Fall 2014

Automatic translation of non-repetitive OpenMP to MPI

Fahed Jubair
Purdue University

Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Fahed Jubair

Entitled
Automatic Translation of Non-Repetitive OpenMP to MPI

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

RUDOLF EIGENMANN Chair	VIJAY S. PAI
MILIND KULKARNI	
MITHUNA S. THOTTETHODI	
SAMUEL P. MIDKIFF	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): RUDOLF EIGENMANN

Approved by: M. R. Melloch 10-09-2014
Head of the Graduate Program Date

AUTOMATIC TRANSLATION OF NON-REPETITIVE OPENMP TO MPI

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Fahed A. Jubair

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

December 2014

Purdue University

West Lafayette, Indiana

To my precious family: my parents Elham and Ahmad, my brother Mohammad,
and my sisters Asma and Hanan.

ACKNOWLEDGMENTS

I would like to show my utmost appreciation to Professor Rudolf Eigenmann for being an extraordinary advisor throughout the years of getting my PhD degree. I also would like to thank professor Samuel Midkiff for his valuable help with my research. My deepest gratitude to my laboratory mates: Okwan Kwon, Amit Sabne, Putt Sakdhnagool and Aurangzeb for all the interesting discussions about research and about life. Lastly, I thank my dear friends Mohammad Hajjat and Jonathan Pullum for making my life at Purdue more enjoyable.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Fahed, Jubair A. Ph.D., Purdue University, December 2014. Automatic Translation of Non-Repetitive OpenMP To MPI. Major Professor: Rudolf Eigenmann .

Cluster platforms with distributed-memory architectures are becoming increasingly available low-cost solutions for high performance computing. Delivering a productive programming environment that hides the complexity of clusters and allows writing efficient programs is urgently needed. Despite multiple efforts to provide shared memory abstraction, message-passing (MPI) is still the state-of-the-art programming model for distributed-memory architectures.

Writing efficient MPI programs is challenging. In contrast, OpenMP is a shared-memory programming model that is known for its programming productivity. Researchers introduced automatic source-to-source translation schemes from OpenMP to MPI so that programmers can use OpenMP while targeting clusters. Those schemes limited their focus on OpenMP programs with repetitive communication patterns (where the analysis of communication can be simplified). This dissertation reduces this limitation and presents a novel OpenMP-to-MPI translation scheme that covers OpenMP programs with both repetitive *and* non-repetitive communication patterns. We target laboratory-size clusters of ten to hundred nodes (commonly found in research laboratories and small enterprises).

The presented translation scheme consists of a compiler and a runtime system. The compiler analyzes the OpenMP program and converts it into message-passing form. In the translated code, the compiler provides information about produced and consumed shared array elements by each thread, information that is collected by an array data flow analysis. The runtime system uses this information to schedule communication.

Limitations in the compiler analysis can lead to excessive communication. To this end, we present novel compiler algorithms that perform accurate array data flow analysis for OpenMP programs. This is accomplished by the following contributions: (i) The π *operator*: an abstract representation that exploits high level information about the partitioning used in parallel loops to improve the accuracy of cross-thread analysis; (ii) *Delayed symbolic evaluation*: a compiler algorithm that performs all operations in the dataflow analysis without conservative approximation; and (iii) The *variant-set* and *region-based* analyses: compiler algorithms that enable the dataflow analysis to reason about Gen and Kill sets that vary across different instances of a loop or statement, a pattern that often exists in non-repetitive programs.

Similarly, limitations in the runtime schemes for communication generation can lead to high overheads. To this end, we present a new runtime communication scheme that generates messages for repetitive *and* non-repetitive communication patterns with low runtime overheads. This is accomplished by an algebra of the π operator that reduces the needed computation to schedule communication.

With our contributions, six non-repetitive and four repetitive OpenMP benchmarks have been efficiently scaled to a cluster of 64 cores. In contrast, the state-of-the-art translator scaled only the four repetitive benchmarks. In addition, our translation scheme was shown to outperform or perform as well as the state-of-the-art translator. We also compare the translation scheme with available hand-coded MPI and *Unified Parallel C* (UPC) programs.

1. INTRODUCTION

1.1 Motivation

The state of the art of programming distributed memory machines – today's most common high-performance computing (HPC) platforms – is dire. Despite multiple efforts to provide better programming environments, most software engineers still have to use the "assembly language of parallel programming", MPI [1], to write efficient HPC code. MPI requires programmers to explicitly partition data and computation, and to insert communication messages. The dissertation improves this situation by allowing programmers to target HPC platforms using a shared-memory abstraction, as represented by standard OpenMP [2].

Among many approaches to enhance productivity of HPC software, this work is related most closely to those that create a shared-memory abstraction of the underlying machine. For example, High Performance Fortran (HPF) [3] provided a shared address space plus user directives for data distribution. Despite the creation of several industrial HPF compilers, this programming model did not succeed. UPC [4] is a more recent effort to provide a global address space abstraction. UPC programmers need to manually confront issues of data distribution and thread-data affinity to assist the compiler.

A shared-memory abstraction can also be provided at runtime, such as in Software Distributed Shared Memory (SDSM) systems. TreadMarks [5] was such a system; it was used in Intel's Cluster OpenMP [6] product and in several OpenMP-related research projects [7–9]. A primary issue is the inherent overhead of page-based coherence mechanisms. Also, in some approaches, OpenMP programs had to be modified for use in SDSMs.

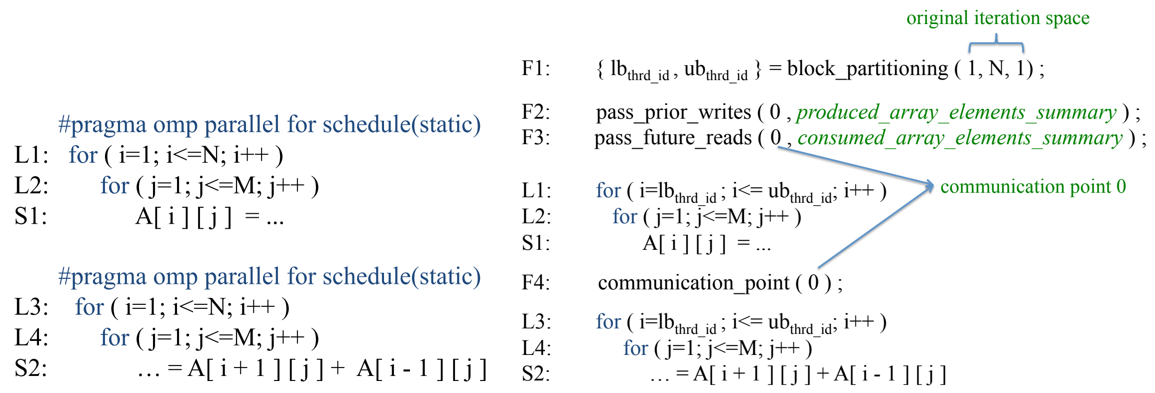
The dissertation presents a programming system that allows standard OpenMP programs to be translated into efficient code for distributed memory machines – we target clusters of 10–100 processors. Those clusters can be commonly found in research laboratories and small enterprises. Similar to HPF and UPC systems, our approach makes use of an optimizing compiler. Unlike previous OpenMP-to-MPI translators [10–13], this dissertation takes into consideration programs with *non-repetitive* communication patterns. We will also evaluate how close our approach can come to the performance of hand-coded MPI and UPC programs (when available).

Another important contribution of this dissertation is providing general concepts for improving accuracy of *Array Data Flow Analysis* (ADFA). In the literature, many compiler optimizations and transformations require, or would benefit from, knowledge about the elements of an array that are accessed within a loop nest. Two examples of where such information is useful are *array privatization* [14, 15] and determining which array elements produced in a parallelized loop need to be communicated to another process or thread [10, 13]. *Array Data Flow Analysis* (ADFA) has been used to obtain such information in both sequential and parallel programs. The accuracy of this analysis is key to the performance of the compiled programs.

1.2 Translation Process Overview

The translation scheme consists of a compiler and a runtime system. The compiler analyzes the OpenMP program and converts it into an *single-program-multiple-data* (SPMD) [16] code, which can be executed on a cluster. During the execution, the runtime system generates communication using the MPI communication library.

A communication point in the SPMD code is a point where communication may be necessary during the execution. For each communication point, the compiler determines (i) an exact description of the most recent produced shared array elements by each thread *in the past*; and (ii) an approximate description of the consumed shared array elements by each thread *in the future*. The compiler passes these descriptions



```
     #pragma omp parallel for schedule(static)
L1:  for ( i=1; i<=N; i++ )
L2:      for ( j=1; j<=M; j++ )
S1:          A[ i ] [ j ]  = ...

     #pragma omp parallel for schedule(static)
L3:  for ( i=1; i<=N; i++ )
L4:      for ( j=1; j<=M; j++ )
S2:          ... = A[ i + 1 ] [ j ] +  A[ i - 1 ] [ j ]
```

(a) The OpenMP input. There is an implicit barrier at the end of each parallel loop.

```
F1:   { lb_thrd_id , ub_thrd_id } = block_partitioning ( 1, N, 1) ;

F2:   pass_prior_writes ( 0 , produced_array_elements_summary ) ;
F3:   pass_future_reads ( 0 , consumed_array_elements_summary ) ;

L1:   for ( i=lb_thrd_id ; i<= ub_thrd_id; i++ )
L2:     for ( j=1; j<=M; j++ )
S1:         A[ i ] [ j ]  = ...

F4:   communication_point ( 0 ) ;

L3:   for ( i=lb_thrd_id ; i<= ub_thrd_id; i++ )
L4:      for ( j=1; j<=M; j++ )
S2:          ... = A[ i + 1 ] [ j ] + A[ i - 1 ] [ j ]
```

(b) The translated code. Statements $F1 - F4$ are function calls inserted by the compiler to initiate proper actions by the runtime system.

Fig. 1.1.: An overview of the presented OpenMP to MPI translation process.

to the runtime system. The runtime system generates communication to satisfy the *inter-thread* dependencies between produced and consumed shared data.

Figure 1.1 shows an example of the presented translation scheme. In the translated code, the compiler inserts the following function calls that initiate proper actions by the runtime system:

- $F1$, which partitions the iterations spaces of the parallel loops $L1$ and $L3$ using block partitioning (as stated by OpenMP *schedule* directives).
- $F2$ and $F3$, which pass information about prior written and future read shared array elements at communication point 0.
- $F4$, which initiates communication generation at communication point 0.

1.3 Challenges

The translation scheme employs a compiler Array Data Flow Analysis (ADFA) framework to analyze accessed array elements in the input OpenMP program. The accuracy of this analysis impacts the performance of our translated programs because accessed array elements information is used to determine communication sets.

```
L0: for( i = 0; i <= N − 1; i++) {

        #pragma omp parallel for
L1:     for ( k = i; k <= N; k++ )
S1:         x [ k ] =  x [ k ] + x [ i ] * r ;

   }
```

Fig. 1.2.: An example of a non-repetitive OpenMP program. Due to the triangular access pattern, the accessed array elements for individual instances of the inner nested loop $L1$ (i.e., Gen and Kill sets) vary for different iterations of the enclosing loop $L0$.

Prior approaches [10, 11] were proposed that extended classical ADFA to analyze OpenMP programs. Those approaches have two accuracy issues. The first issue is assuming that the analyzed program is *repetitive*. A program is repetitive if all parallel loops in the program are repetitive. A parallel loop is repetitive if array elements written and read by a given thread executing the loop are the same for every repetition (i.e. instance) of the loop. Two common reasons for a parallel loop being not repetitive are that it is nested within one or more outer sequential loops and the indices of those outer sequential loops appear in array accesses or the parallel loop is non-rectangular (e.g., triangular) causing different elements to be accessed by each instance of the loop.

For an ADFA, a parallel loop not being repetitive means that the *Gen* and *Kill* sets for different instances of the loop will be different (i.e., variant), as shown in Figure 2.1a. ADFAs that assume repetitive programs are forced to make conservative assumptions when confronted with non-repetitive parallel loops. Generally, they assume that any element accessed by any thread executing the parallel loop is accessed by all threads executing the loop, and any array element accessed by any instance of the loop is accessed by all instances of the loop.

The second issue with prior ADFAs is that they have insufficient knowledge of the partitioning semantics of parallel loops. They represent parallel loops as sequential loops whose bounds have a parameterized thread number. We refer to this method as *explicit static partitioning*.

Explicit static partitioning loses information about the bounds and the partitioning scheme of the original non-partitioned iteration space. Insufficient symbolic information while analyzing cross-thread relationships often forces ADFAs to perform dataflow operations (intersection, union and subtraction) conservatively, reducing accuracy in the process.

Due to the aforementioned two issues, using previous ADFA frameworks in our translation scheme leads to inaccurate descriptions about accessed array elements, which increases the communication volume and lowers performance. This dissertation will introduce compiler algorithms that enable ADFA frameworks to analyze OpenMP programs while overcoming these sources of inaccuracy. In addition, our compiler algorithms include general concepts that can be extended to traditional ADFAs, beyond OpenMP.

The assumption of translated programs having repetitive communication patterns has also been used to simplify the runtime system by previous OpenMP-to-MPI translators [10,11]. A repetitive communication has the same communication schedules for all instances of a particular communication point. Therefore, the runtime overhead of communication scheduling can be amortized because messages need to be computed only once at runtime.

Byn contrast to repetitive communication, communicated array elements for non-repetitive communication patterns vary across different instances of the communication points. Therefore, new communication schedules may be needed for every instance, introducing the problem of potential runtime overhead. This dissertation will introduce a new runtime communication scheduling scheme to overcome this problem.

1.4 Specific Contributions

The key contribution of this dissertation is a fully automatic source-to-source translation scheme from OpenMP to MPI. The translation scheme makes it feasible for programmers to use standard OpenMP for computation-intensive algorithms on clusters of ten to hundred nodes. We target programs with regular write data accesses and both repetitive and non-repetitive communication patterns. By contrast, prior translation schemes only handle programs with repetitive communication patterns.

The translation scheme consists of a compiler and runtime systems that include the following new concepts (which we introduce to overcome the aforementioned challenges):

- A compiler producer-consumer array data flow analysis (PCDFA) for OpenMP programs that accounts for the partitioning semantics.

  We present PCDFA, an ADFA that collects reaching definitions and upwardly exposed uses of shared array elements for each statement in an OpenMP program. PCDFA includes the following concepts:

  - The π *operator*: An abstract representation of partitioned iteration spaces that captures the partitioning semantics implied by OpenMP directives. The π operator retains information about original iteration spaces and the partitioning scheme across threads that is relevant for accurate dataflow operations.
  - *Delayed symbolic evaluation*: A compiler algorithm that postpones the evaluation of a conservative operation occurs during PCDFA's computation by representing this operations as an unevaluated expression. Later in the analysis, postponed unevaluated expressions are either simplified or evaluated as further symbolic information becomes available. Should a full compile-time evaluation of some operations not be possible, delayed symbolic evaluation simplifies and retains these operations as unevaluated expressions, allowing them to be accurately evaluated at runtime.

- Compiler analyses that extend the producer-consumer array data flow analysis (PCDFA) for programs with non-repetitive parallel loops.

  As explained in Section 1.3, classical ADFAs generate conservative array section information when analyzing non-repetitive programs. To overcome this, we present compiler algorithms that allow ADFAs to perform accurately when confronted with loops or statements that have variant Gen and Kill sets, as follows:

  - The *variant-set* analysis: A compiler analysis that aims to find *accurate* (i.e., not approximate) symbolic expressions of dataflow solutions for each statement *instance* within *enclosing loops*. By properly representing variant Gen and Kill sets, dataflow operations can be extended to apply while reasoning about different instances. A key insight is that a *pattern* often holds for variant dataflow sets that allows the desired representation of dataflow solutions to be computed using a bounded number of instances.
  - The *region-based* analysis: A compiler analysis that prevents conservative effects from statements outside enclosing loops to preclude variant-set analysis. This is accomplished by forming enclosing loops in the program into *regions* that get analyzed *in isolation*. Next, each enclosing loop is collapsed into a single node that represents all iterations, and the combined dataflow solution of all statement instances within the loop will be used when analyzing the whole program.

- A Runtime communication scheduling scheme that incurs low runtime overheads.

  We present a runtime communication scheduling scheme that have low runtime overheads for both repetitive *and* non-repetitive communication patterns. By exploiting an algebra of π operators, our scheme reduces the computation needed to determine messages.

Using the aforementioned contributions, we implement a full automatic source-to-souce OpenMP-to-MPI translation system and evaluate its performance on a cluster of 64 cores using ten (six non-repetitive and four repetitive) OpenMP benchmarks. On average, all ten benchmarks achieve a speedup of 3.8x over OpenMP on 8 cores. In contrast, a state-of-the-art translator scaled only the four repetitive benchmarks and obtained an average speedup of 3.3x. Our translation scheme was shown to outperform a state-of-the-art translator for one repetitive benchmark by 1.44x and achieve the same performance for the other three repetitive benchmarks. In addition, we compare against available hand-coded MPI and UPC programs. On average, the OpenMP-to-MPI translator achieves 54% and 60% of the performance of MPI and UPC, respectively. This was achieved with no information beyond a standard OpenMP program is required.

This dissertation targets OpenMP programs that have loop-level parallelism. Handling other forms of parallelism such as task or function parallelism is beyond the scope of this dissertation.

1.5 Dissertation Organization

This dissertation is organized as follows: Chapter 2 describes the producer-consumer array data flow analysis (PCDFA). Chapter 3 presents compiler analyses that extend the producer-consumer array data flow analysis (PCDFA) to non-repetitive OpenMP programs. Both Chapter 2 and Chapter 3 include an early performance evaluation. Chapter 4 describes the new communication scheduling scheme.

Chapter 5 puts everything together and describes the fully automated translation scheme of OpenMP to MPI. We also present an evaluation of overall performance on a cluster of 64 cores. Note that the performance evaluation presented in earlier chapters is now being accumulated in Chapter 5.

Chapter 6 surveys related work. In Chapter 7, we conclude the dissertation and discuss potential future work.

2. A NEW PRODUCER-CONSUMER ARRAY DATA FLOW ANALYSIS FOR OPENMP

Array data flow analysis (ADFA) is a classical method for collecting array section information in sequential programs. When applying ADFA to parallel OpenMP programs, array access information needs to be analyzed in loops whose iteration spaces are partitioned across threads. Insufficient symbolic information while analyzing cross-thread relationships of array section expressions and limitations in internal representations often forces ADFAs to perform array section operations (intersection, union and subtraction) conservatively, reducing accuracy.

In order to use ADFAs with OpenMP programs, previous approaches (such as [10, 11]) proposed expressing the bounds of partitioned parallel loops as symbolic functions of the thread number. This allows the compiler to view a multi-threaded program as a serial program with a parameterized thread number. We refer to this method as *explicit static partitioning*.

Explicit static partitioning is reasonably accurate for representing array sections collected within a parallel loop; however, when analyzing array sections collected across multiple parallel loops, data flow computation tend to introduce inaccuracy for the aforementioned reasons.

We introduce the π *operator* [17], an abstract representation of partitioned iteration spaces that captures the partitioning semantics implied by OpenMP directives. The π operator retains the knowledge of original iteration spaces and the partitioning scheme across threads; this information is lost by explicit partitioning, but is relevant for accurate array section operations.

Using the π operator, we present the producer-consumer array data flow analysis (PCDFA) that collects prior produced and upwardly exposed consumed array sections for an OpenMP program's statements. PCDFA is essentially a classical ADFA that takes the memory model semantics of OpenMP into consideration.

In addition, we introduce the concept of *delayed symbolic evaluation*: If a dataflow step would yield a conservative result of an operation, the algorithm postpones evaluating this operation by representing it as an unevaluated expression. Later in the analysis, postponed unevaluated expressions are either simplified or evaluated as further symbolic information becomes available. Should a full compile-time evaluation of some operations not be possible, delayed symbolic analysis simplifies and retains them as unevaluated expressions, allowing them to be accurately evaluated at runtime.

We evaluate the performance of PCDFA by measuring (i) *operation accuracy*, which measures the difference between precise evaluation of PCDFA array section operations and the actual evaluation (which may be approximate and conservative); and (ii) *array section complexity*, which measures the number of terms in array sections' expressions and delayed operations.

The remainder of this chapter is organized as follows: Section 2.1 introduces the π operator. Section 2.2 introduces PCDFA's algorithm. Section 2.3 describes the compiler framework that performs PCDFA. Section 2.4 describes delayed symbolic evaluation. Section 2.5 provides an evaluation of the PCDFA framework.

2.1 The π Operator

The π operator is an abstract representation that captures the high level semantics of partitioning while hiding its implementation. We describe how to use the π operator for both iteration and data spaces.

2.1.1 Iteration Space Representation

Consider the iteration space $(l{:}u{:}s)$, where l, u, and s are, respectively, the lower bound, upper bound and stride expressions. The π operator represents partitioning on this iteration space using the abstract form $\pi_x(l{:}u{:}s)$, where x is the type of par-

Table 2.1: List of π operators. Block-cyclic partitioning currently is supported conservatively as dynamic partitioning.

$\pi_b(l : u : s)$	Divide into chunks of approximately equal size and map to threads in monotonic order (block partitioning)
$\pi_c(l : u : s)$	Map elements to threads in round-robin fashion (cyclic partitioning)
$\pi_m(l : u : s)$	Map all elements to the master thread
$\pi_s(l : u : s)$	Map all elements to a single thread
$\pi_d(l : u : s)$	Mapping is unknown (dynamic partitioning)
$(l : u : s)$	A non-partitioned space (all elements are mapped to every thread)

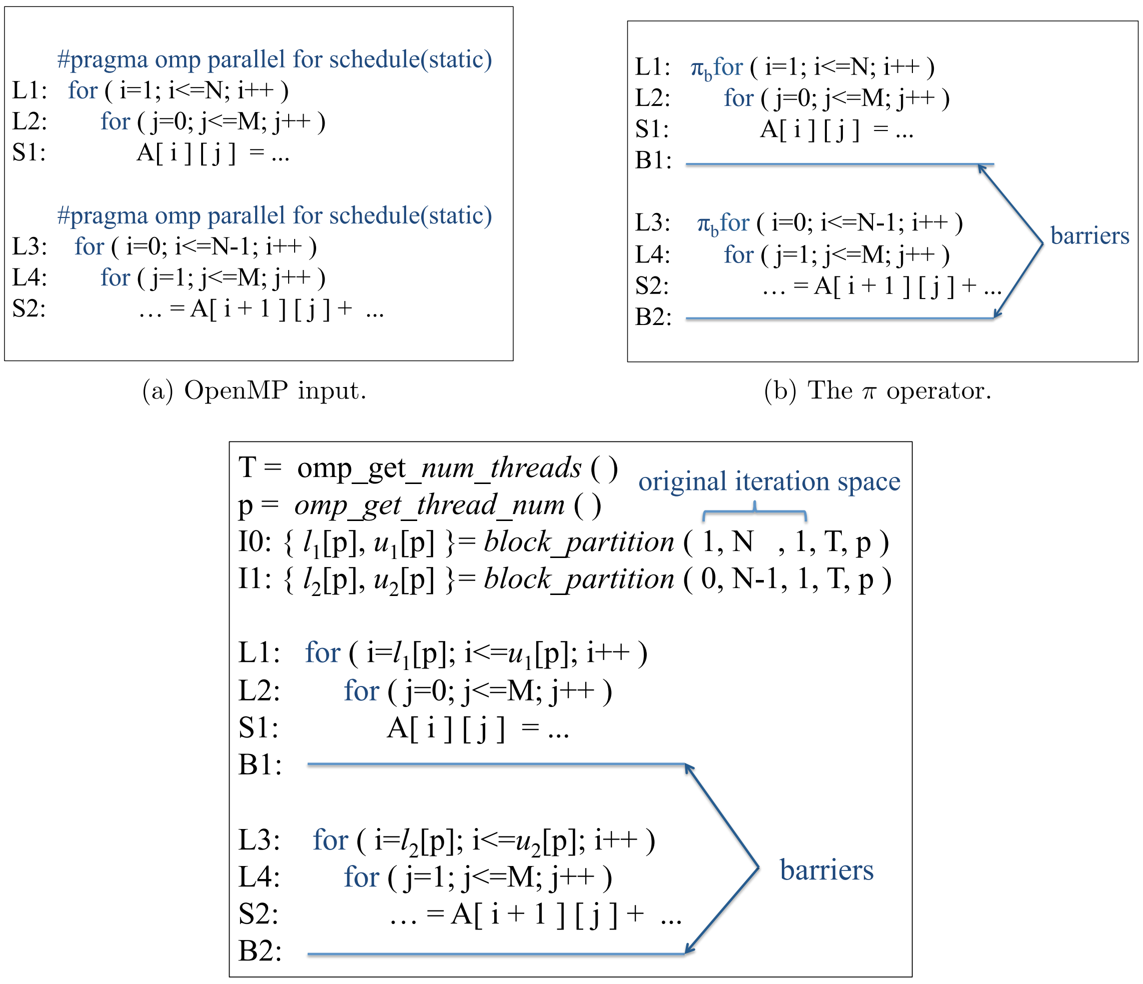
titioning applied to this iteration space as stated or implied by OpenMP directives (see Table 2.1). π operators encapsulate high-level knowledge of partitioning schemes and hide implementation details about how these schemes are actually computed.

The compiler parses OpenMP *schedule* directives of parallel loops and represents their iteration spaces with the appropriate π operator from Table 2.1. π_s and π_m operators represent iteration spaces of loops that are within OpenMP *single* and OpenMP *master* regions, respectively. The π_m operator also represents the iteration space of a sequential loop, since loops that do not correspond to a parallel region execute on the master thread.

Figure 2.1 shows an example of the internal representation by the π operator and state-of-the-art explicit static partitioning.

2.1.2 Data Space Representation

In OpenMP, the mapping of array elements onto threads depends on the partitioning scheme in iteration spaces and the array subscript functions. Hence, a partitioned data space of an array access can be represented using an algebra that applies array subscript functions to π operators.



```
     #pragma omp parallel for schedule(static)
L1:  for ( i=1; i<=N; i++ )
L2:      for ( j=0; j<=M; j++ )
S1:          A[ i ] [ j ]  = ...

     #pragma omp parallel for schedule(static)
L3:  for ( i=0; i<=N-1; i++ )
L4:      for ( j=1; j<=M; j++ )
S2:          ... = A[ i + 1 ] [ j ] +  ...
```

(a) OpenMP input.

```
L1:  π_b for ( i=1; i<=N; i++ )
L2:      for ( j=0; j<=M; j++ )
S1:          A[ i ] [ j ]  = ...
B1:  ________________________________

L3:  π_b for ( i=0; i<=N-1; i++ )
L4:      for ( j=1; j<=M; j++ )
S2:          ... = A[ i + 1 ] [ j ] + ...
B2:  ________________________________
```

(b) The π operator.

```
T =  omp_get_num_threads ( )
p =  omp_get_thread_num ( )
I0: { l1[p], u1[p] }= block_partition ( 1, N   , 1, T, p )
I1: { l2[p], u2[p] }= block_partition ( 0, N-1, 1, T, p )

L1:  for ( i=l1[p]; i<=u1[p]; i++ )
L2:      for ( j=0; j<=M; j++ )
S1:          A[ i ] [ j ]  = ...
B1:  ________________________________

L3:  for ( i=l2[p]; i<=u2[p]; i++ )
L4:      for ( j=1; j<=M; j++ )
S2:          ... = A[ i + 1 ] [ j ] +  ...
B2:  ________________________________
```

(c) Explicit static partitioning.

Fig. 2.1.: The internal representation of an OpenMP program. The explicit static partitioning method explicitly expresses loop partitions; it introduces new complex loop bounds that are parameterized by the thread number. By contrast, the π operator keeps both original iteration spaces and the partitioning semantics.

We first describe the *regular section descriptor (RSD)* [18], a previously introduced array section representation that is accurate for array accesses with linear subscripts. Let $A[f_1][f_2]\ldots[f_m]$ be an m-dimensional array access, where f_j is the subscript expression for dimension j, $1 \leq j \leq m$. Let A be contained in a loop nest with depth n, where the outer most loop has the index variable i_1 and the innermost loop has the index variable i_n. Let array subscript f_j be a linear function of 0 or 1 indices in

$i_1, \ldots, i_n$, i.e., subscripts are not coupled. The same index can appear in more than one dimension. Using RSDs, the array section of A is $(l_1{:}u_1{:}s_1)\ldots(l_m{:}u_m{:}s_m)$, where the bounds in $(l_j{:}u_j{:}s_j)$ are computed from applying array subscript f_j to the bounds of corresponding iteration spaces.

We build on the RSD and introduce the πRSD representation, a simple extension of the RSD representation such that π operators can represent dimensions that have partitioned data spaces. For example, $\pi_b(l_1{:}u_1{:}s_1)(l_2{:}u_2{:}s_2)$ has a block-partitioned data space in the first dimension and a non-partitioned data space in the second dimension.

A partitioned data space for an array access with a linear subscript function is computed by the following algebraic property:

$$a + b \times \pi(l : u : s) = \pi(a + b \times l : a + b \times u : b \times s)$$

. For example, consider the read array access $A[i{+}1][j]$ of statement *S2* in Figure 2.1b. The iteration spaces corresponding to the first and second dimensions are $\pi_b(0{:}N{-}1{:}1)$ and $(1{:}M{:}1)$, respectively. Therefore, the read array section of this access is $\pi_b(1{:}N{:}1)(1{:}M{:}1)$.

A partitioned data space for an array access with a non-linear subscript function (e.g.,$A\ [B\ [j\ ]\ ]$) is conservatively approximated (using overestimation or underestimation). Note that there are no accuracy constraints for linear subscript functions that have non-linear bounds. For example, if $A[1{+}i]$ is being accessed inside a parallel loop with the partitioned iteration space $\pi_b(1{:}B(j){:}1)$, then the partitioned data space of this array access is $\pi_b(2{:}1{+}B(j){:}1)$.

In this work, we build on RSDs to represent partitioned data spaces. However, our new representation (the π operator) requires no alteration to RSDs' expressions or operations. In the implementation, π operators are essentially notations that are attached to array section expressions to describe additional information. This abstraction allows the π operator to be integrated with other array section representations as well.

2.1.3 The π Operator VS Explicit Static Partitioning

Compared to explicit static partitioning, the abstract representation provided by π operators is more concise and enables improved cross-thread analysis of array section expressions. This is because: (i) it hides the complexity of partitioning and keeps expressions simple (functions of original data spaces), and (ii) it provides high-level knowledge about the partitioning semantics.

Consider the written array section in Statement *S1* and the read array section in statement *S2* in the OpenMP code of Figure 2.1a. With explicit static partitioning, the written and the read sections are $(l_1[p]{:}u_1[p]{:}1)(0{:}M{:}1)$ and $(l_2[p]{+}1{:}u_2[p]{+}1{:}1)(1{:}M{:}1)$, respectively (see Figure 2.1c). Because the number of threads and the thread number are unknown at compile time, the cross-thread relationships of the parameterized bounds are also unknown. For example, the result of an intersection operation is unknown. With the π operator, the written and the read sections are $\pi_b(1{:}N{:}1)(0{:}M{:}1)$ and $\pi_b(1{:}N{:}1)(1{:}M{:}1)$, respectively (see Figure 2.1b). Partitioned dimensions in both sections are shown to be the same. For example, the result of an intersection operation is $\pi_b(1{:}N{:}1)(1{:}M{:}1)$. We discuss array section operations in Section 2.2.

In general, explicit static partitioning explicitly represents the complex loop partitions, which tends to lead to inaccurate cross-thread analysis of array section expressions and therefore inaccurate array section operations. The π operator improves this accuracy.

2.2 The Analysis Algorithm

The producer-consumer array data flow analysis (PCDFA) collects prior produced and future consumed array section information by each thread for the statements in an OpenMP program.

$$DEF_{in}(e) = \bigcup_{x \in Pred(e)} DEF_{out}(x)$$

$$DEF_{out}(e) = \begin{cases} \phi & \text{, } e \text{ is barrier node} \\ \Big(DEF_{in}(e) - KILL_{\text{all}}(e)\Big) \bigcup wGEN(e) & \text{, otherwise} \end{cases}$$

Fig. 2.2.: Reaching Definitions analysis. Note that $KILL_{\text{all}}$ is across all threads, all other sets are for the current thread.

$$USE_{out}(e) = \bigcup_{x \in Succ(e)} USE_{in}(x)$$

$$USE_{in}(e) = \Big(USE_{out}(e) - KILL_{\text{all}}(e)\Big) \bigcup rGEN(e)$$

Fig. 2.3.: Liveness analysis. Note that $KILL_{\text{all}}$ is across all threads, all other sets are for the current thread.

We first describe the *Producer-Consumer Flow Graph (PCFG)* [19]. PCFG is a *Control Flow Graph* that represents both the control flow and the relevant memory model semantics of an OpenMP program. In particular, *barrier* nodes, which denote points where memory is to be made coherent across threads, are placed at the end of parallel loops that do not have an OpenMP *nowait* directive. Each node in the PCFG corresponds to a program statement or an OpenMP directive.

For a node e in PCFG: (i) $Succ(e)$ and $Pred(e)$ are the sets of successor and predecessor statements, respectively; (ii) $wGEN(e)$ and $rGEN(e)$ contain the shared array elements that are written and read, respectively, by a thread in e; and (iii) $KILL_{\text{all}}(e)$ contains the aggregated shared array elements that are written by *all threads* in e.

The PCDFA consists of Reaching Definition analysis (Figure 2.2) and Liveness analysis (Figure 2.3). For every node e in PCFG: (i) Reaching Definition analysis computes $DEF_{in}(e)$ and $DEF_{out}(e)$, which are the reaching definitions of shared array elements at the entry and the exit of e, receptively, for a thread; and (ii) Liveness analysis computes $USE_{in}(e)$ and $USE_{out}(e)$, which are the upwardly exposed uses of shared array elements at the entry and the exit of e, respectively, for a thread.

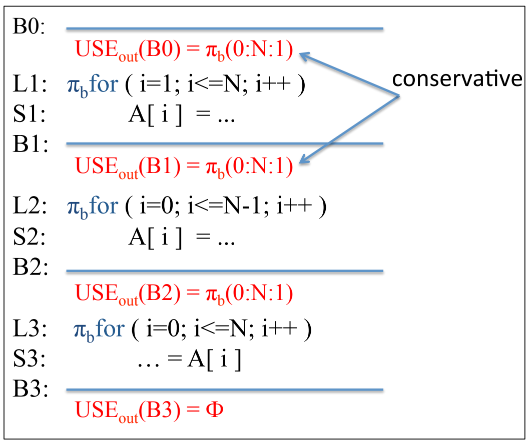
The PCDFA is similar to classical Liveness and Reaching Definition analyses, but takes the coherence semantics of OpenMP's memory model into consideration, as follows: (i) The definitions $DEF_{in}(e)$ and $DEF_{out}(e)$ are the result of writes that have occurred since the *last barrier* (i.e., the most recent produced data since the last global coherent point), and (ii) $KILL_{\text{all}}(e)$ contains the aggregated kills across *all* threads. This is because an element killed in a thread is killed in every thread (their copy of this element becomes invalid).

2.3 The Compiler Framework

We now describe the compiler framework that performs PCDFA while using the π operator for representing partitioned iteration and data spaces. The compiler uses a set of array sections (represented with πRSDs) to represent Gen, Kill, use and definition sets.

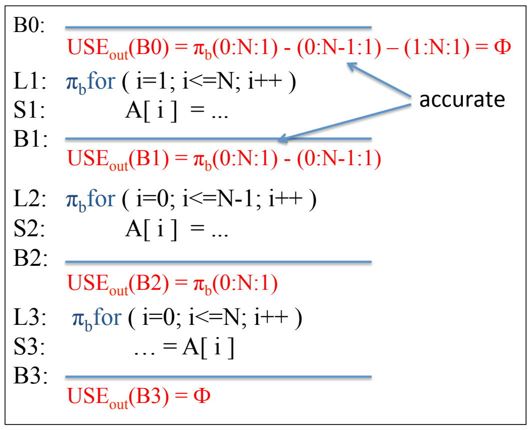
First, the compiler computes Gen (*wGEN* and *rGEN*) and Kill ($KILL_{\text{all}}$) sets for each node in PCFG. Then, the compiler performs PCDFA. During PCDFA's computation, the analysis needs to perform array section operations (as shown in Figure 2.2 and Figure 2.3). We classify operations into two types: *tractable* or *intractable* operations. An operation is tractable if the partitioning semantics of its result can be described using a known partitioning scheme in Table 2.1. Otherwise, the operation is *intractable.*

For example, the intersection operation of $\pi_b(1{:}N{:}1)$ and $(0{:}N{:}1)$ is tractable (can be statically computed) and the result is $\pi_b(1{:}N{:}1)$. This is because all partitions before the intersection operation are the same after performing the operation and therefore can be described using the π_b operator with the same original data space. On the other hand, the intersection operation of $\pi_b(1{:}N{:}1)$ and $(2{:}N{:}1)$ is intractable.



```
B0:
      USE_out(B0) = π_b(0:N:1)
L1:  π_bfor ( i=1; i<=N; i++ )
S1:         A[ i ] = ...
B1:
      USE_out(B1) = π_b(0:N:1)
L2:  π_bfor ( i=0; i<=N-1; i++ )
S2:         A[ i ] = ...
B2:
      USE_out(B2) = π_b(0:N:1)
L3:  π_bfor ( i=0; i<=N; i++ )
S3:         ... = A[ i ]
B3:
      USE_out(B3) = Φ
```

(a) Without delayed symbolic evaluation.



```
B0:
      USE_out(B0) = π_b(0:N:1) - (0:N-1:1) – (1:N:1) = Φ
L1:  π_bfor ( i=1; i<=N; i++ )
S1:         A[ i ] = ...
B1:
      USE_out(B1) = π_b(0:N:1) - (0:N-1:1)
L2:  π_bfor ( i=0; i<=N-1; i++ )
S2:         A[ i ] = ...
B2:
      USE_out(B2) = π_b(0:N:1)
L3:  π_bfor ( i=0; i<=N; i++ )
S3:         ... = A[ i ]
B3:
      USE_out(B3) = Φ
```

(b) With delayed symbolic evaluation.

Fig. 2.4.: The result of PCDFA at barrier nodes. Delayed Symbolic evaluation avoids conservative approximation of the intractable subtract operation at barrier *B1*.

This is because there is at least one partition (which has the element 1) that is changed while performing the intersection operation.

The algorithms for performing union, intersection and subtraction operations for πRSD sections are as described for RSD sections [18]. However, their results are kept only if they are tractable. Intractable operations would lead to approximation. Section 2.4 provides an accurate solution for intractable operations.

As discussed earlier, we build on RSDs to represent partitioned data spaces. π operators require no alteration to RSDs' expressions or operations. In the implementation, π operators are notations that are attached to array section expressions to abstractly describe partitioning semantics.

2.4 Delayed Symbolic Evaluation

We introduce delayed symbolic evaluation to improve the accuracy of PCDFA's computation in the presence of intractable operations. At a dataflow step that has an intractable operation, the analysis delays this operation to later dataflow steps by representing it as an unevaluated expression. The key observation is that the unevaluated expressions at later dataflow steps can be simplified (i.e., do not grow in complexity) because additional symbolic information becomes available.

Subtraction operations are most critical in this context. To represent the unevaluated expression of a subtraction operation, we extend the πRSD representation and introduce the ERSD representation, as follows: ERSD = $\pi\mathrm{RSD}_1 - \pi\mathrm{RSD}_2 - \ldots - \pi\mathrm{RSD}_n$, where n is the number of terms. During PCDFA's computation, operations with ERSD sections get performed or simplified using the mathematical rules of set theory.

The unevaluated expression of an intractable union operation is a set of two sections. In PCDFA, unevaluated intersections operations are not needed. However, in general, the unevaluated expression of an intersection operation can also be represented as a set of sections, i.e., a set can have an implicit union or intersection operation based on the dataflow equations of the ADFA.

The ERSD representation retains subtracted sections that would otherwise be lost by conservative approximation. In doing so, ERSDs (i) provide additional symbolic information at later computation steps, which allow simplification, and (ii) allow PCDFA to terminate with simplified and unevaluated but accurate expressions of intractable operations. Expressions that remain unevaluated when PCDFA terminates will be evaluated at runtime with minimal cost.

Consider PCDFA's computation result for the OpenMP example code in Figure 2.4. Without delayed symblic evaluation, intractable subtract operations get approximated. This yields inaccurate array sections at barriers *B0* and *B1*, as shown in Figure 2.4a. In Figure 2.4b, delayed symbolic evaluation yields accurate array sections because: (i) the intractable subtract operation at barrier *B1* is postponed, and (ii) the symbolic information at barrier *B0* is sufficient to perform the postponed subtraction operation from *B1* accurately.

An important property of PCDFA is that subtracted terms ($\pi\mathrm{RSD}_2$, …, $\pi\mathrm{RSD}_n$) in ERSD expressions are $\mathit{KILL}_{\mathrm{all}}$ sets (see Figure 2.3 and Figure 2.2). These sets contain non-partitioned RSD sections that the analysis, in practice, can merge (i.e., minimize) into a small number of terms (no more than 3 terms in tested benchmarks).

As a concept, delayed symbolic evaluation is orthogonal to the π operator and can be applied with other representations such as explicit static partitioning. However, the additional complexity of doing so is significantly higher than in the case of π operators, as will be shown in our performance evaluation.

2.5 Performance Evaluation

We evaluate the performance of PCDFA using both representations, the π operator and state-of-the-art explicit static partitioning. We also evaluate the impact of using delayed symbolic evaluation on each representation.

2.5.1 Performance Metrics

We evaluate using two metrics: *operation accuracy* and *array section complexity.* Operation accuracy describes the difference between precise evaluation of PCDFA array section operations and the actual evaluation (which may be approximate and conservative). Array section complexity measures the number of terms in array sections' expressions, as well as the number of delayed subtract operations.

To measure PCDFA's operation accuracy, we compute the volume of the overlap between prior produced and future consumed (DEF_{in} and USE_{out}) at barriers (i.e., communication volume). The operation accuracy is then given as a percentage of the ideal volume to this volume. The ideal volume is obtained by a separate runtime computation of PCDFA, where all operations are kept precise due to available full knowledge about cross-thread relationships at runtime. We choose communication volume because its accuracy directly impacts the performance of several optimizations such as barrier elimination [20] and OpenMP-to-MPI translation [10].

A 100% operation accuracy means that no conservative PCDFA operations were performed. Note that operation accuracy does not account for the inaccuracy that results from approximating indirect memory accesses or other non-linear subscripts that cannot be represented accurately by RSDs.

We measure array section complexity by counting the number of array section terms and delayed subtraction operations. This metric represents the work (i.e., the overhead) needed to evaluate array section expressions at runtime.

2.5.2 Experimental Setup

We implemented the PCDFA compiler frameworks, including π operators and explicit static partitioning, in the Cetus Compiler Infrastructure [21]. We also implemented a runtime tool that receives produced and consumed array sections from the compiler and determines operation accuracy and array section complexity. Figure 2.5 shows an example of the function calls used by the compiler to pass produced and consumed array sections to the runtime tool.

We evaluate using four OpenMP benchmarks taken from the NAS Parallel Benchmarks suite [22]: FT, SP, BT and SP. All functions were automatically inlined by the Cetus Compiler. FT benchmark was optimized using the *owner alignment* technique presented in [11], which was applied before performing PCDFA with all FT experiments.

2.5.3 Evaluation of the Producer-Consumer Array Data Flow Analysis Accuracy

Figure 2.6 shows the operation accuracy and the array section complexity for BT, SP, CG and FT. Without delayed symbolic evaluation, on average, the π operator and explicit static partitioning have operation accuracy of 76% and 46%, respectively. In addition, the π operator reduces array section complexity by 33%, compared to explicit static partitioning. When applying delayed symbolic evaluation, both representations have 100% operation accuracy, while increasing array section complexity by 1.96x with explicit static partitioning and by no more than 1.1x with the π operator.

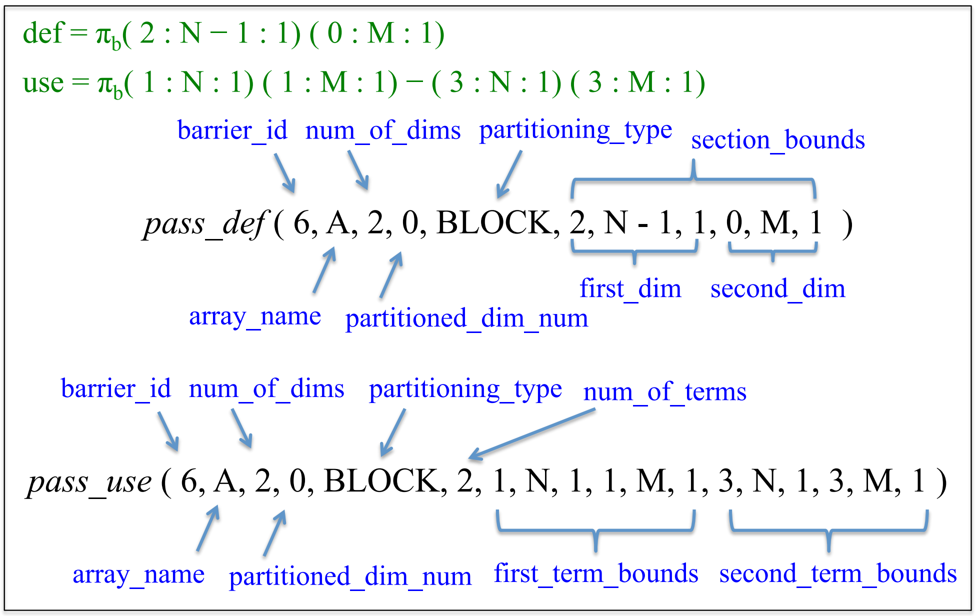
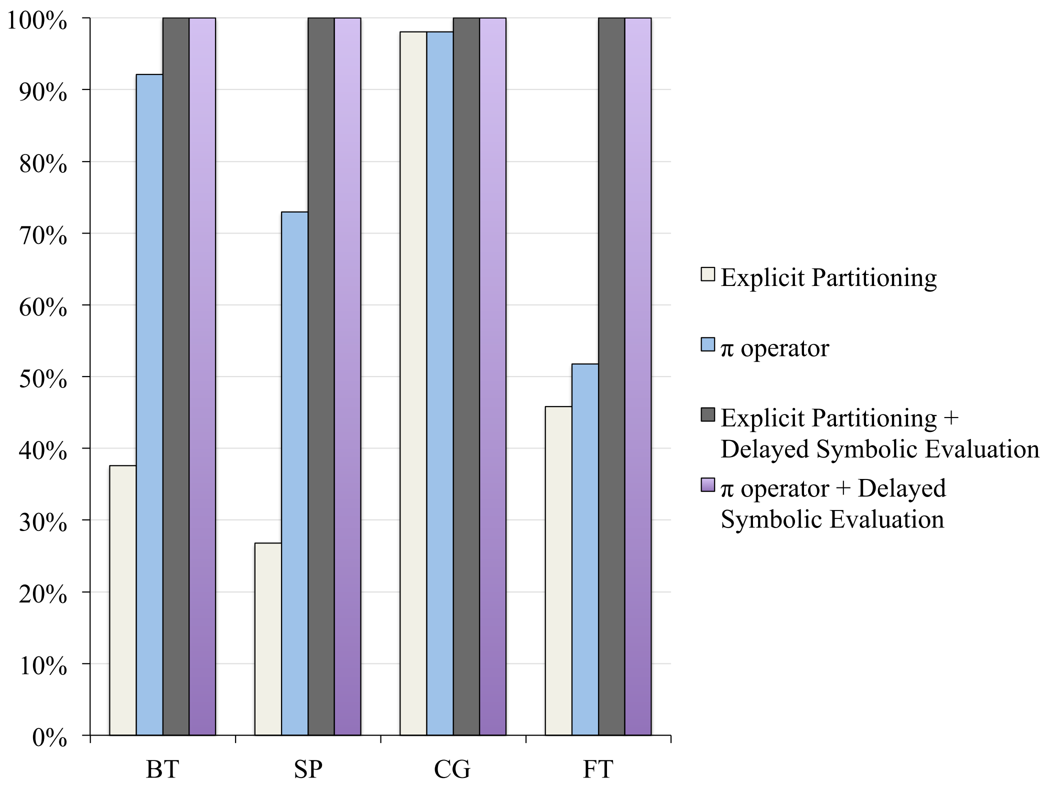

def = π_b(2 : N − 1 : 1) (0 : M : 1)
use = π_b(1 : N : 1) (1 : M : 1) − (3 : N : 1) (3 : M : 1)
barrier_id
num_of_dims
partitioning_type
section_bounds
pass_def (6, A, 2, 0, BLOCK, 2, N - 1, 1, 0, M, 1)
first_dim
second_dim
array_name
partitioned_dim_num
barrier_id
num_of_dims
partitioning_type
num_of_terms
pass_use (6, A, 2, 0, BLOCK, 2, 1, N, 1, 1, M, 1, 3, N, 1, 3, M, 1)
array_name
partitioned_dim_num
first_term_bounds
second_term_bounds

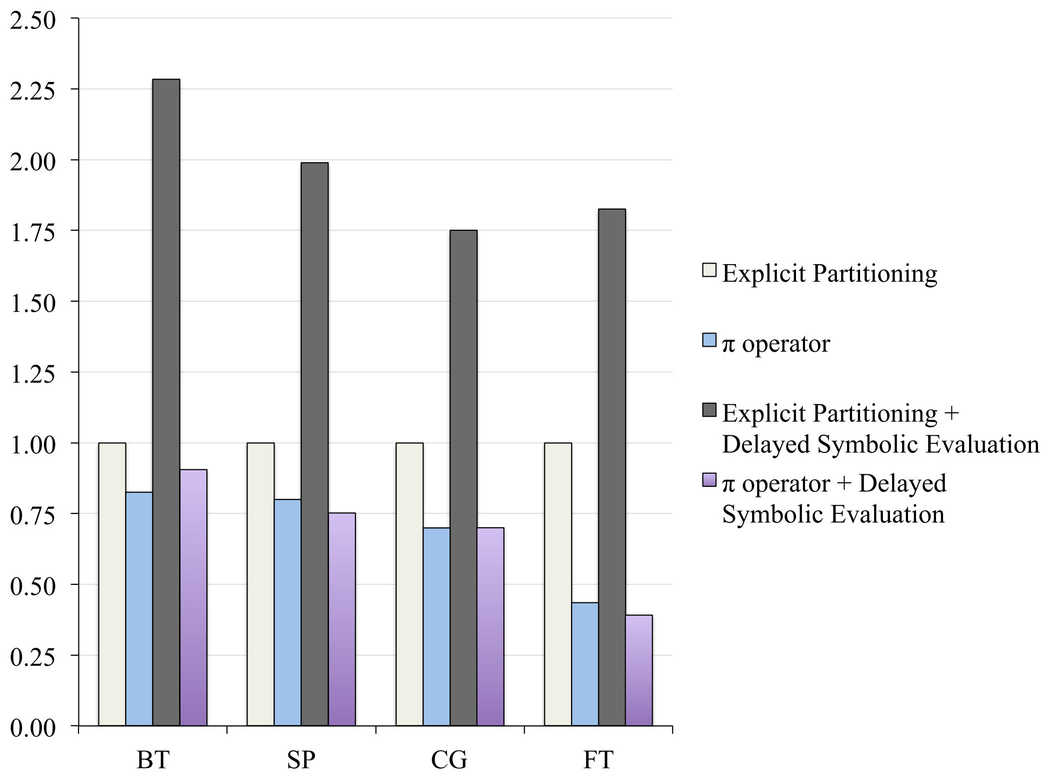
Fig. 2.5.: An example of the function calls used by the π operator's compiler to pass produced and consumed array sections to the runtime tool for a particular barrier. Function calls only specify one partitioned dimension because all tested benchmarks have one-dimensional parallelism (no nested parallelism). In the case of explicit static partitioning, the same function calls are used except that there are no *partitioned_dim_num* or *partitioning_type* fields. The total number of function calls is equal to the total number of array sections.

Explicit static partitioning has insufficient knowledge of cross-thread relationships during PCDFA's computation. Without delayed symbolic evaluation, this causes a large number of conservative subtraction operations and explains the inferior operation accuracy of explicit static partitioning compared to the π operator. This holds for all benchmarks except for CG. CG is a case where subtract operations are less frequent and can be performed accurately with explicit static partitioning. Our solution obtains the same accuracy for CG but reduces array section complexity to 70%.

Figure 2.7 shows the number of generated array sections by PCDFA for the π operator and explicit static partitioning. It also shows the impact of delayed symbolic evaluation on both representations. For two benchmarks (FT and SP), using delayed symbolic evaluation with the π operator reduced the total number of array



(a) Operation accuracy (higher is better). 100% means all operations are performed accurately.



(b) IR complexity (number of terms and delayed operations) normalized to IR complexity obtained with explicit static partitioning (lower is better).

Fig. 2.6.: Operation accuracy and IR complexity of PCDFA averaged over 8, 16, 32 and 64 threads.

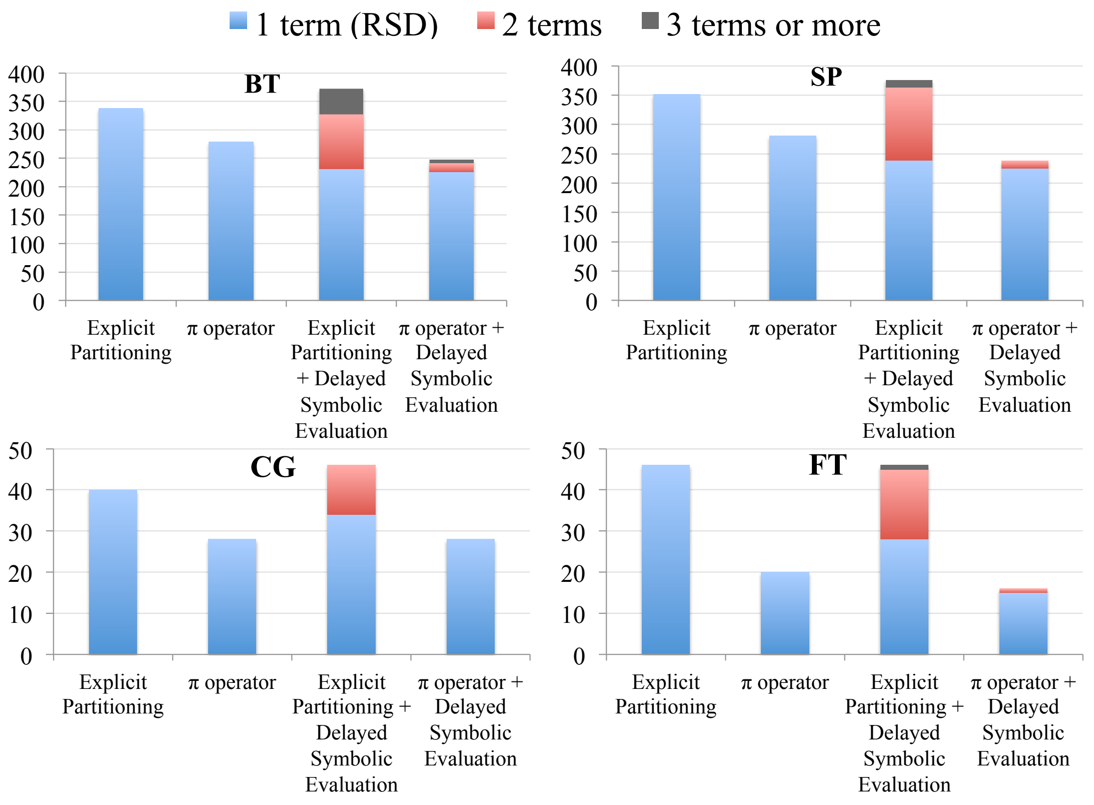


Fig. 2.7.: The number of array sections generated by PCDFA, categorized into RSDs (which have 1 term) and ERSDs (which can have 2 terms or more). Without delayed symbolic evaluation, π operators lead to 33% fewer array sections, on average, than explicit static partitioning. When applying delayed symbolic evaluation, the ratio of generated ERSDs (delayed subtract operations) is below 8% with the π operator and is 26%−39% with explicit static partitioning. The largest number of terms in an ERSD section is 3 with π operators and 5 with explicit static partitioning.

sections. This is due to the additional symbolic knowledge provided by delayed symbolic evaluation during PCDFA's computation, which increases accuracy and reduces the number of array sections. With explicit static partitioning, delayed symbolic evaluation has high complexity because of the insufficient symbolic information during PCDFA's computation (i.e., large number of conservative subtract operations were delayed).

In general, the π operator is a better representation than explicit static partitioning. In addition to being more concise, the π operator improves operation accuracy and reduces the complexity of computed array sections. Delayed symbolic evaluation is a useful technique for both representations that can eliminate conservative operations. Its complexity, however, is dependent on the representation.

The π operator and delayed symbolic evaluation provide general concepts that can be extended to other parallel programs, beyond OpenMP. To use the π operator, user directives or compiler analyses are needed to retrieve high level information about the partitioning applied to parallel loops. In addition, the abstraction allow π operators to be implemented as annotations that describe parallel semantics without the need to alter the implementation of array section representations or operations by the compiler. To use delayed symbolic evaluation, the compiler needs to represent unevaluated expressions of delayed operations such that they can be simplified along the dataflow computation.

3. EXTENDING THE PRODUCER-CONSUMER ARRAY DATA FLOW ANALYSIS TO NON-REPETITIVE OPENMP

An analysis that determines the array elements accessed by a loop or an instance of a loop is a classical problem in optimizing compilers targeting both sequential and parallel programs. *Array Data Flow Analysis* (ADFA) is a traditional compiler analysis that has been heavily used for analyzing accessed array elements in sequential programs, and also have been extended to parallel programs. When analyzing an instance of a loop (or loop nest), current ADFAs develop dataflow information that is *invariant* across all instances of the loop. In programs where different instances of the loop read and write different array elements, the dataflow analysis result, while valid for all instances, will be conservative for any given instance.

Common cases where the aforementioned weakness of ADFA occurs are in triangular loop nests or, more generally, where the indices of enclosing loops appear in array accesses of inner loops. For an example, consider the loop nest in Figure 3.1. Let $L1$ be the loop where the compiler performs optimizations while taking into consideration accessed array elements in *multiple* instances. Traditional ADFAs will force Gen and Kill sets of $L1$ to be invariant and therefore approximate them over all iterations of $L0$. This leads ADFAs to generate conservative result.

In this context, we build on traditional ADFAs and introduce the producer-consumer array data flow analysis (PCDFA) for analyzing OpenMP programs. Similar to traditional ADFAs, PCDFA generates conservative array section information in non-repetitive OpenMP programs [1]. This can preclude the performance of our trans-

[1]An OpenMP program is non-repetitive if it has at least one parallel loop with variant Gen or Kill sets, otherwise, the program is repetitive

```
L0: for ( i = 1; i <= N; i++) {
L1:       for ( k = i ; k <= N; k++ )
S1:            x [ k ] =  x [ k ] + …
      }
```

Fig. 3.1.: Due to the triangular access pattern, the accessed array elements for individual instances of the inner nested loop $L1$ (i.e., Gen and Kill sets) vary for different iterations of the enclosing loop $L0$.

lation scheme because the accuracy of PCDFA's result impacts the communication volume.

In this Chapter, we present algorithms that overcome this weakness in current ADFAs, and in doing so, improve the accuracy of the information obtained. Our solution retains the generality of ADFA frameworks but most beneficial for the aforementioned cases.

We divide our solution into two parts. Let an enclosing loop be a loop that has inner loops with variant Gen and Kill sets. The first part is the *variant-set* analysis, which aims to find *accurate* (i.e., not approximate) symbolic expressions of dataflow solutions for each statement *instance* within *enclosing loops* (we discuss an exception in the second part of the solution). By properly representing variant Gen and Kill sets, dataflow operations can be extended to apply while reasoning about different instances. A key insight is that a *pattern* often holds for variant dataflow sets that allows the desired representation of dataflow solutions to be computed using a bounded number of instances.

In some cases, conservative effects from statements outside enclosing loops may cause variant-set analysis to develop conservative information. Figure 3.2 shows an example of such a case. To overcome this, the second part of our solution is the *region-based* analysis, where enclosing loops in the program form *regions* that get analyzed

```
L0:  for( i = 1 ; i <= N ; i++) {
L1:      for ( k = i ; k <= N; k++ )
S1:          x [ k ] =  x [ k ] +  …
     }
     /* verification code */
L2:  for ( k = 1; k <= N; k++ )
S2:      verify_value ( x[ k ] ) ;
```

Fig. 3.2.: Consider a parallelization transformation where explicit communication is generated. The access pattern for consumers inside the enclosing loop $L0$ is triangular, and for consumers outside the loop is rectangular. Combining use information for these patterns may lead individual instances of the loop $L1$ (where data is produced) to generate excessive communication.

in isolation using variant-set analysis. Next, each enclosing loop is collapsed into a single node that represents all iterations, and the combined dataflow solution of all statement instances within the loop will be used when analyzing the whole program.

By contrast to the classical analysis, the region-based analysis computes dataflow solutions within an enclosing loop for the effects in this loop only while the combined effect of all iterations is used for dataflow solutions outside the loop. By doing so, both dataflow solutions inside and outside enclosing loops are accurately determined.

The dataflow solutions produced by region-based analysis can be beneficial for many compiler transformations. For example, communication inside the enclosing loop $L0$ in Figure 3.2 can be generated for producers and consumers inside the loop, while communication with the outside consumers can be deferred to the last iteration. In general, region-based analysis cannot be applied when compiler transformations require dataflow solutions within enclosing loops to account for the full effect of the program.

Our solution can be applied to traditional ADFAs with sequential programs, and extended to ADFAs with parallel programs. In this dissertation, we demonstrate the

$$USE_{out}(e) = \bigcup_{x \in Succ(e)} USE_{in}(x)$$

$$USE_{in}(e) = \Big(USE_{out}(e) - KILL(e)\Big) \bigcup GEN(e)$$

Fig. 3.3.: Liveness analysis.

utility of our compiler analyses into improving the accuracy of PCDFA and reducing communication while translating non-repetitive OpenMP programs. By doing so, our translation scheme efficiently scales non-repetitive OpenMP programs to clusters. In contrast, prior translation schemes [10–13] only covered the easier-to-analyze repetitive OpenMP programs (where Gen and Kill sets are invariant).

The remainder of this chapter is organized as follows: Section 3.1 and Section 3.2 present variant-set and region-based analyses, respectively. Section 3.3 extends these analyses to PCDFA. Section 3.4 evaluates the accuracy impact of using variant-set and region-based analyses for six non-repetitive and four repetitive OpenMP benchmarks.

3.1 Variant-Set Analysis

We present the variant-set analysis, which performs ADFAs while allowing Gen and Kill sets that change from iteration to iteration of the enclosing loops.

3.1.1 A Running Example

We use classical Liveness analysis as an example to illustrate our solution. Liveness analysis determines the upward exposed uses of array elements for each statement in a sequential program (see Figure 3.3). Figure 3.4 shows the corresponding control flow graph (CFG). Applying classical analysis to the non-repetitive code example in Figure 3.1 would result in conservative Use sets information, as shown in Figure 3.5.

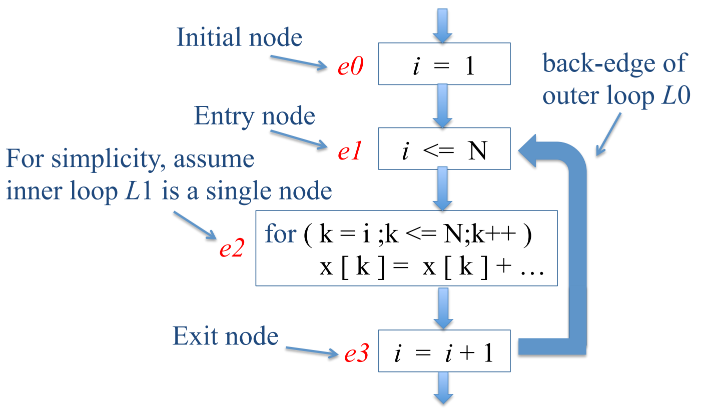

Initial node
e0
i = 1
back-edge of outer loop L0
Entry node
e1
i <= N
For simplicity, assume inner loop L1 is a single node
e2
for (k = i ;k <= N;k++)
x [k] = x [k] + …
Exit node
e3
i = i + 1

Fig. 3.4.: The *control Flow Graph* (CFG) representation for the example in Figure 3.1.

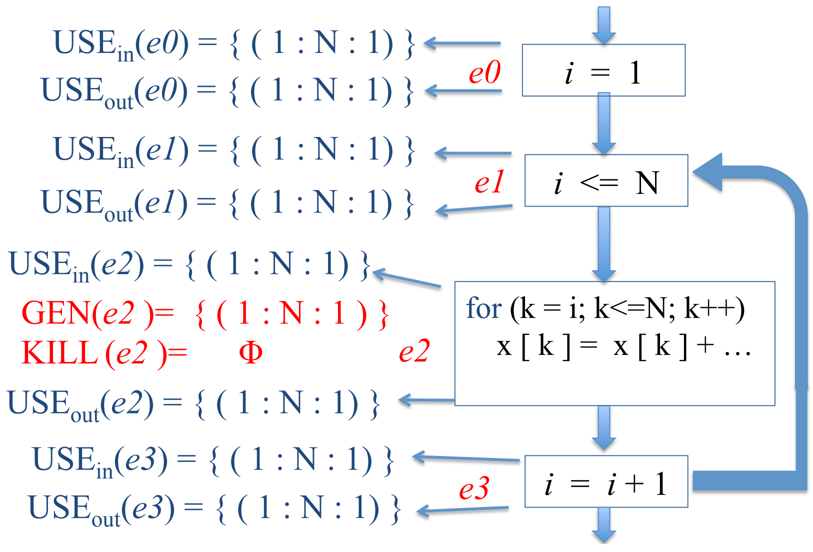

USEin(e0) = { (1 : N : 1) }
USEout(e0) = { (1 : N : 1) }
e0
i = 1
USEin(e1) = { (1 : N : 1) }
USEout(e1) = { (1 : N : 1) }
e1
i <= N
USEin(e2) = { (1 : N : 1) }
GEN(e2)= { (1 : N : 1) }
KILL (e2)= Φ
e2
for (k = i; k<=N; k++)
x [k] = x [k] + …
USEout(e2) = { (1 : N : 1) }
USEin(e3) = { (1 : N : 1) }
USEout(e3) = { (1 : N : 1) }
e3
i = i + 1

Fig. 3.5.: Classical Liveness analysis applied to the example in Figure 3.1. In order to make $GEN(e2)$ and $KILL(e2)$ invariant, they are approximated over all instances of $e2$ (gens get overestimated and kills get underestimated).

3.1.2 Approach

Our goal is to find dataflow solutions that are accurate by using variant Gen and Kill sets. To do so, the key challenges that we address are (i) Representing Gen and Kill sets accurately for each statement instance (i.e., each iteration in the CFG cycle); and (ii) Extending dataflow operations so that they can operate on dataflow sets that involve different instances. We address these challenges using two key insights: (i) Individual instances of Gen and Kill sets can often be represented using expressions that are functions of enclosing loop indices; and (ii) The algebraic relationship of iterations (which form a sequence) can be exploited to perform operations on dataflow set expressions that contain different instances.

Our solution is enabled by a pattern that we have found to be common: dataflow solutions are functions of Gen and Kill sets for only a *bounded* number of instances. Using this *fixed dataflow* pattern, our solution traverses a bounded number of iterations in the CFG cycle representing the enclosing loop and finds dataflow solutions that are valid for *every* instance.

3.1.3 The Analysis Algorithm

Figure 3.6 describes the dataflow computation of a backward ADFA performed using variant-set analysis (forward ADFAs are analogous). The statements within the enclosing loop L may have variant Gen and Kill sets. During the dataflow computation, the iterations of each enclosing loop L in the CFG are examined; traversing the back-edge of L indicates that a *new* iteration is now being examined. In this case, a special function S is invoked to enable dataflow operations to operate on dataflow sets from *previously* examined iterations. Below, we explain the dataflow set representation, the function S, and the convergence scheme of variant-set analysis in more detail. For the presentation, we assume the CFG is traversed backwards starting from the program exit node.

Dataflow sets use *regular section descriptors* (RSDs) [18] to represent array sections. Given a multi-dimensional array access, each dimension is represented using symbolic lower, upper and stride expressions that can be variant (often functions of the enclosing loop index) and either linear or non-linear. Our solution is accurate for linear expressions and handles non-linear expressions conservatively. In addition, our solution allows symbolic reasoning to be performed for each dimension in RSDs

$$OUT(e\ ,\ i) = \bigcup_{x \in Succ(e)} IN^{*}(x\ ,\ i)\ ,\ where:$$

$$IN^{*}(x\ ,\ i) = \begin{cases} S\big(IN(x,\ i)\ ,\ L\big)\ , & \text{e} \rightarrow \text{ x is a back-edge of an enclosing loop } L \\ IN(x\ ,\ i)\ , & \text{otherwise} \end{cases}$$

$$IN(e\ ,\ i) = \Big(OUT(e\ ,\ i) - KILL(e\ ,\ i)\Big) \bigcup GEN(e\ ,\ i)$$

Fig. 3.6.: A backward dataflow analysis performed using variant-set analysis. i is the index of the enclosing loop L being currently examined. When traversing the back-edge of L, the function S (shown in Figure 3.7) is invoked.

independently. This can be useful where the analysis is conservative for dimensions with non-linear bounds and accurate for dimensions with linear bounds, achieving an accurate overall analysis (e.g., two array sections can be shown to be independent by proving independence for one dimension). This dissertation uses RSDs; however, other array section representations can also be used with variant-set analysis.

The function S is invoked when a new iteration is being examined. Consider the two array sections $\text{RSD}_1(i')$ and $\text{RSD}_2(i)$, where i' and i are the previously and currently examined iterations, respectively. Using the algebraic relationship of iterations, function S expresses i' as a function of i. By doing so, dataflow operations on $\text{RSD}_1(i')$ and $\text{RSD}_2(i)$ can be performed in the usual way.

As shown by Figure 3.7, function S expresses each array section in dataflow sets from the previously examined iteration as expressions of the current iteration. A special case is when a dependency test (described in Figure 3.8) proves that a particular array section from the previously examined iteration has array elements that are independent from the array elements of Kill sets within the enclosing loop L. In this case, dataflow sets for *all* previously examined iterations will be fully exposed to the new examined iteration. In cases where the dependency test cannot prove or disprove dependence (result is unknown), function S produces a conservative result.

Convergence occurs, as in classical ADFA, when a fix-point is reached. The only difference is that our solution has variant dataflow sets in the enclosing loop L and thus symbolic expressions need to be compared.

For an enclosing loop L, a fix-point is reached when the aforementioned fixed dataflow pattern holds. In this pattern, the array elements of Gen sets in a previously examined iteration $i2$ are *fully* covered by the array elements of Kill sets in a later examined iteration $i1$ and the distance d between $i1$ and $i2$ is constant. In this case, reaching a fix-point is bounded because dataflow sets are functions of Gen and Kill sets for the iterations $i1$, $i1 + s$, $\ldots$, $i2$ only, where s is the stride of L. An enclosing loop can have multiple fixed dataflow patterns (see Figure 3.9). Array elements in Gen sets that are independent from array elements in Kill sets within the enclosing loop L are handled as a special case (as shown in Figure 3.7).

In Figure 3.1, the enclosing loop has a fixed dataflow pattern because consumed array elements in each iteration $i + 1$ (Gen sets) are fully covered by iteration i (Kill sets). Fixed dataflow patterns are often found in the algorithms of numerical applications. All of our tested benchmarks have these patterns except for one program that we included specifically to exercise this corner case.

Figure 3.10 performs Liveness analysis using variant-set analysis for the example in Figure 3.1. After the analysis reaches the initial node $e0$, Use sets are no longer variant within $L0$.

3.1.4 Implementation

The function S performs the dependency test given by Figure 3.8. This test represents a classical dependency test found in automatic parallelizers. In our implementation, we use an existing stand-alone Satisfiability Modulo Theories (SMT)

```
// $L$ is an enclosing loop and $i$ is the iteration being currently examined
// IN ($L$_entry , $i$ ) is the data flow set from the previously examined iteration
$S$ ( IN ($L$_entry , $i$ ) , $L$) {
    ( $l : u : s$ ) = (lower bound: upper bound: stride) of $L$'s iteration space
    $KILL_{\text{iteration}}$ ( $i$ ) = kills of all statements in the iteration $i$
    $DFS_{\text{substituted}} = \Phi$
    **for each** section $sec(i)$ in IN ($L$_entry , $i$ ) and $sec(i)$ is variant **do**
        **if** dependency_test ( $sec(i)$, $KILL_{\text{iteration}}$ ( i ), $L$ ) returns **true do**
            $T = sec($ substitute $i$ by $i + s)$ // $i + s$ is previously examined iteration
        **else** // $sec$ is independent -- special case
            $all\_previous\_iterations = \{ i + s, i + 2 \times s, i + 3 \times s, \dots , u \}$
            $T = \bigcup_{j \in all\_previous\_iterations} sec$ ( substitute $i$ by $j$ )
        **end if**
        add $T$ to $USE_{\text{substituted}}$
    **end for**
    **return** $USE_{\text{substituted}}$
}
```

Fig. 3.7.: The function S for backward dataflow analyses (where previously examined iterations are later executed iterations). The dependency test is shown in Figure 3.8.

solver [23] to perform the dependency test. The analysis sends queries to the SMT solver with the dependency test parameters.

The SMT solver [23] can also compute integer solutions. In Figure 3.8, d represents the dependency distance between iterations $i1$ and $i2$. Using the distance d, the SMT solver finds, for a given array section $sec(i)$ in Figure 3.7, the distance to the next iteration with dependent Kill sets. Therefore, variant-set analysis can skip examining the $d{-}1$ iterations in between (which have independent Kill sets) and instead account for them when performing the substitution, which now becomes $sec(i+s) \cup \cdots \cup sec(i+d \times s)$.

With our implementation, fix-points are reached in our tested benchmarks with no more than three examined iterations, except for one case. In that case, reaching a fix-point can be unbounded because a fixed dataflow pattern is not present. When reaching a user-specified number of iterations (a threshold), variant-set analy-

```
// L is an enclosing loop
// i is the loop index of L
// sec1( i ) and sec2( i ) are array sections
dependency_test ( sec2( i ) , sec1( i ) , L ) {
    Let d, p , q be integers
    set  i1  = i
    set  i2  = i + d x s
    set  d  ≥  1          // i2 executes after i1 and i2 ≠ i1
    set  d  <   iteration count of L
    set  q  ∈  sec1 ( substitute i by i1 )
    set  p  ∈  sec2 ( substitute i by i2 )
    set  p  =  q
    if  the system has an integer solution do
        return true       // dependent
    else
        return false      // independent
}
```

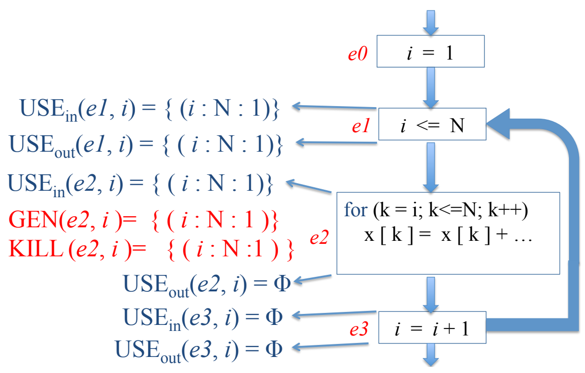
Fig. 3.8.: The dependency test solves a system of equations and constraints to proves/disproves dependence between two array sections sec1 and sec2 in an enclosing loop L such that sec1 occurs before sec2.

```
L0: for ( i = 1; i <= N; i++) {
L1:     for ( k = 1 ; k <= N; k++ )
S1:         … =  x [ i - 2] [ k ] + x [ i - 4] [ k ]
L2:     for ( k = 1 ; k <= N; k++ )
S2:         x [ i ][ k ] =   …
    }
```

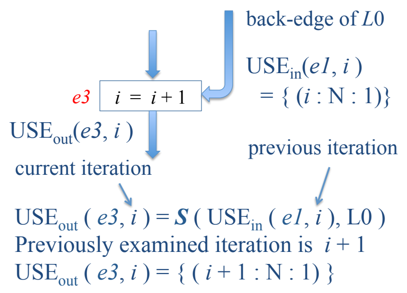
Fig. 3.9.: Assuming a backward analysis where reads are Gens and writes are Kills, the enclosing loop $L0$ has two fixed dataflow patterns with distances 2 and 4.

sis restarts while approximating variant Gen and Kill sets, obtaining the same result as the classical dataflow analysis.

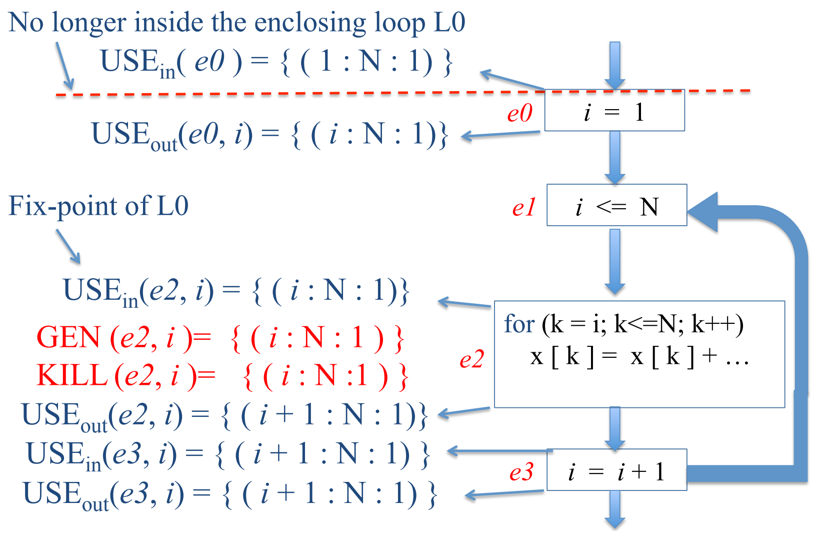
The function S in Figure 3.7 also has a special case where a union operation may be performed for an unbounded number of array sections. In some cases, it


e0
i = 1
USE_in(e1, i) = { (i : N : 1)}
USE_out(e1, i) = { (i : N : 1)}
e1
i <= N
USE_in(e2, i) = { (i : N : 1)}
GEN(e2, i)= { (i : N : 1)}
KILL (e2, i)= { (i : N :1) }
e2
for (k = i; k<=N; k++)
x [k] = x [k] + …
USE_out(e2, i) = Φ
USE_in(e3, i) = Φ
USE_out(e3, i) = Φ
e3
i = i + 1

(a) The first examined iteration.


back-edge of L0
USE_in(e1, i)
= { (i : N : 1)}
e3
i = i + 1
USE_out(e3, i)
current iteration
previous iteration
USE_out (e3, i) = S (USE_in (e1, i), L0)
Previously examined iteration is i + 1
USE_out (e3, i) = { (i + 1 : N : 1) }

(b) The substitution function S is invoked when traversing the back-edge of $L0$.


No longer inside the enclosing loop L0
USE_in(e0) = { (1 : N : 1) }
e0
i = 1
USE_out(e0, i) = { (i : N : 1)}
Fix-point of L0
e1
i <= N
USE_in(e2, i) = { (i : N : 1)}
GEN (e2, i)= { (i : N : 1) }
KILL (e2, i)= { (i : N :1) }
e2
for (k = i; k<=N; k++)
x [k] = x [k] + …
USE_out(e2, i) = { (i + 1 : N : 1)}
USE_in(e3, i) = { (i + 1 : N : 1) }
USE_out(e3, i) = { (i + 1 : N : 1) }
e3
i = i + 1

(c) The second examined iteration.

Fig. 3.10.: Liveness analysis using variant-set analysis performed for the example in Figure 3.1, which has a fixed dataflow pattern of distance 1. Fix-point of $L0$ occurs at $e2$, which has the same $USE_{in}(e2,\ i)$ in the second and first examined iterations.

is not possible to simplify this union without approximation. To retain accuracy, we build on the RSD representation and present the *Range Section* representation $\bigcup_{j \in rng}$RSD(j), which represents the union of multiple sections whose symbolic expressions are represented by the single array section RSD(j) and j iterates over the range $rng = (l : u : s)$.

During dataflow computation, operations with Range Sections get performed using of the rules of set theory. Additionally, our implementation takes advantage of the

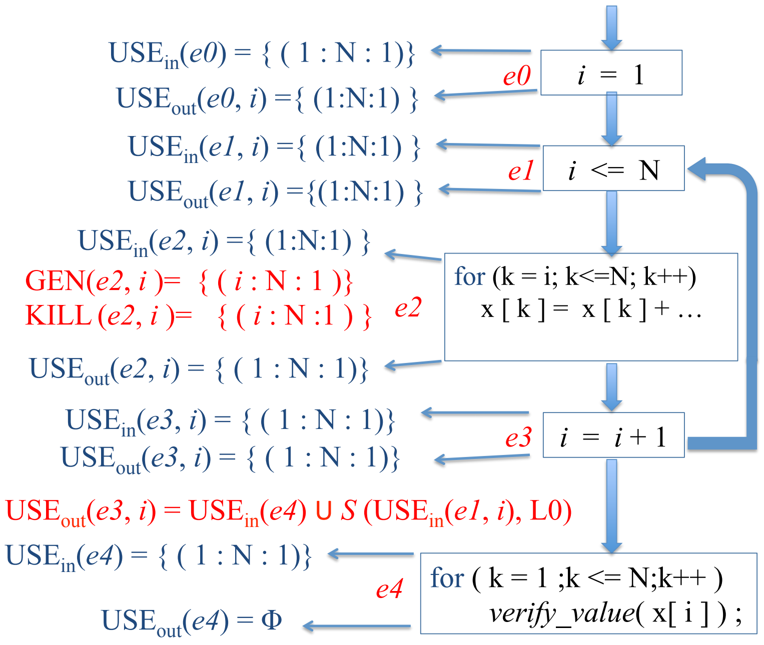

USE_in(e0) = { (1 : N : 1)}
USE_out(e0, i) ={ (1:N:1) }
e0
i = 1
USE_in(e1, i) ={ (1:N:1) }
USE_out(e1, i) ={(1:N:1) }
e1
i <= N
USE_in(e2, i) ={ (1:N:1) }
GEN(e2, i)= { (i : N : 1)}
KILL (e2, i)= { (i : N :1) }
e2
for (k = i; k<=N; k++)
x [k] = x [k] + …
USE_out(e2, i) = { (1 : N : 1)}
USE_in(e3, i) = { (1 : N : 1)}
USE_out(e3, i) = { (1 : N : 1)}
e3
i = i + 1
USE_out(e3, i) = USE_in(e4) ∪ S (USE_in(e1, i), L0)
USE_in(e4) = { (1 : N : 1)}
e4
for (k = 1 ;k <= N;k++)
verify_value(x[i]) ;
USE_out(e4) = Φ

Fig. 3.11.: Liveness analysis performed for the example in Figure 3.2 using variant-set analysis. Due to the cyclic representation, $e3$ represents the exit of *every* iteration in the enclosing loop $L0$ (not only the last iteration). As a result, $USE_{in}(e4)$ gets fully exposed to *every* iteration. This is conservative because $USE_{in}(e4)$ is being partially killed at node $e2$, i.e., not fully exposed to earlier iterations.

SMT solver. For example, the SMT solver can prove/disprove if all members of a Range Section are disjoint from a given section. The analysis may approximate the result of operations on Range Sections to reduce complexity, but this was not needed in our tested benchmarks.

3.2 Region-Based Analysis

We introduce the *region-based* analysis, which accounts for the effects of statements outside enclosing loops.

3.2.1 A Running Example

We again use classical Liveness analysis to illustrate our solution. As shown in Figure 3.11, the conservative effects from the upward exposed uses of the statements following the enclosing loop $L0$ reduce the accuracy of variant-set analysis.

3.2.2 Approach

Our approach is to form each enclosing loop in the CFG into a region, which gets analyzed independently using variant-set analysis. Next, each enclosing loop in the CFG is collapsed into a single node where Gen and Kill sets are computed by summarizing the analysis of the enclosing loop. Collapsed nodes are used when analyzing the whole program. With our approach, computed dataflow solutions within enclosing loops accurately represent the effects of statements within enclosing loops. Dataflow solutions outside enclosing loops are computed while accounting for the full effect of all iterations in enclosing loops. The limitation of our approach is that its applicability depends on the compiler transformation.

3.2.3 The Analysis Algorithm

Algorithm 1 describes region-based analysis. First, steps 1 and 2 identify enclosing loops in the CFG and determine their *analysis dependence* order, as follows: if $L1$ and $L2$ are two enclosing loops such that $L1$ is nested within $L2$, then $L1$ is analyzed before $L2$. Any two enclosing loops that are not nested can be analyzed in any order.

Second, steps $3 - 22$ in Algorithm 1 perform ADFA for each enclosing loop L according to their analysis dependence order. This is accomplished as follows: (i) Steps $6 - 9$ extract the region of L in the CFG; (ii) Step 10 performs ADFA for L using variant-set analysis; and (iii) Steps $12 - 21$ represent the region of L as a single node x in the CFG. Gen and Kill sets for node x are obtained by summarizing the analysis of all statements in all iterations of L.

Algorithm 1 Region-based analysis. CFG is the control flow graph.

```
1: $loop\_list \leftarrow$ a list of all enclosing loops in CFG
2: $ordered\_list \leftarrow$ sort $loop\_list$ by their $analysis\ dependence$ order
3: while $ordered\_list$ is not empty do
4:     $L \leftarrow$ pick and remove head of $ordered\_list$
5:     $(l : u : s) \leftarrow$ iteration space of $L$
6:     Original Predecessor Set $\leftarrow$ Pred ( $L\_initial\_node$)
7:     Original Successor Set $\leftarrow$ Succ ( $L\_exit\_node$)
8:     Pred ( $L\_initial\_node$) $\leftarrow \phi$
9:     Succ ( $L\_exit\_node$) $\leftarrow \phi$
10:    perform ADFA for $L$ using variant-set analysis
11:    create node $x$
12:    Pred ( $x$ ) $\leftarrow$ Original Predecessor Set
13:    Succ ( $x$ ) $\leftarrow$ Original Successor Set
14:    $KILL_{iteration}(i) \leftarrow$ kills within an iteration $i$
15:    $KILL(x) \leftarrow \bigcup_{i \in rng} KILL_{iteration}(i)$ , $rng = (l : u : s)$
16:    $GEN(x) \leftarrow$ summary of ADFA for all iteration of $L$
17:    represent $L$ region by $x$ in CFG
18: end while
19: perform ADFA for the whole CFG
```

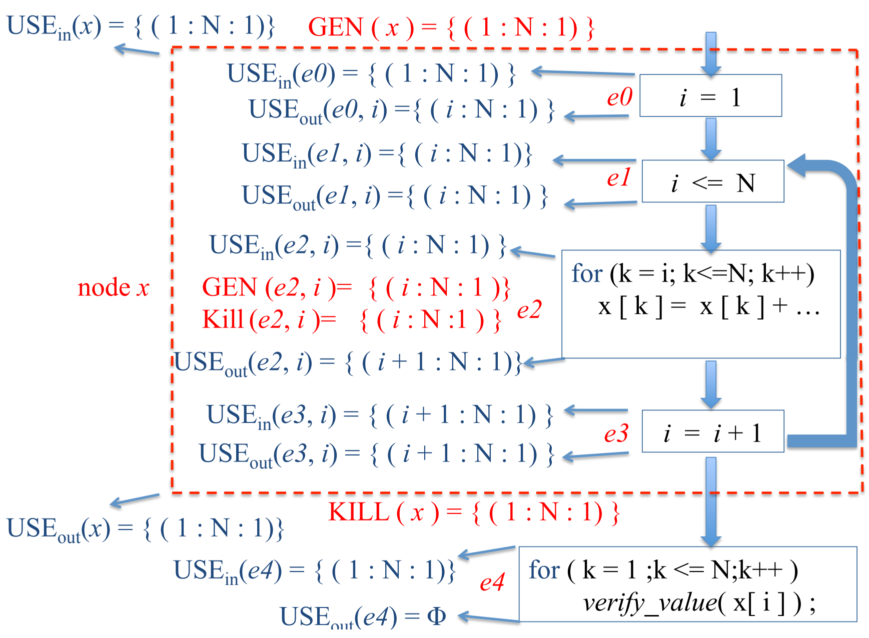
Finally, step 23 in Algorithm 1 performs ADFA for the entire CFG region, which now has each enclosing loop represented as a single node.

Figure 3.12 performs Liveness analysis using region-based analysis for the example in Figure 3.2. $GEN(x)$ represents the upward exposed uses from all iterations in $L0$ (which can be found at the initial node of $L0$). $KILL(x)$ represents the union of Kill sets for all iteration in $L0$. By contrast to Figure 3.11, $USE_{in}(e4)$ is killed by $KILL(x)$.

When extending region-based analysis to other ADFAs, steps $14 - 16$ can be extended to determine the combined effect depending on the ADFA.

3.3 Extending The solution to the Producer-Consumer Array Data Flow Analysis

We now apply variant-set and region-based analyses to producer-consumer array data flow analysis (PCDFA). As was presented in Chapter 2, PCDFA collects the



USE_in(x) = { (1 : N : 1)}
GEN (x) = { (1 : N : 1) }
USE_in(e0) = { (1 : N : 1) }
USE_out(e0, i) ={ (i : N : 1) }
e0
i = 1
USE_in(e1, i) ={ (i : N : 1)}
USE_out(e1, i) ={ (i : N : 1) }
e1
i <= N
USE_in(e2, i) ={ (i : N : 1) }
node x
GEN (e2, i)= { (i : N : 1)}
Kill (e2, i)= { (i : N :1) }
e2
for (k = i; k<=N; k++)
x [k] = x [k] + …
USE_out(e2, i) = { (i + 1 : N : 1)}
USE_in(e3, i) = { (i + 1 : N : 1) }
USE_out(e3, i) = { (i + 1 : N : 1) }
e3
i = i + 1
USE_out(x) = { (1 : N : 1)}
KILL (x) = { (1 : N : 1) }
USE_in(e4) = { (1 : N : 1)}
e4
for (k = 1 ;k <= N;k++)
verify_value(x[i]) ;
USE_out(e4) = Φ

Fig. 3.12.: Liveness analysis performed using region-based analysis for the example in Figure 3.2.

prior reaching definitions and upward exposed uses of shared array elements for each statement in an OpenMP program.

3.3.1 Extending Variant-Set Analysis

Performing PCDFA using variant-set analysis is as described in Section 3.1. In the implementation, π operators are annotations that specify parallelism semantics for partitioned iteration and data spaces while hiding the complexity of actual partitioning. This improves the symbolic reasoning of variant-set analysis because Gen and Kill sets retain the original bounds of data spaces. As explained in Section 3.1.3, those bounds can be variant (functions of enclosing loops indices) and therefore variant-set analysis can reason about different instances of these sets.

In general, the accuracy of variant-set analysis, when extended to an ADFA, depends on the power of the symbolic reasoning provided by the underlying ADFA framework.

Our results show that variant-set analysis improves the accuracy of PCDFA for Use sets in non-repetitive benchmarks and has no negative impact in repetitive benchmarks. By contrast to uses, definition sets remain the same with variant-set analysis. This is because definitions are the most recent produced data since the last barrier present in the current iteration (i.e., functions of the current instance only).

3.3.2 Extending Region-Based Analysis

Performing PCDFA using region-based analysis is as described by Algorithm 1 with a barrier placed at the end of each enclosing loop L. The barrier is needed to account for synchronization between the summarized definitions of L and the upward exposed uses from the statements outside and following L. This barrier can be redundant, e.g., the same synchronization between the same definitions and uses already occurred in the last iteration of L. Our implementation includes an additional step that eliminates redundant barriers for enclosing loops when redundancy can be proven.

Our results show that region-based analysis is beneficial for PCDFA with both repetitive and non-repetitive benchmarks.

3.4 Performance Evaluation

We evaluate the impact of using variant-set and region-based analyses into improving the accuracy of the PCDFA framework.

3.4.1 Performance Metrics

We evaluate the accuracy of PCDFA by measuring the maximum communication volume per thread. This directly evaluates the impact of using variant-set and region-based analyses into eliminating communication that was generated due to conservative array access information by PCDFA alone.

3.4.2 Experimental Setup

We evaluate using ten benchmarks: (i) Multi-Grid (MG), Conjugate Gradient (CG), Fourier Transform (FT), Block Tri-diagonal (BT), and Scalar Penta-diagonal (SP) (taken from the NAS Parallel Benchmark suite [22]); (ii) LU reduction (taken from the OmpSCR benchmark suite [24]); (iii) Gram-Schmidt process and Cholesky decomposition (taken from the PolyBench suite [25]); and (iv) LU_v and Gram-Schmidt_v (the same LU and Gram-Schmidt benchmarks but with additional verification codes as shown in Figure 3.2).

All benchmarks are available in OpenMP, except for the PolyBench benchmarks, which are sequential (we create OpenMP versions by hand). Verification codes in LU_v and Gram-Schmidt_v were also parallelized. All parallel loops are block-partitioned. NAS benchmarks are also available in MPI. We found UPC versions available online [26] for MG and CG.

We use CLASS C input data size for all NAS benchmarks. The input size for Cholesky, LU and LU_v is 10000×10000 and the input size for Gram-Schmidt and Gram-Schmidt_v is 5000×5000. The input size is only known at runtime – parameters that describe input sizes are unknown during the compiler analysis. Tested benchmarks consist of six non-repetitive benchmarks (MG, LU, LU_v, Gram-Schmidt, Gram-Schmidt_v and Cholesky), and four repetitive benchmarks (CG, SP, FT and BT).

We implement variant-set and region-based analyses using the Cetus Compiler infrastructure [21]. We use the same runtime tool described in Section 2.5 to measure performance metrics.

Except for Cholesky, variant-set analysis converged with no more than 3 examined iterations in all benchmarks. Unlike other benchmarks, Cholesky has an enclosing loop where a fixed dataflow pattern is not present. This is because Gen sets in previously examined iterations have array elements that overlap with array elements of Kill sets in an *unbounded* number of later examined iterations.

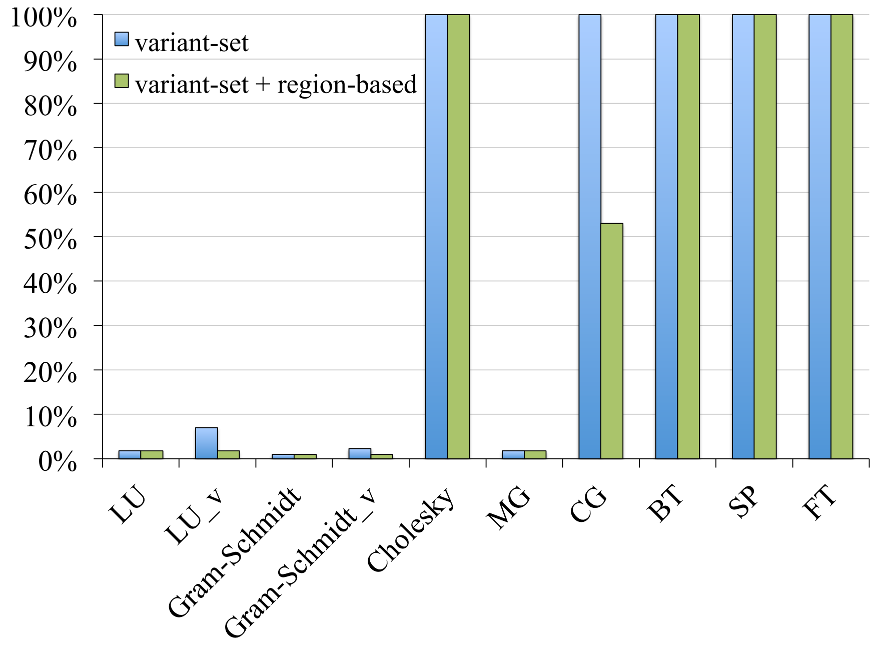
3.4.3 Evaluation of the Producer-Consumer Array Data Flow Analysis Accuracy

Figure 3.13 shows the measurements of use information and communication volumes. Use information measurements show that the reduction in communication is due to improving accuracy while determining Use sets by our compiler analyses compared to PCDFA alone. Definition information with our analyses remain the same as the original PCDFA (as explained in Section 3.3.1).

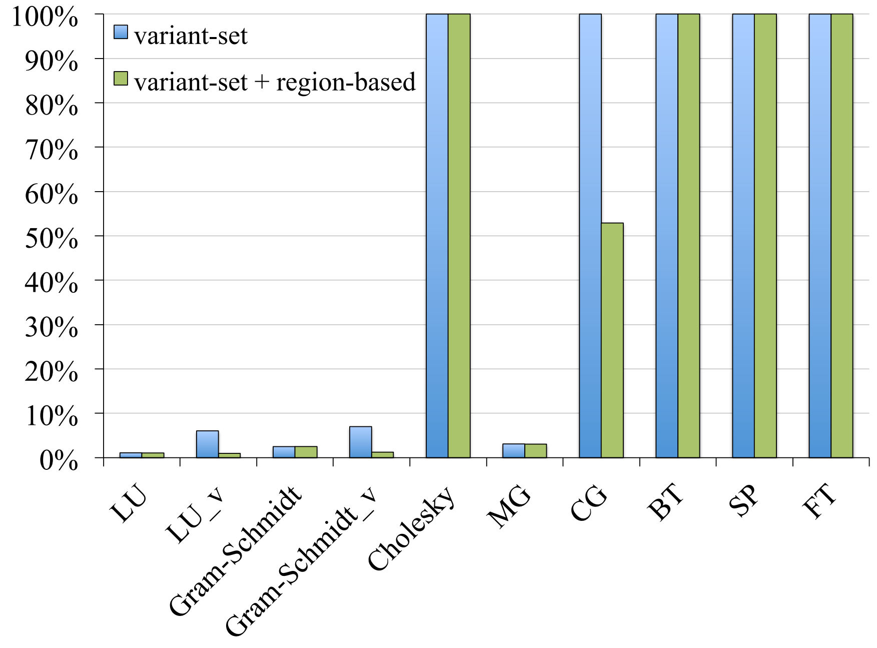
In the case of non-repetitive benchmarks, the percentage of communication with variant-set analysis is 7% or less, except for Cholesky. When also using region-based analysis, communication volume is reduced for LU_v and Gram-Schmidt_v and the percentage of communication becomes 3% or less. When performing PCDFA alone for non-repetitive benchmarks, Use sets are conservative due to approximating variant Gen and Kill sets; Gens are overestimated by assuming all threads read all data in all instances and Kills are underestimated. By contrast, our analyses generate variant and accurate Use sets.

In the case of repetitive benchmarks (where Gen and Kill sets are invariant), variant-set analysis shows no negative impact. When using region-based analysis, the communication percentage of CG decreases to 52% due to removing the outside conservative effects while enclosing loop regions are analyzed (as explained in Section 3.2). This shows that region-based analysis can also be beneficial for repetitive benchmarks. For BT and SP, our implementation determines that the inserted barriers by region-based analysis are redundant and therefore are removed (as explained in Section 3.3.2).

Overall, region-based analysis enables PCDFA to perform accurately for non-repetitive benchmark and yields no negative impact on repetitive benchmarks. Region-based analysis is beneficial with both repetitive and non-repetitive benchmarks.



(a) Use information percentage



(b) Communication percentage.

Fig. 3.13.: Maximum volume of use information and communication (per thread) with our compiler frameworks normalized to the volume obtained with the PCDFA framework when performed alone, computed as a percentage (less is better). We show the average percentage over 8, 16, 32 and 64 threads.

4. RUNTIME COMMUNICATION SCHEDULING

The role of the runtime system is to generate MPI communication. At each communication point in the translated program, communication satisfies dependencies between *prior* produced shared array elements by a thread and the *upward exposed* uses by another thread. We introduce a scheduling scheme that determines shared array elements that need to be communicated (i.e., communication sets) at runtime.

Prior translation schemes [10, 11] made the simplifying assumption that communicated array elements are invariant across all instances of communication points. Using this *repetitive* communication property, the runtime overhead can be amortized because communication schedules need to be computed only once.

In contrast to repetitive communication patterns, communicated array elements vary across different instances of communication points for non-repetitive communication patterns. Therefore, new communication schedules are needed for every instance of communication points, introducing the potential problem of high runtime overheads for communication scheduling.

To reduce runtime overheads, we introduce a new communication scheduling scheme that exploits an algebra of π operators to determine communication partners for each thread at a given communication point. In doing so, the work needed to schedule communication is proportional to the number of communication partners. In practice, threads communicate with a small subset of other threads. Therefore, our scheme can schedule communication without incurring high runtime overheads with both repetitive and non-repetitive communication patterns.

The remainder of this chapter is organized as follows: Section 4.1 describe the algebra that determines communication partners. Section 4.2 describes the communication scheduling algorithm. We delay evaluating runtime overheads to overall performance evaluation in Chapter 5.

4.1 Determining Communication Partners

As presented in Chapter 2, π operators provide an abstract representation that contain information about how shared array elements are partitioned across threads in array sections (e.g., block partitioning, cyclic partitioning, etc). Because the partitioning scheme is known, our idea is to use an algebra of π operators to determine the *communication range* that describes, for each thread at a communication point, other threads where communication may be needed. These ranges will be then used for communication scheduling.

First, we describe an example to explain how the aforementioned algebra can be obtained. Consider a communication point that have the produced array section π_b(0:99:1) and the consumed array section π_b(1:100:1), where π_b is the block partitioning operator. Assuming that the total number of threads is 4, let the actual partitioning of produced data be (0:24:1), (25:49:1), (50:74:1), and (75:99:1), which are mapped to threads 0, 1, 2, and 3, respectively. Similarly, let the actual partitioning of consumed data be (1:25:1), (26:50:1), (51:75:1), and (76:100:1), which are mapped to threads 0, 1, 2, and 3, respectively.

In the previous example, block partitioning divides array elements into approximately even partitions and assign them to threads in one-to-one fashion. Therefore, there exists an inverse function that determines the thread that corresponds to a particular array element. Using this function, a thread can determine other threads that need communication. For example, thread 1 (which has the produced data partition (25:49:1)) can determine which threads in the block-partitioned consumed space π_b(1:100:1) correspond to elements 25 and 49 (the lower and upper bound of thread 1's produced data partition). In the example, those threads would be 0 and 1 and therefore the communication range of send messages is [0:1]. Similarly, thread 1 can find the communication range of receive messages by finding which threads in the block-partitioned produced space π_b(0:99:1) correspond to the lower and upper elements of thread 1's consumed data partition (26:50:1).

We now generalize. Consider the partitioned space $\pi_x(l{:}u{:}s)$, where l, u, and s are, respectively, the original lower bound, upper bound and stride expressions, and x is the type of partitioning applied to this space (Table 2.1 shows the list of available π operators). We introduce the function $\psi_x(\ l,\ (l{:}u{:}s)\ )$, which finds the threads in $\pi_x(l{:}u{:}s)$ that correspond to the particular element l. The implementation of ψ_x depends on the implementation of π_x, which is obtained by an algebra that takes the inverse function and the total number of threads into consideration, as explained in the previous example.

In this dissertation, we use *regular section descriptors* (RSDs) [18] to represent array sections. In Chapter 2, we introduced the πRSD representation, which extends the RSD to allow dimensions with partitioned data spaces to be represented using π operators. Given two array sections πRSD1 and πRSD2, the function $\Psi(c,\ T,\ \pi\text{RSD1},\ \pi\text{RSD2})$ determines the communication range for thread c by finding the threads in πRSD2 that correspond to the partition in πRSD1 of thread c (T is the total number of threads). This function is implemented using the function ψ_x.

4.2 The Scheduling Algorithm

We now describe Algorithm 2, which determines communication schedules at a communication point p. First, steps $1-8$ determine communication ranges for send and receive messages using the Ψ function. Next, steps $9-22$ create messages by examining communication ranges and finding the common shared array elements between produced and consumed array sections. Note that the specific array section produced or consumed by a thread is obtained by applying π operators in πRSD sections, which yield normal RSD sections.

The communication volume in Algorithm 2 depends on the accuracy of the array section information obtained from the compiler. The runtime system makes no approximation during communication scheduling. In addition, the runtime system

Algorithm 2 Scheduling algorithm for communication generation at a given communication point p.

```
1:  $T \leftarrow$ total number of threads
2:  $c \leftarrow$ current thread number
3:  $\pi\mathrm{RSD}_{def} \leftarrow$ *produced array section at p*
4:  $\pi\mathrm{RSD}_{use} \leftarrow$ *consumed array section at p*
5:  $[R1{:}R2] \leftarrow \Psi(c,\ T,\ \pi\mathrm{RSD}_{def},\ \pi\mathrm{RSD}_{use})$
6:  $[S1{:}S2] \leftarrow \Psi(c,\ T,\ \pi\mathrm{RSD}_{use},\ \pi\mathrm{RSD}_{def})$
7:  for each  thread $y \in [R1 : R2]$ and $y \neq$ c  do
8:      $\mathrm{RSD}^{c}_{def} \leftarrow$ *the specific array section produced by thread c* at $p$
9:      $\mathrm{RSD}^{y}_{use} \leftarrow$ *the specific array section consumed by thread x* at $p$
10:     common_array_elements $\leftarrow \mathrm{RSD}^{c}_{def} \cap \mathrm{RSD}^{y}_{use}$
11:     if *common_array_elements* $\neq \phi$ then
12:         create *message*( sender= $c$, receiver= $y$, data= common_array_elements )
13:     end if
14: end for
15: for each  thread $y \in [S1 : S2]$ and $y \neq$ c  do
16:     $\mathrm{RSD}^{c}_{use} \leftarrow$ *the specific array section consumed by thread c* at $p$
17:     $RSD^{y}_{def} \leftarrow$ *the specific array section produced by thread y* at $p$
18:     common_array_elements $\leftarrow RSD^{c}_{use} \cap RSD^{y}_{def}$
19:     if *common_array_elements* $\neq \phi$ then
20:         create *message*( sender= $y$, receiver= $c$, data= common_array_elements )
21:     end if
22: end for
```

merges communication schedules such that array elements are not redundantly communicated.

The runtime overhead of Algorithm 2 is proportional to the number of threads in communication ranges. In our tested benchmarks, the dominant communication pattern is point-to-point, where a thread communicates with a small subset of other threads. Therefore, communication scheduling overheads are proportional to this small subset of threads. In the case of all-to-all communication (the less frequent type in our tested benchmarks), communication ranges are proportional to the total number of threads.

5. PUTTING EVERYTHING TOGETHER: THE OPENMP-TO-MPI TRANSLATOR

We combine the concepts introduced in earlier chapter and describe a source-to-source automatic compiler-runtime scheme to translate shared-address programs written in standard OpenMP to distributed-address programs written in MPI [1]. The scheme covers OpenMP computational programs with regular write data accesses and both repetitive *and* non-repetitive communication patterns. By contrast, prior translation schemes only target programs with repetitive communication patterns.

The translation process starts by the compiler converting the input OpenMP program into *single-program-multiple-data* (SPMD) [16] form. The resulting code represents a *node program* for each thread that operates on partitioned data. The SPMD code has function calls inserted by the compiler that contain information needed by the runtime system to generate communication, which is performed using the MPI communication library.

Using ten (six non-repetitive and four repetitive) OpenMP benchmarks, we measure the speedups obtained by scaling these OpenMP benchmarks to a cluster of 64 cores. We also compare against state-of-the-art OpenMP-to-MPI translation system, and available hand-coded MPI and UPC programs.

The remainder of this chapter is organized as follows: Section 5.1 describes the translation process by the compiler and Section 5.2 describes communication generation by the runtime system. Section 5.3 evaluates the performance.

[1]OpenMP thread corresponds to MPI process

5.1 Compiler Code Generation

The compiler starts by parsing the OpenMP directives in the input program to identify parallel and serial regions and synchronization statements. All *omp master* and *omp single* constructs are conservatively treated as serial regions. To identify shared variables for each parallel region using a previulsy introduced shared variables analysis [27]. The compiler rely on internal annotations to keep information about shared variables.

The compiler also identifies reduction operations. There are two sources of reduction operations: (i) Explicit `omp reduction` clauses in the source program; and (ii) `omp critical` or `omp atomic` constructs that are used for reduction operations. A common practice in OpenMP programs is to code array or scalar reductions using a critical section where each thread updates the global copy of the shared data using its own local copy. The compiler identifies such critical sections. All critical sections in the tested benchmarks are used as reduction operations. Handling general critical sections is beyond the scope of our translation scheme.

The compiler annotates all reduction operations. When the compiler generates the final code, all reduction operations are converted into *MPI_reduce* operations, which are communication routines provided by the MPI library to perform reduction operations.

The compiler uses the *Producer-Consumer Flow Graph* (PCFG) [19] to represent the analyzed OpenMP program. As mentioned in Section 2.2, PCFG is a Control Flow Graph that represents both the control flow and the relevant memory model semantics of an OpenMP program. In particular, *barrier* nodes, which denote points where memory is to be made coherent across threads, are placed at the end of parallel loops that do not have an OpenMP *nowait* directive. Each node in the PCFG corresponds to a program statement or an OpenMP directive.

Using the producer-consumer array data flow analysis (PCDFA) presented in Chapter 2, the compiler analyzes prior reaching definitions and upwardly exposed uses of shared array elements for the OpenMP program's statements. The compiler uses π operators to represent partitioned iteration and data spaces. In addition, the compiler uses delayed symbolic evaluation to ensure that all operations during the dataflow computation are performed accurately. In Section 2.5, we presented early results to demonstrate the impact of using π operators and delayed symbolic evaluation on the accuracy of PCDFA.

Furthermore, the compiler uses variant-set and region-based analyses presented in Chapter 3 for optimizing the accuracy of PCDFA in the presence of non-repetitive loops. The impact of doing so was evaluated in Section 3.4.

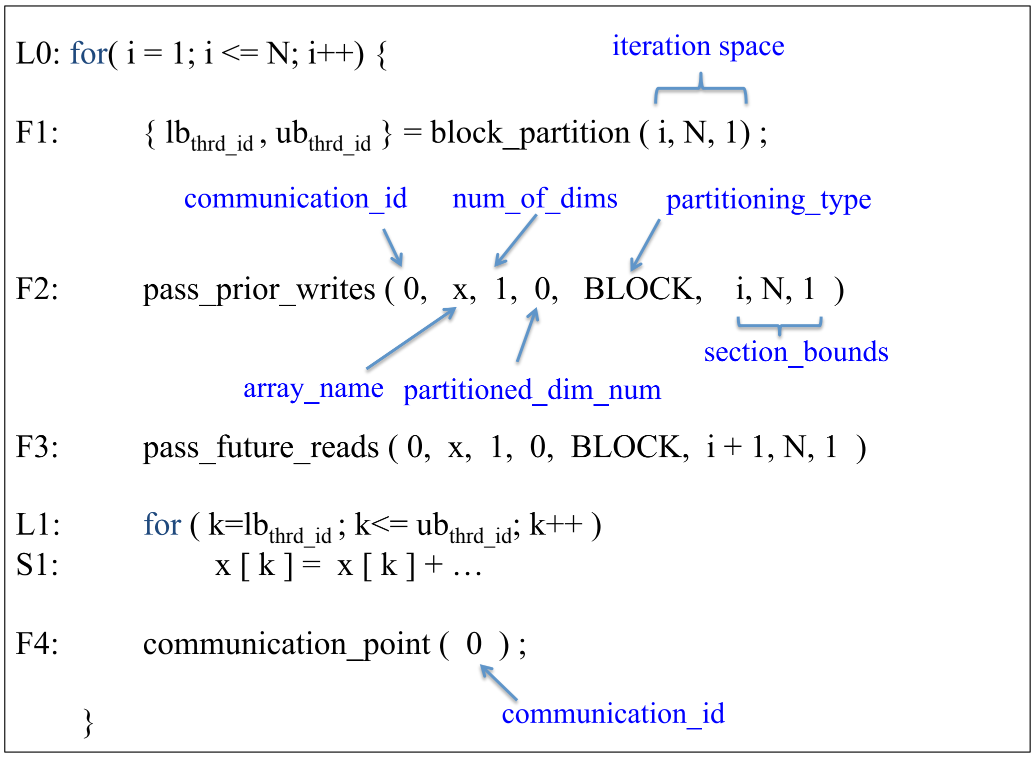
As described in Section 1.2, information that describe prior definitions of shared array elements for each thread at communication points need to be exact. To ensure correctness, the compiler serializes parallel loops that yield imprecise definitions information (e.g., a parallel loop that has an irregular write array access). In addition, the compiler performs both *must* and *may* Reaching Definitions analyses and serializes parallel loops where the result of both analyses are not equal. This eliminates imprecision due to control dependencies. Tested benchmarks in this dissertation have regular write accesses and therefore were analyzed precisely by the compiler. Handling irregular write array accesses is beyond the scope of this dissertation (we describe techniques to handle these accesses in Future Work in Section 7.2).

In contrast to definition information, information about future uses of shared array elements for each thread at communication points can be approximated. Therefore, the volume of communication in our translation scheme depends on the accuracy of use information.

The final step is code generation. The generated SPMD code has the the following properties:

i. The work of parallel loops is partitioned among processes according the OpenMP *schedule* directives (all tested benchmarks use block partitioning).

ii. Serial regions are redundantly executed by all participating processes.



```
L0: for( i = 1; i <= N; i++) {
F1:      { lb_thrd_id , ub_thrd_id } = block_partition ( i, N, 1) ;
F2:      pass_prior_writes ( 0,  x,  1,  0,  BLOCK,  i, N, 1  )
F3:      pass_future_reads ( 0,  x,  1,  0,  BLOCK,  i + 1, N, 1  )
L1:      for ( k=lb_thrd_id ; k<= ub_thrd_id; k++ )
S1:           x [ k ] =  x [ k ] + …
F4:      communication_point (  0  ) ;
     }
```

Fig. 5.1.: The translated SPMD code generated by the compiler system for the OpenMP example in Figure 3.1.

iii. The virtual address space of shared data is replicated on all processes. Shared data is not physically replicated, only the data actually accessed by a process is physically allocated on that process.

In the SPMD code, the compiler inserts function calls that initiates the runtime system to perform the following actions: (i) compute partitioned loops' bounds; (ii) generate communication at each communication point; and (iii) obtain information about prior definitions and future uses of shared array elements that correspond to each communication point. Figure 5.1 shows the translated code for the OpenMP example in Figure 2.1a.

In this dissertation, we consider OpenMP programs with loop-level parallelism. Handling OpenMP programs with `omp section` and `omp task` constructs is beyond the scope of our work.

5.2 Runtime Communication Generation

At each communication point, the runtime system uses the communication scheduling scheme presented in Chapter 4 and generates MPI messages. Needed information about produced and consumed array sections are obtained from the compiler.

As shown in Figure 5.1, new communication schedules are needed for every iterations in the outer sequential loop $L0$. This because produced and consumed array sections at communication point p are variant (the inner parallel loop $L1$ is non-repetitive). The communication pattern at communication point p is point-to-point. Using our communication scheduling scheme, the overhead time of communication scheduling is reduced.

No additional functionality is needed beside communication scheduling at runtime. In contrast, the state-of-the-art OpenMP-to-MPI translation system [11] relies on a runtime data flow analysis to obtain 100% operation accuracy. Our translation scheme obtains this accuracy using delayed symbolic evaluation with no significant runtime overheads (as shown in Chapter 2).

5.3 Evaluation of Overall Performance

We evaluate our translation system on a cluster of 64 cores. In addition, we compare with a state-of-the-art translator [11], referred to as the hybrid translator, which targets repetitive OpenMP benchmarks. We also compare against available hand-coded MPI [1] and UPC [4] programs.

5.3.1 Performance Metrics

In Chapter 3, performance evaluation showed the impact of variant-set and region-based analyses into reducing communication. We now evaluate the impact of this optimization on overall performance. We do so by measuring the speedups of our

Table 5.1: Benchmark set.

Benchmark	Suite	Type	Input Size	Availlable in
FT	NAS	Repetitive	CLASS C	OpenMP, MPI, UPC
BT	NAS	Repetitive	CLASS C	OpenMP, MPI
SP	NAS	Repetitive	CLASS C	OpenMP, MPI
CG	NAS	Repetitive	CLASS C	OpenMP, MPI, UPC
MG	NAS	Non-Repetitive	CLASS C	OpenMP, MPI, UPC
LU	OmpSCR	Non-Repetitive	10000 x 10000	OpenMP
LU_v	OmpSCR	Non-Repetitive	10000 x 10000	OpenMP
Gram-Schmidt	PolyBench	Non-Repetitive	5000 x 5000	Sequential
Gram-Schmidt_v	PolyBench	Non-Repetitive	5000 x 5000	Sequential
Cholesky	PolyBench	Non-Repetitive	10000 x 10000	Sequential

translated benchmarks on a cluster of 64 cores (8 nodes $\times$ 8 cores). Additionally, we measure runtime overheads of communication scheduling.

5.3.2 Experimental Setup

The translation process from OpenMP to MPI is fully automatic. We implemented the full compiler system using the Cetus infrastructure [21]. We also implemented a runtime system for MPI communication generation. The hybrid translator was obtained from the author.

We evaluate using the same ten benchmarks used in Section 3.4, which are shown in Table 5.1. We performed experiments using a community cluster, with available 8-core Intel Xeon-E5 processor and 64 GB of memory per node. Each node runs eight MPI processes or OpenMP threads. The underlying operating system is Red Hat Linux kernel 2.6.32. Nodes communicate using a 40 Gbps FDR10 Infiniband network. The back-end compiler is Intel64 13.1, and the MPI runtime environment is MVAPICH2 1.9. We compile and execute UPC benchmarks using Berkeley UPC translator and runtime version 2.18.2.

5.3.3 Evaluation on a Cluster of 64 Cores

Figure 5.2 and Figure 5.3 show the speedups of the translated non-repetitive and repetitive benchmarks on 1, 2, 4 and 8 nodes (or 8, 16, 32 and 64 cores), respectively. The speedups of available hand-coded MPI and UPC benchmarks are also shown.

In the case of non-repetitive benchmarks, the translated programs achieve, on average, a speedup of 3.8x on 8 nodes. By comparison, the translated programs with PCDFA alone is slower than OpenMP programs due to excessive communication, except for Cholesky, which has a small communication volume. As mentioned before, translating non-repetitive OpenMP programs with the hybrid translator is not supported.

MG is the only non-repetitive benchmark with available hand-coded MPI and UPC versions. On 8 nodes, the speedups of MPI and UPC programs are 5.2x and 4.8x, respectively. This is approximately 2x faster than our translator. The difference in performance is due to using advanced multidimensional partitioning schemes that improve scalability as the number of cores increases in both MPI and UPC codes. Note that the OpenMP code uses one-dimensional parallelism where only one loop is partitioned in loop nests. The UPC code is also tuned for optimizing thread locality and communication prefetching.

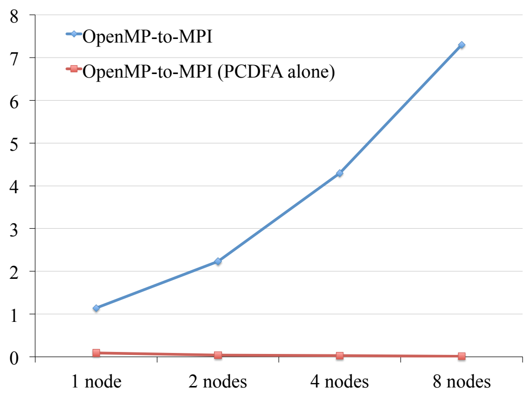
We also analyze runtime overheads. Figure 5.4 shows the overall runtime overhead of communication scheduling in all iterations of enclosing loops in LU and MG. As explained in Chapter 4, the runtime communication scheduler takes advantage of point-to-point communication (the dominant pattern) to reduce runtime overheads. Except for Cholesky, runtime overheads in all other non-repetitive benchmarks have similar behavior (we choose LU and MG as a representatives). Cholesky has all-to-all communication and therefore the runtime overhead is proportional to the total number of threads. The average overhead ratio is 19% of the execution time.

In the case of repetitive benchmarks, the translated programs achieve an average speedup of 3.6x on 8 nodes. Compared to the hybrid translator, our translator is 1.44x faster with CG (due to region-based analysis), and has the same performance with FT, BT and SP. MPI and UPC codes are approximately 1.8x faster than the OpenMP-to-MPI translator, on average. MPI codes of BT and SP use advanced multidimensional partitioning schemes. CG has sparse data access patterns, the algorithms of MPI and UPC codes are optimized by hand to reduce communication. In the case of FT, MPI code exceeds our translator due to indirect read memory accesses that caused our translator to generate extra communication. The UPC version of FT is slower than both MPI and our translator. Unlike other UPC benchmarks, FT was not optimized by hand for communication prefetching.

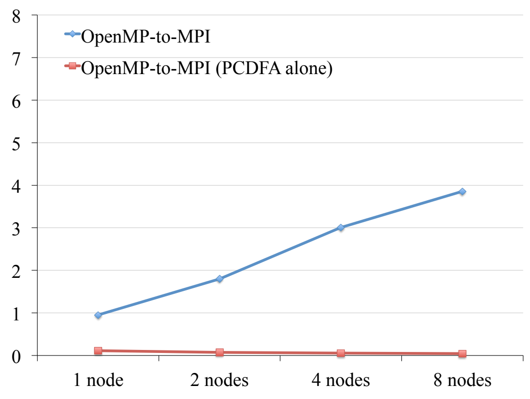
Runtime overheads with repetitive benchmarks are negligible. This is because communicated array elements are invariant for all instances of communication points. Communication schedules need to be computed only once, and then reused for all instances.

Overall, variant-set and region-based analyses improve PCDFA's array access information accuracy and therefore performance by 8.6x and 1.11x, on average, for the six non-repetitive and four repetitive benchmarks, respectively. Compared to OpenMP on 8 cores, all ten benchmarks achieve a speedup of 3.8x, on average. By comparison, the hybrid translator only scaled the three repetitive benchmarks with an average speedup of 3.3x. Our translator is faster than the hybrid translator by 11%, on average. The performance of our translated programs came within 54% and 60% of the performance of hand-coded MPI and UPC programs, respectively.

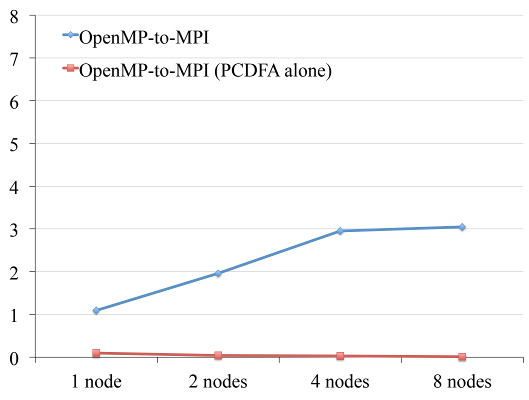
As shown by our results, our work overcomes the weakness in current OpenMP-to-MPI translation schemes and allows non-repetitive programs to be efficiently scaled to clusters. In future work (Section 7.2), we discuss techniques that can be combined with our work to reduce the gap to hand-coded MPI and UPC programs.



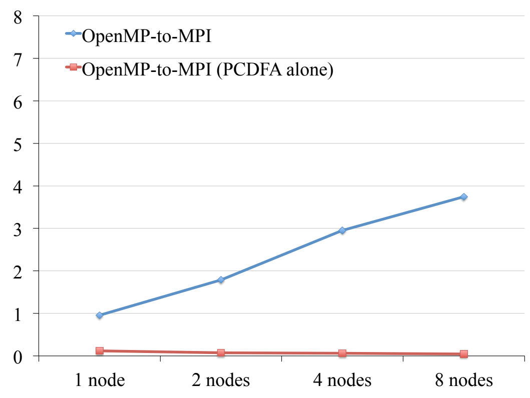
(a) LU



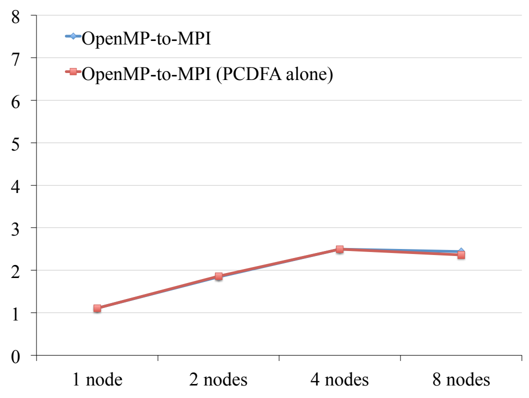
(b) Gram-Schmidt



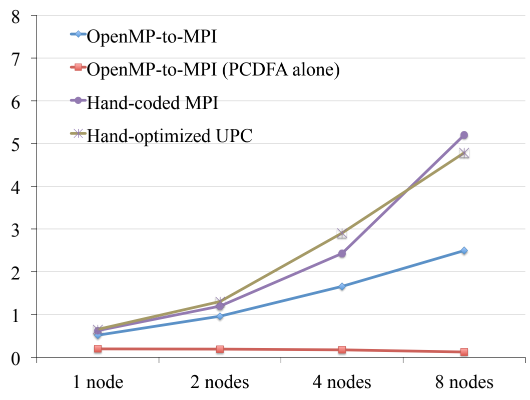
(c) LU_v



(d) Gram-Schmidt_v

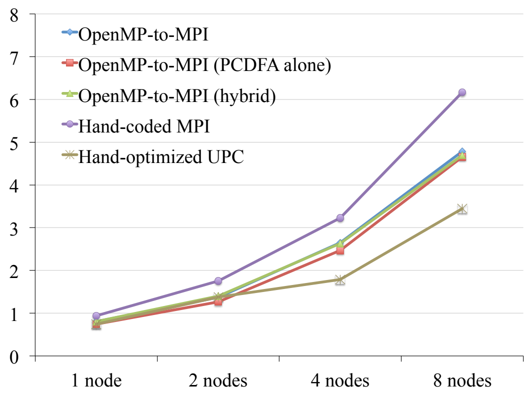


(e) Cholesky

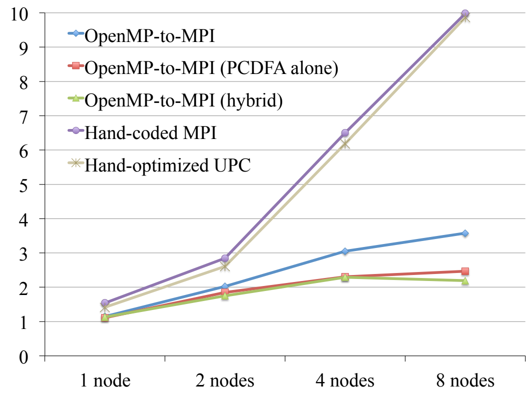


(f) MG

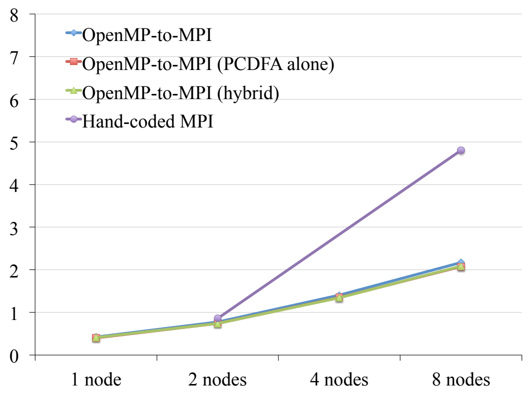
Fig. 5.2.: The speedups of the translated *non-repetitive* OpenMP to MPI programs and the hand-coded MPI and UPC programs (when available) on a cluster of 1, 2, 4 and 8 nodes (or 8, 16, 32 and 64 cores) over OpenMP on 1 node (8 cores). Measurements for translated programs are shown for both cases: (i) when PCDFA is performed using variant-set and region-based analysis; and (ii) when PCDFA is performed alone.



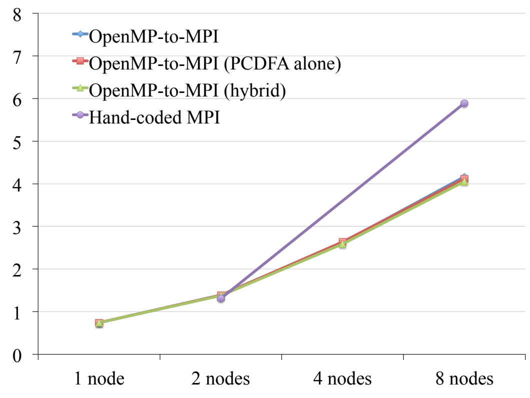
(a) FT



(b) CG

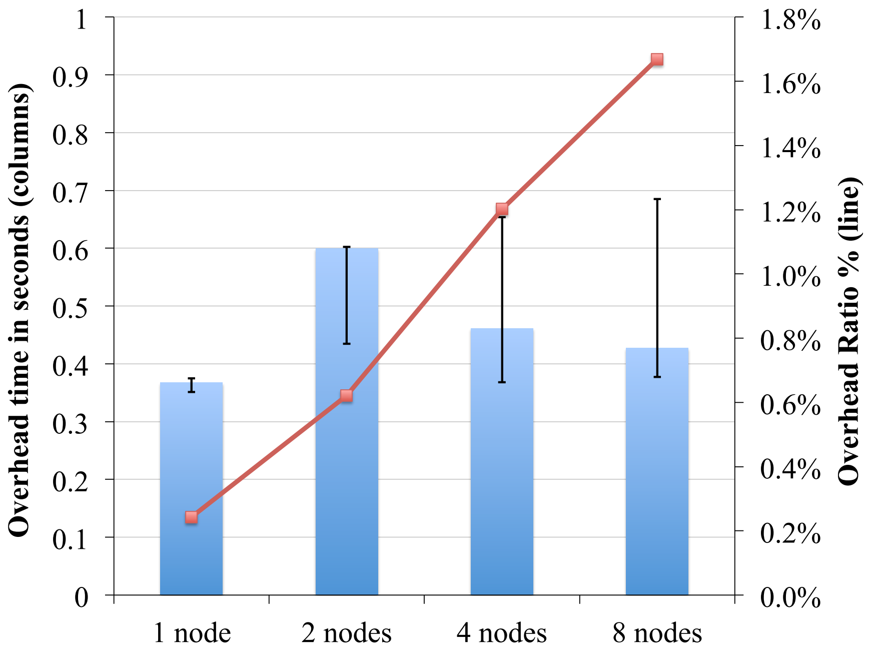


(c) SP

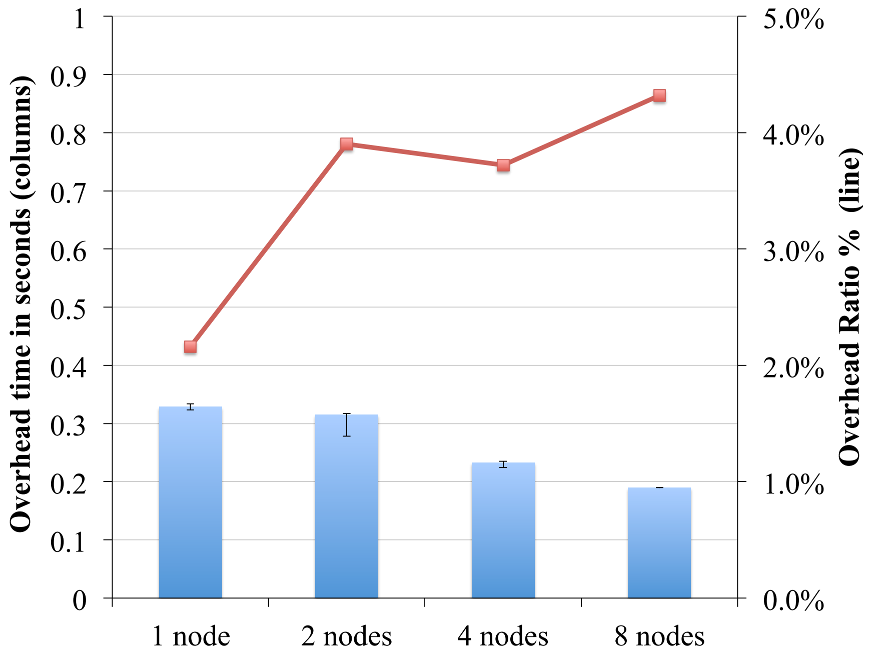


(d) BT

Fig. 5.3.: The speedups of the translated *repetitive* OpenMP to MPI programs and the hand-coded MPI and UPC programs (when available) on a cluster of 1, 2, 4 and 8 nodes (or 8, 16, 32 and 64 cores) over OpenMP on 1 node (8 cores). MPI codes for BT and SP run only with a square number of cores. Measurements for translated programs are shown for both cases: (i) when PCDFA is performed using variant-set and region-based analysis; and (ii) when PCDFA is performed alone. They are also shown for the hybrid translator [11], the current state-of-the-art translator.



(a) LU



(b) MG

Fig. 5.4.: The runtime overhead during the execution of the translated LU and MG benchmarks. Error bars show the ranges of the absolute overhead time in five runs. Because we use strong scaling (input size is fixed), the overhead ratio generally increases while increasing the number of nodes.

6. RELATED WORK

Our work is directly related to approaches that provide a shared memory abstraction for distributed memory programming. In prior work [28], a manual proof of general concepts showing the feasibility of automatic translation of OpenMP to MPI was presented. These concepts were incorporated in [10] by presenting a compiler-runtime system for automatic translation of OpenMP programs with regular write data accesses and repetitive communication patterns to MPI. A unique and important feature of this dissertation is the ability to also handle OpenMP applications with non-repetitive communication patterns. We survey other models that provide shared memory abstraction for distributed memory architectures in Section 6.1.

In addition, this dissertation presented new concepts in the context of Array Data Flow Analysis (ADFA). In the literature, ADFAs have been used for analyzing array section information in OpenMP programs, as well as other programming models. We survey prior ADFA frameworks in more detail in Section 6.2.

6.1 Prior Shared-Address Space Programming Models for Distributed-Address Space Architectures

High Performance Fortran (HPF) [3] is an early effort for improving the productivity of programming with distributed memory architectures. HPF compilers automatically translate shared memory programs, written in Fortran, to message-passing programs. This is accomplished using data partitioning directives that are provided by the programmer and assist the compiler to physically distribute shared arrays among processes. This distribution will hold for the entire execution.

Many implementations of HPF (such as [29–31]) used the *owner computes* rule, where computation is always performed on the process that owns the left hand side of the computation. The owner computes rules assists the compiler while performing communication optimizations due to available static information about owners.

By contrast to HPF, our approach provides a standard OpenMP programming interface, where data partitioning directives or the owner computes rule are not required. Another difference is that data distribution among processes with our approach may vary during the execution.

Software Distributed Shared Memory (SDSM) systems have also been proposed to provide a global shared address space abstraction on distributed memory architectures *at runtime*. TreadMarks [5] is a well-known SDSM system that tracks accesses to shared data at runtime using page faults mechanisms. In addition, researchers have proposed compile-time optimizations [7–9] to SDSM systems, where the compiler inserts directives to generate prefetch instructions at runtime. Intel's Cluster OpenMP [6] was a commercial SDSM system product, but is no longer supported.

All page-based SDSM approaches, while support OpenMP or a similar interface, have inherent significant overheads and false sharing issues due to the page granularity of tracking shared accesses at runtime. Our approach does not rely on expensive runtime mechanisms. Instead, communication sets are determined based on a compiler analysis of shared array sub-ranges representing the written and read data elements.

Recent SDSM schemes [32–34] proposed a source-to-source translation approach from OpenMP to *Global Arrays* (GA). GA [35] provides a virtual shared address space to programmers that gets supported by a runtime system. Compared to prior SDMS approaches, the OpenMP-to-GA approach provides a better scalability as communication is not performed at a page level. Instead, threads update global arrays at the end of each parallel region by their local writes. Compared to our translation scheme, this approach still generates excessive communication because it does not account for an analysis of future read shared array elements.

Other related efforts are the *Partitioned Global Address space* languages such as UPC [4], Co-array Fortran [36], X10 [37] and Titanium [38]. In general, PGAS models provide a global shared address space that is logically divided among threads. Programmers specify thread-data affinity and optimize computation locality by having threads compute on their local data.

Compared to our programming model, UPC programmers generally hand-tune their codes by include additional optimizations such as privatizing local shared accesses or perfecting remote shared data using block transfer (optimizations described in [39]). The programming productivity of UPC is studied and quantified in [40]. Our scheme supports standard OpenMP interface, where programmers need not specify thread-data affinity or include manual optimizations.

Similar to our approach, other researchers [12, 13] have also presented translation schemes from OpenMP to MPI. In contrast to our work, these schemes do not collect information about future uses of shared array elements for threads. Instead, they analyze produced array elements within each parallel loop, which are sent to all nodes at the end of the loop. While this simplifies the compiler analysis, these translation schemes generate a lot of unnecessary communication.

Kwon et al. [10] also presented an OpenMP-to-MPI translator that collects both prior produced and future consumed array elements using an ADFA framework and targets repetitive OpenMP programs. We consider Kwon's work to be the state-of-the-art. We compare against Kwon's work in Section 5.3 and show that, with the new introduced contribution in this dissertation, our scheme (i) outperforms or performs as well as Kwon's work with repetitive benchmarks; and (ii) efficiently scales non-repetitive benchmarks to clusters, which are a new class of programs that was not handled by Kwon's work.

A recent approach by Bondhugula [41] was proposed to allow using sequential programs while targeting distributed memory architectures. In particular, this approach used the polyhedral model and presented a compiler for automatic translation of sequential programs into MPI. However, this approach is restricted to programs with affine loop nests. Our approach allows a larger set of applications.

6.2 Prior Compiler Frameworks for Array Data Flow Analysis

There is a rich literature on array access analyses and their use in compiling both parallel and sequential programs. We divide prior work based on the statement-level internal representation by the compiler, as follows: (i) work that used classical array data flow analysis (ADFA) frameworks; and (ii) work that used integer linear programming (ILP) frameworks. With ILP frameworks, a statement in a loop nest is represented as an individual instance using a matrix representation. By comparison, ADFA frameworks represent all instances of a statement as a single global instance.

Compared to ADFA frameworks, ILP frameworks provide more powerful symbolic reasoning. However, they are more complex in their implementation and commonly more restrictive (only applicable to the subset of programs that have affine loop nests). Our tested benchmarks include cases that do not confirm to this model such as MG and CG.

Analyzing accessed array elements in programs where different instances of a loop read and write different array elements favors using ILP frameworks. However, this paper presents concepts that allow ADFA frameworks (a more general and less complex approach) to represent and reason about array elements written or read by individual instances of a statement. This holds in the presence of recurring *patterns* that describe dataflow computation across all instances. Such patterns can be found in numerical solvers, as shown in our performance evaluation.

Exploiting recurring patterns in loops is a general concept. For example, Haghighat and Polychronopoulos [42] used this concept during induction variables analysis in paralellizing compilers. This paper describes the algorithms and the implementation that enable a similar concept for ADFA.

We first survey some of the prior work on ADFA frameworks. Gu et al. [14] described a symbolic ADFA framework for sequential programs and demonstrated its value for array privatization. Granston et al. [43] targeted parallel programs with *doall* constructs and presented an ADFA for detecting redundant shared array references.

Rus et al. [15] proposed an array *single static assignment* representation and used it for a classical ADFA in an automatic parallelization. Li et al. [44] described an ADFA for parallel programs written with *POSIX* threads [45] for optimizing thread-data locality while allocating cores on Chip Multiprocessors platforms. By contrast to our work, these approaches do not consider variant Gen and Kill sets analyzed across multiple instances.

In addition, this work has presented delayed symbolic evaluation. While the general concept of delaying inaccurate compiler analysis to runtime is not new, to the best of our knowledge, this concept has not been incorporated in prior ADFA frameworks. In particular, we claim that postponing performing a conservative operation that happens at a particular point in the data flow computation is counterintuitive. Instead of growing in complexity, expressions of postponed operations are simplified or performed when additional information is available in later analysis steps.

We now survey some of the prior work on ILP frameworks. Collard [46, 47] presented an early work for array section analysis in explicit parallel programs with strong or weak memory consistency models by expressing the relative execution order of threads in its analysis. Our work eliminates the need for expressing the execution order in the analysis by using a control flow graph that captures the implied execution order by the memory consistency model of the analyzed program.

The polyhedral model [48] is the current de facto ILP framework. Yuki et al. [49] used the polyhedral model for analyzing array section information in *X10* programs [37] with finish/asynch parallelism for the purpose of detecting race conditions. Similar to Collard's work, this analysis also takes the relative execution order across threads into account. Our work eliminates the need for considering the execution order for the aforementioned reason.

Bondhugula [41] used the polyhedral model and presented a compiler for automatic translation of sequential programs with affine loop nests into MPI. The polyhedral model enabled this work to reason about different instances of program statements. Our work is the same but builds on an ADFA. In the implementation, the ADFA framework takes advantage of ILP by sending queries to a stand-alone SMT solver without the need to expose the complexity of ILP to the compiler internal representation.

To the best of our knowledge, no ILP frameworks have been presented in the literature that start from OpenMP as the input program and analyze accessed array elements while considering the partitioning semantics of iteration and data spaces across threads.

7. EPILOGUE

7.1 Conclusions

The development of high-productivity programming environments that support the development of efficient programs on distributed-memory architectures is one of the most pressing needs in parallel computing today. Many of today's parallel computer platforms have a distributed memory architecture, as most likely will future multi-cores.

Despite many approaches to provide improved programming models, the state of the art for cluster platforms is to write explicit message-passing programs, using MPI. This process is tedious, but allows high-performance applications to be developed.

This dissertation showed the feasibility of allowing programmers to write computation-intensive algorithms in OpenMP while targeting middle-size clusters. This was achieved by presenting an automatic source-to-source translation scheme of OpenMP to MPI. OpenMP hides the complexity of data partitioning and explicit communication generation, allowing our approach to be accessible to the typical programmer.

A key feature of our work is translating OpenMP programs with regular write memory accesses and both repetitive *and* non-repetitive communication patterns. Prior translation schemes only covered OpenMP programs with repetitive communication patterns.

In order to generate efficient translated programs, this dissertation presented new compiler and runtime techniques that overcome limitations in prior work. Our compiler included an array data flow analysis framework that accounts for partitioning semantics of parallel loops using the π *operator*, which is an abstract representation that hides the complexity of partitioning andallows simple array section expressions during the dataflow computation. The compiler also included *delayed symbolic eval-*

uation, a compiler algorithm that ensures all operations are performed accurately during the dataflow computation. This is accomplished by representing conservative operations as simplified unevaluated expressions that get evaluated at runtime.

In addition, this dissertation showed that traditional ADFAs perform conservatively when confronted with loop nests that have variant Gen and Kill sets. Such issue appears when dealing with programs that have non-repetitive communication patterns. To overcome this issue, we presented the *variant-set* analysis, which allows traditional ADFAs to accurately represent and reason about variant Gen and Kill sets. Our solution is enabled by a common pattern that bounds the dataflow computation. In addition, we presented the *region-based* analysis that enables variant-set analysis to eliminate conservative effects from statements outside loop nests.

This dissertation also presented a runtime communication scheduling scheme that generates messages with low runtime overheads. By contrast to prior work, our scheduling scheme handles both repetitive and non-repetitive communication patterns.

With our contributions, we presented a fully automatic OpenMP-to-MPI translation system and evaluated its performance. On a cluster of 64 cores, our translator scaled six non-repetitive and four repetitive OpenMP benchmarks and achieved an average speedup of 3.8x over OpenMP on 8 cores. By comparison, a state-of-the-art translator only scaled the four repetitive benchmarks and obtained an average speedup of 3.3x.

We also compared the performance of our translation scheme against available hand-coded MPI and UPC programs. Those programs were tuned by hand to achieve high performance. The current performance of our translation scheme came within 54% and 60% of the performance of MPI and UPC programs, respectively.

Overall, this dissertation has advanced the state-of-the-art of cluster programming by allowing a new class of OpenMP applications to be efficiently and automatically scaled to clusters. In doing so, users can use OpenMP for a larger number of applications while targeting clusters. Performance results show that our approach is within $50 - 60\%$ of hand-coded distributed memory programs. To bridge this gap, we identified optimizations that can be combined with our translation scheme in future work, which we discuss in the next section.

7.2 Future Work

Several compiler analyses have been presented in the literature to promote the locality of accessed array elements in multi-threaded applications. Such analyses can be combined with our work to reduce communication. For example, Kwon [50] presented in his dissertation compiler analyses that adjust iteration spaces of parallel loops in an OpenMP program such that the effect of using the same threads to access the same array elements across the entire execution is increased. Using this analysis with our translation scheme can reduce the array elements that need to be communicated at barriers.

Another important extension of this work is supporting the translation of OpenMP programs with nested parallelism. By doing so, the compiler can apply multi-dimensional partitioning schemes where more than one loop is partitioned in loop nests. Multi-dimensional partitioning schemes have been used in hand-coded MPI and UPC programs to improve communication/computation ratio and therefore scalability.

Next, we describe two other extensions for our work and provide early result to show their feasibility. By doing these two extensions, the remaining benchmarks (LU and IS) in the NAS Parallel Benchmark suite that were not included in our performance evaluation can now be translated.

7.2.1 Pipeline Parallelism

Pipeline parallelism is an important technique used by programmers for parallelizing loops with carried dependencies that otherwise are sequential. Figure 7.1a shows an example of an OpenMP code with pipeline parallelism taken from NAS LU benchmark.

Our translation scheme can be extended to account for loops that have pipeline parallelism. This is accomplished by MPI *blocking* communication. As shown by the translated code in Figure 7.1b, send and receive messages can be used to both communicate data and force the synchronization implied by the pipeline paralellism of Figure 7.1a. In order to generate this code automatically, a compiler analysis is needed to identify the start and end points of the code region being executed using pipeline parallelism.

By hand, we apply the presented translation scheme for NAS LU benchmark and combine it with MPI blocking communication. The performance of the translated program is shown in Figure 7.2. This early result shows the feasibility of the proposed extension.

7.2.2 Runtime Inspection

Indirect memory accesses are generally not analyzable by compilers. This dissertation uses the Regular Section Descriptor representation for array sections, which represent indirect *read* memory accesses conservatively. As a result, communication that is relevant to these accesses may communicate array elements that are not actually needed. Figure 7.3 shows an example of an indirect read memory access that we obtain from NAS IS benchmark.

To improve accuracy, we propose using the runtime inspection technique proposed by Basumallik and Eigenmann [51]. At runtime, the actual array elements consumed by each thread are found by inspecting the indirection vector (see Figure 7.3). By providing this information to the runtime communication scheduler, precise communication sets can be computed.

By hand, we apply our translation scheme for IS and combine it with runtime inspection. The performance of the translated IS program is shown in Figure 7.4. This early result shows the feasibility of the proposed extension for the OpenMP-to-MPI translator.

```
#pragma omp parallel
{
  synch_left ( ) ;  // thread x is waiting to be enabled by thread x – 1

  #pragma omp for
  for ( j = jst; j <= jend; j++ )
      for ( i = ist; i <= iend; i++ )
          for ( m = 0; m < 5;  m++ )
              v [ k ][ j ][ i ][ m ] =  v [ k ][ j − 1 ][ i ][ m ] +
                                        v [ k ][ j ][ i − 1 ][ m ] +
                                        v [ k ][ j ][ i ][ m − 1 ]  + …

    synch_right ( ) ; // thread x finishes and enables thread x + 1
}
```

(a) The synchronization functions *synch_left* and *synch_right* are coded such that they ensure the loop nest in between (which has carried dependencies) is executed using pipeline parallelism.

```
// the current thread is x
  mpi_block_receive (sender= x − 1, receiver= x, data ) ;

   for ( j = lb_x; j <= ub_x; j++ )
       for ( i = ist; i <= iend; i++ )
           for ( m = 0; m < 5;  m++ )
               v [ k ][ j ][ i ][ m ] =  v [ k ][ j − 1 ][ i ][ m ] +
                                         v [ k ][ j ][ i − 1 ][ m ] +
                                         v [ k ][ j ][ i ][ m − 1 ]  + …

    mpi_send (sender= x, receiver= x + 1, data ) ;
```

(b) In the translated code, we propose to use the MPI blocking communication, which can be used to both communicate data and force synchronization.

Fig. 7.1.: An example of an OpenMP code taken from NAS LU benchmark.

IS benchmark also has other code versions that have irregular write accesses. In previous work, Min and Eigenmann [52] used runtime inspection and proposed advanced communication generation techniques for irregular write accesses in a software distributed shared memory system. Those techniques can be integrated with our translation scheme to handle OpenMP programs with irregular write array accesses.

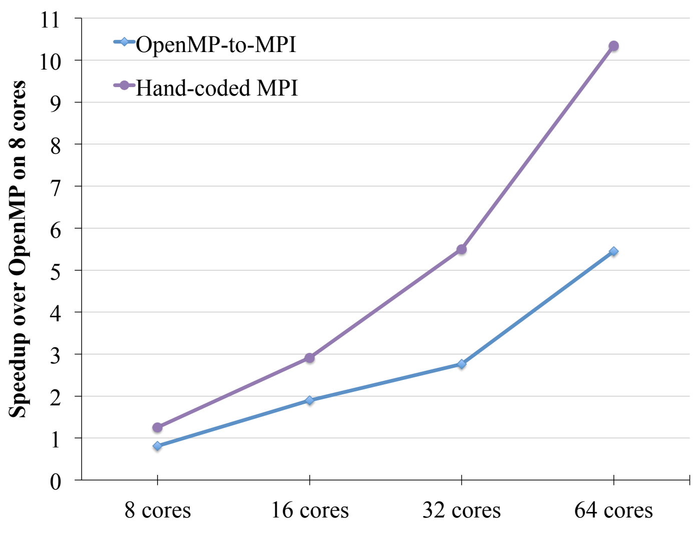


Fig. 7.2.: An early performance evaluation of the translated NAS LU benchmark. The translated code achieves 53% of the performance of the hand-coded MPI version of NAS LU.

```
#pragma omp for
 for (i = 0; i <= N; i++ )
      A[ i ] = …                    ;  // producers

#pragma omp for
 for (i = 1; i <= N-1; i++ )
       … = A[ ind_vec[ i ] ] ;  // consumers
```

Fig. 7.3.: Threads read array elements of A via the indirection vector *ind_vec*. As a result, the read Gen set is overestimated. We propose using the *runtime inspection* [51] technique to find precise descriptions of which elements were actually read by which threads.

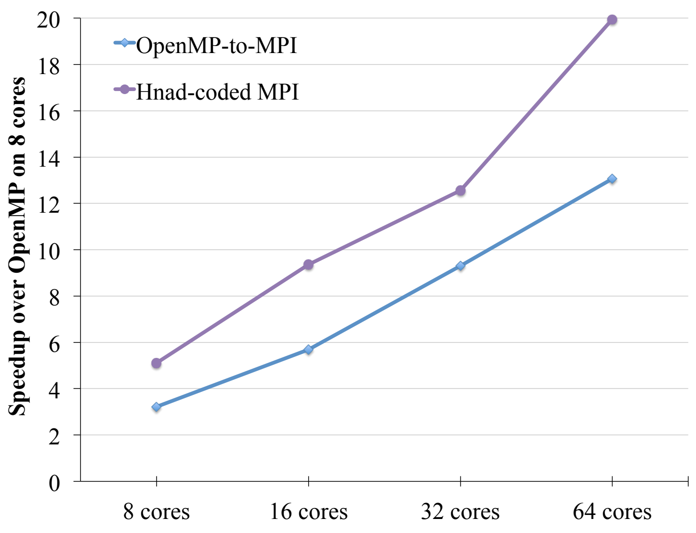


Fig. 7.4.: An early performance evaluation of the translated NAS IS benchmark. The translated code achieves 65% of the performance of the hand-coded MPI version of NAS IS.

LIST OF REFERENCES

LIST OF REFERENCES

[1] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference.* Cambridge, MA, USA: MIT Press, 1995.

[2] "OpenMP Application Programming Interface 4.0. Available: http://openmp.org."

[3] High Performance Fortran Forum, "High Performance Fortran language specification, version 2.0," tech. rep., 1997.

[4] UPC Consortium, "UPC Language Specifications, version 1.2," tech. rep., Lawrence Berkeley National Laboratory, 2005.

[5] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel, "TreadMarks: Shared Memory Computing on Networks of Workstations," *IEEE Computer*, vol. 29, pp. 18–28, Feb. 1996.

[6] J. P. Hoeflinger, "Extending OpenMP to Clusters." White Paper, 2006.

[7] S. Dwarkadas, A. L. Cox, and W. Zwaenepoel, "An Integrated Compile-Time/Run-Time Software Distributed Shared M emory System," in *Proc. of the 7th Symposium on Architectural Support for Progra mming Languages and Operating Systems (ASPLOS)*, pp. 186–197, 1996.

[8] S. Min and R. Eigenmann, "Combined Compile-time and Runtime-driven, Pro-active Data Movement i n Software DSM Systems," in *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time support for scalable systems*, pp. 1–6, 2004.

[9] P. J. Keleher and C.-W. Tseng, "Enhancing Software DSM for Compiler-Parallelized Applications," in *Proceedings of the 11th International Symposium on Parallel Processing (IPPS)*, (Washington, DC, USA), pp. 490–499, IEEE Computer Society, 1997.

[10] O. Kwon, F. Jubair, S.-J. Min, H. Bae, R. Eigenmann, and S. Midkiff, "Automatic Scaling of OpenMP Beyond Shared Memory," in *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, (Fort Collins, CO, USA), pp. 75–84, 2011.

[11] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff, "A Hybrid Approach of OpenMP for Clusters," in *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming (PPoPP)*, (New Orleans, Louisiana, USA), pp. 75–84, 2012.

[12] A. M. Kouadri-Mostéfaoui, D. Millot, C. Parrot, and F. Silber-Chaussumier, "Prototyping the Automatic Generation of MPI Code from OpenMP Programs in GCC," in *International Workshop on GCC Research Opportunities (GROW)*, Paphos, Cybrus, 2009.

[13] D. Millot, A. Muller, C. Parrot, and F. Silber-Chaussumier, "STEP: A Distributed OpenMP for Coarse-grain Parallelism Tool," in *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism (IWOMP)*, pp. 83–99, 2008.

[14] J. Gu, Z. Li, and G. Lee, "Symbolic Array Dataflow Analysis for Array Privatization and Program Parallelization," in *Proceedings of the ACM/IEEE Conference on Supercomputing (SC)*, 1995.

[15] S. Rus, G. He, C. Alias, and L. Rauchwerger, "Region Array SSA," in *Proceedings of the 15th international conference on Parallel architectures and compilation techniques (PACT)*, (Seattle, Washington, USA), 2006.

[16] F. Darema, D. A. George, V. A. Norton, and G. F. Pfister, "A Single-Program-Multiple-Data Computational Model for Epex/Fortran," *Parallel Computing*, 1988.

[17] F. Jubair, O. Kwon, R. Eigenmann, and S. Midkiff, "π Abstraction: Parallelism-Aware Array Data Flow Analysis for OpenMP," in *Proceedings of the 27th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, (Hillsboro, OR, USA), 2014.

[18] P. Havlak and K. Kennedy, "An Implementation of Interprocedural Bounded Regular Section Analysis," *IEEE Transactions on Parallel and Distributed Systems*, pp. 350–360, 1991.

[19] A. Basumallik and R. Eigenmann, "Incorporation of OpenMP Memory Consistency Into Conventional Dataflow Analysis," in *Proceedings of the 4th international conference on OpenMP in a new era of parallelism (IWOMP)*, (West Lafayette, IN, USA), pp. 71–82, 2008.

[20] S. Satoh, K. Kusano, and M. Sato, "Compiler optimization techniques for openmp programs," *Scientific Programming*, pp. 131–142, 2001.

[21] H. Bae, D. Mustafa, J.-W. Lee, Aurangzeb, H. Lin, C. D. R. Eigenmann, and S. Midkiff, "The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation," *International Journal of Parallel Programming (IJPP)*, pp. 1–15, 2012.

[22] The NAS Parallel Benchmarks suite version 3.3.

[23] J. Christ, J. Hoenicke, and A. Nutz, "SMTInterpol: An Interpolating SMT Solver," in *Proceedings of the 19th International Conference on Model Checking Software (SPIN)*, (Oxford, UK), pp. 248–254, 2012.

[24] OmpSCR Repository [version 2.0].

[25] The Polyhedral Benchmark suite version 3.2.

[26] UPC NAS Parallel Benchmarks suite version 2.4.

[27] S. Min, A. Basumallik, and R. Eigenmann, "Optimizing OpenMP programs on Software Distributed Shared Memory Sys tems," *International Journal of Parallel Programming*, vol. 31, no. 3, pp. 225–249, 2003.

[28] A. Basumallik and R. Eigenmann, "Towards Automatic Translation of OpenMP to MPI," in *Proceedings of the 19th Annual International Conference on Supercomputing (ICS)*, pp. 189–198, 2005.

[29] M. Gupta and P. Banerjee, "PARADIGM: A Compiler for Automatic Data Distribution on Multicomputers," in *Proceedings of the 7th International Conference on Supercomputing (ICS)*, pp. 87–96, 1993.

[30] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, D. Shields, K.-Y. Wang, W.-M. Ching, and T. Ngo, "An HPF Compiler for the IBM SP2," in *Proceedings of the 1995 ACM/IEEE Conference on Supercomputing (SC)*, 1995.

[31] S. Hiranandani, K. Kennedy, and C.-W. Tseng, "Compiling Fortran D for MIMD Distributed-memory Machines," *Communications of the ACM*, vol. 35, pp. 66–80, 1992.

[32] L. Huang, B. Chapman, and R. Kendall, "OpenMP for Clusters," in *In The Fifth European Workshop on OpenMP (EWOMP)*, pp. 22–26, 2003.

[33] Z. Liu, L. Huang, B. Chapman, and T.-H. Weng, "Efficient Implementation of OpenMP for Clusters with Implicit Data Distribution," in *Proceedings of the 5th International Conference on OpenMP Applications and Tools: Shared Memory Parallel Programming with OpenMP (WOMPAT)*, pp. 121–136, 2005.

[34] D. Eachempati, L. Huang, and B. Chapman, "Strategies and Implementation for Translating OpenMP Code for Clusters," in *Proceedings of the Third international conference on High Performance Computing and Communications (HPCC)*, pp. 420–431, 2007.

[35] J. Nieplocha, R. J. Harrison, and R. J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-performance Computers," *The Journal of Supercomputin*, vol. 10, pp. 169–189, June 1996.

[36] R. W. Numrich and J. Reid, "Co-array Fortran for Parallel Programming," *SIGPLAN Fortran Forum*, vol. 17, pp. 1–31, 1998.

[37] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: An Object-Oriented Approach to Non-uniform Cluster Computing," in *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2005.

[38] K. A. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. N. Hilfinger, S. L. Graham, D. Gay, P. Colella, and A. Aiken, "Titanium: A high-performance java dialect," *Concurrency - Practice and Experience*, vol. 10, no. 11-13, pp. 825–836, 1998.

[39] T. A. El-Ghazawi and S. Chauvin, "UPC Benchmarking Issues," in *Proceedings of the International Conference on Parallel Processing (ICPP)*, pp. 365–372, 2001.

[40] F. Cantonnet, Y. Yao, M. M. Zahran, and T. A. El-Ghazawi, "Productivity Analysis of the UPC Language," in *In the 18th International Parallel and Distributed Processing Symposium*, 2004.

[41] U. Bondhugula, "Compiling Affine Loop Nests for Distributed-memory Parallel Architectures," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, pp. 33:1–33:12, 2013.

[42] M. R. Haghighat and C. D. Polychronopoulos, "Symbolic Analysis for Parallelizing Compilers," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, pp. 477–518, 1996.

[43] E. D. Granston and A. V. Veidenbaum, "Combining Flow and Dependence Analyses to Expose Redundant Array Accesses," *International Journal of Parallel Programming (IJPP)*, pp. 423–470, 1995.

[44] Y. Li, A. Abousamra, R. Melhem, and A. K. Jones, "Compiler-Assisted Data Distribution for Chip Multiprocessors," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*, (Vienna, Austria), pp. 501–512, 2010.

[45] D. R. Butenhof, *Programming with POSIX threads.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[46] J.-F. Collard, "Array SSA for Explicitly Parallel Programs," in *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pp. 383–390, 1999.

[47] J.-F. Collard and M. Griebl, "Array Dataflow Analysis for Explicitly Parallel Programs," in *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pp. 406–413, 1996.

[48] P. Feautrier, "Automatic Parallelization in The Polytope Model," in *The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications*, (London, UK), pp. 79–103, 1996.

[49] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat, "Array Dataflow Analysis for Polyhedral X10 Programs," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, (Shenzhen, China), pp. 23–34, 2013.

[50] O. Kwon, *Automatic scaling of OpenMP applications beyond shared memory.* PhD thesis, Purdue University, School of Electrical and Computer Engineering, August 2013.

[51] A. Basumallik and R. Eigenmann, "Optimizing Irregular Shared-memory Applications for Distributed-memory Systems," in *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*, pp. 119–128, 2006.

[52] S.-J. Min and R. Eigenmann, "Optimizing Irregular Shared-Memory Applications for Clusters," in *Proceedings of the ACM International Conference on Supercomputing (ICS)*, pp. 256–265, 2008.

VITA

VITA

Fahed Jubair earned his B.Sc. in Computer Engineering from University of Jordan in August 2006. On August 2007, Fahed joined the graduate school of Electrical and Computer Engineering at Purdue University where he earned his M.Sc. degree on May 2009. Starting August 2009, Fahed joined the Paramount research laboratory where he continued his studies under the supervision of professor Rudolf Eigenmann and obtained a Ph.D. degree on December 2014. In his research, Fahed worked on developing compiler and runtime techniques that extend the ease of use of shared-address space programming, written in OpenMP, to clusters. Fahed was also involved in the development of Cetus Compiler Infrastructure.