KANOR: A LANGUAGE FOR DECLARATIVE COMMUNICATION

Nilesh Mahajan

Submitted to the faculty of the University Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Mathematics,

Indiana University

October 5, 2016

v0.1
Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.
Doctoral Committee

Arun Chauhan, Ph.D.

Andrew Lumsdaine, Ph.D.

Randall Bramley, Ph.D.

Martin Swany, Ph.D.

October 14, 2016
DEDICATION

To my parents and my brother.
Nilesh Mahajan

KANOR: A LANGUAGE FOR
DECLARATIVE COMMUNICATION

Writing efficient parallel programs continues to be a challenge for the programming community. Large-scale parallel programs are usually coded using the single program multiple data (SPMD) model in which processes communicate by sending explicit messages using a standardized and portable Message Passing Interface (MPI) library. Writing efficient MPI programs often requires a deep understanding of how a parallel program works and forces programmers to compromise readability by strewing communication primitives all over the unrelated computational code.

This thesis describes a domain-specific language (DSL), called Kanor, that takes a different approach. Kanor allows programmers to specify communication patterns at a high level, in Bulk Synchronous Parallel (BSP) style. The semantics of the language are carefully defined to guarantee correctness properties, such as deadlock freedom and determinism, while allowing efficient execution. The language is highly expressive, able to succinctly describe all the existing MPI collective operations, and allowing users to create their own custom collectives that could be detected and optimized. The BSP style of Kanor also makes it amenable to source-level optimizations that are well understood, including those that exploit shared memory for efficient intra-node communication.
We start by describing the syntax and semantics of Kanor and discuss certain properties of Kanor programs, such as deadlock freedom and determinism. Next, we present an implementation of Kanor, that is embedded in C++ and provided as a library, along with runtime optimizations. In addition, we present an optimizing transformation of Kanor programs that tries to overlap communication with computation. Next, we present a backend for Kanor that uses shared memory to reduce buffer copies while communicating between processes. We conclude by discussing future directions for Kanor that include heterogeneous backends and optimizing Kanor programs for irregular domains such as graph algorithms.

----------------------------------

Arun Chauhan, Ph.D.
(Dissertation Chair)

----------------------------------

Andrew Lumsdaine, Ph.D.

----------------------------------

Martin Swany, Ph.D.

----------------------------------

Randall Bramley, Ph.D.
CONTENTS

DEDICATION v

1 INTRODUCTION 5
   1.1 IMPLICIT VS EXPLICIT COMMUNICATION ...................... 6
   1.2 LIMITATIONS OF MPI ........................................... 7
   1.3 KANOR .......................................................... 9
   1.4 THESIS STATEMENT .............................................. 13
   1.5 THESIS ORGANIZATION ........................................... 13

2 BACKGROUND AND RELATED WORK 15
   2.1 AUTOMATIC PARALLELIZATION ................................. 16
   2.2 BULK SYNCHRONOUS PARALLELISM ............................ 17
   2.3 DOMAIN SPECIFIC LANGUAGES ................................. 18
   2.4 PROGRAMMER ANNOTATIONS ..................................... 19
   2.5 LANGUAGE FEATURES ........................................... 20

3 KANOR DESIGN 25
   3.1 KANOR DESIGN CHOICES ....................................... 25
      3.1.1 PROGRAMMING MODEL ..................................... 25
LIST OF FIGURES

1.1 MPI Send Receives introduce control flows. ......................... 8
1.2 Example of a communication statement. .............................. 10
1.3 Reduction using MPI; even and odd processors perform different reductions. .................................................. 11
1.4 Reduction is expressed much more concisely and clearly in Kanor. ... 12

3.1 Formal syntax of Kanor. A communication statement expressed in Kanor is represented by the non-terminal, CommStmt. ............... 28
3.2 MPI collectives encoded in Kanor. In “One-to-All”, blk refers to the size of rb. ......................................................... 30
3.3 Formal syntax of $K_T$. ...................................................... 31

4.1 VarWrapper constructor and getter/setter types ...................... 42

5.1 Global Case Inference .................................................. 51
5.2 Comparing Kanor implementations of MPI collectives to their counterparts for 32 processes. ........................................... 53
5.3 Comparison of Kanor to MPI implementations of NAS Benchmarks
Integer Sort (IS) and Fourier Transform (FT). NAS Benchmark class
(S, W, A, B, C) denotes the size of the problem to be solved. Procs
denote the number of processes used to solve the problem.

5.4 Cholesky: Kanor vs MPI.

6.1 Overlap Example

6.2 CommBuff get method Example

6.3 Non-Blocking Communication Safety

6.4 Sweep3D kernel in Kanor

6.5 Piplined Sweep3D kernel in Kanor

7.1 Local Writes Case

7.2 A simple example of local writes into a globalized variable.

7.3 CFG of a hypothetical example showing writes into a globalized variable within complex control flow. Dotted path is possible only with unstructured control-flow, such as goto.

7.4 Examples CFGs illustrating the subtleties involved in computing locking sets.

7.5 Example for which locking set is not dynamically minimal.

7.6 C++-like pseudocode for smart synchronization. Localized is a per-process Boolean valued hash table.

7.7 Communication benchmarks in Kanor and their equivalent MPI and shared memory code generated by Kanor compiler.
7.8 Experimental evaluation of some of the commonly occurring collective communication patterns, compared to MPI.
LIST OF TABLES

3.1 Local semantics in $K_T$. ................................................. 33
3.2 Communication semantics in $K_T$. ................................. 34
CHAPTER 1

INTRODUCTION

Over the past few decades, technological changes such as parallel hardware (multicores, Graphics Processing Units) and the need to process large amounts of data, require programmers to write parallel programs. The task of reasoning about the issues of correctness and efficiency of parallel programs is substantially more complicated compared to sequential programs. A spectrum of programming language approaches exists towards simplifying this task, ranging from fully automatic (where the compiler, runtime manage all the parallelism details) to fully manual (where there is minimal compiler, runtime support and the programmer is responsible for the correctness, performance of the program). Even with a plethora of these approaches, writing efficient parallel programs continues to be a challenge for the programming community.
1.1 IMPLICIT VS EXPLICIT COMMUNICATION

One of the many decisions the language designer has to make is whether the communication between different computing entities (we call them processes) is explicitly specified by the programmer or if the communication is implicit and generated by language compiler and runtime.

The implicit approach to communication is exemplified by Partitioned Global Address Space (PGAS) languages such as HPF [50], UPC [20], Co-array Fortran [64] and more recently X10 [71] and Chapel [17]. These languages expose the memory hierarchy, to varying degrees, to the programmer while still providing a uniform means of accessing local and non-local data. This approach delivers parallelism by composing several well-understood sequential components and the programmer does not have to worry about communication details like placement of the communication calls, serialization of complex data structures, buffering of data etc. The programmer can provide hints describing the partitioning of data among processes. These data partitions can be predefined or provided by the programmer. The compiler can make use of these hints to generate efficient communication code. This approach starts from a shared view of data and tries to approach performance comparable to the explicit message passing approach with the use of compiler, runtime. This implicit approach to communication fails to exploit the underlying structure of the communications between sequential components. Naively writing PGAS code might lead to serious performance hits and the programmer needs to implement further tricks to get acceptable performance [83].
The de facto standard for explicit communication is the Message Passing Interface (MPI) [60], which is typically used to program large-scale computers. Large-scale parallel computers consist of many nodes connected by a high-performance network. Parallel programs written for these machines are usually coded using the partitioned address space model in which processes have separate address spaces and communicate by sending explicit messages using a standardized MPI library, which provides a highly portable interface. MPI provides communication primitives that the programmer uses to specify communication pattern. The program is written in Single Program Multiple Data (SPMD) style, multiple processes executing the same program on different data. The programmer is responsible for the management of this distributed nature of the data structures as well as other performance issues like load balancing, message sizes etc. MPI has been a successful library based solution widely adopted across the high performance computing community to solve many scientific problems and owes its success to a variety of factors [38].

1.2 LIMITATIONS OF MPI

Although MPI has been successful and widely adopted, writing correct and efficient MPI programs requires considerable programming skill. The interface offered by MPI is too low-level and often requires users to sacrifice abstraction by forcing them to delve into the system details. As a result, programming style is crippled and the benefits of higher-level abstractions such as type-safety and genericity are lost. The set of communication patterns, provided as collectives in MPI, is limited. Hence, programmers are forced to resort to point-to-point send/receive calls to encode the
communication pattern. Although more general, send/receives introduce additional control and data flows within the SPMD program as shown in figure 1.1. The programmer as well as the compiler has to keep in mind these additional control flows while reasoning about the parallel program. The point-to-point messaging style of parallel programming has been compared to goto statements in sequential programming and considered harmful [36].

A consequence of this programming style is that the programs become unnecessarily complicated and lack expressiveness. Additionally, since MPI does not see the communication pattern that is deeply embedded within the point-to-point operations, opportunities for optimization are lost. Another important drawback of MPI and many other approaches mentioned above is their failure to provide an uniform interface to exploit both fine-grained and coarse-grained parallelism. Often, MPI users rely on OpenMP to benefit from the largely ubiquitous multi-core
processors. This forces two separate models of parallelism where one would have sufficed. Projects such as MPI.net [37], Boost MPI [9], and OOMPI [74] have tried to alleviate this shortcoming by attempting to provide a higher-level interface. However, the performance drops sharply when the high-level constructs can not be expressed through MPI primitives.

### 1.3 KANOR

Implicit approaches are easier to program, but have more opaque performance characteristics, and thus their performance is harder to predict or tune. The explicit approach taken by MPI is more difficult to program, requiring communication to be specified at a very fine-grained level of detail and thus leading to more errors, but allow more knowledge and control over a programs behavior and thus its performance.

Kanor takes a different approach. It is a domain-specific language (DSL) that allows programmers to specify communication patterns *declaratively*, at a high level, in Bulk Synchronous Parallel (BSP) style [78]. The semantics of the language are carefully defined to guarantee correctness properties, such as deadlock freedom and determinism, while allowing efficient execution. The language is highly *expressive*, able to succinctly describe all the existing MPI collective operations, and allowing users to create their own custom collectives that could be detected and optimized. The BSP style of Kanor also makes it amenable to source-level optimizations that are well understood [26], including those that exploit shared memory for efficient intra-node communication [47].
As an example of a communication statement consider the Kanor code in Fig. 1.2.
The statement updates the variable rval at a Kanor process denoted by root. Only
the processes with even process IDs send value stored in the variable sval. The
receiver process set consists of a single process (root) and the sender process set
consists of processes with even IDs, assuming that Topology defines integral type
process IDs. The sender process set is formed with the help of Kanor constructs
.for_each and .if, and .with specifies the topology. Finally, only the memory
location rval[sndr] is updated with the value receiver from sndr.

```
Topology topology;
rval[sndr].at_root << sval.at sndr |
.for_each(sndr, topology.world) &
.if (sndr % 2 == 0) .with topology;
```

Figure 1.2: Example of a communication statement.

As a slightly more realistic example, consider the MPI code in
Fig. 1.3 where separate reductions are performed by even and odd
processes with the sender processes sending different values. The
functions MPI_Isend and MPI_Recv perform non-blocking send and blocking receive,
respectively. This pattern can be written in other ways in MPI but the code will end
up either using send and receive calls or MPI derived types and subcommunicator
manipulations. Even with the new neighborhood collectives in MPI-3 standard, this
pattern cannot be represented as a single function call. The programmer must also
make sure the sent and received messages match and there is no deadlock. Various
MPI implementations manage the temporary buffers differently so the deadlock bug
might show up in some and not others. Finally, a better algorithm might be
implemented with better knowledge of the communication pattern.

In contrast, in order to express this communication in Kanor, process sets, messages
int rval;
std::vector<int> sbuff;
...
std::vector<MPI_Request> reqs;
int rmdr = me % 2;
for (int i = 0; i < nprocs; i++) {
    if (i % 2 == rmdr) {
        MPI_Request req;
        MPI_Isend(&sbuff[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD, &req);
        reqs.push_back(req);
    }
}
MPI_Waitall(reqs.size(), reqs.data(), MPI_STATUSES_IGNORE);
for (int i = 0; i < nprocs; i++) {
    if (i % 2 == rmdr) {
        int r;
        MPI_Recv(&r, 1, MPI_INT, i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        rval += r;
    }
}

Figure 1.3: Reduction using MPI; even and odd processors perform different reductions.

sizes, and memory locations can all be specified in a single communication statement. The operation can be viewed as a single parallel assignment of receiver memory locations by the senders. Kanor constructs are similar to list comprehensions seen in languages like Python and Haskell. The example communication in Fig. 1.3 can be expressed in Kanor succinctly, as shown in Fig. 1.4.

In brief Kanor addresses the following issues:

- **Smooth Learning Curve** Kanor is not a revolutionary new language like Chapel,
int rval;
kanor::CommBuff<int> sbuff;
...
Topology t;
rval _at_ rcvr << std::plus<int>() << sbuff[rcvr] _at_ sndr |
_for_each(sndr, t.world) & _for_each(rcvr, t.world) &
_if ((sndr % 2) == (rcvr % 2)) _with t & GLOBAL;

Figure 1.4: Reduction is expressed much more concisely and clearly in Kanor.

X10. The programmers familiar with C, C++, MPI should not face a steep
learning curve. This also provides a path to incremental adoption of Kanor. By
adhering to existing tools, languages and paradigms, Kanor simplifies the
deployment of the solution while delivering optimized performance.

• Correctness Kanor communication statements provide a logical barrier for the
communicating processes. Explicitly specifying communication avoids the pitfalls
like data races associated with shared memory programming. The programmer
does not use complicated protocols employing locks, condition variables to ensure
correctness.

• Performance Kanor forces the programmer to write in the BSP style. Explicit
specification of communication also means the compiler need not make heroic
attempts to generate efficient code thus simplifying its design. By separately
specifying communication and computation, Kanor is able to deliver better
performance on hierarchical memory machines while providing a uniform
communication interface.

With this background, we now present the thesis statement.
1.4 THESIS STATEMENT

Declarative specification of communication can help programmers write correct and efficient parallel programs. Kanor is an evolutionary approach that helps us achieve this goal.

1.5 THESIS ORGANIZATION

The rest of this dissertation is organized as follows:

- **Background and Related Work** This chapter describes approaches towards simplifying parallel programming and relates them to Kanor.

- **Kanor Design** This chapter explains the rationale behind some of the design choices in Kanor. Next, it explains the syntax with multiple examples and a grammar. It also gives an operational semantics for a simplified subset of the language. Finally, it discusses correctness properties satisfied by Kanor programs.

- **Kanor Implementation** This chapter presents an overview of our implementation of Kanor embedded in C++. It explains key abstractions provided by Kanor that help programmers express the communication pattern.

- **Runtime Optimizations** This chapter describes our implementation of Kanor embedded in C++. Kanor runtime optimizations are discussed and comparison with equivalent MPI codes is presented.

- **Compute Communicate Overlap** This chapter presents compiler and runtime approaches that help Kanor overlap computation with communication.
Correctness of the approaches and state of the implementation is discussed also discussed.

• **A Shared Memory Backend** This chapter presents a backend for Kanor that uses shared memory to reduce buffer copies while communicating between processes and shows comparison with MPI.

• **Conclusions and Future Work** This chapter summarizes the thesis and discusses possible directions to be pursued that build on top of the current work.
Writing correct parallel programs, that use hardware resources efficiently, is a challenging task. Reasoning about the control and data flows in parallel programs is a much more complex task compared to sequential programs. Due to non-linearity of the control flow, the number of different orderings of the instructions might be exponential in the number of instructions in the program. Further, an instruction or statement in a higher level language might be translated to a number of instructions at the hardware level which can be interleaved. If not managed, this complexity requires the programmer to reason about a large number of interleavings of the program instructions. Typically, programmers end up sequentializing large fragments of the program to guarantee correctness. Unfortunately, this results in underutilization of the parallel hardware and subpar performance.

With modern hardware that includes multicores and specialized units such as GPUs, parallel programming has become even more challenging. Programmers need
to learn the subtleties of different architectures to make full use of the hardware.

The large amount of literature on algorithms [32, 65, 62] tuned for GPU architectures is an example of this trend. The scale of the machine also affects the kind of issues programmers need to worry about. For example, typically fault tolerance is not an issue for a machine running four cores but becomes an issue at higher scales. Thus hardware heterogeneity adds another layer to the complexity of programming parallel machines.

Several approaches have been tried over the years to make parallel programming easier, spanning a wide range of solutions that includes automatic parallelization, domain specific languages, new parallel languages etc. In this chapter we will discuss some of the approaches from this parallel programming landscape and how they relate to Kanor.

### 2.1 AUTOMATIC PARALLELIZATION

The goal of automatic parallelization is to convert a program written in a sequential style to a parallel one without minimal if any programmer intervention. The compiler/runtime takes the responsibility to generate a correct and efficient parallel program for different hardware configurations. It is difficult to statically estimate if a certain compiler transformation will benefit performance. Also, the compiler has to search a large space of valid target programs, which is typically exponential in the size of the input program. The selected target program might not make optimal use of different hardware resources such as GPUs. Compiler transformations used for automatic parallelization are typically complex and limited to a subset of valid
programs. Due to these difficulties, the automatic parallelization approach has seen limited use, especially in large-scale parallel programming community.

Some of the recent approaches to automatically generating code for distributed memory involve the use of polyhedral compilation methods [69, 27]. Inspector-Executor transformations have also been tried to automatically parallelize irregular codes [76]. Jablin et.al. [46] use a run-time library to simplify compiler transformations.

Kanor asks the programmer to manage parallelization details such as distribution of data among processes, what data to communicate and to whom etc. Kanor does not rely on complicated compiler transformations while simplifying expression of communication patterns.

Due to the limits on auto-parallelization, programmers explicitly specify parallelism details in the form of hints, library calls, new language features etc. We discuss some of the ways in which programmers can control the parallelism details in the following sections.

2.2 BULK SYNCHRONOUS PARALLELISM

The Bulk Synchronous Parallel (BSP) model was originally proposed by Valiant [78] as a bridging model for parallel computation. It was intended as a model between hardware and higher level programming models. A program written in BSP style consists of a sequence of supersteps. Each superstep at a process consists of some combination of local computation, message transmissions and (implicitly) message arrivals from other processes. The communication is intended to be one-sided
(\texttt{bsp\_get()} and \texttt{bsp\_put()}) but can be implemented using two-sided libraries like MPI. The local computation steps on each process run independently of each other. There is an implicit barrier at the end of each superstep which does a global check if all the processes have completed the current superstep.

Several implementations of the BSP model have been provided such as BSPLib [41], BSPonMPI, Oxford BSP toolset, MulticoreBSP for C [82] among others. Recent programming systems such as MapReduce, graph libraries such as Pregel [56], Apache Giraph are inspired by the BSP model.

Kanor combines the communication and barrier stages into one statement. This makes reasoning about the program much easier and also represents the communication pattern in a single statement. Further, Kanor communication statements are powerful so that programmers can specify complicated communication patterns in a single statement instead of multiple calls to \texttt{bsp\_get()} and \texttt{bsp\_put()} methods mixed with other computation in a superstep. Kanor runtime and compiler try to overlap computation and communication to minimize overheads associated with the global synchronization step while preserving BSP semantics.

### 2.3 DOMAIN SPECIFIC LANGUAGES

Domain specific languages (DSLs) take a restricted view of the world as opposed to general languages and are focused on a particular problem domain [30, 33]. This restriction to a specific domain simplifies reasoning about parallelism and the compiler/runtime can use well-known strategies specific to the domain to
automatically parallelize the program. The MapReduce framework [28] is an important example of this paradigm. In this framework, the programmer defines two functions, map that takes a key and a value as input and produces a list of key-value pairs, and reduce that takes a key and a list of associated values as input and outputs a list of values. The MapReduce framework is responsible sorting, grouping, and routing results from the mappers to reducers. The framework also takes care of distributed computing issues such as scheduling of mappers and reducers on physical machines as well as fault tolerance. Languages and libraries that work on stencil computations are another important class of DSLs. Some examples of stencil DSLs include Pochoir [77], LibGeoDecomp [73], Physis [58] etc. Kanor works on the domain of programs written with SPMD model where communication is done in BSP style. So Kanor is not restricted to a particular domain like stencil computations but requires the programmer to worry about certain aspects of parallelism, thus simplifying the task of compiler/runtime.

2.4 PROGRAMMER ANNOTATIONS

OpenMP [25] programs are sequential programs with statements or block of statements annotated with pragma directives to help the compiler identify parallelism in the program. Mainstream compilers supporting OpenMP, such as gcc and icc, generate multithreaded as well as task-parallel code for multicores. Researchers [59, 7, 80] have tried generating code for distributed memory architectures from OpenMP programs. These efforts have seen limited use in the HPC community which uses hybrid MPI+OpenMP approach [67] to program for
distributed memory architectures.

Kanor requires simple annotations from programmer specifying the nature of communication. The annotations only need to be provided in case the compiler is not able to figure out certain properties of the communication statement.

2.5 LANGUAGE FEATURES

Partitioned Global Address Space (PGAS) languages, such as UPC [20], Co-array Fortran [64], X10 [71], and Chapel [17], provide programmers with explicit control over data placement, but still use shared-memory-like semantics with implicit communication. They distinguish between references to local vs. remote memory, however; the earlier High-Performance Fortran (HPF) language was similar to PGAS, but without this explicit distinction [50]. In this section, we will describe some of these languages in brief and compare them with Kanor.

Fortran has remained the preferred language for development of scientific applications. Co-Array Fortran is an extension to Fortran 95/2003. It has syntax for specifying data distribution which is very Fortran like. The coarray syntax lets each object image access remote images. The Coarray syntax is a visual flag for both the programmer and compiler that some communication is imminent and use the hint for optimization. All data and computations in co-array fortran are local and all communication is one-sided through libraries like ARMCI or GASNet. Kanor’s remote references are inspired by Co-Array Fortran.

High performance Fortran was heavily influenced by Fortran D and Vienna Fortran which were the data parallel versions of Fortran. HPF 1.0 had support of data
distribution, data parallel execution, extended intrinsic functions and a library for special operations like sum reduction, gather and scatter. The user is responsible for finding an appropriate data distribution for the application. It has the owner-compute rule which suggests that the owner of the L.H.S is responsible for computing the R.H.S. The communication is implicit. Various directives help in optimizations e.g. collocation of data and partitioning of data to create the location abstraction. HPF 2.0 was more "declarative" in the sense that the "ON" directive gave user the a way of overriding the owner-compute rule and specify the location of computation. Support for nonblocking execution was introduced, but communication was still implicit.

UPC is a language designed for writing high-performance applications for large parallel machines. The variables are associated with particular processors, but the user is provided with view of single thread, shared partitioned address space. Remote references are translated to communication calls implemented in a library. Thus communication in UPC is not explicit.

X10 is an Asynchronous PGAS (APGAS) language in the Java family. It is designed to provide higher level of abstractions to write concurrent programs, to exploit non-uniform data access, portable across multiple architectures and varied workloads. Some key concepts in X10 include:

- **Places**: correspond loosely to process or processor.

- **Activities**: An activity is a statement being executed, independently, with its own local variables; it may be thought of as a very light-weight thread.

- **Typing**: Typing is a substantial extension of Java typing. Generic types, function
types, inference etc. are provided. Dependent types are used to check locality properties.

- Arrays: Arrays are defined using distributions, regions and points.

Chapel is a new language that provides programmers with high level abstractions to write parallel code. You can write data, task, nested parallel programs (and any combinations of those). They also provide control over locality. Key concepts in Chapel include:

- Task Parallelism: Keywords like `begin` and `sync` are provided for task creation. Using `cobegin` and `coforall` you can invoke multiple tasks in a structured manner.

- Locality: `Locales` provide a way to reason about locality similar to `places` in X10.

- Data Parallelism: Arrays are defined using `domains`. Domains are integer sets that support iteration and various other operations. They can also be distributed across multiple locales. Built-in as well as user-defined scans and reductions over arrays are provided.

- Distributions: Domain maps define ownership of domain indices. `Layouts` are single-locale domain maps and `distributions` are multi-locale. User defined distributions as well as well-known ones such as block, cyclic etc. are supported.

These languages differ from Kanor in several ways: they provide a global address space, they do not allow (or expect) programmers to specify communication explicitly, they do not segregate communication from computation, and they do not provide collective semantics for general communications. Kanor, on the other hand,
exposes a distributed address space without direct access to remote memory; communication operations must be specified explicitly, rather than implicitly through a memory consistency model. Kanor also has separate communication blocks, separated from an applications computation. These communication operations are collective, matching the BSP model often used in message-passing programs.

Erlang is a declarative programming language designed for developing highly scalable, reliable concurrent applications. An overview of the development of Erlang is given in [4]. After writing telecom software in many languages, the designers found that the versions written in declarative languages tended to be much shorter and easier to understand than those written using traditional imperative languages. Early versions of Erlang were prototyped in Prolog, but Erlang has since developed into a standalone language. Erlang supports concurrency by providing language support for easily spawning processes and sending messages between processes. Although Erlang is a declarative, functional programming language, its message passing abstractions resemble those in imperative languages. Programmers write explicit sends and receives. Erlang does not use the Single Program, Multiple Data model that Kanor uses, and thus the notion of declaratively specifying communication patterns is absent. It may, however, be possible to emulate many of Kanor’s idioms using Erlang’s list comprehensions and other declarative features. Data Parallel Haskell (DPH) [16] adds parallel arrays to Haskell along with operations on parallel arrays, e.g., fold. DPH lacks mechanisms to send messages explicitly, and lacks X10- or Chapel-like constructs to express locality. However, it supports nested parallelism, similar to NESL [8], which is a nested data-parallel
XcalableMPs [81] gmove construct specifies collective communications as
pragmas using concepts similar to those in Kanor. However, a single gmove
statement cannot perform data reductions and covers only the global knowledge
case while other cases require a mix of multiple pragmas and serial code.

Eden [54] is a high-level declarative language for parallel programming. Their
approach is to start with a more declarative language (Haskell) and add support for
parallelism. In contrast, we are adding declarative features to C++. While
performing on par with MPI is an explicit non-goal for Eden, there is nothing
fundamental about the design of Kanor that prevents it from achieving performance
similar to pure MPI programs. Eden has a much richer processor abstraction than is
provided by Kanor.

The design choice of explicit specification of communication in Kanor is motivated
by our desire to leverage highly tuned hardware-specific communication libraries for
communication optimization. Past efforts at optimizing communication when it is
implicitly specified, as it is in the vast majority of parallel languages, have had
limited success. Kanor also takes an incremental approach in that it is built on top
of C++ and can be compiled with a standard C++ compiler. New languages like
Chapel, X10 require writing new compilers and runtimes which requires considerable
time and effort. These new languages also require a strong user community that
uses the language and helps contribute to its development. It is also difficult for the
new language to work with legacy codes written in languages such as C,
FORTRAN. These challenges have motivated Kanor’s incremental approach.
CHAPTER 3

KANOR DESIGN

3.1 KANOR DESIGN CHOICES

In the previous chapter Background and Related Work, we described various approaches towards simplifying parallel programming and how they relate to Kanor. In this sections, we discuss aspects of Kanor’s design and the rationale behind them.

3.1.1 PROGRAMMING MODEL

Kanor programs are partitioned address space programs. Processes with separate address spaces work on local data and communicate with each other using Kanor communication statements if required. This approach is in contrast with languages that provide some form of sharing of data structures. The amount of sharing can vary. For sequential programs, all data is shared while for PGAS languages such as UPS, programmers need to explicitly declare shared data. Shared data structures might be easier to use compared to data structures in partitioned address space languages, but programmers need to be careful to ensure correctness of the program.
Providing shared memory as a language feature also requires that the hardware
provides shared memory capability or some simulation of shared memory over the
underlying distributed architecture needs to be provided, hindering performance.
Kanor avoids these issues by requiring programmers to think and write programs in
terms of communicating processes. Distributed address spaces naturally map to
modern hardware so that Kanor does not need to simulate sharing. Further, as we
discuss in chapter A Shared Memory Backend, Kanor can exploit shared memory
backends for efficient execution.

### 3.1.2 SYNTAX

Kanor communication statements can be thought of as parallel assignments. The
memory on receivers is updated in parallel with sender values. The relationship
between sender and receiver processes is specified with syntax similar to list
comprehensions used in the Python language (Python Software Foundation,
https://www.python.org/). The simplicity of Kanor syntax means that
programmers do not need to remember complicated syntactic forms. The syntax
forms provided to construct process sets are, at the same time, expressive enough to
encode complex communication patterns. Finally, the syntax should be familiar to
programmers using C-like languages.

### 3.1.3 SEMANTICS

If the communication code is mixed with computation, the compiler/runtime as well
as the programmer might find it difficult to analyze the communication pattern.
Kanor addresses this problem by separating communication code into a single statement, thus making the task of communication analysis much simpler. Since, the communication pattern is specified as a single statement, it becomes easier for the runtime to analyze and optimize the pattern. This is in contrast to MPI where the runtime perform complex optimizations to figure out the communication pattern [43]. In section 5.1.2, we explain how this design choice also helps Kanor runtime cache communication patterns bypassing costly analysis.

3.1.4 IMPLEMENTATION APPROACH

Kanor is embedded in C++ using operator overloading and metaprogramming techniques as described in chapter Kanor Implementation. This design choice means that C++11 compliant compilers can be used to compile and run Kanor. Also Kanor programs, being standard C++11 programs, can work seamlessly with existing C++ code. This is in contrast with systems such as Charm++ [48], which require a custom parser to generate standard C++ code. Further, Kanor implementation does not use any compiler-specific features making it portable. New parallel languages such as X10 and chapel have seen limited acceptance since there is a non-significant learning curve for these languages. Kanor is designed to avoid this issue with an incremental approach that builds on top of the popularity of a mainstream language like C++.

With this background, we discuss Kanor syntax in more detail in the next section. In the subsequent sections, we present an operational semantics for a subset of Kanor called \( K_T \) and discuss some useful properties of the language.
\[\text{CommStmt} ::= DstSpec \ll< \text{Operation} \ll< SrcSpec | \text{Conditions [WithStmt]}\]

\[\text{DstSpec} ::= Lval \text{at ProcIdExpr}\]

\[\text{SrcSpec} ::= Rval \text{at ProcIdExpr}\]

\[\text{Conditions} ::= \text{Clause [\& Clause]*}\]

\[\text{Clause} ::= \text{ForEach} | \text{If} | \text{Let} | \text{TopoSpec} | \text{Hints}\]

\[\text{ForEach} ::= \text{for\_each}(PVar, ProcSet) | \text{for\_each}(PVar, Beg, End)\]

\[\text{If} ::= \text{if ( Cond )}\]

\[\text{Let} ::= \text{let ( PVar, Val )}\]

\[\text{WithStmt} ::= \text{with Hints [\& TopologyObject]} | \text{with TopologyObject}\]

\[\text{Hints} ::= \text{Hint [\& Hint]*}\]

\[\text{Hint} ::= \text{GLOBAL} | \text{CORRESPONDING} | \text{SENDER} | \text{INVARIANT}\]

Figure 3.1: Formal syntax of Kanor. A communication statement expressed in Kanor is represented by the non-terminal, \text{CommStmt}.

### 3.2 Kanor Syntax

Fig. 3.1 shows the BNF grammar for Kanor. A communication statement in Kanor consists of four parts: destination specification, operation, source specification, and condition. The operation is bracketed by the literal \ll< and the condition is preceded by the literal |, which makes the communication statement read like set comprehension. The destination specification is any valid lvalue in C++ and source specification is any valid rvalue. If the source type is not assignable to the destination, compilation fails, as would be expected in standard C++. The operation bracketed by \ll< can be any binary functor, which is applied to the source and the destination to update the destination. The operation may be omitted, in which case the operation defaults to identity operation. If specified the operation is assumed to be associative and commutative. Note that assigning multiple values to a single destination violates the condition of associativity, in which case the
program’s behavior is undefined.

The condition consists of a clauses separated by the literal &. A _for_each clause binds a process ID variable, PVar, to a set of process IDs (ProcSet). The set can be generated with the range Beg to End. An _if clause is used to filter out certain process IDs from a generated set; Cond is a boolean expression and may involve the PVars bound with a _for_each. The _let clause assigns a value Val to a PVar.

In destination and source specifications, Lval refers to an expression that evaluates to an lval, Rval to an expression that evaluates to an rval, and ProcIdExpr to an expression that evaluates to a valid process ID. The _with clause is used to specify a topology and provide certain hints to Kanor about the communication (see Section 5.1).

Kanor is sufficiently expressive to encode all the MPI collectives. Communication characteristics, such as, process sets, message lengths, and destination addresses can be specified as part of the statement itself. This means that a communication pattern like the MPI collective MPI_Alltoallv can be encoded in many ways with a Kanor communication statement. Some of the encodings are shown in Fig. 3.2.

Kanor makes extensive use of operator overloading for clean syntax and C++ expression templates [22] for performance. The users only need to include a single header, kanor.h.

3.3 KANOR SEMANTICS AND PROPERTIES

In order to make the language behavior precise we give a big-step operational semantics for a subset of Kanor. We have chosen to restrict ourselves to a subset of
### All-to-All:

<table>
<thead>
<tr>
<th>All</th>
<th>rb[s] <em>at</em> r &lt;&lt; sb <em>at</em> s</th>
<th>_for_each(s, t.world) &amp; _for_each(r, t.world) _with t;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gather</td>
<td>rb[Slice(dspls[s],counts[s])] <em>at</em> r &lt;&lt; sb[Slice(0, counts[s])] <em>at</em> s</td>
<td>... ;</td>
</tr>
<tr>
<td>Reduce</td>
<td>rb[Slice(0, count[r])] <em>at</em> r &lt;&lt; std::plus&lt;btype&gt;() &lt;&lt; sb[Slice(displ[r], count[r])] <em>at</em> s</td>
<td>... ;</td>
</tr>
</tbody>
</table>

### All-to-One:

| Gather       | rb[Slice(s*blk_sz, blk_sz)] _at_ root << sb[Slice(0, blk_sz)] _at_ s | ... ;                                                                 |
| Gatherv      | rb[Slice(displ[s], counts[s])] _at_ root << sb[Slice(0, counts[s])] _at_ s | ... ;                                                                 |
| Reduce       | rb _at_ root << kanor::sum<btype> << sb _at_ s | ... ;                                                                 |

### One-to-All:

| Bcast        | rb _at_ r << sb _at_ root | ... ;                                                                 |
| Scatter      | rb _at_ r << sb[Slice(r*blk, blk)] _at_ root | ... ;                                                                 |
| Scatterv     | rb _at_ r << sb[Slice(displ[r], counts[r])] _at_ root | ... ;                                                                 |

### Other:

| Scan         | rb _at_ r << std::plus<btype>() << sb _at_ s | _for_each(s, t.world) & _for_each(r, s, t.world.size()) _with t; |
| Exscan       | rb _at_ r << std::plus<btype>() << sb _at_ s | _for_each(s, t.world) & _for_each(r, s+1, t.world.size()) _with t; |

Figure 3.2: MPI collectives encoded in Kanor. In “One-to-All”, blk refers to the size of rb.

the entire language for space considerations and also to keep the proofs of determinism and deadlock freedom tractable. We call this restricted language $K_T$.

### 3.3.1 SEMANTICS

The syntax for $K_T$ is shown in Fig. 3.3. A $K_T$ program consists of a sequence of commands denoted by $c$ in the table. Traditional control flow constructs are represented by the if and while commands. Commands are sequenced with ;
\[ a\exp \ ::= \ n \mid x \mid a_0 \oplus a_1 \mid \text{me} \mid \text{np} \]
\[ \oplus \ ::= + \mid \times \mid - \]
\[ b\exp \ ::= \text{true} \mid \text{false} \mid a_0 \odot a_1 \mid b_0 \odot b_1 \mid \neg b \]
\[ \odot \ ::= < \mid > \mid \leq \mid \geq \mid = \]
\[ \oslash \ ::= \land \mid \lor \]
\[ \text{cmd} \ ::= \text{skip} \mid x := a \mid c_0 ; c_1 \mid \text{if } b \text{ then } c_1 \text{ else } c_2 \mid \text{while } b \text{ do } c \mid \text{comm} \]
\[ \text{comm} \ ::= x_1 \odot p_1 \leftarrow \text{op} \leftarrow x_2 \odot p_2 \text{ where clause}^* \text{filter}^* \]
\[ \text{clause} \ ::= \text{foreach}(v, s\exp) \]
\[ s\exp \ ::= \text{list}(a_0, a_1, \ldots) \mid \text{rep}(a_0, a_1) \mid \text{range}(a_0, a_1) \]
\[ \text{filter} \ ::= \text{BExp expression} \]
\[ \text{op} \ ::= \text{reduction op} \]

Figure 3.3: Formal syntax of K_T.

(semi-colon): \( c_1 ; c_2 \) means that \( c_1 \) is executed before \( c_2 \). Variables, \( x \), in the language represent memory locations that can be updated during execution of the program. Variables can be updated with the assignment command (:=) and with the communication command denoted by \text{comm}. Expressions can be arithmetic \((a\exp)\) or boolean \((b\exp)\).

All \( K_T \) processes execute the same program similar to the single process multiple data (SPMD) model. Each process has its private memory, called that process’s \text{store}. Each process starts with its own store with the variables \text{me} and \text{np} denoting the process rank and the total number of processes respectively. Communication can only be done with the \text{comm} commands.

The operational semantics for \( K_T \) consist of local rules (Table 3.1) and communication rules (Table 3.2). The local semantics specify how processes compute values locally. Local process stores are modified with variable assignment denoted by the \text{E-ASSIGN} rule. The semantics for communication are specified by the \text{E-COMM} rule in Table 3.2. The communication command can be thought of as
parallel assignment of receiver locations by sender values. The sender process \( s \) evaluates the expressions \((p_1, x_2, p_2, \text{foreach}(i, ...), ..., \text{pred}_1, ... \) with the store \( \sigma \) producing an environment \( \rho_s \) \( (\text{E-SENDABLES}) \). \( \rho_s \) maps the sent variable \((x_2)\) to a value \((v_s)\) and also binds the generator-bound variables \((i)\) to set of process IDs.

The set of process IDs is generated after the evaluation of conditions 
\text{foreach}(i, ...), ..., \text{pred}_1, .... Only the values of \( i \) that evaluate \( p_2 \) to \( s \) are stored in \( \rho_s \). The operation \( \uplus \) represents communication of data(\( \rho_s \)) from senders to receivers.

The environment \( \rho_r \) is formed on receiver \( r \), by combining mappings from \( \rho_s \) with \( p_r = r \). All the sent variables are distinct from each other and their mappings are preserved in \( \rho_r \). Finally, receiver \( r \) evaluates and updates memory location(s) \( x_1 \) by applying \( \text{op} \) to received values in the combined environment \( \rho_r \oplus \sigma \). Application of \( \text{op} \) is a local computation on the receiver \( (\text{E-APPOP}) \). The environment \( \rho_r \) contains a set of values for \( x_2 \) received from different senders. Variable \( x_1 \) in \( \sigma \) is updated by combining all these values with operator \( (\text{op}) \). The updated store is denoted by \( \sigma_1 \).

### 3.3.2 PROPERTIES

We first define what well-formedness means for \( K_T \) programs. We assume \( K_T \) programs are well-formed in the ensuing discussion.

**Definition 1.** \textbf{Well-formedness} \( K_T \) programs are said to be well-formed iff 

- All processes participating in communication, \( C \), execute \( C \).

- All processes participating in communications, \( C_1 \) and \( C_2 \), execute \( C_1 \) and \( C_2 \) in the same order.

- There are no local errors, including the application of the reduction operator and
First two requirements for well-formedness are the same as that for an MPI collective. The problem of checking well-formedness is undecidable in general. Kanor does not provide syntactic support or semantic guarantees to ensure well-formedness. Well-formedness could be checked in limited cases (global knowledge) by the compiler, but not in general.
∀r, s ∈ P, ∃p₁s, x₂s, p₂s  ⟨(p₁s, x₂s, p₂s, foreach(i, ..., pred1, ...) | σ) ↓_s ρ_s⟩

∀r, s ∈ P, ⊎ρ_s ↓_r ρ_r

∀r, s ∈ P, ρ_s ↓_r ρ_r  ⟨(x_1, op, x_2) | ρ_r ⊕ σ⟩ ↓_r σ_1

⟨x_1@p_1 ← op ← x_2@p_2 where ... | σ⟩ ↓_r σ_1

Table 3.2: Communication semantics in K_T.

DETERMINISM

We would like K_T programs to produce the same output on same inputs. This means that each K_T process starting with some initial configuration always ends up with the same final configurations on each run of the program. We say that K_T is deterministic if all programs satisfy this property. More formally, we say that K_T programs are deterministic if and only if for a given process set P, each process with an initial store σ_p, p ∈ P, all executions of the program c satisfy the following property: For each process p ∈ P, if two executions of c evaluate to final stores σ_1 and σ_2 then σ_1 = σ_2. Note that processes share the same program c but they can have different initial and final configurations (memory stores denoted by σ).

We divide the proof of determinism into two parts, proving that expression evaluation is deterministic and that the execution of commands is deterministic.

Lemma 1. K_T expression evaluation is deterministic.

• ∀e ∈ AExp, ∀σ ∈ Σ, ∀n, n’ ∈ ℤ, ⟨e, σ⟩ ↓_a n ∧ ⟨e, σ⟩ ↓_a n’ ⇒ n = n’

• ∀b ∈ BExp, ∀σ ∈ Σ, ∀t, t’ ∈ ℂ, ⟨b, σ⟩ ↓_b t ∧ ⟨b, σ⟩ ↓_b t’ ⇒ t = t’

Proof. By induction on the structure of arithmetic expression e. The base cases are numeric constants n, me and np. The conclusion follows from reflexivity of integers.
In case $e$ is a variable $x$, $x$ evaluates to a unique $n$ in a given store $\sigma$. The inductive case $(a_0 \oplus a_1)$ follows from the deterministic nature of arithmetic operations. The proof for boolean expressions is similar.

A potential source of non-determinism is the communication command. The reduction operator might be non-commutative, e.g. assignment. If such an operator operates with different values on the same memory location, then the result might be non-deterministic. In this case, we make the operator application ($\downarrow_s$ of rule E-APPOP in Table 3.2) deterministic by choosing a particular evaluation order. Also, we assume the network is reliable so that the $\uplus$ operator in E-UNION always produces the same environment after a union over the sent environments.

In the presence of commands like while, we cannot use induction on the structure of commands to prove determinism. Instead, we use induction on derivation trees. A judgment $D$ of the form $c \downarrow \sigma$ says that the command evaluates to final configuration $\sigma$ without errors. The derivation of $D$ starts by selecting the operational semantics rule (Tables 3.1 and 3.2) for which $D$ is the consequent. The derivation then branches out, each branch representing a derivation for each premise of the selected rule. Derivation along a branch of ends when a rule with no premise is found. Thus the derivation for $D$ forms a tree with $D$ at its root. We prove determinism of command evaluation by induction over derivation trees.

**Lemma 2.** $K_T$ command evaluation is deterministic.

$$\langle c \mid \sigma \rangle \downarrow \sigma_1 \land \langle c \mid \sigma \rangle \downarrow \sigma_2 \Rightarrow \sigma_1 = \sigma_2$$

Proof. The most interesting case here is the rule E-COMM in Table 3.2. Let $D$ be the derivation when $c$ evaluates to $\sigma_1$ and $D'$ be the derivation when $c$ evaluates to
σ₂. Derivation tree for D must have two branches (subderivations) from the root, one for rule e-union (D₁) and other for rule e-appop (D₂). At the end of D₁ we should get the store ρ₁ and at the end of D₂ we should get σ₁.

By inversion, since D’ uses the rule e-comm again with two subderivations D’₁ and D’₂ with stores ρ₂ and σ₂ respectively. By induction hypothesis on D₁ with D’₁, we have ρ₁ = ρ₂ and by induction hypothesis on D₂ with D’₂ and ρ₁ = ρ₂, we have σ₁ = σ₂.

A Kₜ program is a command with initial store. Hence, determinism of Kₜ programs follows from lemma 2.

**DEADLOCK FREEDOM**

A message passing program might deadlock when a process blocks waiting for a message that is never sent. Kₜ programs are deadlock-free by construction.

**Lemma 3.** Kₜ programs are deadlock free.

∀c ∈ WC, ∀p ∈ P, ⟨c | σₚ⟩ ↓ σ’ₚ where WC is the set of well-formed Kₜ programs.

**Proof.** In well-formed Kₜ programs, application of the rule e-appop, is always successful across all processes. All other commands act locally and do not block, hence there is no deadlock. The proof follows similar pattern to the determinism proof. The induction is on the structure of derivations. □
KANOR IMPLEMENTATION

Having described Kanor’s syntax and semantics, we will look at our implementation of Kanor as a DSL embedded in C++. Kanor was initially implemented as an extension to C++ with special operators such as @ that were not part of the standard C++ syntax. The translation to standard C++ required tokenizers that take the non-standard code, tokenize it based on special operators and output standard C++ with pragmas and function calls to identify Kanor specific code. Though the benefit of this approach was that the syntax looked good, there were many disadvantages to the approach:

- **Complex Tokenizer** - The tokenizer that converts from the extended C++ syntax to standard C++ has to still handle C++ complexity and it is difficult to guarantee its correctness. The tokenizer gets complicated trying to identify Kanor syntax forms embedded in surrounding C++.

- **Difficult to Extend** - The tokenizer has to be modified if we want to add new syntax to Kanor. C++ also keeps adding new features which might impact the
tokenizer. It is difficult to extend the tokenizer to handle this added complexity and ensure its correctness.

- *Interaction with C++ code* - Users need the tokenizer executable to use Kanor. It becomes difficult to use Kanor with existing code since Kanor programs need to go through the tokenizer first. Existing C++ tool niceties such as syntax highlighting, auto-completion do not work with the extended syntax.

Due to these limitations, we decided to use C++ features such as operator overloading and expression templates to embed Kanor in C++. Thus Kanor programs are standard C++ program. Programmers only need to include `kanor.h` to use Kanor as a library. Kanor programs can be compiled with C++11 standard compliant compilers. The C++ standard compliance of Kanor programs facilitates seamless integration with existing codebases and tools.

Providing Kanor abstractions as a library enables Kanor runtime analyze individual communication statement and optimize them. The runtime has no control over surrounding C++ code and it is unable to perform optimizations that depend on analyzing and transforming non-Kanor C++ code. To enable these optimizations, users need the source-to-source translator discussed in section Clang Libtooling. In this chapter, we discuss two components of the Kanor system, namely the runtime and the source-to-source translator.

### 4.1 Kanor Abstractions

We first describe abstractions provided by Kanor to programmers. Kanor users write programs mainly with three classes, namely `CommBuff`, `ProcID`, and `Topology`.
4.1.1 PROCID

As described previously, Kanor programs consist of processes with separate address spaces that exchange data with each other using communication statements. Each process in Kanor is associated with an id represented by an object of the ProcID class. A process ID can be thought as an integer tuple which maps to a unique integer used by the runtime to communicate with the associated process.

Representing process IDs as tuples helps programmers arrange processes logically, e.g. if the programmer wants to arrange processes in a two dimensional grid, the process IDs will be a two dimensional tuple where the first tuple element represents process row and the second tuple element represents process column in the process grid. Helper methods for comparison and printing ProcID objects are provided.

Overloaded arithmetic and comparison operators are defined for ProcID, which return a Kanor node, which will be evaluated by runtime while executing communication. ProcID arithmetic is useful in specifying memory locations to be updated or sent in the communicated data.

4.1.2 COMMBUFF

Kanor communication statements can send and receive variables whose type is either a primitive type or a std::vector of a primitive type. Subregions of the data to be communicated can be specified using a combination of the CommBuff, ProcID, and kanor::Slice classes. The data to be communicated is wrapped in a CommBuff object. We refer to the type of wrapped data as boxed type. CommBuff has an overloaded index operator (operator[]), which takes a Kanor node as an argument.
The argument Kanor node is a Kanor expression which might consist of `kanor::Slice`, arithmetic on ProcIDs, and `CommBuff` nodes. The overloaded index operator returns a Kanor node that is evaluated by runtime and allows programmers to specify data subregions as part of the communication pattern.

`CommBuff`s can be constructed from objects of primitive types, pointers of primitive types, `std::vector`s of primitive types, and `VarWrapper` objects explained in subsection 4.1.3. `CommBuff` is a template class and its template type is the primitive type of `boxed` variable, `std::vector` or pointer. In case `CommBuff` is constructed from a `VarWrapper`, `CommBuff` template type is `VarWrapper`.

The `CommBuff` class is also used if programmers want non-blocking communication. Kanor treats the communication of `CommBuff` objects as non-blocking by default. When the programmer wants to access the `boxed` data, they call `get` method on the `CommBuff` object. `get` returns a pointer which can be read or updated.

### 4.1.3 VARWRAPPER

Restricting `boxed` types of `CommBuff` objects to primitive types, vectors or pointers forces programmers to serialize complex type into contiguous types. Serialization might also force programmers to create temporary copies only for communication purposes. The serialized copies need to be communicated differently from the contiguous subregions of the same structure. Another disadvantage of this restriction is that Kanor might not be able to leverage features provided by backend libraries, such as MPI derived data types.

Kanor provides a class called `kanor::VarWrapper` to overcome these restrictions. The class constructor shown in figure 4.1 takes four arguments:
• a reference \( v \) of type \( T \). This is the wrapped variable whose subregions are communicated.

• a \textit{setter} function object \( f_1 \), whose type is shown in figure 4.1 line 12. The setter function is meant to update subregion of memory denoted by the first argument \( (T*) \) with contents from the second argument \( (\text{std::pair<R*, std::size_t>} \)\).
The variable list of arguments \( (\text{Args...}) \) help specify the subregion to be updated. The setter function is used on the receiver side to update subregion of some larger structure wrapped in the \texttt{VarWrapper} class.

• a \textit{size} function object \( f_2 \), whose type is shown in figure 4.1 line 15. The runtime needs to specify the size of data to be received from receivers. The size function gives this information to the runtime.

• a \textit{getter} function object \( f_3 \), whose type is shown in figure 4.1 line 18. This function is meant to return some subregion \( (\text{std::pair<R*, std::size_t>} \) of a larger structure \( (T*) \) specified by a variable list of arguments \( (\text{Args...}) \). Getter functions are used on the sender side to specify subregions to be sent.

4.1.4 TOPOLOGY

The \texttt{Topology} class is used by Kanor runtime to send and receive messages.
Programmers can provide their own implementation of topologies. Kanor runtime expects the topology class to have the following requirements:

• a member variable \texttt{me} of type \texttt{ProcID} denoting the ID of the current process,
template <typename T, typename F1, typename F2, typename F3>
class VarWrapper {
public:
// constructor
VarWrapper(T& v, const F1& f1, const F2& f2, const F3& f3) {
...
}
...;
}

template <typename T, typename R, typename... Args>
using setter_type = std::function<void(T*, std::pair<R*, std::size_t>, Args...)>
;

template <typename T, typename R, typename... Args>
using getter_type = std::function<std::pair<R*, std::size_t>(T*, Args...)>
;

template <typename T, typename R, typename... Args>
using size_func_type = std::function<std::pair<R*, std::size_t>(T*, Args...)>
;

Figure 4.1: VarWrapper constructor and getter/setter types

- a member variable `max_rank` of type `ProcID` denoting the ID of the highest ranked process,

- a member variable `world`, which is the set of all the processes in the system,

- a member function `to_int`, which takes a `ProcID` object and returns the int that the process ID maps to,

- a member function `exec_comm`, which takes runtime representation of the communication statement, executes the communication and returns `true` if the communication was successful, `false` otherwise.

The runtime additionally checks for a method `exec_cached_comm` which executes cached communication, as explained in section Communication Invariance. The topology class implementation is not part of `kanor.h` and needs to be included separately.

Kanor provides an in-built `MPI_OneToOne` class which maps each Kanor process to
an MPI process. The processes for this class are arranged as a single dimensional array. The class identifies collective communication patterns and calls optimized MPI collectives. It caches communication patterns and executes the cached patterns on subsequent executions of the communication statement. The class also detects if shared memory can be leveraged to reduce message copies if the processes are executing on a single node, as explained in chapter A Shared Memory Backend.

4.1.5 COMMUNICATION STATEMENT

All of the above classes are used by Kanor programmers to specify different aspects of communication. We will now look at implementation of the Kanor communication statement.

Recall from section Kanor Syntax, that Kanor communication statement looks like

\[
e_1 \text{ } \_at\_ \text{ } p1 \Leftrightarrow \text{op} \Leftrightarrow e_2 \text{ } \_at\_ \text{ } p2 \mid [\text{clauses}] \_with \text{ topology} \& [\text{hints}];
\]

where \(e_1\) and \(e_2\) are Kanor expressions denoting destination and source communication buffers respectively, \(p1\) and \(p2\) denoting receiver and sender ranks respectively, \(\text{op}\) is the reduction operator, \text{clauses} specify the sender/receiver relationships between processes, \text{topology} object helps runtime communicate data, and \text{hints} specify certain properties of the communication statement that help in runtime optimizations.

The syntax required by the communication statement is implemented using C++ techniques such as operator overloading, expression templates, and template metaprogramming \[22\]. The overloaded operators build and return objects of type \text{kanor::internal::Node} which encode the structure of the communication statement. The \text{Node} objects represent different types of Kanor expressions such as generators
(kanor::for_each), filters (kanor::if), hints (e.g. kanor::GLOBAL), kanor::ProcID arithmetic expressions etc. The type of the node is represented by a member variable of the Node class. The children of a Node might be other Nodes, ProcIDs, Topology objects etc. The children are stored as a tuple member variable in a Node object.

Kanor runtime analyzes the communication statement by inspecting the Abstract Syntax Tree (AST) formed out of the Node objects. The communication case (global, corresponding, sender) is statically deduced by looking at the type of hints and appropriate implementation is called. Kanor syntax requirements are similarly checked at compile time. The runtime passes the AST to an Evaluator class that visits each AST node, evaluates it and creates a graph out of the communication statement. This graph is passed to the topology object for execution. The topology implementation can pattern match on AST nodes to identify the communication pattern and optimize for the same. The topology object is responsible for updating the receiver buffer. A class called KRuntime helps maintain state in the runtime which is needed in caching communication structure, implementation of non-blocking communication semantics etc. A RuntimeException class is provided to signal any runtime exceptions. Helper classes are provided to help programmers with debugging, common typedefs, timing the code.

4.2 CLANG LIBTOOLING

We have described Kanor as a library so far in this chapter. The library can be compiled using any standard C++11 compliant compiler. The library design of
Kanor also means that the runtime does not have access to the surrounding C++ code and is unable to perform optimizations that require reasoning and/or modifying the surrounding code. We have used the Clang Libtooling framework [19] to analyze and perform these optimizations.

The Libtooling framework parses the C++ source code and produces an AST which can be walked using the `clang::RecursiveASTVisitor` class. We use this class to collect and parse Kanor communication statements. The Libtooling framework can also be used to produce control flow graph (CFG) from the AST. We perform a reaching definition analysis on the CFG to generate use-def chains which are needed in overlap optimizations described in chapter Compute Communicate Overlap. The dataflow analysis is performed using a simple worklist based algorithm based on the in-built Liveness Analysis algorithm provided as part of the Clang source code. We also use the dominator tree provided by Clang to calculate control dependences using a well-known algorithm [49]. Finally, we use the `clang::Rewriter` class to insert, delete code in the original Kanor program.

The ROSE compiler infrastructure [66] is a C++ library for source-to-source transformations. ROSE can parse C, C++, and FORTRAN source codes and provides a common AST for these languages. ROSE was one of the first frameworks to represent the AST in object oriented form. Visitor classes are provided to traverse AST along with several functions to manipulate (insert, delete) nodes. The library also contains an analysis framework that can be extended to write custom analyses. Several in-built analyses such as Liveness Analysis, Alias Analysis are provided with the library. Despite these niceties, the C++ support for ROSE is still evolving and we decided to use Clang for its more mature C++11 support.
CHAPTER 5

RUNTIME OPTIMIZATIONS

5.1 OPTIMIZING COMMUNICATION

Having established precise semantics and basic correctness of Kanor, we next identify opportunities to optimize it. Our core technique is based on inferring the collective operation at run-time the first time a communication statement is executed. Subsequent executions of the statement use the previously computed (cached) inference, which eliminates the overhead of the run-time inference of the pattern, which can take $O(n^2)$ time for $n$ processes.

5.1.1 COMMUNICATION KNOWLEDGE

In order to understand when and how collective patterns can be detected we need to define communication knowledge cases, which describe the extent to which the processes agree on the values of the expressions involved in a communication statement.

The receiver lval ($Lval$ in $DstSpec$ in Fig. 3.1) is computed on the receiver.
Similarly, the sender rval is computed on the sender. This is necessary, because the lval might not make sense on the sender and rval might be meaningless on the receiver. However, the process sets—the sets of senders and receivers that are computed using DstSpec, SrcSpec, and Conditions—need to be computed by both the senders and the receivers in order for two-way communication to take place. If the sender and receiver process sets evaluate to exactly the same values on all the processes we call it the global knowledge case. This is the simplest of all cases.

It is possible that the receiver sets evaluate to different values on different processes. In such cases, Kanor assumes the communication to be sender-driven, i.e., the receiver process sets computed by senders take precedence\(^1\). Thus, the senders know which processes they are sending to, but the receivers may not know their senders. We call this the sender knowledge case. To illustrate it, suppose the sender process \(s\) computes the sender set \(S_s\) and receiver set \(R_s\). The receiver process \(r\) computes the sender set \(S_r\) and the receiver set \(R_r\). If \(r \in R_s\) but \(s \notin S_r\), the sender \(s\) still sends the message which must be received by \(r\).

Finally, it is possible that the sender and the receiver processes agree on their corresponding receivers and senders, but other processes might not. Thus, if the communication statement requires process \(A\) to send data to \(B\) then both \(A\) and \(B\) agree on it, but a third process \(C\) might not, although, \(C\) knows that it is not involved in this communication. Such a case, which might be relatively rare compared to other cases, is called the corresponding knowledge case.

Note that there is no receiver knowledge case, since the communication in Kanor is

\(^1\)This is motivated by the fact that one-sided put operations are usually more efficient than one-sided get operations.
sender driven. If sender expression evaluates to different values on different processes, it is still the senders’ versions that take precedence.

It is possible to detect these cases using compiler analysis, however, that is beyond the scope of this paper. In this paper we assume that the users provide appropriate annotations with a communication statement (Hint in Fig. 3.1) to identify the knowledge case. For the rest of the discussion we assume global knowledge case, which is by far the most common. Other cases can also be handled similarly, but usually require additional communication. Assuming local computations are error-free, the global case guarantees well-formedness 1. It is left to the programmer to make sure that non-global Kanor programs are well-formed.

5.1.2 COMMUNICATION INVARIANCE

A communication statement whose process set calculation depends on an enclosing loop’s index may use different process sets in each iteration. Thus, certain aspects of a communication statement might change with each invocation. This is a property distinct from knowledge case.

We identify three core characteristics of a communication statement: length of the messages, the contiguity of the data in memory, and the process sets involved in sending and receiving data. We say that the communication is invariant if none of the communication characteristics change. Invariance of communication allows us to cache the communication pattern and reuse it in later instances of the same communication statement.

If a communication statement is both global knowledge case and invariant then each process can independently infer the communication pattern and cache it, with the
assurance that every other process will make an identical inference. The communication pattern is inferred using Algorithm 1 and cached for subsequent use. Our evaluation (section 5.4) shows that the inference cost gets amortized quickly as message size increases.

```
1 Input: Communication Statement S
2 Output: Set of Collective Calls C
// G is a directed graph, in which vertices are process IDs,
// an edge connects sender to a receiver
3 G = build from S;
4 n = number of vertices in vertex set V(G);
5 if each vertex v in V(G) has degree n then
6     if send and receiver buffers contiguous then
7         C = {Alltoall};
8     else
9         C = {AllGather};
10    return;
// build rooted collectives to be executed independently
11 foreach v in V(G) with no incoming edges do
12     if send and receiver buffers contiguous then
13         C = C ∪ {broadcast};
14     else
15         C = C ∪ {scatter};

Algorithm 1: Algorithm to detect MPI collectives.
```

### 5.2 KNOWLEDGE CASE INFERENCE

If each free variable in a communication block has the same value on all processes then that implies global knowledge case. The inverse is not necessarily true, since it may be possible for processes to completely infer the communication pattern even when some of the free variables have differing values. Two processes differ on a variable if its value is rank-dependent or obtained through an I/O operation. By
restricting ourselves to those global knowledge cases in which free variables have identical values on all ranks, we can identify those cases by ensuring that no information flows from any value representing the rank of a process, or an I/O operation, to any free variable within a communication block. Information flow relations may be caused either by explicit or implicit flow dependencies.

- **Explicit:** When \( x \) is data-dependent on \( me \). Flow dependence is transitive, i.e., if \( x \) is data-dependent on \( y \) and \( y \) is data-dependent on \( me \) then \( x \) is flow-dependent on \( me \).

- **Implicit:** When \( x \) is control-dependent on \( me \). As before, the dependence may be transitive.

We present a list-based algorithm that a compiler can use to look for this transitive data- and control-dependence of free variables on \( me \). It starts by initializing the list to all free variables inside a communication block. For each definition of a variable on the list, if the right hand side contains \( me \) then that variable is dependent on \( me \). If not, all the variables on the right hand side are added to the list. Inter-procedural analysis must be used if a variable is passed by reference to a function. If the function cannot be analyzed, the compiler conservatively assumes that it causes a flow dependence from \( me \). Control dependencies are handled by adding to the list those variables that occur in the condition expressions of any branches or loops surrounding the definition of a variable on the list. Flow from I/O operations can be handled similarly.

Aliasing can be a problem when we calculate dependencies, but the compiler makes

\(^2\)We use \( me \) as a generic stand-in for any variable that might be used in Kanor to refer to a process’s own rank.
conservative assumptions. For example, if we can not prove that two arguments passed by reference are not aliases of each other then we assume conservatively that the free variable is also dependent on me.

```cpp
list<Variable*> wlist;

vector<Variable*> fv = get_var_refs(comm_stmt);
copy(fv.begin(), fv.end(), wlist.end());

while(!wlist.empty()) {
    Variable* dep_target = *(wlist.begin());

    wlist.erase(wlist.begin());

    if (/* dep_target */) {
        return false;
    }
    vector<Variable*> implicits = get_implicit_dependences(defuse, dep_target);
    vector<Variable*> explicits = get_explicit_dependences(defuse, dep_target);
    copy(implicits.begin(), implicits.end(), inserter(wlist, wlist.end()));
    copy(explicits.begin(), explicits.end(), inserter(wlist, wlist.end()));
}
return true;
```

Figure 5.1: Global Case Inference

### 5.3 IMPLEMENTATION STATUS

We have implemented Kanor as an embedded DSL in C++. We make use of operator overloading, template meta-programming and certain C++11 features, such as lambdas, to achieve this. The library will be released in open source.

Kanor process ranks are expressed as members of the `kanor::ProcID` class. Arithmetic and comparison operators are overloaded for the `ProcID` class. Programmers can use list comprehensions provided by Kanor to bind `ProcID` variables to sets. Other entities, including communication buffers and slices, are also
provided as convenient Kanor classes. All communication is implemented using MPI as the underlying communication mechanism. This allows existing MPI programs to be converted to Kanor incrementally.

The implementation uses type traits to perform several compile-time checks, for example, to make sure that the sender and receiver expressions will evaluate to process IDs, and to make sure that the left hand side (receiver expression) is a valid lval. In order to implement communication pattern detection, carefully overloaded operators work together to construct an abstract syntax tree (AST) out of the communication statement. Once the AST is complete, Algorithm 1 is used to infer and cache the pattern as a lambda that can be invoked directly the next time. With the programmer-supplied hint the library generates optimized implementation for each knowledge case using expression templates.

5.4 EVALUATION

We evaluated the pattern identification and caching mechanism implemented in the library with several benchmarks. First set of benchmarks consists of well-known MPI collectives working with different process sets and buffer sizes. Each collective is executed in a loop. First iteration of the loop incurs detection and caching overhead. The runtime overhead for subsequent iterations is minimal compared to actual communication. We also evaluated our system on three other benchmarks, including one dense matrix kernel, Cholesky and two NAS parallel benchmarks, IS (Integer Sort), and FT (Fourier Transform) [5]. We selected Cholesky, where the matrix columns are cyclically distributed across processors, for an example of dense
matrix computation with complex communication patterns. The NAS IS benchmarks models irregular communication seen in typical N-Body codes. NAS FT represents regular communication on a subset of processes.

The experiments were conducted on the Big Red II infrastructure at Indiana University. Big Red II is a Cray XE6/XK7 supercomputer with a hybrid architecture providing a total of 1,020 compute nodes. It consists of 344 CPU-only compute nodes, each containing two AMD Opteron 16-core Abu Dhabi x86_64 CPUs and 64 GB of RAM. It also has 676 CPU/GPU compute nodes, each containing one AMD Opteron 16-core Interlagos x86_64 CPU, one NVIDIA Tesla K20 GPU accelerator with a single Kepler GK110 GPU, and 32 GB of RAM. Big Red II runs a proprietary variant of Linux called Cray Linux Environment (CLE).

**MICRO BENCHMARKS** Fig. 5.2 shows the results for the collective micro-benchmarks. Timings for six communication statements representing MPI
alltoall, allreduce, broadcast, scatter, scattery and gather are shown. The collectives were run for different message sizes, processors and loop iteration counts. We only show the results for 32 processors with variable sized messages of double precision values. Each vertical bar represents total time (in milliseconds) it took for the communication statement to finish. The bars are shown in groups of three. First bar shows the time taken by MPI collective. Next two bars show the time taken by an equivalent Kanor communication statement with caching enabled and disabled respectively. To enable caching, we provide the \texttt{INVARIANT} hint. With caching disabled (third bar in a group), the runtime incurs pattern detection overheads on each iteration. With caching enabled (second bar), the runtime incurs overheads related to the caching mechanism only. The pattern detection overheads (third bar) are considerable for small messages sizes and all-to-all patterns. Detection starts to match MPI for larger sizes. Kanor collectives with caching enabled, start to match MPI even for smaller message sizes.

\textbf{APPLICATION BENCHMARKS}  The comparison results for Cholesky are shown in Fig. 5.4. In our implementation, the matrix columns are cyclically distributed and the main computation loop is strip-mined. A process operates on a block it owns and broadcasts the calculated column to downstream processes that require it. The message lengths may vary hence this is not an invariant communication statement. The computation time dominates the communication time so the detection overheads do not cause significant performance degradation.
Figure 5.3: Comparison of Kanor to MPI implementations of NAS Benchmarks Integer Sort (IS) and Fourier Transform (FT). NAS Benchmark class (S, W, A, B, C) denotes the size of the problem to be solved. Procs denote the number of processes used to solve the problem.

Fig. 5.3 shows the results of the NAS Integer Sort (IS) and Fourier Transform (FT) benchmarks. The benchmarks were run for classes S, W, A, B and C. IS processes send variable number of keys to other processes and the number of keys are not known a-priori. So an alltoall exchange happens to let the receivers of the number of keys they are receiving. Next an alltoallv actually sends the keys. The second alltoallv sends variable length messages, hence it is not an invariant communication. The detection and caching overhead shows up for smaller
problem sizes (S, W). For larger sizes, the computation and communication time hides this overhead. Finally, FT is regular alltoall communication on a subset of processes and it is also invariant. Consequently, Kanor begins to match MPI even for smaller problem sizes.

5.5 RELATED WORK

Kanor’s operational semantics were described in a previous paper [21]. This paper treats communication as parallel assignment, simplifying reasoning. Callahan et.al [14] detail small-step operational semantics for the BSPLib library. Gava et.al. [34] give big-step operational semantics for a subset of BSPLib. Kanor communication statements are treated as parallel assignments in our approach. We do not work or reason with message queues which simplifies the semantics a lot. Using expression templates [22] helps us pattern match AST nodes at compile time and inline code based on the match. This is in contrast to other embedding technique, used for example in Halide [68], that identifies AST nodes by casting pointers. New features provided in C++11 such as static assert help us provide useful error messages in case the communication statement is ill-formed.

Collective detection efforts in MPI have mostly focused on analyzing traces of programs and detect patterns in them. Hoefler et.al. [43] present an online algorithm to detect collective patterns in codes with point to point messages. Kanor communication statements enable easier detection of collectives. Also, we can detect reductions like allreduce.
CHAPTER 6

COMPUTE COMMUNICATE

OVERLAP

6.1 INTRODUCTION

Kanor communication statements, as seen previously, are blocking. This means that the statement does not complete until all the data has been communicated and the reduction operator applied, if necessary. Each process waits for others to finish the communication statement. Programs written in this style are easier to reason about and maintain but can be substantially less efficient due the blocking nature of communication calls.

We observe that we can start sending data as soon as it is available and wait for the sends to finish just before the sent data is updated again. Similarly, we can start receiving data as early as possible and we wait for the receives to finish just before the receive buffer is used or updated again. The computation that does not use or update the communicated buffers can proceed without waiting for the
communication to finish. Thus the communication can be *overlapped* with computation that is independent of it. Overlapping communication with computation is an important optimization for BSP style programming exemplified by Kanor programs.

In this chapter, we describe two approaches taken by Kanor to help programmers achieve this overlap. First of these approaches is a purely library based approach and the second is a hybrid compiler/runtime approach. We compare the two approaches and evaluate them on example benchmarks. After that, we discuss further opportunities of overlap in Kanor programs. Finally, we describe how this work relates to prior work on overlap in other languages.

Consider the example Kanor program shown in figure 6.1a. The program packs data in the buffer $sb$ and receives in the buffer $rb$. Communication is performed at the start of the loop which is executed till some condition is met. The buffer $rb$ is used in computation in the loop and $sb$ is updated again to be sent in the next
iteration. The communication statement at the start of the loop blocks until rb is
updated at all processes represented by rcvr with value of sb from corresponding
senders represented by sndr.

The code in figure 6.1b represents an overlapped version of the code in figure 6.1a.
The sends are started as soon as the sb is ready to be sent (start_send calls). A
receive call (recv(rb)) is posted at the start of the loop. A wait call for rb is
posted just before it is used. Finally, a wait call for sb is posted just before it is
used in the loop body and another wait call is posted outside the loop body in case
the loop was never executed. We see that the independent computations, comp1() and comp2, were overlapped with communication of sb and rb.

6.2 RUNTIME APPROACH

As described earlier, programmers can use CommBuff class objects to pack and
communicate data. CommBuff instances store the data as a pointer internally and
programmers cannot directly modify the pointer data. The get method must be
called on a CommBuff instance, which returns the wrapped pointer which then can
be used/updated. The ownership of the data returned by get is transferred to the
caller of get, similar to C++ smart pointers. Programmers need to use objects of
the CommBuff class to employ the non-blocking semantics of a communication
statement. C++ variables of primitive types or std::vectors of primitive types
can be communicated, but the communication is immediate since the get method
cannot be called on these objects.

In figure 6.2, we see a transformation of the blocking communication example seen
before in the chapter. The code with blocking communication is shown in figure 6.2a. This code can be transformed to use non-blocking communication semantics with the use of get calls as shown in figure 6.2b. In the transformed code, the sent and received buffers, denoted by sent.data and rcvd.data respectively, are packed in CommBuff objects. The type of sent and received buffers are stype and rtype respectively. The CommBuff objects are used in the communication statement at the start of the loop. If the library detects a communication pattern and a non-blocking collective is available for that pattern, the non-blocking collective is issued. If no pattern is found, asynchronous send and receives are issued. The get method is called on the communicated CommBuff objects just before the data is needed. The get call blocks till the data is available. This means the asynchronous sends finished and the wrapped send buffer is available for use again and similarly for the receives. The data is updated or used and communicated in the next loop iteration.

It is the programmer’s job to ensure that get method is called before access to the wrapped data. In the example, rb.get() is called before use of rcvd.data. Similarly, sb.get() is called before the use of sent.data. The programmers only need to place these calls before accesses to the wrapped data. CommBuff class does not provide any methods to access the wrapped data. CommBuff constructors do not copy non-const data to be wrapped, but store a pointer to it. This pointer is an alias to the wrapped data. Any updates to the wrapped data outside CommBuff will be visible to the runtime when CommBuff is used by the runtime. Thus Kanor programmers need to make sure that there are no modifications to the wrapped data after communication and before the get call. Kanor runtime does not have
access to the surrounding C++ code, hence it cannot prevent accidental writes to the wrapped data. C++’s move semantics could be used to ensure that ownership of the wrapped data is transferred to CommBuff at the time of its creation.

Similarly, move semantics can ensure that the ownership is released by CommBuff through the get method. Unfortunately, this approach does not allow any reads after communication and before get is called. Enabling these intermediate reads might help programmers achieve overlap and was one of the main motivations for our current design of the CommBuff class.

For each execution of the communication statement, the runtime updates a registry storing the addresses pointed by the internal pointers of the sent and receiver CommBuffers. The CommBuff get method consults the registry to check if the address has been waited upon. If yes, the communication for the buffer represented by the
address is complete and the underlying pointer is returned. If not, get method issues a wait on the address and blocks till the communication is complete. After the completion, the registry is updated and the underlying pointer is returned. The get method must be called before the received data is used or modified and before the sent data is modified. The registry check is performed in the destructor of CommBuff and finish method of the Topology object. This ensures that all the communication which was initiated but not waited on is finished, preventing undesirable results.

Consider the example code shown in figure 6.3. Similar to previous examples, CommBuff objects are used for non-blocking communication. The received data is accessed inside an if statement (line 12). get method is called to retrieve data from rb before accessing the data (line 11). This get call will only be executed if and only if the if statement condition evaluates to true. For the object sb, the get method is never called. The Kanor runtime calls wait methods on the wrapped data, rcvr_data and sndr_data, as soon as the CommBuff objects, rb and sb, go out of scope on line 18. This ensures that all the asynchronous communication that has been started, will be waited upon. The runtime also does not allow the same data to be communicated before it has been waited upon, either through a get call or through a CommBuff destructor. The runtime checks the registry for the presence of data to be communicated to ensure that the same data is not in flight in another communication statement. For the code shown in figure 6.3, if the communication call on line 8 is executed again without any intervening get calls on sb or rb, a Kanor runtime exception will be thrown. The two conditions, 1. waiting on data in get or CommBuff destructor and 2. not allowing communication of same data
rtype* rcvr_data = ...;
stype* sndr_data = ...;
{
  // pack data in CommBuffs
  CommBuff<rtype> rb(rcvr_data);
  CommBuff<stype> sb(sndr_data);
  rb _at_ rcvr << sb _at_ sndr | ...;
  ...
  if(condition) {
    rcvd_data = rb.get();
    ... = rcvd_data[];
  } else {
    ...
  }
// CommBuffs rb and sb out of scope

Figure 6.3: Non-Blocking Communication Safety

without prior get calls, together ensure that each asynchronous call will be waited once and exactly once.

Kanor communication statements are non-blocking by default. If blocking semantics are desired, the hint IMMEDIATE must be used. In case of non-blocking communication, in the absence of a source-to-source translator, the programmer is responsible for the placement of get calls before use or update of communicated data. In the next section, we see a static analysis that tries to do this placement automatically.

6.3 STATİC APPROACH

In this section, the data being sent is denoted by the sender buffer variable svar and data being receive is denoted by receiver variable rvar. We assume that the control flow graph (CFG) has one entry and one exit node. The communication
A definition of a variable is a potential update node for that variable. A use of a variable accesses that variable but does not define it. Every path from entry to the communication statement node has at least one definition of the sender variable $svar$. The communication node defines the receiver variable $rvar$. A definition $d_1$ reaches a node $n$ iff there is a path from the $d_1$ node to $n$ without an intervening definition node $d_2$ that defines the same variable as $d_1$.

We would like to replace the blocking communication statement into non-blocking receives for $rvar$ and non-blocking sends for $svar$ and corresponding waits. We want to maximize the overlap of communication with computation. This means that we would like to place the non-blocking calls as early as possible and the corresponding waits as late as possible.

We start with the placement of non-blocking receives and corresponding waits for $rvar$. On all paths from the start node to the communication node, one and only one node must start non-blocking receives. All other paths that do not reach the communication node, no non-blocking receive must be started. On all paths from the non-blocking receive node to the communication node, there should be no definition or use of $rvar$. All pending non-blocking receives must finish before $rvar$ is used or defined. Finally, every placement of non-blocking receives must be balanced by exactly one placement of wait calls. There can be multiple placements that satisfy these conditions. We want to find the placement that places non-blocking receives as early as possible and the waits as late as possible.

We first make some observations regarding the placement of non-blocking receives.

**Observation 1.** A node $n$ is a safe placement for non-blocking receive if 1. $n$ does
not define or use \texttt{rvar} 2. all paths from \( n \) to exit go through the communication node \( n_c \).

We cannot start non-blocking receives before any definition (other than at \( n_c \)) since we do not want to overwrite \texttt{rvar} before \( n_c \). Also we do not want the overwrite \texttt{rvar} before a use. The second condition implies that we only start non-blocking receives when needed. This observation suggests we perform an analysis similar to anticipable expressions.

\[
\text{ANTC}(n_f) = \text{false}
\]

\[
\text{ANTC}(n) = \text{COMM}(n) \lor (\text{TRANS}(n) \land \bigwedge_{s \in \text{Succ}(n)} \text{ANTC}(s))
\]

\text{COMM}(n) \text{ is true if } n = n_c \text{ and false otherwise. } \text{TRANS}(n) \text{ is true if } n = \text{does not modify our use } \texttt{rvar} \text{ and false otherwise. } \text{ANTS}(n) \text{ is true if on all paths from } n, \text{ communication occurs and there is no use or def of } \texttt{rvar}. \text{ Note that } \text{ANTS}(n_c) \text{ is always true. We call a node } n \text{ safe if } \text{ANTS}(n) \text{ is true.}

Next we define an predicate on edges connecting nodes.

\[
\text{EARLIEST}(i,j) = \overline{\text{ANTS}(i)} \land \text{ANTS}(j) \quad i,j \text{ are nodes}
\]

**Lemma 4.** On all paths from entry node \( n_e \) to communication node \( n_c \), there is one and only one edge for which \text{EARLIEST} is true.

*Proof.* We consider an arbitrary path \( p \) from \( n_e \) to \( n_c \).

Case 1: There is no safe node on \( p \) other than \( n_c \). The predecessor node \( n_p \) must not be safe. Hence, \text{EARLIEST}(n_p,n_c) = \text{true}. For all other nodes \( n_1, n_2 \) on \( p \),
ANTC = false, hence EARLIEST(n₁, n₂) = false.

Case 2: There are one or more safe nodes on p. In this case, we look at the first safe node on p, n₁. All nodes from n₁ to nₖ on p must be safe according to the second equation for ANTC calculation. The predecessor of n₁, nₚ must not be safe, hence EARLIEST(nₚ, n₁) = true. For all other edges, EARLIEST is false.

If EARLIEST(i, j) = true and i has one successor, then we place the non-blocking receives at the end of i. If j has one predecessor, we place non-blocking at the start of j. Otherwise, (i, j) is a critical edge so we split it and place the non-blocking receives on the newly created node.

Lemma 5. EARLIEST(i, j) = true implies that non-blocking receives cannot be moved earlier on the path.

Proof. EARLIEST(i, j) = true implies that the node i is not a safe node. All predecessors of i cannot be safe because of the second equation in calculation of ANTC.

Lemmas 4 and 5 together imply that EARLIEST is a safe and optimal placement for non-blocking receives.

The problem of placing waits is essentially the reverse of the problem of placing non-blocking receives. In this case, we perform a forward dataflow analysis to find the all nodes which are dominated by nₖ and which do not use or define rvar. Then on all paths from nₖ to nᵢ, we find the farthest such node and insert wait calls on the edge that connects that node to its unsafe successor. The dataflow equations
are as follows:

\[
WSAFE(n_e) = false
\]

\[
WSAFE(n) = COMM(n) \lor (TRANS(n) \land \bigwedge_{p \in PRED(n)} WSAFE(p))
\]

Next we define a predicate \( LATEST \) on edges.

\[
LATEST(i, j) = WSAFE(i) \land WSAFE(j) \quad i, j \text{ are nodes}
\]

We place wait calls on the edge \((i, j)\) if \( LATEST(i, j) = true \) similar to \( EARLIEST \).

For the placement of non-blocking sends for \texttt{svar}, we only need to look for defs of \texttt{svar}. Similarly, for wait placement, we only need to place waits before any defs of \texttt{svar}. The dataflow equations are similar as above.

### 6.3.1 EVALUATION

We are in the process of evaluating the two approaches on a Jacobi iteration benchmark with the matrix partitioned block-wise on a two-dimensional grid of processes.

### 6.4 SOFTWARE PIPELINING IN KANOR

The two approaches described in this chapter can be effective but might not exploit all opportunities of overlap. One of the key optimization techniques in high performance parallel programs is software pipelining. Compiler support for
pipelining varies across languages[53]. In this section, we present an algorithm that transforms Kanor programs into pipelined programs.

Pipelining is based on the observation that we can divide communication buffers into smaller chunks and start communicating a chunk while the computation is working on getting the next chunk ready, thus achieving overlap. Pipelining is especially useful in cases where there are cross-processor data dependencies that limit parallelism, as the receivers wait for the senders to finish. Strip-mining the computation creates a pipeline, leading to parallelism that did not exist in the original form of the code. The algorithm first identifies the cases where opportunities for software pipelining might exist and then transforms the communication and its dependent computation to take advantage of the pipeline.

In order for a communication step to lead to software pipelining, certain conditions must hold.

**GLOBAL KNOWLEDGE CASE** All free variables in the communication statement must evaluate to the same value on all processors (we refer to this as the global knowledge case [44]). This means that no free variable is rank- or I/O-dependent. The compiler performs information flow analysis to make this determination. The dependence may be due to either data or control.

**SENDING AND RECEIVING RANKS** All processors must take part in communication. Specifically, for $N$ processors, senders $\in \{0\ldots N-2\}$, and receivers $\in \{1\ldots N-1\}$. To verify this, the compiler analyzes the filters in the where clause, which are boolean conditions on the rank variables. The filter conditions, formulated as a constraint satisfaction problem, must have a solution such that
sender ranks $\in \{0 \ldots N - 2\}$ and receiver ranks $\in \{1 \ldots N - 1\}$. Strictly speaking, not all processors need to participate in a communication step for software pipelining to be useful, only a sufficiently large number of them. However, a collective communication operation involving all processors is a common case in most current data-parallel applications.

**DIRECTIONAL COMMUNICATION** We require that there exist a total ordering of processor ranks (say, denoted by $\prec$), such that $p$ sends data to $q \Rightarrow p \prec q$. Unconstrained communication patterns can substantially complicate the problem of determining whether all the data flows in a single direction. Fortunately, most practical applications that can benefit from software pipelining follow simple communication patterns. Nevertheless, multi-dimensional processor grids—common in high-performance code—make the problem non-trivial to solve. An imprecise test that works in practice is verifying that the data flows in only one direction individually along each processor dimension. If it does then the overall data movement is also in a single direction. If it does not, there might still be a total ordering of processors that results in a directional movement of data, however we interpret the outcome to mean that software pipelining does not apply.

**GRANULARITY OF COMMUNICATION** Since we are primarily interested in coarse-grained software pipelining that is useful on clusters, it is desirable to have larger granularity of communicated data. Thus, we assume that communication has already been aggregated where possible [49]. Only those communicate steps are candidates for software pipelining where the communicated
Algorithm: Generate Software Pipeline

Input: Code region, $C$; Communicate block, $B$; Strip-mining factor, $S$; Dependence graph, $G$

Output: Software pipelined code

1. $F \leftarrow$ empty set of loops
2. $F \leftarrow$ Strip-mine $B$
3. for each loop, $L \in C$, that uses $B$’s send or receive buffer do
   4. $F \leftarrow F \cup$ Strip-mine $L$
5. $F' \leftarrow$ typed-loop-fusion($F$, $G$) // apply Allen and Kennedy’s algorithm [49]
6. abort, if fusion fails
7. return $C$, with $F$ replaced by $F'$

Algorithm 2: Algorithm to generate software pipelined code from Kanor.

data size, after aggregation, is “sufficiently” large. Once a candidate communication statement has been identified, the compiler uses Algorithm 2 to generate pipelined code. It takes the communicate block and its surrounding code block, which it will attempt to pipeline. The algorithm also takes the strip-mining factor as a parameter. We leave determining the exact value of this factor for future work.

Figure 7.7 shows examples of the benchmarks we studied, and the pipelined version for one of them. Note that the transformed code is still in Kanor, not MPI. The code is eventually translated into MPI through a sequence of steps that have been omitted here for the sake of brevity. Correctness of the transformation follows from the observation that it is always legal to strip-mine communication, as well as element-wise array computation, and from the correctness of Allen and Kennedy’s loop-fusion algorithm.
for (int i = 0; i < OCTANTS; i++) {
    for (int j = 0; j < ANGLES; j++) {
        // loop though the diagonals, N is the number of processors
        for (int diag = 0; diag < 2 * N + 1; diag++) {
/* wave front */
            if ((myid.x + myid.y) == diag) { compute(); }

            // sending row
            temp_sသို့ rcvr << A[Slice(lastrow, n_elems)] သို့ sndr |
            _for_each(sndr, mpi.world) & _for_each(rcvr, mpi.world) &
            _let(r_x, rcvr / np) & _let(r_y, rcvr % np) &
            _let(s_x, sndr / np) & _let(s_y, sndr % np) &
            _if(((s_x + s_y) == diag) && (r_x == s_x) && (r_y == (s_y + 1)))
            _with mpi;

            // sending column
            temp_eသို့ rcvr << A[Slice(0, n_elems, n_elems)] သို့ sndr |
            _for_each(sndr, mpi.world) & _for_each(rcvr, mpi.world) &
            _let(r_x, rcvr / np) & _let(r_y, rcvr % np) &
            _let(s_x, sndr / np) & _let(s_y, sndr % np) &
            _if(((s_x + s_y) == diag) && (r_x == (s_x + 1)) && (r_y == s_y))
            _with mpi;
        }
    }
}

Figure 6.4: Sweep3D kernel in Kanor

6.4.1 EVALUATION

We studied three well-known kernels used in high performance numerical
applications, iterative Jacobi, Cholesky factorization, and Sweep3D. Each of these
kernels operates on dense matrices and is amenable to software pipelining, but each
exhibits distinct communication and computation patterns.

Figure 6.4 shows how one of the benchmarks (Sweep3D) is coded in Kanor. The
figure 6.5 shows the resulting software pipelined code after applying Algorithm 2 on
one of the Sweep3D benchmark. The code in the figure uses a two-dimensional
virtual processor grid. Pipelined versions of others are similar and have been
omitted for the sake of brevity.

1Determining the right threshold data size is an orthogonal problem, which is outside the scope
of this paper.
for (int i = 0; i < OCTANTS; i++) {
    for (int j = 0; j < ANGLES; j++) {
        for (int s = 0; s < min(SIZE, s + BLOCK_SIZE); s+=BLOCK_SIZE) {
            // loop though the diagonals, N is the number of processors
            for (int diag = 0; diag < 2 * N + 1; diag++) {
                if ((myid.x + myid.y) == diag) { strip_mined_compute(); }
                // sending row
                temp_s[Slice(lastrow+s, BLOCK_SIZE)] _at_ rcvr <<
                A[Slice(lastrow+s, BLOCK_SIZE)] _at_ sndr |
                _for_each(sndr, mpi.world) & _for_each(rcvr, mpi.world) &
                _let(r_x, rcvr / np) & _let(r_y, rcvr % np) &
                _let(s_x, sndr / np) & _let(s_y, sndr % np) &
                _if(((s_x+s_y)==diag) && (r_x==s_x) && (r_y==(s_y + 1)))
                _with mpi;
                // sending column
                temp_e[Slice(lastrow+s, BLOCK_SIZE)] _at_ rcvr <<
                A[Slice(0+s, BLOCK_SIZE, nelems)] _at_ sndr |
                _for_each(sndr, mpi.world) & _for_each(rcvr, mpi.world) &
                _let(r_x, rcvr / np) & _let(r_y, rcvr % np) &
                _let(s_x, sndr / np) & _let(s_y, sndr % np) &
                _if(((s_x+s_y)==diag) && (r_x==(s_x+1)) && (r_y==s_y))
                _with mpi;
            }
        }
    }
}

Figure 6.5: Piplined Sweep3D kernel in Kanor

Performance benefits of software pipelining are well-established. Therefore, a higher performance from pipelined code should be expected. However, in order to gain maximum possible benefit the compiler will need to account for the target platform characteristics to choose an appropriate blocking factor or even decide when software pipelining could be advantageous. We expect to incorporate a cost model into the Kanor compiler to equip it to make platform-specific decisions.

Since Kanor is designed to target clusters as well as shared memory architectures [47], software pipelining decisions may be dramatically different depending on the availability of shared memory for inter-process communication.

For example, lower inter-process communication latencies with shared memory may
make software pipelining worthwhile in a larger set of scenarios.

6.5 RELATED WORK

Bamboo MPI [63] discusses communication and computation overlap by executing MPI programs in dataflow style with the help of programmer annotations. Iancu et.al. [45] discuss a compiler-runtime system to achieve compute-communicate overlap for UPC programs. Their compiler optimizations are targeted for loops with one-sided communication operations for UPC programs. They do not support communication scheduling for irregular collective operations. Also they do not support global scheduling of communication. A hybrid approach with MPI/SMPSs [57] has been tried where the programmer again has to annotate to help the translator figure out tasks. SMPSs is a shared memory based task library so the generated code takes advantage of heterogeneous architectures. Our approach automatically identifies tasks.

Chakrabarti et.al. [15] try to schedule communication globally for pHPF programs. Kanor programs are partitioned address space programs where the remote references are identified by the explicit communication statements. Hoefler et al. [43] discuss algorithms to detect MPI collectives at runtime. Their approach does not require static matching of send, receives. This work can be used in Kanor to detect complex communication patterns out of reach for compiler. The compiler can provide useful information to the runtime in the detection.

Early work on software pipelining in compilers focused on pipelining across loop iterations for instruction-level parallelism [51]. Recently, streaming languages, such
as StreamIt, have implemented coarse-grained software pipelining [35]. However, these languages usually have a more restrictive programming model than Kanor, with emphasis on streaming applications.

Software pipelining has also been studied in the context of PGAS (Partitioned Global Address Space) languages, including X10, UPC, and HPF [6, 18, 42]. Fortran D compiler identified cross-processor loops and tried to strip-mine them [42]. Even though similar in its goal, our analysis occurs in the context of an explicitly parallel program, which requires us to first infer the communication pattern. However, the declarative specification of communication helps the compiler extract an accurate high-level picture of the communication pattern and dependencies.
CHAPTER 7

A SHARED MEMORY BACKEND

7.1 COMPILING FOR SHARED MEMORY

7.1.1 LEVERAGING SHARED MEMORY

An easy way to get correct execution of a partitioned address space program on a shared-memory machine is to map the partitioned address spaces to independent processes that communicate using some inter-process communication mechanism. In particular, MPI is one such mechanism. Unfortunately, even though there are versions of MPI that are optimized for shared memory, MPI semantics limit the optimizations that the MPI library could incorporate. For example, for a send operation MPI must copy the send buffer at least once before the call to send can finish since the sender might modify the buffer content subsequently. However, a compiler that is not tied to using MPI for communication could eliminate the copy
if it can prove that the sender does not modify the buffer after the communication step. Instead, the compiler could generate code to allocate the sender buffer in shared memory and have the receiver read the buffer directly using a shared-memory pointer, eliminating any need for copying the buffer. This could eliminate significant overheads when the communication buffer is large (say, a large array) or has multiple readers (say, in a broadcast).

Consider the following simple hypothetical example in Kanor:

\[ A \text{ at } i \ll B \text{ at } j \mid \ldots \]

Buffer \( B \) at process \( j \) is sent to buffer \( A \) at process \( i \). When hardware shared memory is available, the compiler can implement this communication step by aliasing the variable \( A \) at \( i \) with the variable \( B \) at \( j \), eliminating any buffer copy. However, such an aliasing operation may create data dependencies when there were none in the original program.

The compiler needs to consider two scenarios that might complicate the aliasing: 1. The sender may modify the send buffer (in the above example process \( j \) may modify \( B \) after the communication step); or 2. The receiver may modify the receive buffer (in the above example process \( i \) may modify \( A \) after the communication step).

A second consideration is the range of code within which the variables are aliased. The data dependencies need to be considered only for the duration for which the variables are aliased.

As a code generation strategy, our compiler generates partitioned address space code (i.e., each node code executes in its own independent address space as a

---

1We employ the convention of using names starting with lowercase for scalar variables, names starting with uppercase for array variables, and names in all uppercase for constants.
separate process), with selected portions of data globalized by allocation in shared memory. Globalized data can then be indexed using the process index. Further, since Kanor works with SPMD (Single Program Multiple Data) style of programming, any variable $\alpha$ declared in the original program is available on all processes, each having its local copy. If the compiler chooses to globalize $\alpha$ it does so by allocating $\text{sizeof}(\alpha) \times P$ space in shared memory, where $P$ is the number of processes sharing memory. The compiler modifies the declaration of $\alpha$ by performing array expansion on it, i.e., expanding it by one dimension—if $\alpha$ was a scalar it becomes a vector, and if it was an $n$-dimensional array it becomes an $(n + 1)$-dimensional array. The entire globalized variable $\alpha$ can now be accessed by any process with one extra process dimension.

As an example, consider a left-shift operation, which represents the common pattern of nearest-neighbor communication. $A\_\text{at}_-i << A\_\text{at}_-(i + 1) \ldots i\_\text{right}$ denotes the right neighbor of process $i$. This is a case of ownership transfer [61]. As a result, as long as the compiler can identify and transfer ownership from sender to receiver, no further synchronization is needed. By expanding $A$ along the process dimension, when it is globalized, ownership transfer can be implemented by adjusting the process subscript in globalized $A$. For example, an expression $A[i]$ in the original program translates to $A[i][\text{myShmRank}]$ after $A$ is globalized, where $\text{myShmRank}$ refers to the rank of the current process among those sharing memory. Subsequent to a left-shift all references to $A[i]$ get replaced by $A[i][\text{myShmRank}+1]$. Such array subscript transformations, called beating and dragging along, are well known in optimizing compilers [2]. In certain cases, such as broadcast, it is possible to avoid array expansion.
7.1.2 IDENTIFYING GLOBALIZATION CANDIDATES

In principle, buffer copying can be eliminated in any communication step by appropriate aliasing. However, this can be counterproductive in certain cases, since to maintain correctness the compiler may need to insert shared memory synchronization that might result in serializing the parallel program. Note that the original (partitioned address space) program contains no shared memory synchronization. We divide the possible scenarios into three cases, described next.

CASE 1: NO LOCAL WRITES

Consider the following Kanor code that specifies broadcast:

\[ A \text{ }_{\text{at}} \text{ } i \leftarrow A \text{ }_{\text{at}} \text{ } 0 | \ldots \]

Here, WORLD denotes the set of all processes. Replacing a message-passing broadcast by shared-memory access could potentially lead to big wins, since the communication step reduces to a barrier (or alternatively, a signal-to-all by process 0 on which all others wait).

CASE 2: LOCAL WRITES

Suppose that each process, after consuming the values from A, proceeds to make local changes to A. This is a reasonable operation with partitioned address spaces, where each process has (semantically) local copy of A. However, when the compiler optimizes the step for shared memory by having each process refer to a global A, the compiler must also serialize certain parts of code to maintain the semantics of partitioned address space.

Thus, for the case shown in figure 7.1, all code between overwrite(A) and the
Figure 7.1: Local Writes Case

following `consume(A)` must be enclosed within a critical section, protected by mutex locks. Clearly, if this region of code is a substantial fraction of computation then making it a critical section could more than offset the cost of buffer copying. In Section 7.1.5 we present a strategy to overcome this performance issue. This is an example, where globalization of `A` is not accompanied by expansion. Note that before locally overwriting `A` there needs to be another barrier to ensure that communicated values do not get corrupted by local updates before they get consumed.

**CASE 3: FORCED COPYING**

If the receive buffer is different from send buffer, the compiler could consider aliasing the two. However, the aliasing is not possible if the two variables have overlapping live ranges. In such a case, the compiler resorts to using buffer copying to implement the communication step.

A subtle variation occurs when multiple communication steps involving pairs of processes use common variables and there are local writes into those variables. Globalizing such variables runs the risk of serializing the entire computation through a “domino effect” of aliasing several variables across multiple processes. Fortunately, such occurrences are rare—most communication steps are collective operations. As a result, we consider only those variables candidates for globalization.
x _at_ i << x _at_ 0 | _for_each(i, 1, t.world.size());
...
... = x; // read x
...

x = ... // write x
...
... = x; // read x

Figure 7.2: A simple example of local writes into a globalized variable.

that participate in collective communication. (Handling of array subsections is described in Section 7.1.4.)

7.1.3 AN ALGORITHM TO MINIMIZE CONTENTION

The Control Flow Graph (CFG) of a program is a directed graph where nodes represent simple statements or condition expressions\(^2\) and edges represent possible direct jumps between those nodes on some execution of the program. We work with the CFG of one function at a time. For convenience we assume that there is a unique entry node and a unique exit node in the CFG of each function. Thus, any return statement results in an edge from the return statement to the exit node.

We will assume that all nodes in a CFG are reachable from the entry node.

Fig. 7.2 shows a simple example when \(x\) can be globalized. For our discussion we assume that the sharing occurs subsequent to a communication step. A similar analysis applies if it occurs before the communication step as well. It is easy to infer that correct semantics can be maintained if the shaded area is treated as critical section.

\(^2\)A simple statement is a simple assignment or an expression treated as a statement, which includes function calls. A condition expression occurs, say, in a for- or if-statement. In general, CFGs can be constructed with nodes representing basic blocks. For this paper, we will consider CFG nodes to always be simple statements or condition expressions.
Figure 7.3: CFG of a hypothetical example showing writes into a globalized variable within complex control flow. Dotted path is possible only with unstructured control-flow, such as goto.

In general, when local writes and reads could occur inside arbitrary control flow, the compiler would need to identify an enclosing region of code as the critical section. Fig. 7.3 shows a situation in which the globalized variable could be written and read along multiple paths in an arbitrarily complex portion of the CFG. The circle at the bottom of the figure denotes the exit node, $E$, for the code, which could also be the next communication block. The edges in the graph denote paths—nodes that do not access $x$ have been omitted for clarity. Notice that all paths exiting the region between the communication node, $C$, and the exit node, $E$, go through $E$. We also assume that $C$ is the entry node for all paths leading to any of the nodes shown in the figure, including $E$. This assumption is made here only to simplify the discussion of our algorithm, which works without making any such assumption.

A naïve solution to ensure correctness is to treat the entire subgraph from $C$ to $E$ as critical section. More generally, the critical section lies between a node that
dominates all nodes that access x and a node that post-dominates all nodes that access x. However, this is overly conservative. For example, the leftmost path from C to E needs no locking since it involves only reads from x. In general, we would like to hold the lock for minimum possible amount of time. In order to do that, we define locking set.

**Definition 2.** Locking Set: The set of CFG nodes that lie on a path from a node containing local write into a globalized variable to a node containing read of that value.

Note that the definition precludes nodes that only read global (i.e., communicated) values, because those can be done safely without locking; and also local writes that do not reach any reads, because such writes are redundant.

**Theorem 1.** If the locking set belongs to a critical section then the partitioned address space semantics are maintained.

*Proof.* Any path from a local write to read passes only through nodes in the locking set, by definition. Since all these nodes belong to a critical section, only one process can write into, and read from, the globalized variable at a time. Therefore, each process sees exactly the same values for all the variables that it would see under a strict partitioned address space execution.

Theorem 1 indicates that is the compiler should ensure that a process holds a mutex while executing any node in the locking set. In the absence of any more information about which control flow edges may or may not be taken at runtime, this defines the minimal set of CFG nodes over which the mutex must be held. However, such a
Figure 7.4: Examples CFGs illustrating the subtleties involved in computing locking sets.

region of code may have multiple entries and exits. The compiler must ensure that
the lock is acquired and released exactly once, no matter which path is taken
through the locking set. At the same time, to minimize serialization, we would like
to avoid holding locks for any longer than necessary. Next, we describe a strategy
that achieves both.

Suppose that $W_\alpha$ denotes the set of CFG nodes that contain writes to a variable $\alpha$.
Similarly, suppose that $R_\alpha$ denotes the set of CFG nodes that read the variable $\alpha$.
The locking set is then denoted by $L_\alpha$. It is tempting to compute $L_\alpha$ as the
intersection of nodes that are reachable from $W_\alpha$ and the nodes that can reach $R_\alpha$.
However, this has several problems as illustrated by Fig. 7.4. In each example, grey
colored boxes are not part of the locking set. As before, the circle marked $E$
represents the exit node. The globalized variable is $x$ is the first two examples and $A$
in the third example.

In the leftmost example, since $x$ is written again, the first write to $x$ cannot reach
the second read. Thus, the grey box in the middle represents a statement that
should not be part of the locking set. A simple intersection based approach, as suggested above, would erroneously add that node into the locking set.

In the second example, there is a loop carried dependency due to reuse of x, but each iteration defines a new value that get used in the next one. Notice that the first iteration uses the “global” value of x that comes from the communication statement. Thus, the middle grey box represents a statement during which a local value of x is never live.

Before we discuss the final example, we review the terminology related to data dependencies. A data dependence exists from a statement $S_1$ to $S_2$ if: 1. $S_1$ and $S_2$ access a common memory location, $M$; 2. at least one of the accesses to $M$ is a write; and 3. there is a control flow path from $S_1$ to $S_2$. $S_1$ is said to be the source, and $S_2$ the sink, of the dependence. If $S_1$ and $S_2$ are inside a loop, and the accesses to $M$ occur on different iterations then the dependence is called a loop-carried dependence. Inside a loop-nest, the loop that causes the dependence is said to carry the dependence. The level of the loop-carried dependence is the level of the loop that carries the dependence, the outermost loop being at level 1, as in Fig. 7.4. Any dependence that is not carried by a loop is called a loop-independent dependence.

The dependence distance of a loop-carried dependence is the number of iterations that separate the source of the dependence from the sink.

The rightmost example in Fig. 7.4 illustrates the subtle problems that arrays can cause. There is a loop-carried dependence that is carried by the j-loop, which is at level 2. We use the convention that a statement that is not inside any loop is considered to be at level 0. In this case, all CFG nodes that are at level greater than or equal to those that carry the dependence are part of the locking set.
Finally, we note that a read that has no incoming dependencies should cause no locking, since that indicates read of the global (communicated) value. Similarly, a write without outgoing dependencies should cause no locking. In the rightmost example of Fig. 7.4, this could happen if the i-loop went from 1 to \( N \) and the reference to \( A[i,j+2,k] \) was replaced by, say, \( A[N+1,j+2,k] \).

In order to take such subtleties into account, we make use of data dependence analysis, which is a well-established technique in compilers [49]. We will use the term *looping back-edge* to refer to the critical edge from the last statement in a loop-body to the head node of the loop. In a depth-first search starting from the head node this edge can be detected as a back edge to the head node. We assume that there is a unique last node of the loop-body so that if there are statements that allow the rest of the loop body to be skipped for the current iteration, such as *continue* or *next*, they cause jumps to this unique last node, instead of directly to the head node.

Fig. 3 shows the helper algorithm \textsc{Paths} that computes the set of all nodes lying on any path from \( s \) to \( t \).

**Theorem 2.** Algorithm 3 computes the set of all nodes that lie on any path from node \( s \) to \( t \) in time \( O(|E| + |V|) \).

*Proof.* Lines 7–13 mark all nodes reachable from \( s \) “red”, by doing a BFS starting at \( s \). Similarly, lines 14–22 do a backward BFS (on reverse edges) starting at \( t \), which visits any node that can reach \( t \). Thus, any node added to \( P \) is on a path from \( s \) to \( t \). On the other hand, if there is a path from \( s \) to \( t \) then any node on that path must be reachable from \( s \), and \( t \) must be reachable from any such node. Thus,
Algorithm 3: Algorithm to compute the set of all nodes that lie on any path from $s$ to $t$.

the algorithm will discover that node in the BFS from $s$ as well as in the reverse BFS from $t$, adding it to $P$. Finally, the two BFS steps in the algorithm lead directly to the time complexity of $O(|E| + |V|)$.

In order to arrive at an algorithm to compute the locking set, we make several observations in the form of following lemmas.

**Lemma 6.** For a loop-carried dependence, carried by loop level $l$, all dependence carrying edges in the CFG lie at loop level $l$ or higher and any dependence carrying path must traverse the looping back-edge at level $l$. 
Proof. The proof follows directly from the definition of loop-carried dependencies [49].

Lemma 7. For a loop-independent dependence between statements that are at the common level $l$, no dependence carrying path in the CFG goes through the looping back-edge at level $l$ or lower.

Proof. If the looping back-edge at level $l$ was involved in the dependence it would be a loop-carried dependence.

Lemma 8. Suppose that there is a loop-carried true dependence from a CFG node $w$ to a CFG node $r$ with dependence distance 1 due to a variable $x$, carried by a loop with the head node $h$. Suppose that $P_{u,v}$ denotes the set of nodes on all possible simple paths from $u$ to $v$. Then, the locking set for $x$ due to the dependence from $w$ to $r$ is given by $L_x = P_{w,h} \cup P_{h,r}$.

Proof. For a loop-carried dependence with dependence distance 1, any dependence-carrying path goes through the looping back-edge exactly once. Thus, any such path must start from the write node, $w$, go through the looping back-edge to the head node $h$, and finally to the read node $r$. Since any sub-path from $w$ to $h$ could be composed with any sub-path from $h$ to $r$, the locking set consists of the union of the two.

Lemma 9. Suppose that there is a loop-carried true dependence from a CFG node $w$ to a CFG node $r$ with dependence distance greater than one, due to a variable $x$, carried by a loop with the head node $h$. Then, the locking set for $x$ due to the dependence from $w$ to $r$ is the set of all nodes inside the body of the loop and the head node $h$.
Proof. Since the dependence distance is greater than one, a dependence carrying path from $w$ to $r$ may go through any arbitrary cycle from $h$ to itself, which may involve any nodes from the loop body. Thus, all nodes in the loop body, and the head node, are part of the locking set.

These observations lead us directly to Algorithm 4. The following theorem proves its correctness and time bound.

**Theorem 3.** For a CFG, $G = (V, E)$, Algorithm 4 computes the locking set, $L_x$, associated with a globalized variable, $x$, in $O(\delta \cdot (|E| + |V|))$ time, where $\delta$ is the number of true dependencies (read-after-write) involving $x$.

Proof. The correctness of the algorithm follows in a straightforward manner from the preceding lemmas. Line 6 tests if the dependence between $w$ and $r$ is loop-independent. The if in line 7 succeeds if the loop-independent dependence lies outside any loop, in which case the locking set is computed simply as all possible paths that lie between $w$ and $r$. If the loop-independent dependence is inside a loop-nest, then the algorithm removes all the looping back-edges that cannot lie on a dependence carrying path, according to Lemma 7. If the dependence distance is 1 then the algorithm computes the locking set using Lemma 8. Otherwise, when a loop-carried dependence has dependence distance greater than one (lines 16–18) any path from $w$ to $r$ may carry dependencies, per Lemma 9.

From Theorem 2, each call to PATH costs $O(|V| + |W|)$, leading to the overall time complexity of $O(\delta \cdot (|V| + |E|))$.

As the last step, the locking set is divided into connected components to identify the control-flow edges along which lock acquires and releases should be inserted.
Algorithm: Compute-Locking-Set

Input: CFG $G(V,E)$ of code region over which variable $x$ is globalized, with
level-annotated nodes;
dependence levels, $l_x$, for dependencies involving $x$;
dep. distances, $d_x$, for dependencies involving $x$;

Output: Locking set $L$

$L = \phi$

for each node pair $(w,r)$ with an entry in $l_x$ do
  if $d_x(w,r) = 0$ then
    if $l_x(w,r) = 0$ then
      $L \leftarrow L \cup \text{Paths}(G, w, r)$
    else
      $G'(V', E') \leftarrow G$ without any looping back-edges at level $l_x(w,r)$ and lower
      $L \leftarrow L \cup \text{Paths}(G', w, r)$
  else if $d_x(w,r) = 1$ then
    $h \leftarrow$ head node of loop at level $l_x(w,r)$
    $G'(V', E') \leftarrow G$ restricted to levels $l_x(w,r)$ and higher
    $L \leftarrow L \cup \text{Paths}(G', w, h) \cup \text{Paths}(G', h, r)$
  else
    $G'(V', E') \leftarrow G$ restricted to levels $l_x(w,r)$ and higher
    $L \leftarrow L \cup \text{Paths}(G', w, r)$

return $L$

Algorithm 4: Algorithm to compute the locking set for a given globalized variable.

This is done by splitting an edge and inserting a CFG node to place the lock operation. If all predecessors (or successors) of a node have lock acquires (or releases) then the acquire (or release) can be moved into the node, obviating the need for edge splitting.

Definition 3. Locking section: A connected component of a locking set.

Lemma 10. A critical section consisting of nodes from the locking set can be implemented by inserting lock acquires along each edge going into a locking section and inserting lock releases along each edge exiting a locking section.

Proof. Follows directly from Theorem 1.
We note that while the locking set defines the minimal static set of nodes defining
the critical section, it does not necessarily result in a minimum number of lock
acquires at runtime. In Fig. 7.5, grey boxes represent CFG nodes that are not part
of the locking set. Since the join point of the branch is in the locking set, the read
branch (left branch) needs to acquire the lock before entering that node. However,
that is unnecessary. In an execution where the read branch often executes
consecutively this could lead to a significant overhead of lock acquire and release,
especially, if there is a contention on the lock. Section 7.1.5 addresses this issue.

7.1.4 FUSED GLOBALIZATION

A commonly occurring communication pattern arises when only certain parts of an
array are communicated. For example, here is an abstraction of a communication
step that might occur in 2D successive over-relaxation (SoR) computation, such as
iterative Jacobi. \( A[\text{Slice}((N-1)\cdot N, N)] \_\text{at} \_ i \ll A[\text{Slice}(0, N)] \_\text{at} \_ (i + 1) \_ \ldots \)
This assumes that the parallel program uses shadow columns, which are “synchronized” before each iteration. If we have to eliminate buffer copying here we will need to alias (i.e., overlap) portions of the array $A$ across all processes. In other words, globalization of $A$ involves not just array expansion, but also fusing together certain regions of the expanded array. We call this transformation fused globalization. Once the sections that need to be fused have been identified, the algorithms presented here for inserting synchronization can be applied to the globalized array sections.

Identification of sections to be fused, and proving the validity of fusion, is a difficult problem on its own and orthogonal to the globalization problem addressed in this paper. Consequently, we consider the problem of fusing to be out of the scope of this paper.

### 7.1.5 Runtime Support to Minimize Wait Time

The idea behind reducing synchronization overheads is to dynamically localize those buffers that might cause too much contention, by creating a local (per-process) copy. While at first it appears to be counterproductive to first globalize data, only to be localized again at runtime, in practice this greatly simplifies the compiler and eliminates situations in which conservatively inserted synchronization could lead to unacceptable runtime overheads.

Fig. 7.6 shows the pseudocode for the Kanor compiler’s runtime support for shared memory targets that implements an aggressive synchronization by simultaneously initiating a buffer copy operation in a separate thread. A node can safely enter the “critical section” that was statically identified by the compiler in one of the three
cases, (1) the shared buffer was dynamically localized in an earlier step; (2) the critical section lock is successfully acquired; or (3) the shared buffer is localized by copying it into a local buffer.

Even though the pseudocode suggests that threads are spawned in each call to `acquire_or_lock`, a thread pool could be used if spawning has unacceptable overheads, as turns out to be the case with most current `pthread` implementations.

We emphasize that this approach of *copy-on-conflict* is different from *copy-on-write*, because no copying is performed when there are no runtime conflicts, even if multiple processors write to the same shared memory.

```c++
void acquire_or_copy (Buffer& a, Lock& lock)
{
    if (Localized[a]) return NULL;
    Condition cond;
    enum {COPY_THRD, LOCK_THRD} notifier;
    a_cpy = new Buffer;

    Thread l_thrd =
        spawn(acquire_lock, lock, cond, &notifier);
    Thread c_thrd =
        spawn(buf_copy, a, a_cpy, cond, &notifier);
    wait(cond);

    if (notifier == LOCK_THRD) {
        c_thrd.kill();
        free(a_cpy);
    } else {
        l_thrd.kill();
        if (lock.held()) lock.release();
        delete a;
        a = a_cpy;
        Localized[a] = true;
    }
}
```

Figure 7.6: C++-like pseudocode for smart synchronization. *Localized* is a per-process Boolean valued hash table.
7.2 EXPERIMENTAL EVALUATION

7.2.1 BENCHMARKS

We study four commonly occurring communication patterns, all-to-all, broadcast, shift, and reduction. In each case the MPI and shared memory codes are those emitted by a proof-of-concept Kanor compiler using the algorithms presented in this paper. Fig. 7.7 summarizes the benchmarks.

In each case, if the send and receive buffers are different variables then those two variables are aliased by the compiler. Clearly, such an aliasing is not possible if the original code had overlapping live ranges of the variables. In that case, the shared memory code would also include call to memcpy (not shown in the figure).

We measured the performance of MPI on a multicore machine with 8 cores (AMD Opteron 2356, Gentoo Linux 2.6, two quad cores, 8GB memory), with MPI libraries (OpenMPI 1.4.3) optimized for shared memory. We compared that to a version that used shared memory directly, using our algorithms and our runtime system. We used 8 processes in each case and increased the buffer size until MPI’s shared memory version could no longer handle it. The shared memory version translated from Kanor used mmap. Fig. 7.8 shows four graphs corresponding to the three cases listed in Section 7.1.2 with the case 2 divided into two subcases: 2(a) when the process successfully acquires the lock; and 2(b) when the process localizes the buffer while waiting for the lock. For case 2, we use a pool of waiting threads instead of dynamically spawning threads.

For each benchmark, we observe that shared memory versions consistently perform better than MPI for each case. In the best case, there could be several orders of
<table>
<thead>
<tr>
<th>Op</th>
<th>Kanor</th>
<th>MPI</th>
<th>Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>all</td>
<td>A[j] <em>at</em> i &lt;&lt; A[i] <em>at</em> j _for_each(i, world) &amp; _for_each(j, world)</td>
<td>MPI_Alltoall (...)</td>
<td>barrier();</td>
</tr>
<tr>
<td>b’cast</td>
<td>A <em>at</em> i &lt;&lt; A <em>at</em> 0 _for_each(i, world)</td>
<td>MPI_Bcast(A, ..., ..., 0, ...)</td>
<td>barrier();</td>
</tr>
<tr>
<td>shift</td>
<td>A <em>at</em> i &lt;&lt; A <em>at</em> (i+1) _for_each(i, 0, n -1)</td>
<td>if (Rank == (numprocs - 1)) { dest = 0; } else { dest = Rank + 1; } MPI_Send(A, array_size, ...) ; MPI_Recv(A, array_size, ...) ;</td>
<td>barrier();</td>
</tr>
<tr>
<td>reduce</td>
<td>A <em>at</em> 0 &lt;&lt; op &lt;&lt; A <em>at</em> i _for_each(i, world)</td>
<td>MPI_Reduce (...) // or specialized code for // tree-reduction of ‘’op’’ loop for // tree-reduction for (i ...) { A[i] = op(..); }</td>
<td>// or specialized code for // tree-reduction of ‘’op’’ loop for // tree-reduction for (i ...) { A[i] = op(..); }</td>
</tr>
</tbody>
</table>

Figure 7.7: Communication benchmarks in Kanor and their equivalent MPI and shared memory code generated by Kanor compiler.
Figure 7.8: Experimental evaluation of some of the commonly occurring collective communication patterns, compared to MPI.
magnitude difference. In the worst case, exemplified by cases 2(b) and 3, when buffers need to be copied, the compiler-generated shared memory version is no worse than MPI but, unsurprisingly, approaches MPI performance for very large buffers as the copy cost dominates. Similarly, in the case of reduction, the reducing cost dominates for large buffer sizes. Note that only case 1 applies for reduction since there is no need to ever copy the buffers when performing reduction. However, the speedup over MPI asymptotically approaches one as the time spent in performing the reduction operation dominates the communication time for very large buffers.

7.3 RELATED WORK

Denning [29] introduced information flow analysis as a means to check security violations inside a program. Since then the information flow methods have been used in security related analyses [70].

Two kinds of information flows are defined:

(1) explicit flow, \( x \Rightarrow y \) occurs whenever the operations generating it are independent of the value of \( x \), such as assignment statements, I/O statements, and value-returning procedure calls; and (2) implicit flows, \( x \Rightarrow y \) occurs whenever a statement specifies a flow from some arbitrary \( z \) to \( y \), but execution depends on the value of \( x \).

We have used information flows to discover an interesting property of parallel program in Kanor, the global knowledge case, which is described in details elsewhere [44].

Several past efforts have tried to optimize MPI on shared memory. These
approaches concentrate on optimizing the sends and receives and the collectives at runtime. Buntinas et. al. [13], for example, use a different interface to send large messages and reduce buffering overhead at runtime. In contrast, we statically analyze the program to produce optimal code. To the best of our knowledge, ours is the first compiler-based approach that eliminates redundant buffer copies for message passing programs on shared memory.

There have been various approaches to show absence of deadlocks or race conditions in a program. Sarkar and Simons [72] talk about using parallel program graphs to show the absence of deadlocks. Since then people have used their approach to extend traditional optimizations for sequential programs [52], including model checking [1] and dataflow analysis [79]. Bronevetsky et. al. [11] describe a form of CFG for data parallel message passing programs. One of the uses of the pCFG is to do send-receive matching in MPI programs. For our analysis, the sends and receives are generated by the compiler and hence already matched. Similarly, compiler can ensure that it never inserts locks that might lead to deadlocks, as we have illustrated in this paper.

Negara et al. [61] discuss ownership transfer for efficient message passing. Their analyses are based on the actor model of communication. Various other researchers have tried to provide type systems to infer ownership of objects [39]. These type systems are too restrictive. Approaches have been proposed to infer these types automatically. Our approach uses a mix of static and dynamic techniques.
CHAPTER 8

CONCLUSIONS AND FUTURE WORK

8.1 CONCLUSIONS

Parallel programming remains a challenging activity for programmers. Large-scale parallel programs are written with the standardized and portable MPI library. Although MPI remains popular, the communication interface offered by MPI is too low-level. Programmers often end up writing point-to-point send/receive calls to encode the communication pattern for clarity and performance.

In this thesis, we have presented a Kanor language for declarative communication. We have seen how Kanor provides abstractions that help programmers specify communication patterns declaratively, at a higher level of abstraction. Declarative nature of Kanor allows programmers to write complex communication patterns including but not limited to MPI collectives.

In chapter Kanor Design, we explained Kanor syntax with multiple examples and
saw how it is more expressive than MPI. We presented an operational semantics for a subset of Kanor and discussed properties of Kanor programs. We saw how the well-formedness property influences deadlock freedom and determinism.

In chapter Kanor Implementation, we saw the design of the library that provides Kanor abstractions using techniques such as operator overloading, expression templates, and metaprogramming. These C++ features help the runtime analyze communication statement structure and check syntactic requirements at compile time. Kanor provides several abstractions such as the ProcID class, CommBuff class, Topology class. The ProcID abstraction provides a way to give a logical structure to Kanor processes. Kanor runtime uses the Topology class to communicate data. Users can provide their own implementation of Topology and thus provide different backends for Kanor. Programmers use CommBuff objects in non-blocking Kanor communication. CommBuff with its get method is similar to a smart pointer in C++. The smart pointer like semantics make sure of disciplined use of non-blocking semantics and help with correctness. CommBuff and VarWrapper classes help in efficient communication of subregions of a larger structure in local memory.

We presented runtime techniques to identify and optimize communication patterns in chapter Runtime Optimizations. The runtime statically matches different Kanor syntactic forms, e.g. Slice, to help with communication pattern detection. We described different types of process knowledge that help runtime in optimization. We presented an information flow analysis that helps determine global vs other kinds of knowledge in Kanor. Communication invariance helps us cache communication pattern and helps us avoid the analysis used to determine the pattern. Determination of global knowledge and communication invariance helps
us match MPI performance in several different benchmarks.

In chapter Compute Communicate Overlap, we discussed two approaches to overlap computation with communication in Kanor programs. Programmers use smart pointer like semantics of CommBuff to use non-blocking communication semantics. This is a pure runtime approach to overlap. A compiler transformation based on ideas from partial redundancy elimination (PRE) [10] is used to find earliest placements for asynchronous send/receives, as well as latest placement of the matching wait calls. Kanor’s approach of encoding communication in a single statement makes the static analyzer’s job much easier compared to the static analyzer for MPI where communication is strewn all over the program. We also discussed how this analysis can be used in tandem with other analyses to find more overlap opportunities.

Finally, in chapter A Shared Memory Backend, we presented techniques that exploit shared memory to reduce message copies in communication. The runtime uses Boost managed shared memory to allocate communication buffers and data is directly copied from the sender buffer to receiver buffer. Compared to MPI, which has to make at least two copies because of partitioned address spaces, Kanor is able to minimize copies. The runtime also uses a hybrid lock or copy approach to minimize contention and still ensure correctness. The evaluation shows performance benefits compared to MPI. This approach is a step towards supporting heterogeneous backends involving GPUs, distributed clusters with many cores on a node etc.
8.2 FUTURE WORK

The work presented in this thesis can be extended in several directions. We discuss two important extensions in this section.

8.2.1 SCALABILITY

Results from our experiments have shown that Kanor matches MPI at the scale of hundreds of processes. While these results are encouraging, larger applications need to be written in Kanor and tested up to millions of processes. This might just mean that we translate existing MPI codes to their Kanor equivalent codes. This translation requires that the programmers to think about the communication pattern and express that in terms of a Kanor communication statement. This translation should still be easier than porting the program to a completely new language. Running at the scale of millions of processes brings challenges such as fault tolerance and it would be interesting to solve these challenges for a BSP styled language like Kanor.

8.2.2 IRREGULAR APPLICATIONS

The thesis presented optimizations that work well for benchmarks that have a regular communication pattern. Irregular application kernels, such as graph algorithms, contain communication patterns that might be data dependent and change over program execution. Graph frameworks such as Pregel [56], Giraph [40] have shown that graph applications can be written in BSP style and benefit from its simplicity. Ediger et.al. [31] discuss implementation of several graph algorithms in
the BSP style. Similarly, others [55, 24, 23] have shown that irregular applications can be written in and optimized in the BSP style.

We have written graph algorithms like Breadth First Search in Kanor. This BFS implementation is uses a two dimensional block distribution of the adjacency matrix [12]. It will be interesting write other graph benchmarks and irregular applications such as Algebraic Multigrid in Kanor and measure their performance against optimized implementations.

One possible approach to tackle these problems, is to convert Kanor programs to use task based runtimes such as HPX-5 [75]. HPX-5, for example, is a lightweight task based runtime meant for programming exascale machines. HPX-5 programs are programmed in continuation passing style (CPS) [3], where continuation functions are tasks and the programmer writes all the code to wire all the tasks together with different control, synchronization constructs. While CPS simplifies the job of HPX-5 runtime, programmers find it difficult to keep track of the control flow. Kanor can use HPX-5 as a backend. The translator from Kanor to HPX-5 will have to identify tasks and their dependencies. There are interesting research problems, such determining task granularity, that need to explored as part of this work.
BIBLIOGRAPHY


Workshop on Leveraging Abstractions and Semantics in High-performance Computing (LASH-C; Workshop at PPoPP’13) (February 2013).

[22] CZARNECKI, K., O’DONNELL, JOHN, T., STRIEGNITZ, J., AND TAH, W. 


[59] MILLOT, D., MULLER, A., PARROT, C., AND SILBER-CHAUSSUMIER, F.  
From openmp to mpi: first experiments of the step source-to-source  

[60] MPI FORUM. MPI: A Message-Passing Interface Standard. Version 2.2,  
September 4th 2009.

[61] NEGARA, S., KARMANI, R. K., AND AGHA, G. Inferring ownership transfer  
for efficient message passing. SIGPLAN Not. 46, 8 (Feb. 2011), 81–90.


Computing, Networking, Storage and Analysis (Los Alamitos, CA, USA, 2012),  


[65] PHARR, M., AND FERNANDO, R. GPU Gems 2: Programming Techniques for  
High-Performance Graphics and General-Purpose Computation (Gpu Gems).  
Addison-Wesley Professional, 2005.

[66] QUINLAN, D., LIAO, C., PANAS, T., MATZKE, R., SCHORDAN, M.,  
VUDUC, R., AND YI, Q. ROSE User Manual: A Tool for Building  
Source-to-Source Translators, version 0.9.5a ed. Lawrence Livermore National  
Laboratory, Livermore, CA 94550, Nov. 2010.
[67] Rabenseifner, R. Hybrid mpi and openmp parallel programming.


Machine and Message Passing Interface (Berlin, Heidelberg, 2008),

[74] Squyres, J. M., McCandless, B. C., and Lumsdaine, A. Object
oriented mpi: A class library for the message passing interface.

[75] Sterling, T., Anderson, M., Bohan, P. K., Brodowicz, M.,
Kulkarni, A., and Zhang, B. Towards exascale co-design in a runtime
system. In Exascale Applications and Software Conference (Stockholm,
Sweden, Apr 2014).

[76] Strout, M. M., LaMielle, A., Carter, L., Ferrante, J., Kreaseck,
B., and Olschanowsky, C. An approach for code generation in the sparse

[77] Tang, Y., Chowdhury, R. A., Kuszmaul, B. C., Luk, C.-K., and
Leiserson, C. E. The pochoir stencil compiler. In Proceedings of the
Twenty-third Annual ACM Symposium on Parallelism in Algorithms and

[78] Valiant, L. G. Bulk-synchronous parallel computers. In Parallel Processing

analysis and optimization of shared memory parallel programs. Tech. rep.,
University of Delaware, 2005.

