumerated task is local or remote (using the placement PTF), and then perform an action dependent on that. Table 4.2 summarizes this for each PTF that assists scheduling.

Communication and placement

In Chapter 3, we generated efficient data movement code for distributed-memory architectures by parameterizing communication on an iteration of the innermost parallelized loop. Since the data to be communicated could be discontiguous in memory, the sender packs it into a buffer before sending it, and the receiver unpacks it from the buffer after receiving it. We adapt the same techniques to parameterize communication on a task.

A PTF is generated to pack elements of a data variable written in a task from local shared-memory into a buffer that should be communicated to a node. Similarly, a PTF is generated to unpack elements of a data variable written in a task to local shared-memory from a buffer that has been communicated to a node. Any of the communication schemes described in Chapter 3 can be used - Flow-out (FO) scheme, Flow-out intersection Flow-in (FOIFI) scheme or Flow-out partitioning (FOP) scheme. We use FOP scheme since it communicates the minimum volume of data for a vast majority of cases, and outperforms the other schemes, as demonstrated in Section 3.6. Therefore, the code generated for the pack or unpack PTF iterates over each communication partition of the given task, and packs or unpacks it only if the given node is a receiver of that communication partition.

A PTF is generated to determine if a node is a receiver of the elements of a data variable written in a task. This corresponds to the $receivers_x$ function described in Section 3.2.1. Since we are using FOP scheme, a PTF is generated for each communication partition to determine if a node is a receiver of that communication partition of a task. These PTFs are not included in Table 4.1 for the sake of brevity. The pi function (Table 4.1) provides the placement of tasks. It is the same function used in Chapter 3. Information on when the placement is determined and specified will be discussed in Section 4.2.3.

Computation

We enumerate all tasks and extract computation for a parameterized task using techniques described by Baskaran et al. [8]. For each innermost parallelized loop in the transformed iteration space, from the iteration domain of a statement within the loop, all dimensions inner to the innermost parallelized loop are projected out. The code generated to traverse this domain will enumerate all tasks in that parallelized loop nest at runtime. To extract the computation PTF, the iteration domain of all statements within the innermost parallelized loop is considered. All outer dimensions up to and including the innermost parallelized loop are treated as parameters, and code is generated to traverse dimensions inner to the innermost parallelized loop.

Thread-safety

A concurrent priority queue is used as the readyQueue. Atomic increments and decrements are used on the elements of numTasksToWait. *unpack* is the only PTF that modifies original data variables in local shared-memory. So, the runtime has to ensure that the function is called according to the inter-task dependence constraints of the program. As

Algorithm 2: Distributed Function-based Dynamic Scheduling (DFDS)

```
1 ⟨numTasksToCompute, numTasksToReceive⟩ ← initTasks()
2 begin parallel region
3     if $thread\_id == 0$ then
          // single dedicated receiver thread
4
5         receiveDataFromTasks(numTasksToReceive)
      // compute threads
6     computeTasks(numTasksToCompute)
```

long as the unpack PTF respects the inter-task dependence constraints, all PTFs can be simultaneously called with different parameters by different threads in a node without affecting program semantics.

4.2.3 Distributed Function-based Dynamic Scheduling (DFDS)

Compiler assistance or hints can make a runtime more efficient by reducing runtime overhead. A runtime, that a compiler can automatically generate code for, is even more useful since efficient parallel code is directly obtained from sequential code, thereby eliminating programmer burden in parallelization. As mentioned earlier, our goal is to build a runtime that is designed to be targeted by a compiler. In particular, we design a distributed decentralized runtime that uses the SRI generated by a compiler to dynamically schedule tasks on each node. Hence, we call this runtime Distributed Function-based Dynamic Scheduling (DFDS). Algorithm 2 shows the high-level code generated for DFDS that is executed by each node. Initially, each node initializes the status of all tasks. It also determines the number of tasks it has to compute and the number of tasks it has to receive from. After initialization, a single dedicated thread receives data from tasks executed on other nodes, while the rest of the threads compute tasks that are assigned to this node and these could send data to other nodes.

Algorithm 3 shows the code generated to initialize the status of tasks. For each local task, its numTasksToWait is initialized to the sum of the number of local and remote tasks that it is dependent on. If a local task has no tasks that it is dependent on, then

Algorithm 3: initTasks()

```
1  my_node_id ← node_id of this node
2  numTasksToCompute ← 0
3  numTasksToReceive ← 0
4  for each task_id do
5      if pi(task_id) == my_node_id then // local task
6          numTasksToCompute++
7          incrementForLocalDependent(task_id)
8          incrementForRemoteDependent(task_id)
9          if numTasksToWait[task_id] == 0 then
10             readyQueue.push(task_id)
11     else // remote task
12         numReceivesToWait[task_id] ← 0
13         for each data_id do
14             if isReceiver(my_node_id,data_id,task_id) then
15                 numReceivesToWait[task_id]++
16         if numReceivesToWait[task_id] > 0 then
17             numTasksToReceive++
18             incrementForLocalDependent(task_id)
   Output: ⟨numTasksToCompute, numTasksToReceive⟩
```

it is added to the readyQueue. For each remote task, a counter numReceivesToWait is determined, which indicates the number of data variables that this node should receive from that remote task. If any data is going to be received from a remote task, then its numTasksToWait is initialized to the number of local tasks that it is dependent on. This is required since the unpack PTF cannot be called on a remote task until all the local tasks it depends on have completed. Note that the *for-each* task loop can be parallelized with numTasksToCompute and numTasksToReceive as reduction variables, and atomic increments to elements of numReceivesToWait.

Algorithm 4 and Algorithm 5 show the generated code that is executed by a compute thread. A task is fetched from the readyQueue and its computation is executed. Then, for each data variable and receiver, the data that has to be communicated to that receiver is packed from local shared-memory into a buffer which is not in use. If all the buffers in the pool are being used, then a new buffer is created and added to the pool. The task_id

Algorithm 4: computeTasks()

```
Input: numTasksToCompute
1 while $numTasksToCompute > 0$ do
2     $(pop\_succeeded, task\_id) \leftarrow$ readyQueue.try_pop()
3     if $pop\_succeeded$ then
4         compute($task\_id$)
5         sendDataOfTask($task\_id$)
6         decrementDependentOfLocal($task\_id$)
7         atomic numTasksToCompute−−
```

Algorithm 5: sendDataOfTask()

```
Input: $task\_id$
1 $my\_node\_id \leftarrow node\_id$ of this node
2 for each $data\_id$ do
3     for each $node\_id \neq my\_node\_id$ do
4         if $isReceiver(node\_id,data\_id,task\_id)$ then
5             Let i be the index of a send_buffer that is not in use
6             Put $task\_id$ to send_buffer[i]
7             pack($data\_id$, $task\_id$, $node\_id$, send_buffer[i])
8             Post asynchronous send from send_buffer[i] to $node\_id$
```

of this task is added as meta-data to the buffer. The buffer is then sent asynchronously to the receiver, without waiting for confirmation from the receiver. Note that the pack PTF and the asynchronous send will not be called if all the tasks dependent on this task due to RAW dependences will be executed on the same node. A local task is considered to be complete from this node's point-of-view only after the data it has to communicate is copied into a separate buffer. Once a local task has completed, numTasksToWait of its dependent tasks is decremented. This is repeated until there are no more tasks to compute.

Algorithm 6 shows the generated code that is executed by the receiver thread. Initially, for each data variable, an asynchronous receive from any node (anonymous) is preemptively posted to each buffer for the maximum number of elements that can be received from any task. Reasonably tight upper bounds on the required size of buffers are determined from the communication set constraints that are all affine. This is used

Algorithm 6: receiveDataFromTasks()

```
Input: numTasksToReceive
1  my_node_id ← node_id of this node
2  for each data_id and index i of receive_buffer do
3      Post asynchronous receive to receive_buffer[i] with any node_id as source
4  while numTasksToReceive > 0 do
5      for each data_id and index i of receive_buffer do
6          if asynchronous receive to receive_buffer[i] has completed then
7              Extract task_id from receive_buffer[i]
8              if numTasksToWait[task_id] == 0 then
9                  unpack(data_id,task_id, my_node_id, receive_buffer[i])
10                 numReceivesToWait[task_id]−−
11                 if numReceivesToWait[task_id] == 0 then
12                     decrementDependentOfRemote(task_id)
13                     numTasksToReceive−−
14                 Post asynchronous receive to receive_buffer[i] with any node_id as
                   source
```

to determine the maximum number of elements that can be received from any task.

Each receive is checked for completion. If the receive has completed, then the metadata task_id is fetched from the buffer. If all the local tasks that task_id depends on have completed, then the data that has been received from the task task_id is unpacked from the buffer into local shared-memory, and numReceivesToWait of task_id is decremented. A data variable from a task is considered to be received only if the data has been updated in local shared-memory, i.e., only if the data has been unpacked. Once the data has been unpacked from a buffer, an asynchronous receive from any node (anonymous) is preemptively posted to the same buffer. A remote task is considered to be complete from this node's point-of-view only if it has received all the data variables it needs from that task. Once a remote task has completed, numTasksToWait of its dependent tasks is decremented. If all the receive buffers have received data, but have not yet been unpacked, then more buffers are created and an asynchronous receive from any node (anonymous) is preemptively posted to each new buffer. This is repeated until there are no more tasks to receive from.

While evaluating our runtime, we observed that a dedicated receiver thread is under-utilized since almost all its time is spent in busy-waiting for one of the non-blocking receives to complete. Hence, we believe that a single receiver thread is sufficient to manage any amount of communication. To avoid under-utilization, the generated code was modified such that the receiver thread also executed computation (and its associated functions) instead of busy-waiting. We observed that there was almost no difference in performance between a dedicated receiver thread and a receiver thread that also computed. There is a trade-off: although a dedicated receiver thread is under-utilized, it is more responsive since it can unpack data (and enable other tasks) soon after a receive. The choice might depend on the application. Our tool can generate code for both such that it can be chosen at compile-time. The algorithms are presented as is for clarity of exposition.

Priority

Priority on tasks can improve performance by enabling the priority queue to choose between many ready tasks more efficiently. There are plenty of heuristics to decide the priority of tasks to be executed. Though this is not the focus of our work, we use PTFs to assist in deciding the priority. A task with more remote tasks dependent on it (countRemoteDependent()) has higher priority since data written in it is required to be communicated to more remote tasks. This helps initiate communication as early as possible, increasing its overlap with computation. For tasks with the same number of remote tasks dependent on it, the task with more local tasks dependent on it (countLocalDependent()) has higher priority since it could enable more tasks to be ready for execution. We also assign thread affinity hints by using a block distribution of local tasks onto the threads. When tasks cannot be differentiated on remote or local tasks dependent on it, a task that has affinity to this thread has higher priority over one that does not have affinity to this thread. This can help improve spatial locality because consecutive iterations (in source code) could be accessing spatially-near data. When none of these can differentiate between tasks, a task whose task_id is the lexicographically least is chosen. We use this priority scheme in our evaluation (Section 4.4). The priority scheme in our design is a pluggable component, and we plan to explore more sophisticated priority schemes (or scheduling policies in general) in the future.

Dynamic a priori placement

When the status of the tasks are being initialized at runtime, DFDS expects the placement of all tasks to be known, since its behavior depends on whether a task is local or remote. The placement of all tasks can be decided at runtime before initializing the status of tasks. In such a case, a hash map from a task to the node which will execute that task should be set consistently across all nodes before the call to initTasks() in line 1 of Algorithm 2. The placement PTF would then read the hash map. DFDS is thus designed to support dynamic a priori placement.

To find the optimal placement automatically is not the focus of this work. In our evaluation, we use a block placement function except in cases where non-rectangular iteration spaces are involved – in such cases, we use a block-cyclic placement. This placement strategy yields good strong scaling on distributed-memory for the benchmarks we have evaluated, as we will see in Section 4.4.2. Determining more sophisticated placements including dynamic a priori placements is orthogonal to our work. Recent work by Reddy et al. [45] explores this independent problem.

4.3 Implementation

We implement our compiler-assisted runtime as part of a publicly available source-to-source polyhedral tool chain. Clan [11], ISL [50], Pluto [14], and Cloog-isl [10] are used to perform polyhedral extraction, dependence testing, automatic transformation, and code generation, respectively. Polylib [42] is used to implement the polyhedral operations in Section 4.2.2.

Figure 4.4 shows the overview of our tool. The input to our compiler-assisted runtime

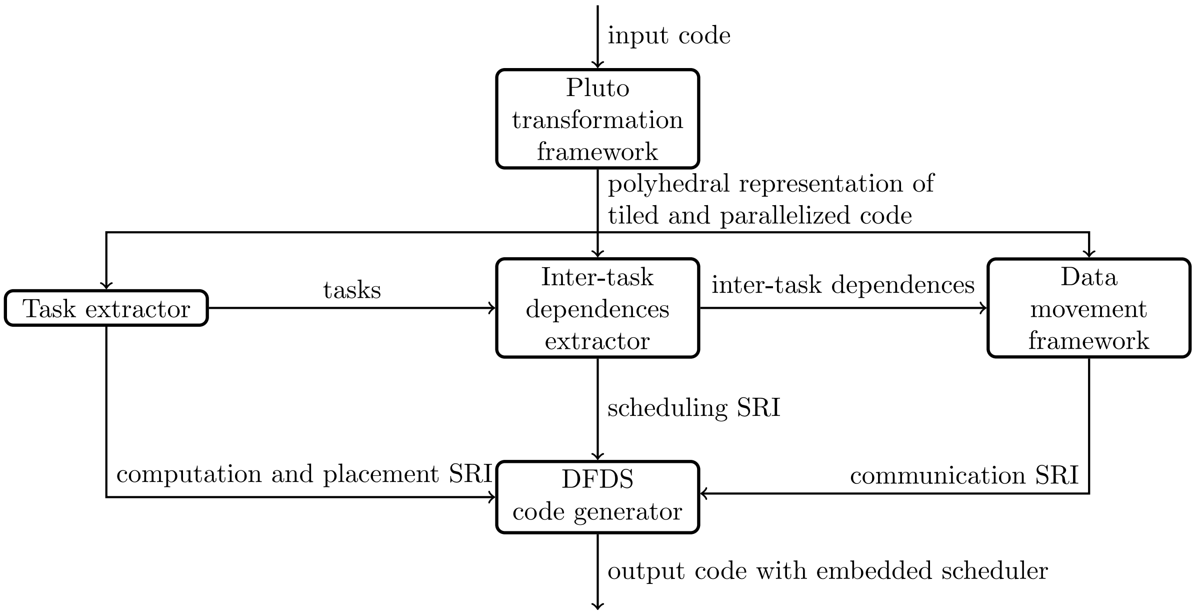

input code
Pluto transformation framework
polyhedral representation of tiled and parallelized code
Task extractor
tasks
Inter-task dependences extractor
inter-task dependences
Data movement framework
scheduling SRI
computation and placement SRI
DFDS code generator
communication SRI
output code with embedded scheduler

Figure 4.4: Overview of our tool

is sequential code containing arbitrarily nested affine loop nests. The sequential code is tiled and parallelized using the Pluto algorithm [5, 15]. Loop tiling helps reduce the runtime overhead and improve data locality by increasing the granularity of tasks. The SRI is automatically generated using the parallelized code as input; the communication code is automatically generated using our own FOP scheme described in Section 3.4. The DFDS code for either shared-memory or distributed-memory systems is then automatically generated. The code generated can be executed either on a shared-memory multicore or on a distributed-memory cluster of multicores. Thus, ours is a fully automatic source-transformer of sequential code that targets a compiler-assisted dataflow runtime.

The concurrent priority queue in Intel Thread Building Blocks (TBB) [29] is used to maintain the tasks which are ready to execute. Parametric bounds of each dimension in the `task_id` tuple are determined, and these, at runtime, yield bounds for each of the outer dimensions that were treated as parameters. A multi-dimensional array of dimension equal to the length of the `task_id` tuple is allocated at runtime. The size of each dimension in this array corresponds to the difference in the bounds of the corresponding dimension in the task_id tuple. This array is used to maintain the task statuses numTasksToWait and numReceivesToWait, instead of a hash map. The status of a task_id can then be accessed by offsetting each dimension in the array by the lower bound of the corresponding dimension in the task_id tuple. Thus, the memory required to store the task status is minimized, while its access is efficient. Asynchronous non-blocking MPI primitives are used to communicate between nodes in the distributed-memory system.

4.4 Experimental evaluation

In this section, we evaluate our compiler-assisted runtime on a shared-memory multicore, and on a distributed cluster of multicores.

Benchmarks

We present results for Floyd-Warshall (`floyd`), LU Decomposition (`lu`), Cholesky Factorization (`cholesky`), Alternating Direction Implicit solver (`adi`), 2d Finite Different Time Domain Kernel (`fdtd-2d`), Heat 2d equation (`heat-2d`) and Heat 3d equation (`heat-3d`) benchmarks. The first five are from the publicly available Polybench/C 3.2 suite [41]; `heat-2d` and `heat-3d` are widely used stencil computations [47]. All benchmarks use double-precision floating-point operations. The compiler used for all experiments is ICC 13.0.1 with options '-O3 -ansi-alias -ipo -fp-model precise'. These benchmarks were selected from a larger set since (a) their parallelization involves communication and synchronization that cannot be avoided, and (b) they capture different kinds of communication patterns that result from uniform and non-uniform dependences including near-neighbor, multicast, and broadcast style communication. Table 4.3 and 4.4 list the problem and tile sizes used. The results are presented for a single execution of a benchmark - the best performing one among at least 3 runs; the variation wherever it existed was negligible.

Benchmark	**Problem sizes**	**Tile sizes**	
		auto	**manual-CnC**
heat-2d	N=8192, T=1024	64 (3d)	256 (2d)
heat-3d	N=512, T=256	16 (4d)	64 (3d)
fdtd-2d	N=4096, T=1024	16 (3d)	256 (2d)
floyd	N=4096	256 (2d)	256 (2d)
cholesky	N=8192	8 (3d)	128 (3d)
lu	N=8192	64 (3d)	128 (3d)

Table 4.3: Problem and tile sizes - shared-memory multicore (Note that computation tiling transformations for auto and manual-CnC may differ; data is tiled in manual-CnC but not in auto)

Benchmark	**Problem sizes**	**Tile sizes**	
		auto	**manual-CnC**
heat-2d	N=8192, T=1024	16 (3d)	256 (2d)
adi	N=8192, T=128	256 (2d)	-
fdtd-2d	N=8192, T=1024	32 (3d)	256 (2d)
floyd	N=8192	512 (2d)	512 (2d)
cholesky	N=16384	128 (3d)	128 (3d)
lu	N=16384	256 (3d)	128 (3d)

Table 4.4: Problem and tile sizes - cluster of multicores (Note that computation tiling transformations for auto and manual-CnC may differ; data is tiled in manual-CnC but not in auto)

Intel Concurrent Collections (CnC) implementations

To compare our automatically generated codes against a manually optimized implementation, we implemented `heat-2d`, `heat-3d`, `fdtd-2d` and `lu` using Intel Concurrent Collections (CnC) [28]. We include `floyd` and `cholesky` from the Intel CnC samples; the `cholesky` benchmark we compare against does not use MKL routines. The Intel CnC version used is 0.9.001 for shared-memory experiments, and 0.8.0 for distributed-memory experiments.

The `cholesky` implementation is detailed in a previous performance evaluation of Intel CnC [22]. Our Intel CnC implementations use computation and data tiling for coarsening task granularity and improving locality. Tables 4.3 and 4.4 show the tile sizes chosen for each benchmark. In case of distributed-memory, we also ensure that communication is precise, i.e., we only communicate that which is necessary to preserve program semantics. Tiling and precise data communication constitute most of the programming effort. Additionally, we specify the nodes which consume the data produced in a task (the consumed_on() tuner) that helps the runtime to push the data to be communicated to the nodes that require it. We observe that the push model performs much better than the default pull model (pulling data when it is required) in our context. We also provide the exact number of uses for each data buffer so that the CnC runtime can efficiently garbage collect it and reduce memory footprint. For each benchmark, we assign higher priority to tasks that communicate to other nodes. Thus, our Intel CnC implementations have been tuned with considerable effort to extract high performance. Note that all these components which are tedious and error prone to write manually are automatically generated by our framework.

4.4.1 Shared-memory architectures

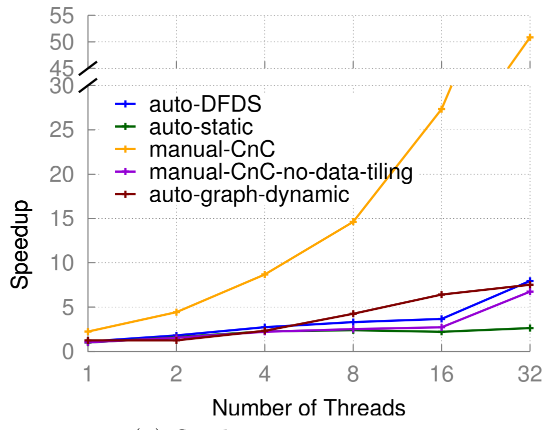
Setup

The experiments were run on a four socket machine of AMD Opteron 6136 2.4 GHz, 128 KB L1, 512 KB L2, and 6 MB L3 cache. The memory architecture is NUMA, and we use numactl to bind threads and pages suitably for all our experiments.

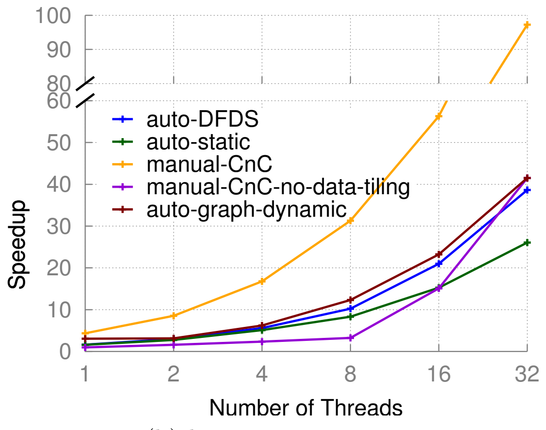
Evaluation

We compare the performance of our automatic approach (auto-DFDS) with:

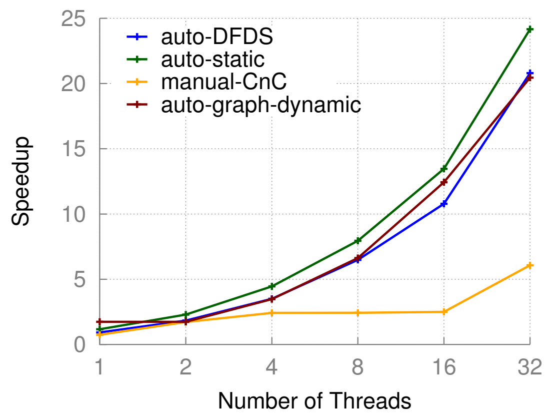
- hand-optimized Intel CnC codes (manual-CnC),
- state-of-the-art automatic dynamic scheduling approach [8] that constructs the entire task dependence graph in memory (auto-graph-dynamic), and
- state-of-the-art automatic static scheduling approach [5,15] that uses *bulk-synchronization* (auto-static).



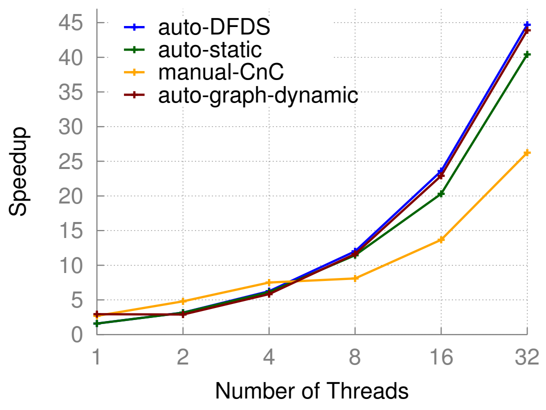
(a) floyd – seq time is 231s



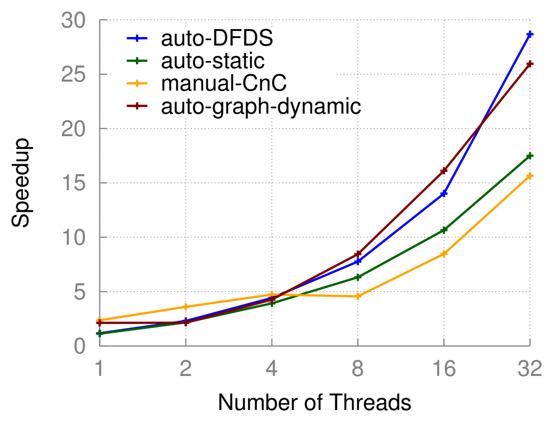
(b) lu – seq time is 796s



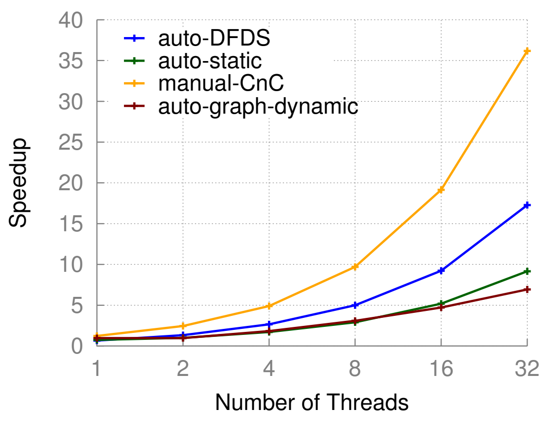
(c) fdtd-2d – seq time is 222s



(d) heat-2d – seq time is 875s



(e) heat-3d – seq time is 609s



(f) cholesky – seq time is 640s

Figure 4.5: Speedup of auto-DFDS, auto-static, and manual-CnC over seq on a shared-memory multicore (Note that performance of auto-DFDS and auto-static on a single thread is different from that of seq due to automatic transformations)

For auto-graph-dynamic, the graph is constructed using Intel Thread Building Blocks (TBB) [29] Flow Graph (TBB is a popular work stealing based library for task parallelism). All the automatic schemes use the same polyhedral compiler transformations (and the same tile sizes). The performance difference in the automatic schemes thus directly relates to the efficiency in their scheduling mechanism.

Analysis

Figure 4.5 shows the scaling of all approaches relative to the sequential version (seq) which is the input to our compiler. auto-DFDS scales well with an increase in the number of threads, and yields a geometric mean speedup of 23.5× on 32 threads over the sequential version. The runtime overhead of auto-DFDS (to create and manage tasks) on 32 threads is less than 1% of the overall execution time for all benchmarks, except `cholesky` for which it is less than 3%.

Benchmarks	**Static**	**DFDS**
heat-2d	23.74	0.65
heat-3d	46.66	0.09
fdtd-2d	22.28	1.14
lu	37.48	1.36
cholesky	57.34	0.39
floyd	102.35	0.08

Table 4.5: Standard-deviation over mean of computation times of all threads in % on 32 threads of a shared-memory multicore: lower value indicates better load balance

auto-DFDS scales better than or comparably to both auto-graph-dynamic and auto-static. For auto-DFDS and auto-static, we measured the computation time of each thread, and calculated the mean and standard deviation of these values. Table 4.5 shows the standard deviation divided by mean, which provides a fair measure of the load balance. auto-DFDS balances load much better than auto-static, thereby decreasing the overall execution time. We also measured the maximum idle time across threads for both auto-DFDS and auto-static, which includes the synchronization time. Figure 4.6 shows that all threads are active for most of the time in auto-DFDS, unlike auto-static.

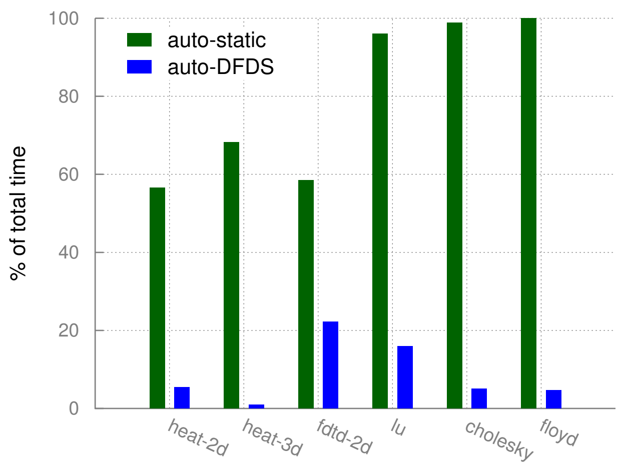


Figure 4.6: Maximum idle time across 32 threads on a shared-memory multicore

Figure 4.7 shows the speedup of auto-DFDS over manual-CnC on both 1 thread and 32 threads. The speedup on 32 threads is as good as or better than that on 1 thread, except for `floyd`. This shows that auto-DFDS scales as well as or better than manual-CnC. In the CnC model, programmers specify tasks along with data they consume and produce. As a result, data is decomposed along with tasks, i.e., data is also tiled. For example, a 2d array when 2d tiled yields a 2d array of pointers to a 2d sub-array (tile) that is contiguous in memory. Such explicit data tiling transformations yield better locality at all levels of memory or cache. Due to this, manual-CnC outperforms auto-DFDS for `floyd`, `lu`, and `cholesky`. manual-CnC also scales better for `floyd` because of privatization of data tiles with increase in the number of threads; privatization allows reuse of data along the outer loop, thereby achieving an effect similar to that of 3d tiling. To evaluate this, we implemented manual-CnC without the data tiling optimizations for both `floyd` and `lu`. Figure 4.5 validates our hypothesis by showing that manual-CnC-no-data-tiling versions perform similar to auto-DFDS, indicating the need for improved compiler transformations for data tiling. For `fdtd-2d`, `heat-2d`, and `heat-3d`, automatic approaches find load-balanced computation tiling transformations [5] that also tile the outer serial loop. These are hard and error prone to implement manually and almost never done in practice. Consequently, manual-CnC codes only tile the parallel loops and not the outer serial

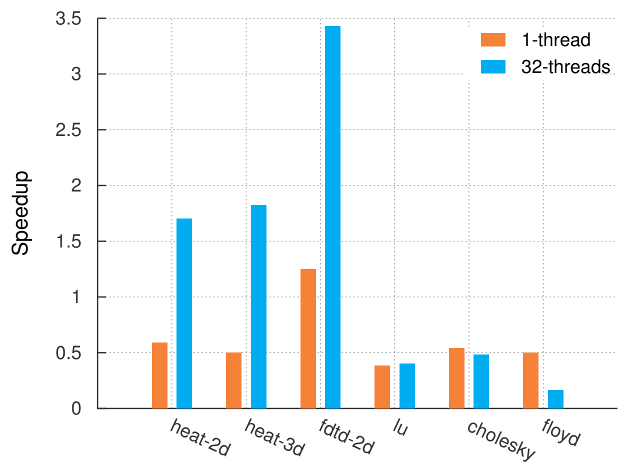


Figure 4.7: Speedup of auto-DFDS over manual-CnC on a shared-memory multicore

loop. In these cases, auto-DFDS significantly outperforms manual-CnC: this highlights the power of automatic task generation frameworks used in conjunction with runtimes. Automatic data tiling transformations [45] can make our approach even more effective, and match the performance of manual implementations like CnC.

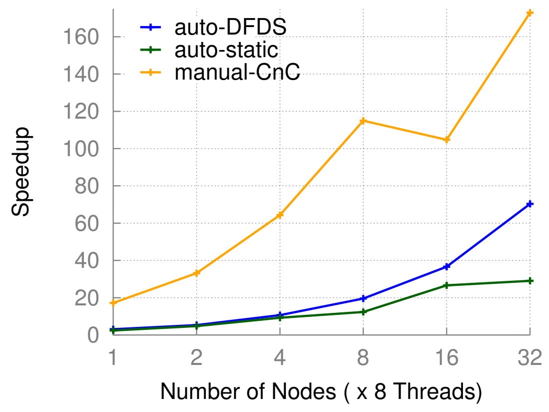
4.4.2 Distributed-memory architectures

Setup

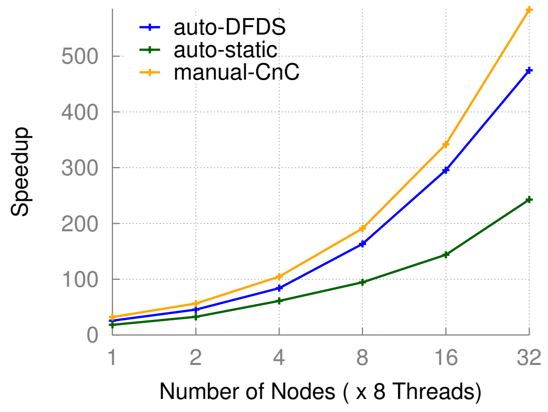
The experiments were run on a 32-node InfiniBand cluster of dual-SMP Xeon servers. Each node on the cluster consists of two quad-core Intel Xeon E5430 2.66 GHz processors with 12 MB L2 cache and 16 GB RAM. The InfiniBand host adapter is a Mellanox MT25204 (InfiniHost III Lx HCA). All nodes run 64-bit Linux kernel version 2.6.18. The cluster uses MVAPICH2-1.8.1 as the MPI implementation. It provides a point-to-point latency of 3.36 μs, unidirectional and bidirectional bandwidths of 1.5 GB/s and 2.56 GB/s respectively. The MPI runtime used for running CnC samples is Intel MPI as opposed to MVAPICH2-1.8.1, as CnC works only with the Intel MPI runtime.

Evaluation

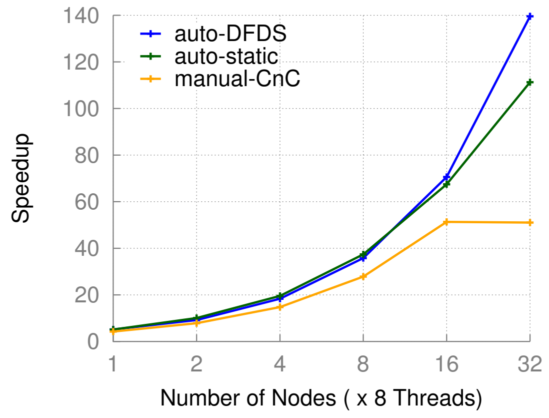
We compare our fully automatic approach (auto-DFDS) with:



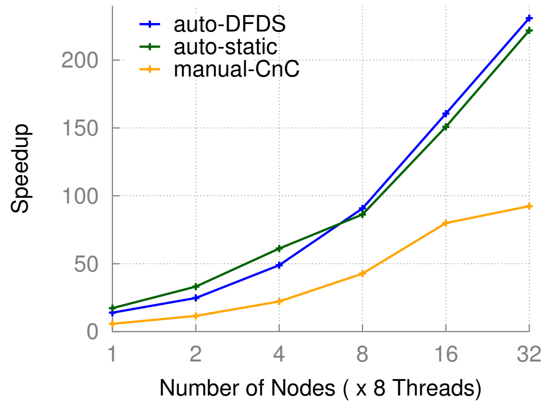
(a) floyd – seq time is 2012s



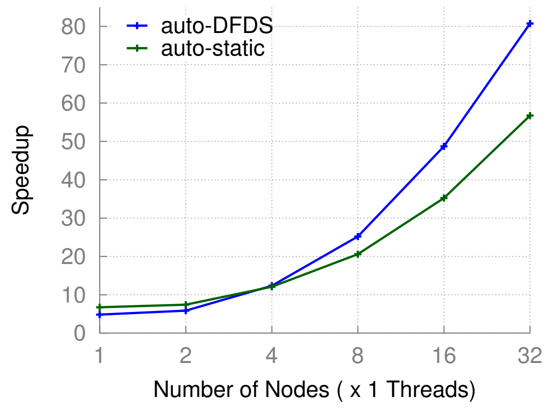
(b) lu – seq time is 5354s



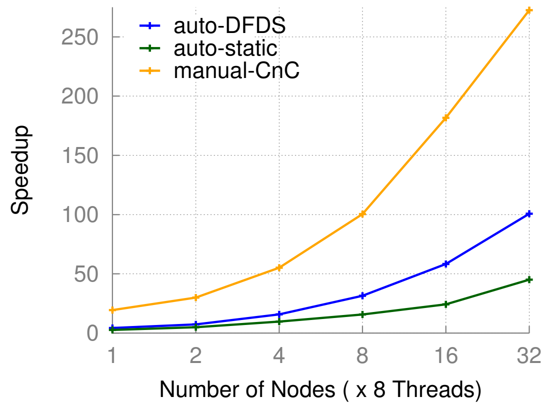
(c) fdtd-2d – seq time is 1432s



(d) heat-2d – seq time is 796s



(e) adi – seq time is 2717s



(f) cholesky – seq time is 2270s

Figure 4.8: Speedup of auto-DFDS, auto-static, and manual-CnC over seq on a cluster of multicores (Note that performance of auto-DFDS and auto-static on a single thread is different from that of seq due to automatic transformations)

- hand optimized Intel CnC codes (manual-CnC), and
- state-of-the-art automatic parallelization approach on distributed-memory [13] that uses *bulk-synchronization*, coupled with our own efficient data movement scheme (auto-static).

Both auto-DFDS and auto-static use the FOP scheme described in Section 3.4. As demonstrated in Section 3.6, FOP significantly improved upon the state-of-the-art automatic approach [13] (FO). Thus, our intention is to evaluate the utility of auto-DFDS on top of our previous state-of-the-art extension (Chapter 3). All the automatic schemes use the same polyhedral compiler transformations (and the same tile sizes). The performance difference in the automatic schemes thus directly relates to the efficiency in their scheduling mechanism.

Analysis

Figure 4.8 shows the scaling of all approaches relative to the sequential version (seq) which is the input to our compiler. auto-DFDS scales well with an increase in the number of nodes, and yields a geometric mean speedup of 143.6× on 32 nodes over the sequential version. The runtime overhead of auto-DFDS (to create and manage tasks) on 32 nodes is less than 1% of the overall execution time for all benchmarks.

Benchmarks	**Static**	**DFDS**
adi	2.58	3.12
heat-2d	3.67	2.22
fdtd-2d	3.52	1.29
lu	67.45	16.13
cholesky	48.96	8.09
floyd	174.78	3.25

Table 4.6: Standard-deviation over mean of computation times of all threads in % on 32 nodes (multicores) of a cluster: lower value indicates better load balance

auto-DFDS yields a geometric mean speedup of 1.6× over auto-static on 32 nodes. For both of them, we measured the computation time of each thread on each node, and calculated the mean and standard deviation of these values. Table 4.6 shows the standard deviation divided by mean, which provides a fair measure of the load balance. auto-DFDS achieves good load balance even though the computation across nodes is statically distributed. auto-DFDS balances load much better than auto-static, thereby decreasing the overall execution time.

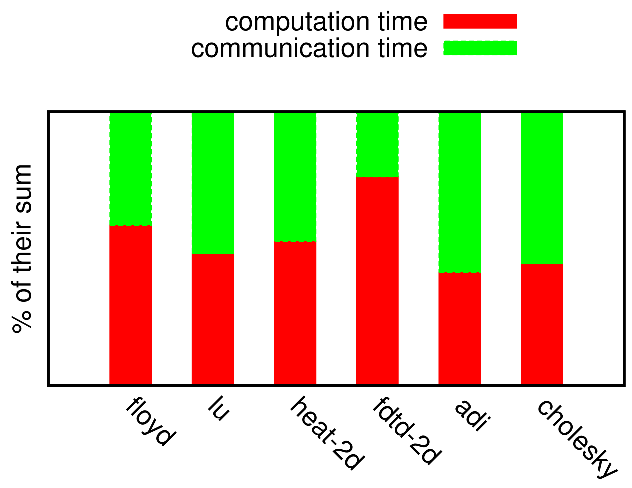


Figure 4.9: Maximum computation time and maximum communication time in auto-static across all threads on 32 nodes (multicores) of a cluster

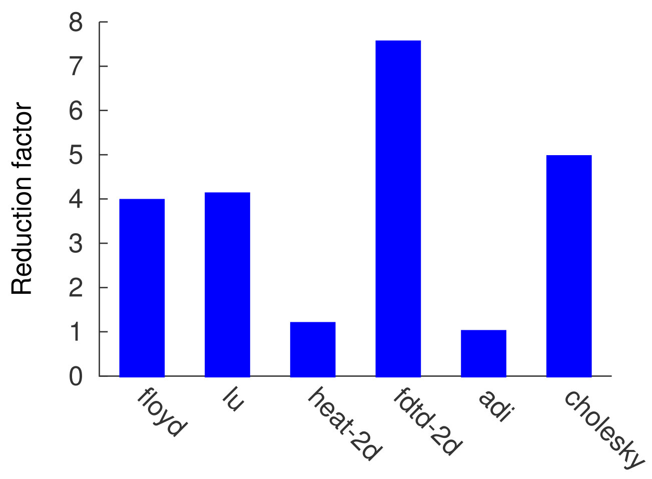


Figure 4.10: Non-overlapped communication time reduction: auto-DFDS over auto-static on 32 nodes (multicores) of a cluster

We measured the maximum communication time across all threads in auto-static, and the maximum idle time across all threads in auto-DFDS, which would include the

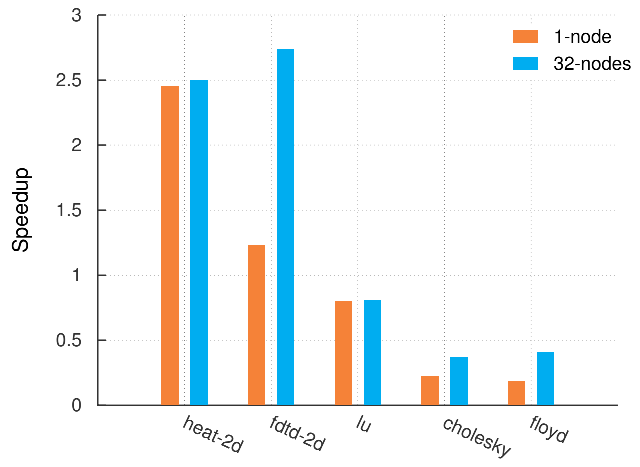


Figure 4.11: Speedup of auto-DFDS over manual-CnC on a cluster of multicores

non-overlapped communication time. Figure 4.9 compares the maximum communication time and the maximum computation time for auto-static on 32 nodes, and shows that communication is a major component of the overall execution time. Figure 4.10 shows the reduction factor in non-overlapped communication time achieved by auto-DFDS on 32 nodes. The graphs show that auto-DFDS outperforms auto-static mainly due to better communication-computation overlap achieved by performing asynchronous point-to-point communication. On 32 nodes, auto-DFDS yields a geometric mean speedup of 1.6× over auto-static.

Figure 4.11 shows the speedup of auto-DFDS over manual-CnC on both 1 node and 32 nodes. The speedup on 32 nodes is as good as or better than that on 1 node. This shows that auto-DFDS scales as well as or better than manual-CnC. The performance difference between auto-DFDS and manual-CnC on 32 nodes is due to that on a single node (shared-memory multicore). Hence, as shown in our shared-memory evaluation in Section 4.4.1, auto-DFDS outperforms manual-CnC when compiler transformations like computation tiling are better than manual implementations, and manual-CnC outperforms auto-DFDS in other cases due to orthogonal explicit data tiling transformations.

Chapter 5

Related Work

In this chapter, we discuss existing literature related to communication code generation (Sections 5.1 and 5.2), automatic parallelization frameworks (Section 5.3), and other dynamic scheduling dataflow runtime frameworks (Section 5.4).

5.1 Data movement for distributed-memory architectures

Works from literature closely related to communication code generation for distributed-memory architectures are:

- LWT – Amarasinghe and Lam [2];
- dHPF – Adve and Mellor-Crummey [1], Chavarría-Miranda and Mellor-Crummey [23];
- CLGR – Claßen and Griebl [24];
- FO – Bondhugula [13]; and
- OMPD – Kwon et al. [34].

These abbreviations will be used to refer to these works.

LWT, dHPF, CLGR and FO statically determine the data to be communicated and generate code for it, whereas OMPD determines the data to be communicated primarily using a runtime dataflow analysis technique. LWT handles only perfectly nested loops; OMPD handles only those affine loop nests which have a repetitive communication pattern (i.e., those which transfer the same set of data on every invocation of the parallel loop); dHPF, CLGR and FO are based on the polyhedral framework, like our schemes, and can handle any sequence of affine loop nests. Our framework builds upon and subsumes the state-of-the-art automatic distributed-memory code generation framework [13], while generalizing it to target heterogeneous architectures.

LWT, dHPF and CLGR use a virtual processor to physical processor mapping to handle symbolic problem sizes and number of processors. If iterations of the innermost distributed loop(s) are treated as virtual processors, then FOIFI statically determines the communication set between two virtual processors, and uses π at runtime as a mapping function from virtual to physical processors. Thus, FOIFI also uses a virtual processor model. For all these schemes, the communication code is generated such that virtual processors communicate with each other only when they are mapped to different physical processors. In spite of this, when the data being sent to different virtual processors is not disjoint and when some of those virtual processors are mapped to the same physical processor, the common portion of the data is sent multiple times to that physical processor. dHPF [23] overcomes some of this redundancy by statically coalescing data required by multiple loop nests. FOP and FO also achieve *communication coalescing* across multiple loop nests by determining the communication set for all dependences, i.e., for all dependent loop nests. Moreover, instead of a virtual processor approach, FOP and FO determine the set of receivers precisely so as to not communicate duplicate data.

dHPF determines communication code by analyzing data accesses as opposed to dependences in a way that lacks *exact dataflow* information. dHPF could be either pulling data just before it is consumed or pushing data soon after it is produced. In the former scenario, when there are multiple reads to the same location that are spread across distributed phases, the read in each distributed phase is expected to get the data at that location from the owning processor, though only the *first read* is required to pull it. In the latter scenario, when there are multiple writes to the same location that are spread across distributed phases, the write in each distributed phase is expected to send the data to all processors which read that location, though only the *last write* is required to be pushed. In contrast, FOP, FOIFI and FO use the *last writer* property of flow dependences to communicate only the *last write*.

In LWT and dHPF, the data is always communicated to and from its owner(s). Only the owner can send the latest value of an element to its subsequent read in another processor, which requires the owner to receive any previous write to that element in any other processor. However, any processor which writes an element could directly send the updated value to its subsequent read in any other processor, as in FO, FOIFI and FOP.

As for *communication coalescing*, LWT and CLGR do not perform it for arbitrary affine accesses, unlike FOP, FOIFI and FO. So, they could communicate duplicate data when there are multiple references to the same data.

FO only ensures that the receiver requires at least one element in the communicated data since it determines the set of receivers for the entire communication set. FO unnecessarily communicates the entire communication set when different receivers require different elements in the communication set. FOP reduces such unnecessary communication by partitioning the communication set and precisely determining the set of receivers for each partition.

Thus, LWT, dHPF, CLGR, FO and FOIFI schemes could lead to substantially larger volume of redundant communication than FOP scheme. Since FOIFI precisely determines the data which needs to be communicated between virtual processors by analyzing multiple dependences simultaneously, any scheme which uses the virtual processor model cannot communicate lesser volume than it. In particular, it is theoretically at least as good as schemes that use the virtual processor model, like LWT, dHPF and CLGR. Our evaluation shows that FOP clearly outperforms OMPD, FO, FOIFI, and consequently, LWT, dHPF and CLGR.

5.2 Data movement for heterogeneous architectures

Existing works for heterogeneous architectures that address problems similar to ours can be classified into those that support computation distribution on multiple heterogeneous devices [21, 32, 36, 40, 46] and those that support only single GPU device, but provide automatic data movement between CPU and GPU [30, 31, 39].

Among existing works that support distributing computation on multiple devices of a heterogeneous system [26, 32, 36, 40, 46], the work of Kim et al. [32] is the only one which completely automates data movement. Their input is an OpenCL program for a single device, which is distributed across multiple compute devices. The kernels in the program can only have affine array accesses. They determine the first and last memory location accessed by a computation partition and then send the entire data in that range to the compute device which would execute that partition. This could lead to false sharing since there could be many memory locations within the range that are not required by the associated partition. Thus, their scheme communicates significantly large volume of redundant data. To ensure consistency, they maintain a separate *virtual* buffer in the host, which is 'diff'ed and 'merge'd with the GPUs' buffers at runtime when required. This introduces additional runtime overhead. FOP, on the other hand, precisely determines memory locations that need to be communicated through static analyses and with minimal runtime overhead. However, we are unable to present an experimental comparison with their scheme as it is not available.

Leung et al. [36] describe an automatic source-level transformer in the RSTREAM compiler [37] which generates CUDA code from serial C code. Their work targets systems with multiple GPUs, but no details on inter-device data movement or results on multiple GPUs are provided.

Song and Dongarra [46] execute linear algebra kernels on heterogeneous GPU-based clusters. They develop a multi-level partitioning and distribution method, which is orthogonal to the problem we address. Their communication scheme is not automatic, but specific to the kernels addressed. Communication is driven by data dependences between atomic tasks. Since they send the entire output data of a task to any task that reads at least one value in that output data, they could send unnecessary data like FO. In contrast, FOP minimizes such redundant communication for the chosen distribution.

Among production compilers, PGI [51] and CAPS [26] have a proprietary directive based accelerator programming model, and also support OpenACC. However, to the best of our knowledge, they do not automatically distribute loop computations across different devices of a heterogeneous system. So, the issues of automatic data movement or synchronization between different devices do not arise.

CGCM [31], DyManD [30], and AMM [39] are recent works that support only a single GPU device, but automate data movement between CPU and GPU. Unlike our schemes, these do not support distributing computation on multiple devices since they cannot handle different elements of the same array being written simultaneously by different devices. They allocate the entire data on every device. AMM moves data to a device only if the existing data is stale. FOP scheme also achieves this by minimizing unnecessary data transfer. Data is transferred at the granularity of an allocation unit in CGCM and DyManD, and at the granularity of a CUDA X10 Rail in AMM, which could lead to redundant communication. FOP, by contrast, is precise in determining data to be transferred at the granularity of array elements. However, our approach is for affine array accesses and is thus complementary to CGCM and DyManD that are designed to also handle pointers and recursive data structures respectively.

Baskaran et al. [6, 7] deal with optimizing data movement between off-chip and on-chip memories on GPUs. These works are orthogonal to the problem we address in this thesis.

5.3 Automatic parallelization frameworks

Previous techniques for automatic parallelization of affine loop nests for distributed-memory architectures [1,13,24] handle only static scheduling of loops across nodes. They perform communication in a bulk-synchronous manner, i.e., after the loop is executed in parallel, communication and synchronization is performed using a "push" approach before the parallel loop is invoked again due to an outer surrounding sequential loop. They do not overlap communication with computation. Our approach uses dynamic scheduling on each node and communication across nodes is done in an asynchronous fashion, thereby achieving communication-computation overlap. Our evaluation demonstrates that it scales better than bulk-synchronous approaches. Our framework builds upon the state-of-the-art automatic distributed-memory parallelization framework [13] and subsumes it to target a dataflow runtime.

Reddy and Bondhugula [45] build upon the work presented in this thesis. They use our dynamic scheduling dataflow runtime framework (along with our data movement techniques) in conjunction with techniques for on-demand data allocation via data tiling, and more general computation placement schemes including *sudoku* mappings as well as arbitrary ones. The computation placement schemes used in their work are all dynamic a priori, but the choice of the placement function is made at compiler time itself. Their work improves data locality further and allows weak scaling.

Baskaran et al. [8] provide techniques for extracting tasks and constructing the task dependence graph. They construct the entire task dependence graph in memory and then schedule it. Since the task graph is shared and modified across all threads, it could become a bottleneck when there are huge number of threads. Moreoever, their compiler-assisted runtime is limited to shared-memory and does not deal with challenges associated with distributed-memory. Their work has been the main motivation behind our compiler-assisted runtime, and our goal was to target a distributed-memory cluster of multicores. In contrast to their techniques, our approach does not build the entire task dependence graph in memory but uses compiler generated routines that semantically encapsulate it. We also generate runtime components to manage memory-based dependences and communication across nodes in distributed-memory. Our tool generates code that can be executed on shared-memory, distributed-memory, or a combination of both. Our evaluation shows that the performance of our approach on shared-memory is similar to or better than that of their approach. Thus, our framework builds upon and subsumes the state-of-the-art automatic dynamic scheduling framework.

5.4 Dataflow runtime frameworks

In this section, we first discuss frameworks that are specific to linear algebra computations, and then discuss frameworks that handle arbitrary computations.

5.4.1 Frameworks for linear algebra computations

A number of works have focused on building scalable frameworks for a certain class of applications, like linear algebra kernels [16,17,46]. Our work has been motivated by these approaches, with the main difference being that we intend to provide a fully automatic solution through compiler support. Our techniques are also applicable to a more general class of computation – affine loop nests.

Directed Acyclic Graph Engine (DAGuE) [17] is driven by a domain specific language in which the user is expected to express the computation as a DAG of tasks, where the nodes are sequential computation tasks, and the edges are dependences between tasks. Hence, the burden of expressing parallelism and locality is shifted to the user. In contrast, our framework automatically extracts such a DAG of tasks. DPLASMA [16] implements linear algebra kernels based on DAGuE. DAGuE does not build or unroll the entire task graph in memory, but encapsulates the DAG concisely in memory using a representation conceptually similar to the *Parameterized Task Graph* (PTG) [25]. Our techniques to compactly represent the task graph use *Parameterized Task Functions* (PTFs). In contrast to PTG, PTFs encapsulate differences in dependences between tasks on the same node and dependences between tasks on different nodes, thereby allowing handling of memory-based dependences; PTFs also encapsulate precise communication semantics (at the granularity of array elements).

In DAGuE, the user also defines how the tasks should be distributed across the cores via the data distribution. DAGuE uses a fully distributed decentralized scheduler without global synchronization, like our framework. The scheduler on each core dynamically schedules tasks waiting only on dependences between tasks, while allowing work stealing between cores of the same node. Communications are determined by the dataflow between tasks, and are handled by a separate thread. Communication is non-blocking and is overlapped with computation. Their communication component is thus conceptually similar to ours, but unlike our approach, the threads on different nodes coordinate with each other (using control messages) to send and receive communication.

Song and Dongarra [46] design a distributed dynamic scheduling runtime system for a cluster of nodes containing CPUs and multiple GPUs. They distribute data to CPUs and GPUs statically. The tasks are statically distributed to CPUs and GPUs using the *owner-computes* rule. Each node runs its own runtime system that schedules tasks within the node dynamically, similar to our framework. For each data tile that a node owns, the node maintains an ordered list of tasks that accesses that data tile. To act on its task instance, the node requires specific information. For example, for a task instance that writes a data tile, a node requires the ready status of the task's inputs to know if it can execute it; for a task instance that reads a data tile, a node requires the location of the task's output to know the node (or device) it should communicate to. They design a distributed protocol such that each node maintains the required information locally for each of its task instances based on a predetermined distribution without any communication with other nodes. We achieve the same objective using the Synthesized Runtime Interface to maintain the status locally for all the tasks that will be computed or received on a node.

In the system designed by Song and Dongarra [46], the node that will execute a task instance maintains its inputs, output, and the ready status of each input. Each node, therefore, has a partition of the DAG of tasks in memory, using which tasks are scheduled dynamically driven by data-availability; the partial task graph is built without any coordination with other nodes. Our framework, on the other hand, does not maintain even a partial task graph in memory, but only maintains status on tasks that is built and maintained without any coordination with other nodes. Communication in their system is determined by the dataflow between tasks at the granularity of data tiles, whereas communication in our system is precise at the granularity of array elements. Communication is asynchronous and is overlapped with computation. There is a separate thread each for intra-node and inter-node communication. The inter-node communication thread preemptively posts an anonymous receive, and checks whether it has finished with busy-polling. Our communication framework is designed similarly.

5.4.2 General-purpose frameworks

Recent works like the codelet model [35, 52] (which is inspired by the EARTH system [48]), StarPU [4], and Concurrent Collections (CnC) [18] focus on providing high-level programming models which enable easy expression of parallelism. These models decouple scheduling and program specification, which is tightly coupled in current programming models. The notion of a “task” in our work is conceptually similar to that of a “codelet” in the codelet model, a “task” in StarPU, and a “step” in CnC. In these models, the application developer or user specifies the program in terms of tasks along with its inputs and outputs, while the system (or system developer) is responsible for scheduling the tasks efficiently on the parallel architecture. However, the user is not completely isolated from locality aspects of modern architectures. As an example, one of the key issues in leveraging task scheduling runtimes such as CnC is in determining the right decomposition into tasks and granularity for the tasks, i.e., block or tile size; a smaller block size increases the degree of available *asynchronous parallelism*, but also increases the overhead in maintaining tasks and managing data. Choosing the right decomposition can improve the performance by orders of magnitude – this is evident from our experimental results. The decomposition into tasks and choice of granularity has a direct connection with loop transformations such as tiling, making a strong case for integration of compiler support.

The CnC model expects the user to provide dependences between tasks. Intel CnC [28] allows the user to also specify the nodes which would consume the data produced in a task. This enables the Intel CnC runtime to push the produced data to the nodes preemptively on task completion, instead of pulling the required data once a task is scheduled. For affine loop nests, our techniques can be used to automatically extract this information parameterized on a task – the tasks that are dependent on it, and the nodes that consume data produced in it (i.e., the receivers).

The CnC model allows the user to provide tags that are used to specify control dependencies between tasks. The Intel CnC runtime, by default, schedules a task when all its tags are prescribed or generated, but if the data required by the task is not available, it is put back on the waiting queue (requeued). The user has the option to either pre-generate the tags or generate them on-the-fly when data becomes available. A recent performance evaluation of the Intel CnC runtime [18] has observed that pre-generating the tags might not perform well due to requeue events, and it might be useful to generate the tags when the data becomes available. In our framework, a task is ready to be scheduled on a node only when all the tasks it depends on have completed, and the data required by it is available on that node. For affine loop nests, our techniques can be used to automatically generated tags in CnC only when the data becomes available.

The CnC model adheres to the dynamic single assignment form; the data is accessed by value and there is no overwriting of values. This helps increase the degree of available *asynchronous parallelism* since there are no memory-based data dependences (WAW and WAR). As a consequence, the memory footprint of the program could be higher. This could drastically affect performance, unless the memory is managed efficiently and the unused data is garbage collected. Intel CnC allows the user to provide the total uses or references of a value, so that its memory location can be garbage collected efficiently. For affine loop nests, our techniques can be used to automatically extract the number of tasks that reference a value produced in a task, which is the same as the number of tasks that are dependent on a task due to flow (RAW) dependences. On the other hand, our framework does not have the dynamic single assignment restriction; our framework handles memory-based dependences by ignoring it across cores which do not share memory, but preserving it across cores which share memory.

The StarPU runtime [3, 4] designs a scheduling framework for heterogeneous computational resources or workers. The framework provides a uniform interface for implementing different scheduling policies. Each worker is associated with an abstract queue, on which it can push and pop tasks. A scheduling policy is defined by the way the workers interact with the set of queues. For instance, the queue could be shared among the workers or each worker could have its own queue. StarPU provides several predefined scheduling policies for the user to choose from. Our framework uses a scheduling policy similar to their greedy scheduling policy with priority, where a worker greedily chooses a task from a shared queue (local to the node) based on some priority; a worker does not choose to stay idle when a task is ready to be executed. Such a greedy scheduling policy might not be suitable for heterogeneous computational resources because some workers might be better equipped to execute a task than others. The scheduling policies in StarPU are orthogonal to our work, and our framework could be adapted to use any of their scheduling policies, including those that require per-worker queues.

Several new dynamic scheduling runtimes have emerged in the recent past and are under active development. These include StarPU [3, 4], ETI SWARM (for the codelet model) [35], Intel CnC [18], and Open Community Runtime (OCR) (inspired by the codelet and CnC models) [38]. The StarPU runtime is targeted for heterogeneous multicore architectures (CPUs along with accelerators like GPUs); SWARM runtime and OCR are targeted for exascale architectures; Intel CnC runtime is targeted for shared and distributed memory architectures. The design of these runtimes is focused on their targeted architectures, but is not limited to it, similar to our work. These runtimes share some of the same design objectives as those of our work, like efficiently utilizing resources on parallel architectures by balancing the load, and overlapping data movement with computation. Our techniques are sufficiently generic, and could possibly be adapted to generate code for these runtimes. However, the thrust of our work is in coupling runtime support with powerful compiler transformation, parallelization, and code generation support to provide a fully automatic solution. This is done so that efficient execution on shared as well as distributed memory is achieved with no programmer input. Our objective here is not to compare the efficiency of the developed runtime with emerging ones. The choice to develop our own runtime was driven by the need to allow sufficient customization and flexibility for current and future compiler support. Our runtime is specialized for affine loop nests; it maintains only a counter or two per task on each node, handles memory-based dependences, and incorporates precise communication semantics by packing and unpacking values of array elements to and from buffers.

Chapter 6

Conclusions and Future Work

In this chapter, we first present conclusions of the work done in this thesis, and then discuss the scope for future work.

6.1 Conclusions

We first proposed compilation techniques to free programmers from the burden of moving data on architectures that do not have a shared address space. We were able to generate efficient data movement code statically using a *source-distinct* partitioning of dependences. Minimum communication volume was achieved with a majority of dependence patterns and for all benchmarks considered. To the best of our knowledge, our tool is the first one to parallelize affine loop nests for a combination of CPUs and GPUs while providing precision of data movement at the granularity of array elements.

On a heterogeneous system, we showed that our data movement scheme (FOP) reduces the communication volume by a factor of 11× to 83×, resulting in a mean execution time speedup of 1.53× over the best existing scheme (FO). For communication-intensive benchmarks like Floyd-Warshall, when running on 4 GPUs, we demonstrated that our data movement scheme yields a mean speedup of 2.56× over 1 GPU. On a distributed-memory cluster, we showed that our scheme (FOP) reduces the communication volume by a factor of 1.4× to 63.5×, resulting in a mean speedup of 1.55× over the best existing scheme (FO). Our scheme gave a mean speedup of 3.06× over another existing scheme (OMPD) and a mean speedup of 2.19× over hand-optimized UPC versions of these codes.

We then described the design and implementation of a new dataflow runtime system for modern parallel architectures that are suitable for target by compilers capable of extracting tasks and dependence information between tasks. We coupled the runtime with a source-to-source polyhedral optimizer to enable fully automatic dynamic scheduling on a distributed-memory cluster of multicores. The design of the runtime takes into account simultaneous load-balanced execution on shared and distributed memory cores, handling of memory-based dependences, and asynchronous point-to-point communication. The resulting system is also the first automatic parallelizer that uses dynamic scheduling for affine loop nests on distributed-memory.

On 32 nodes with 8 threads per node, our compiler-assisted runtime yields a mean speedup of 1.6× over the state-of-the-art automatic approach, and a mean speedup of 143.6× over the sequential version. On a shared-memory system with 32 cores, our runtime yields a speedup of up to 2.5× over the state-of-the-art dynamic scheduling approach, and a mean speedup of 23.5× over the sequential version. Our automatic framework also significantly outperformed hand-optimized Intel CnC codes in some cases due to advanced compiler transformations like loop tiling.

The compiler-assisted dataflow runtime framework we proposed is thus end-to-end fully automatic, thereby providing high productivity while delivering high parallel performance on distributed-memory architectures. We believe that our techniques will be able to provide OpenMP-like programmer productivity and deliver CnC-like parallel performance for distributed-memory architectures if implemented in compilers.

6.2 Future work

- Our framework can be extended to target a distributed-memory cluster of nodes with CPUs and multiple GPUs. Both the data movement techniques and the coarse-grained dataflow parallelism extraction techniques are generic and can be adapted to such systems.

- Sophisticated priority heuristics for scheduling tasks that exploit data locality can be explored to further improve performance.

- Techniques for on-demand data allocation via data tiling can be integrated into our framework to further improve data locality and allow weak scaling.

- Other sub-problems in automatic parallelization of sequential code for distributed-memory architectures, like automatically choosing an optimal computation distribution or placement, are orthogonal and solutions to these orthogonal problems can be coupled with our framework.

References

[1] Vikram Adve and John Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming Language Design and Implementation*, PLDI '98, pages 186–198, New York, NY, USA, 1998. ACM.

[2] Saman P. Amarasinghe and Monica 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*, PLDI '93, pages 126–138, New York, NY, USA, 1993. ACM.

[3] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. StarPU: a Runtime System for Scheduling Tasks over Accelerator-Based Multicore Machines. Technical Report 7240, INRIA, March 2010.

[4] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[5] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. Tiling stencil computations to maximize parallelism. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 40:1–40:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[6] Muthu Manikandan Baskaran, Uday Bondhugula, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic Data Movement and Computation Mapping for Multi-level Parallel Architectures with Explicitly Managed Memories. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 1–10, New York, NY, USA, 2008. ACM.

[7] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA Code Generation for Affine Programs. In Rajiv Gupta, editor, *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin Heidelberg, 2010.

[8] Muthu Manikandan Baskaran, Nagavijayalakshmi Vydyanathan, Uday Kumar Reddy Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Compiler-assisted Dynamic Scheduling for Effective Parallelization of Loop Nests on Multicore Processors. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '09, pages 219–228, New York, NY, USA, 2009. ACM.

[9] Cédric Bastoul. The Clan User guide. `http://icps.u-strasbg.fr/people/bastoul/public_html/development/clan/docs/clan.pdf`.

[10] Cedric 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, pages 7–16, Washington, DC, USA, 2004. IEEE Computer Society.

[11] Cédric Bastoul. Clan: The Chunky Loop Analyzer, 2012. `http://icps.u-strasbg.fr/people/bastoul/public_html/development/clan/`.

[12] Cédric Bastoul. CLooG: The Chunky Loop Generator, 2013. `http://www.cloog.org`.

[13] Uday Bondhugula. Compiling Affine Loop Nests for Distributed-memory Parallel Architectures. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 33:1–33:12, New York, NY, USA, 2013. ACM.

[14] Uday Bondhugula. PLUTO: A polyhedral automatic parallelizer and locality optimizer for multicores, 2013. `http://pluto-compiler.sourceforge.net`.

[15] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '08, pages 101–113, New York, NY, USA, 2008. ACM.

[16] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Herault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asim Yarkhan, and Jack Dongarra. Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA. *Univ. of Tennessee, CS Technical Report, UT-CS-10-660*, 2010.

[17] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pierre Lemarinier, and Jack Dongarra. DAGuE: A generic distributed DAG engine for high performance computing. *Parallel Computing*, 38(1):37–51, 2012.

[18] Zoran Budimlic, Aparna Chandramowlishwaran, Kathleen Knobe, Geoff Lowney, Vivek Sarkar, and Leo Treggiari. Multi-core Implementations of the Concurrent Collections Programming Model. In *CPC09: 14th International Workshop on Compilers for Parallel Computers*, 2009.

[19] Berkeley UPC - Unified Parallel C. `http://upc.lbl.gov`.

[20] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[21] Compiler and Architecture for Superscalar and Embedded Processors. `http://www.irisa.fr/caps/`.

[22] Aparna Chandramowlishwaran, Kathleen Knobe, and Richard Vuduc. Performance evaluation of concurrent collections on high-performance multicore computing systems. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, April 2010.

[23] Daniel Chavarría-Miranda and John Mellor-Crummey. Effective communication coalescing for data-parallel applications. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 14–25, New York, NY, USA, 2005. ACM.

[24] Michael Claßen and Martin 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.

[25] M. Cosnard and M. Loi. Automatic task graph generation techniques. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, HICSS '95, pages 113–, Washington, DC, USA, 1995. IEEE Computer Society.

[26] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A Hybrid Multicore Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, 2007.

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

[28] Intel® Concurrent Collections (CnC) for C/C++, 2013. `http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc`.

[29] Intel® Thread Building Blocks (TBB), 2014. `https://www.threadingbuildingblocks.org/`.

[30] Thomas B. Jablin, James A. Jablin, Prakash Prabhu, Feng Liu, and David I. August. Dynamically managed data for CPU-GPU architectures. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, pages 165–174, New York, NY, USA, 2012. ACM.

[31] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[32] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a single compute device image in OpenCL for multiple GPUs. In *Proceedings of the 16th ACM symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 277–288, New York, NY, USA, 2011. ACM.

[33] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 211–222, New York, NY, USA, 2007. ACM.

[34] Okwan Kwon, Fahed Jubair, Rudolf Eigenmann, and Samuel 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, pages 75–84, New York, NY, USA, 2012. ACM.

[35] Christopher Lauderdale and Rishi Khan. Towards a Codelet-based Runtime for Exascale Computing: Position Paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.

[36] Allen Leung, Nicolas Vasilache, Benoît Meister, Muthu Baskaran, David Wohlford, Cédric Bastoul, and Richard Lethin. A mapping path for multi-GPGPU accelerated

computers from a portable high level programming abstraction. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 51–61, New York, NY, USA, 2010. ACM.

[37] Benoît Meister, Nicolas Vasilache, David Wohlford, Muthu Manikandan Baskaran, Allen Leung, and Richard Lethin. R-Stream Compiler. In David A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1756–1765. Springer, 2011.

[38] Open Community Runtime, 2013. `https://01.org/projects/open-community-runtime`.

[39] Sreepathi Pai, R. Govindarajan, and Matthew J. Thazhuthaveetil. Fast and efficient automatic memory management for GPUs using compiler-assisted runtime coherence scheme. In *Proceedings of the 21st international conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 33–42, New York, NY, USA, 2012. ACM.

[40] Portland Group Inc. Application Programming Interface. `http://www.pgroup.com`.

[41] PolyBench/C - the Polyhedral Benchmark suite, 2012. `http://polybench.sourceforge.net`.

[42] PolyLib - A library of polyhedral functions, 2010. `http://icps.u-strasbg.fr/polylib/`.

[43] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 4–13, New York, NY, USA, 1991. ACM.

[44] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A Language and Compiler for Optimizing

Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM.

[45] Chandan Reddy and Uday Bondhugula. Effective automatic computation placement and data allocation for parallelization of regular programs. In *International Conference on Supercomputing*, 2014.

[46] Fengguang Song and Jack Dongarra. A scalable framework for heterogeneous GPU-based clusters. In *Proceedinbgs of the 24th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 91–100, New York, NY, USA, 2012. ACM.

[47] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The Pochoir Stencil Compiler. In *Proceedings of the 23rd ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[48] Kevin Bryan Theobald. *Earth: An Efficient Architecture for Running Threads*. PhD thesis, Montreal, Que., Canada, Canada, 1999. AAINQ50269.

[49] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[50] Sven Verdoolaege. Integer Set Library - an integer set library for program analysis, 2014. `http://www.ohloh.net/p/isl`.

[51] Michael Wolfe. Implementing the PGI Accelerator model. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 43–50, New York, NY, USA, 2010. ACM.

[52] Stéphane Zuckerman, Joshua Suetterlein, Rob Knauerhase, and Guang R. Gao. Using a "codelet" program execution model for exascale machines: Position paper. In *Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing*

Systems for the Exaflop Era, EXADAPT '11, pages 64–69, New York, NY, USA, 2011. ACM.