

Partial Globalization of Partitioned Address Spaces for Zero-copy Communication with Shared Memory

Fangzhou Jiao Nilesh Mahajan Jeremiah Willcock Arun Chauhan Andrew Lumsdaine

Email: {fjiao, nmmahaja, jewillco, achauhan, lums}@cs.indiana.edu

School of Informatics and Computing, Indiana University, Bloomington, Indiana 47405

Abstract—We have developed a high-level language, called **Kanor**, for declaratively specifying communication in parallel programs. Designed as an extension of C++, it serves to coordinate partitioned address space programs written in the bulk synchronous parallel (BSP) style. **Kanor**'s declarative semantics enable the programmers to write correct and maintainable parallel applications. The communication abstraction has been carefully designed to be amenable to compiler optimizations.

While partitioned address space programming has several advantages, it needs special compiler optimizations to effectively leverage the shared memory hardware when running on multicore machines. In this paper, we introduce such shared-memory optimizations in the context of **Kanor**. One major way we achieve these optimizations is by selectively moving some of the variables into a globally shared address space—a process that we term *partial globalization*. We identify scenarios in which such a transformation is beneficial, and present an algorithm to identify and correctly transform **Kanor** communication steps into zero-copy communication using hardware shared memory, by introducing minimal synchronization. We then present a runtime strategy that complements the compiler algorithm to eliminate most of the runtime synchronization overheads by using a *copy-on-conflict* technique. Finally, we show that our solution often performs much better than shared-memory optimized MPI, and never performs significantly worse than MPI even in the presence of dependencies introduced due to buffer sharing.

The techniques in this paper demonstrate that it is possible to program in a partitioned address space style, without sacrificing the performance advantages of hardware shared memory. To the best of our knowledge no other automatic compiler techniques have been developed so far that achieve zero-copy communication from a partitioned address space program. We expect our results to be applicable beyond **Kanor**, to other partitioned address space programming environments, such as MPI.

Index Terms—Parallel programming; partitioned address space; compilers; shared memory.

I. INTRODUCTION

Parallelism is now an established method to achieve high performance computing. However, writing parallel programs continues to be a challenge. Even though several approaches have been proposed and used for parallel programming [1], they continue to suffer from various shortcomings. One important shortfall of most existing parallel programming models is their failure to present a seamless mechanism that would provide performance portability across the myriad parallel platforms. For instance, tools that simplify programming on shared memory often do not provide any aid in programming clusters, and tools for programming clusters are often awkward to use, or suboptimal, on shared memory.

We have been developing a *declarative language for parallel programming*, called **Kanor**, that aims to achieve performance portability by carefully abstracting out those aspects of parallel programming that are amenable to compiler optimizations, while significantly simplifying the task of writing parallel programs without constraining experienced programmers from developing clever parallel algorithms [2]. **Kanor** achieves this by letting users write parallel programs with partitioned address spaces, so that programmers may exercise complete freedom in deciding data distributions and alignments. At the same time *declarative* specification of communication eliminates the error-prone and tedious task of optimizing data communication, which is the bane of parallel programming with partitioned address spaces. To encourage incremental development and porting of existing code, **Kanor** is designed as a domain-specific language (DSL), extending a host language (currently, C++). Our compiler translates the user program written with **Kanor** extensions to the standard version of the host language, which may then be compiled using any host language compiler.

Partitioned address spaces map naturally to a programming model that is designed for clusters, such as MPI, which is a common target for **Kanor** programs. Indeed, our compiler implements this translation for distributed memory targets [2]. Parallel programs written using partitioned address spaces provide clear indications to the compiler of data locality on each processing unit, opening the path to further locality-based optimizations. By selectively globalizing the local data, such programs can also benefit from the rapid zero-copy sharing of data that would otherwise require complex hybrid-mode programming involving programmer awareness of shared as well as distributed memory.

In this paper we show that partitioned address spaces can be an efficient parallel programming strategy for shared memory targets—and hybrid targets, consisting of clusters of shared-memory machines—with support from a compiler. We describe a compiler algorithm that decides when sharing, rather than copying, of data is desirable and inserts provably minimal synchronization to ensure correctness. The compiler is supported by a smart runtime system that can break dependencies at runtime by copying data, if needed, thus minimizing synchronization overheads. The approach is a part of our under-development **Kanor-to-C++** compiler, which is written using the ROSE compiler framework [3] and our homegrown tree transformation tool called `RubyWrite`. We

report performance results on microbenchmarks designed to stress-test our technique.

The main contributions of this paper include:

- 1) Demonstrating that a partitioned address space programming can achieve the performance of “natively” written global-address programs on shared memory, by eliminating the buffer-copying overheads.
- 2) A theoretical formulation of the problem of leveraging shared-memory to achieve zero-copy messaging.
- 3) A novel algorithm to decide variables to alias (for zero-copy communication) and to insert provably minimal amount of synchronization, for correctness.
- 4) A novel runtime strategy to largely eliminate the synchronization overheads dynamically when write-dependencies on aliased variables lead to too much wait on synchronization.

II. BACKGROUND

Bulk Synchronous Parallel (BSP) is a style of writing parallel programs [4]. Multiple processors run a single program and operate on different data (*Single Program Multiple Data* or SPMD) and each processor alternates between computation and communication steps. One compute and communicate phase is called a *superstep*. There is usually global synchronization at the end of each superstep.

The communication declaration in Kanor assumes BSP style of programming. The parallel program that uses Kanor for communication follows partitioned address space semantics. Local computation steps are standard host language code. The communication steps use *set comprehensions* syntax to specify the communication pattern, emphasizing that all communication within one communication step is concurrent. A communication step has the following syntax:

$$e_0@e_1 \ll op \ll e_2@e_3 \text{ where } e_4$$

From left-to-right, the reduction comprises four major parts: receiver remote reference, reduction operator, sender remote reference, and qualifier. The variable op must evaluate to a reduction function (like sum); e_4 is the *qualifier* of the set comprehension. Because the majority of communication statements simply move data, we allow *transfer* statements, which are merely syntactic sugar for reductions using a special operator that performs assignment. We assume user-defined reduction operators are both commutative and associative. The behavior of non-commutative or non-associative reduction operators is undefined.

The order of evaluation is unspecified between e_1 , e_2 , and e_3 on the sender and between e_0 and op on the receiver. The operator op is call-by-value, and is applied only after all of its arguments have been evaluated.

The dependency chain within a Kanor communication can only be of length one: no read can depend upon another read. Kanor also uses parallel assignment semantics: all reads occur before all writes. This allows us to perform analyses similar to those in static single assignment languages. Parallel assignment is an important part of Kanor’s semantics, as

it guarantees there are no dependencies within a communication block. Parallel assignment makes it much easier to write programs whose communication patterns contain cycles, such as circular shift. Values destined for the same location are accumulated using the reduction operator. However, it is erroneous to make multiple writes to the same location using Kanor’s assignment operator ($\ll=$); the semantics of overlapping assignments is undefined.

III. GLOBAL KNOWLEDGE INFERENCE

Kanor classifies communications into four different categories based on each process’s knowledge of the global communication pattern [2].

- *Global Knowledge* Each process can determine the entire communication pattern.
- *Corresponding Knowledge* Each process knows only a subset of the complete communication topology, but has complete knowledge of the communication in which it will be participating.
- *Sender Knowledge* Senders know only the destinations they will send to; receivers do not know which senders they will receive from. The sender knowledge case requires the receiver to perform a termination protocol. The Kanor runtime uses the Nonblocking Barrier eXchange (NBX) protocol [5], which allows receiver processes to receive an unknown amount of data with minimal overhead.
- *Receiver Knowledge* Receivers know only the senders they will receive from; senders do not know which receivers they will send to. This case requires the receiver to notify the sender processes from which it wishes to receive data. After this notification, communication then becomes equivalent to the corresponding knowledge case.

These categories do not cover all possible applications or communication patterns. For example, some communications might fit a third-party knowledge pattern. However, these categories cover the majority of today’s parallel applications [4], [6], and other types of communication patterns can be transformed into one of the categories that Kanor supports.

Before variables involved in a communication block can be globalized the compiler must determine the knowledge case in order to insert appropriate synchronization. For the remainder of the paper, we focus on the most common “global knowledge” case.

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 $myid$. Flow dependence is transitive, i.e., if x is data-dependent on y and y is data-dependent on $myid$ then x is flow-dependent on $myid$.
- *Implicit*: When x is control-dependent on $myid$. As before, the dependence may be transitive.

The compiler uses a list-based algorithm to look for this transitive data- and control-dependence of free variables on $myid$ ¹. 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 $myid$ then that variable is dependent on $myid$. 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 $myid$. 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.

IV. COMPILING FOR SHARED MEMORY

A. 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 Kanor².

```
@communicate {A@i <<= B@j}
```

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

¹We use $myid$ as a generic stand-in for any variable that might be used in Kanor to refer to a process's own rank.

²We 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.

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 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 α declared in the original program is available on all processes, each having its local copy. If the compiler chooses to globalize α 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 α by performing array expansion on it, i.e., expanding it by one dimension—if α 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 α 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.

```
@communicate {A@i <<= A@i.right}
```

`i.right` denotes the right neighbor of process i . This is a case of *ownership transfer* [7]. 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 `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 [8]. In certain cases, such as broadcast, it is possible to avoid array expansion.

B. 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.

1) *Case 1: No Local Writes:* Consider the following Kanor code that specifies broadcast:

```
@communicate {A@i <=<= A@0, where i in WORLD}
```

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).

2) *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, in the following case,

```
@communicate{A@i <=<= A@0, where i in WORLD}
consume(A); // do something with A
overwrite(A); // reuse A for new data ...
...
consume(A); // ... consume A again
```

all code between `overwrite(A)` and the 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 IV-E 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.

3) *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 that participate in collective communication. (Handling of array subsections is described in Section IV-D.)

C. An Algorithm to Minimize Contention

The Control Flow Graph (CFG) of a program is a directed graph where nodes represent simple statements or condition

expressions³ 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. 1 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

```
@communicate{x@i <=<= x@0,
  where i > 0}
...
... = x; // read x
...
x = ... // write x
...
... = x; // read x
```

Fig. 1. A simple example of local writes into a globalized variable.

maintained if the shaded area is treated as critical section.

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. 2 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 1. *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.

³A 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.

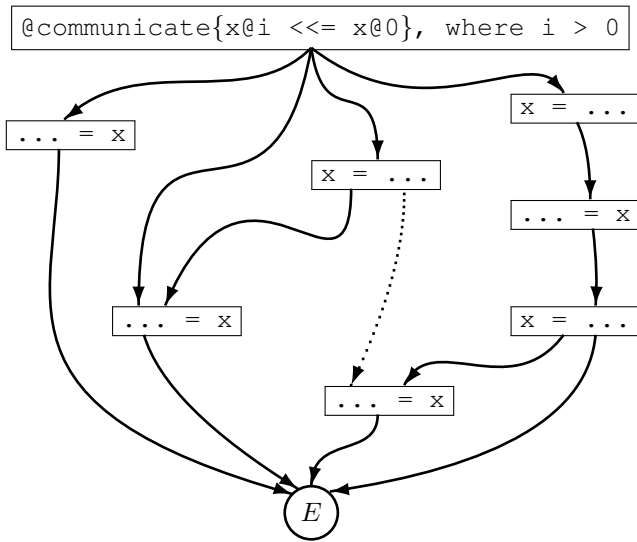


Fig. 2. CFG of a hypothetical example showing writes into a globalized variable within complex control flow. Dotted path is possible only with unstructured control-flow, such as `goto`.

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 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_α denotes the set of CFG nodes that contain writes to a variable α . Similarly, suppose that R_α denotes the set of CFG nodes that read the variable α . The locking set is then denoted by L_α . It is tempting to compute L_α as the intersection of nodes that are reachable from W_α and the nodes that can reach R_α . However, this has several problems as illustrated by Fig. 3. 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. 3. 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. 3 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. 3, 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 [9]. 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. 1 shows the helper algorithm `PATHS` that computes the set of all nodes lying on any path from s to t .

Theorem 2. *Algorithm 1 computes the set of all nodes that lie on any path from node s to t in time $O(|E| + |V|)$.*

Proof: Lines 7–8 mark all nodes reachable from s “red”,

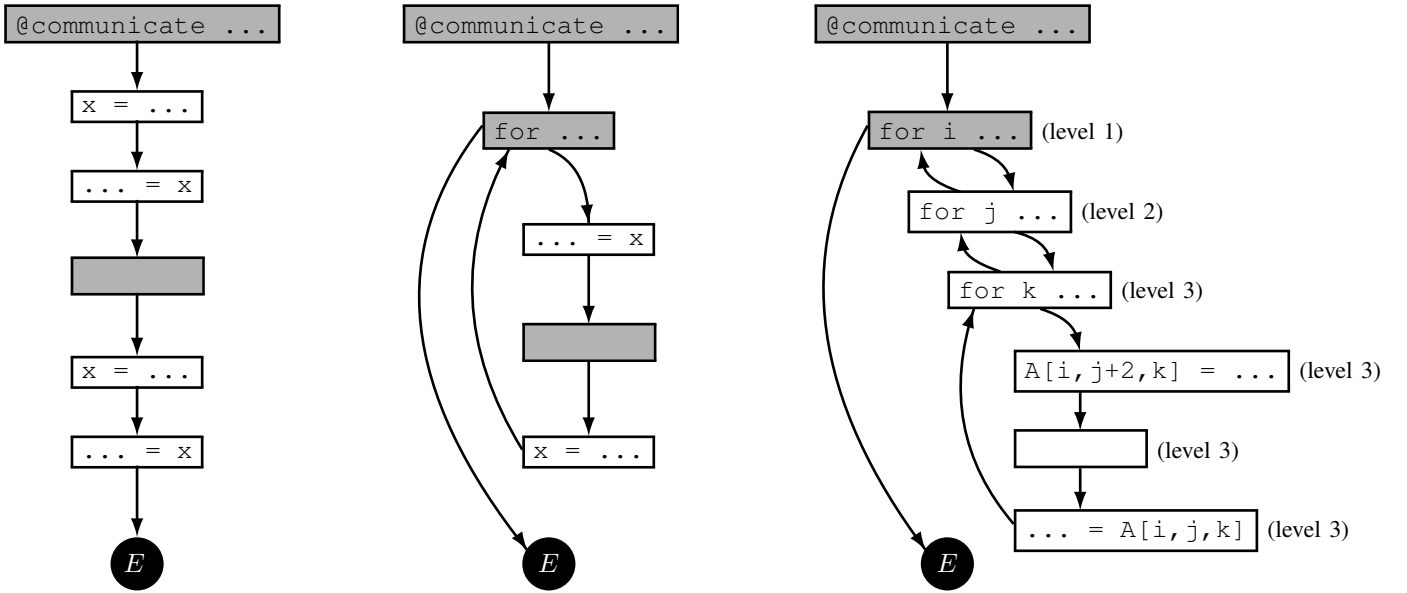


Fig. 3. Examples CFGs illustrating the subtleties involved in computing locking sets.

```

1 Algorithm: PATHS
2 Input: Directed graph  $G(V, E)$ 
   Start node  $s$ 
   End node  $t$ 
3 Output: Set  $P$  of nodes that lie on any path from  $s$  to  $t$ 


---


4  $P \leftarrow \phi$ 
5 for each node  $n$  in  $G$  do
6    $n.color \leftarrow$  "white"
7  $Q \leftarrow [s]$ 
8 while not  $Q.empty$  do
9    $q \leftarrow Q.extract$ 
10  for each edge  $(q, v) \in E$  do
11    if  $v.color \neq$  "red" then
12       $v.color \leftarrow$  "red"
13       $Q.add(v)$ 
14  $Q \leftarrow [t]$ 
15 while not  $Q.empty$  do
16    $q \leftarrow Q.extract$ 
17   for each edge  $(v, q) \in E$  do
18     if  $v.color \neq$  "black" then
19       if  $v.color =$  "red" then
20          $P \leftarrow P \cup \{v\}$ 
21        $v.color \leftarrow$  "black"
22        $Q.add(v)$ 
23 return  $P$ 

```

Algorithm 1: Algorithm to compute the set of all nodes that lie on any path from s to t .

by doing a BFS starting at s . Similarly, lines 14–15 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, 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 1. 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 [9]. ■

Lemma 2. 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 3. 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 4. *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 2. The following theorem proves its correctness and time bound.

Theorem 3. *For a CFG, $G = (V, E)$, Algorithm 2 computes the locking set, L_x , associated with a globalized variable, x , in $O(\delta \cdot (|E| + |V|))$ time, where δ 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 2. If the dependence distance is 1 then the algorithm computes the locking set using Lemma 3. Otherwise, when a loop-carried dependence has dependence distance greater than one (lines 16–5) any path from w to r may carry dependencies, per Lemma 4.

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. 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 2. *Locking section: A connected component of a locking set.*

Lemma 5. *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.

```

1 Algorithm: COMPUTE-LOCKING-SET
2 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  $x$ ;
3 Output: Locking set  $L$ 

4  $L = \phi$ 
5 for each node pair  $(w, r)$  with an entry in  $l_x$  do
6   if  $d_x(w, r) = 0$  then
7     if  $l_x(w, r) = 0$  then
8        $L \leftarrow L \cup \text{PATHS}(G, w, r)$ 
9     else
10       $G'(V', E') \leftarrow G$  without any looping back-edges at
       level  $l_x(w, r)$  and lower
11       $L \leftarrow L \cup \text{PATHS}(G', w, r)$ 
12    else if  $d_x(w, r) = 1$  then
13       $h \leftarrow$  head node of loop at level  $l_x(w, r)$ 
14       $G'(V', E') \leftarrow G$  restricted to levels  $l_x(w, r)$  and higher
15       $L \leftarrow L \cup \text{PATHS}(G', w, h) \cup \text{PATHS}(G', h, r)$ 
16    else
17       $G'(V', E') \leftarrow G$  restricted to levels  $l_x(w, r)$  and higher
18       $L \leftarrow L \cup \text{PATHS}(G', w, r)$ 
19 return  $L$ 

```

Algorithm 2: Algorithm to compute the locking set for a given globalized variable.

In Fig. 4, 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 IV-E addresses this issue.

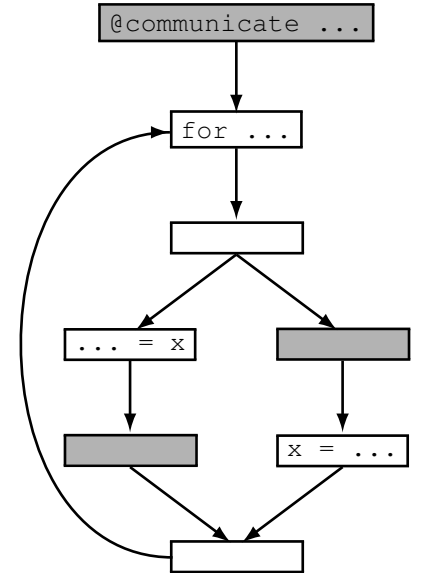


Fig. 4. Example for which locking set is not dynamically minimal.

D. 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.

```
@communicate{A[0:N-1,N]@i <=< A[0:N-1,0]@i.right}
```

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.

E. 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. 5 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.

V. EXPERIMENTAL EVALUATION

A. 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. 6 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 shows four graphs corresponding to the three cases listed in Section IV-B 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 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.

```

1 void acquire_or_copy (Buffer& a, Lock& lock)
2 {
3     if (Localized[a]) return NULL;
4     Condition cond;
5     enum {COPY_THRD, LOCK_THRD} notifier;
6     a_cpy = new Buffer;
7
8     Thread l_thrd =
9         spawn(acquire_lock, lock, cond, &notifier);
10    Thread c_thrd =
11        spawn(buf_copy, a, a_cpy, cond, &notifier);
12    wait(cond);
13
14    if (notifier == LOCK_THRD) {
15        c_thrd.kill();
16        free(a_cpy);
17    } else {
18        l_thrd.kill();
19        if (lock.held()) lock.release();
20        delete a;
21        a = a_cpy;
22        Localized[a] = true;
23    }
24 }

```

Fig. 5. C++-like pseudocode for smart synchronization. `Localized` is a per-process Boolean valued hash table.

<i>Op</i>	<i>Kanor</i>	<i>MPI</i>	<i>Shared Memory</i>
all	$A[j]@i \ll A[i]@j$ where i, j in WORLD	<code>MPI_Alltoall (...)</code>	<code>barrier();</code>
b'cast	$A@i \ll A@0$ where i in WORLD	<code>MPI_Bcast(A, ..., ..., 0, ...);</code>	<code>barrier();</code>
shift	$A@i \ll A@i+1$	if (Rank == (numprocs - 1)) dest = 0; else dest = Rank + 1; <code>MPI_Send(A, array_size, ...);</code> <code>MPI_Recv(A, array_size, ...);</code>	<code>barrier();</code>
reduce	$A@0 \ll_{op} A@i$ where i in WORLD	<code>MPI_Reduce (...)</code> // or specialized code for // tree-reduction of ``op``	// loop for tree-reduction for (i ...) { $A[i] = op(...)$; }

Fig. 6. Communication benchmarks in Kanor and their equivalent MPI and shared memory code generated by Kanor compiler.

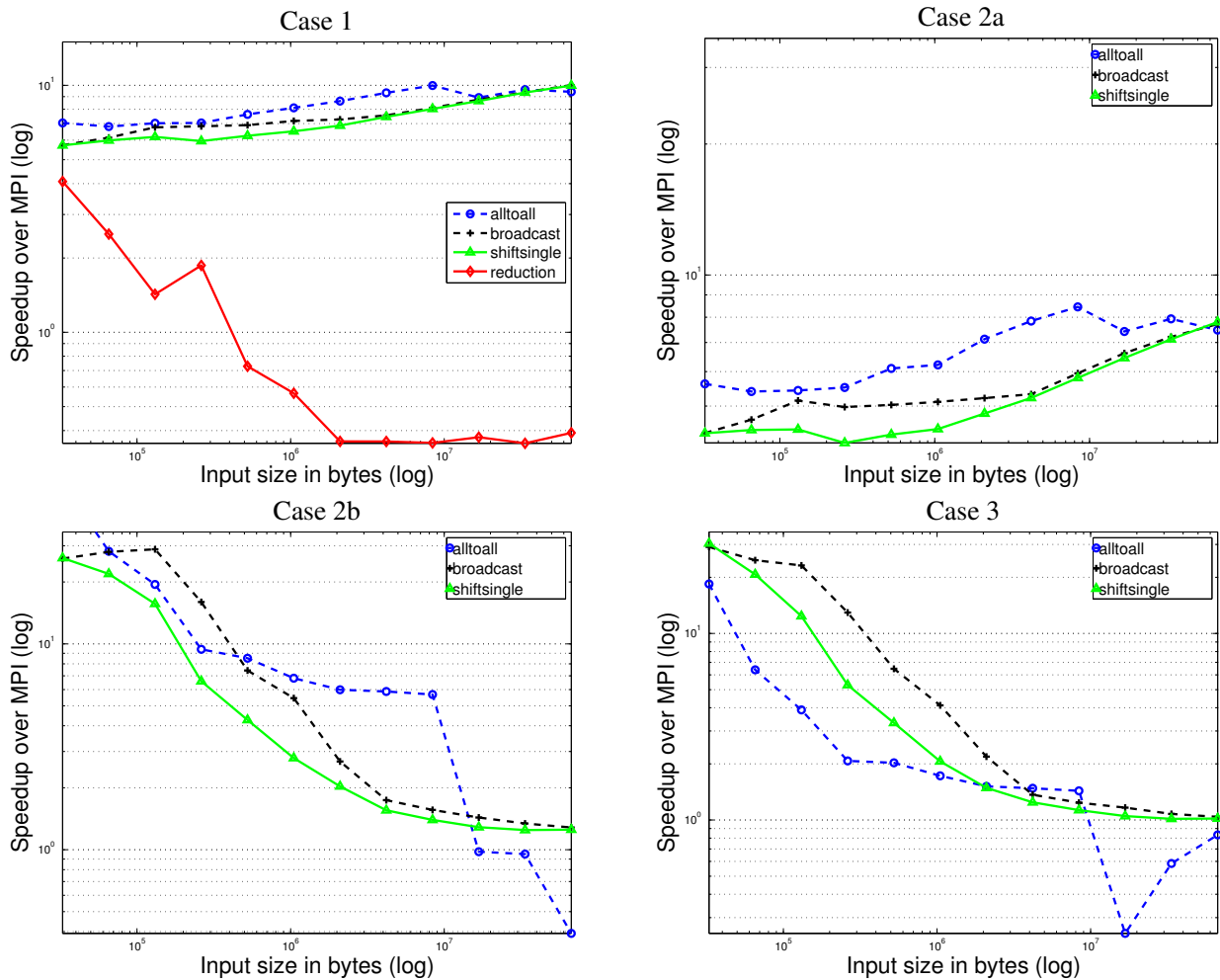


Fig. 7. Experimental evaluation of some of the commonly occurring collective communication patterns, compared to MPI.

VI. RELATED WORK

Denning [10] 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 [11]. 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 [2].

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. [12], 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 [13] 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 [14], including model checking [15] and dataflow analysis [16]. Bronevetsky et. al. [17] 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. [7] 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 [18]. 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.

VII. CONCLUSION

Parallel programming with partitioned address spaces has several advantages, including flexibility in devising parallel algorithms, enabling complicated data distributions, and improved data locality. One issue in programming with partitioned address spaces has been their inability to leverage hardware shared memory on increasingly common multi- and many-core shared-memory hardware, without resorting to complex and error-prone “hybrid” programming models. This paper has presented a compiler strategy to optimize partitioned address space programs for shared memory, in the context of a DSL for declarative communication specification, called Kanor. Our strategy relies on efficient compiler algorithms that build on well established compiler analyses, such as data flow and dependence analysis, and support from a smart runtime system that eliminates overheads when shared memory causes too much runtime contention due to local updates. Our communication benchmarks suggest that our strategy achieves significant performance gains over standard communication over MPI, and incurs minimal overheads when the data must be copied to avoid lock contentions.

We believe that our strategy described in this paper, which has been evaluated in the context of Kanor, is broadly applicable to other similar parallel programming languages that use partitioned address space semantics.

VIII. ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-0834722. We thank our anonymous reviewers for several concrete suggestions for improvements and for pointing out mistakes in the original manuscript.

REFERENCES

- [1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, “The landscape of parallel computing research: A view from Berkeley,” EECS Department, University of California-Berkeley, Berkeley, California, USA, Technical report UCB/EECS-2006-183, Dec. 2006.
- [2] E. Holk, W. E. Byrd, J. Willcock, T. Hoeffer, A. Chauhan, and A. Lumsdaine, “Kanor: A declarative language for explicit communication,” in *Proceedings of the Thirteenth International Symposium on the Practical Aspects of Declarative Languages (PADL)*, 2011.
- [3] D. Quinlan, C. Liao, T. Panas, R. Matzke, M. Schordan, R. Vuduc, and Q. Yi, *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.
- [4] L. G. Valiant, “Bulk-synchronous parallel computers,” in *Parallel Processing and Artificial Intelligence*, M. Reeve, Ed. John Wiley & Sons, 1989, pp. 15–22.
- [5] T. Hoeffer, C. Siebert, and A. Lumsdaine, “Scalable communication protocols for dynamic sparse data exchange,” in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 159–168.
- [6] L. G. Valiant, “A bridging model for parallel computation,” *Communications of the ACM*, vol. 33, no. 8, pp. 103–111, Aug. 1990.
- [7] S. Negara, R. K. Karmani, and G. Agha, “Inferring ownership transfer for efficient message passing,” in *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2011, pp. 81–90.
- [8] P. S. Abrams, “An APL machine,” Doctoral dissertation, Stanford University, Stanford Linear Accelerator Center, Stanford, California, USA, Feb. 1970.
- [9] J. R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. San Francisco, California, USA: Morgan Kaufmann Publishers Inc., 2001.
- [10] D. E. Denning and P. J. Denning, “Certification of programs for secure information flow,” *Communications of the ACM*, vol. 20, pp. 504–513, Jul. 1977.
- [11] A. Sabelfeld and A. C. Myers, “Language-based information-flow security,” *IEEE Journal on Selected Areas in Communications*, vol. 21, p. 2003, 2003.
- [12] D. Buntinas, G. Mercier, and W. D. Gropp, “Implementation and shared-memory evaluation of MPICH2 over the Nemesis communication subsystem,” in *PVM/MPI’06*, 2006, pp. 86–95.
- [13] V. Sarkar and B. Simons, “Parallel program graphs and their classification,” in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, 1994.
- [14] J. Lee, S. P. Midkiff, and D. A. Padua, “A constant propagation algorithm for explicitly parallel programs,” *Int. J. Parallel Program.*, vol. 26, pp. 563–589, Oct. 1998.
- [15] P. A. Abdulla, F. Haziza, and M. Kindahl, “Model checking race-freeness,” *SIGARCH Comput. Archit. News*, vol. 36, pp. 72–79, Jun. 2009.
- [16] Y. Z. Vugranam C. Sreedhar and G. R. Gao, “A new framework for analysis and optimization of shared memory parallel programs,” University of Delaware, Tech. Rep., 2005.
- [17] G. Bronevetsky, “Communication-sensitive static dataflow for parallel message passing applications,” in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’09. IEEE Computer Society, 2009, pp. 1–12.
- [18] P. Haller and M. Odersky, “Capabilities for uniqueness and borrowing,” in *Proceedings of the 24th European conference on Object-oriented programming*, ser. ECOOP’10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 354–378.