Communication-Sensitive Static Dataflow for Parallel Message Passing Applications

Greg Bronevetsky
Center for Applied Scientific Computing,
Lawrence Livermore National Laboratory,
greg@bronevetsky.com

***Abstract*—Message passing is a very popular style of parallel programming, used in a wide variety of applications and supported by many APIs, such as `BSD sockets`, `MPI` and `PVM`. Its importance has motivated significant amounts of research on optimization and debugging techniques for such applications. Although this work has produced impressive results, it has also failed to fulfill its full potential. The reason is that while prior work has focused on runtime techniques, there has been very little work on compiler analyses that understand the properties of parallel message passing applications and use this information to improve application performance and quality of debuggers.**

This paper presents a novel compiler analysis framework that extends dataflow to parallel message passing applications on arbitrary numbers of processes. It works on an extended control-flow graph that includes all possible inter-process interactions of any numbers of processes. This enables dataflow analyses built on top of this framework to incorporate information about the application's parallel behavior and communication topology. The parallel dataflow framework can be instantiated with a variety of specific dataflow analyses as well as abstractions that can tune the accuracy of communication topology detection against its cost.

The proposed framework bridges the gap between prior work on parallel runtime systems and sequential dataflow analyses, enabling new transformations, runtime optimizations and bug detection tools that require knowledge of the application's communication topology. We instantiate this framework with two different symbolic analyses and show how these analyses can detect different types of communication patterns, which enables the use of dataflow analyses on a wide variety of real applications.

I. Introduction

The rise of cluster and multi-core computing has driven a revolution in computer hardware and software. As these machines are best utilized by parallel applications that are explicitly written to take advantage of multiple processing cores, their popularity is pushing the development of a wide variety of parallel applications. In particular, message passing applications, where processes use distributed memory and communicate via explicit `send` and `receive` operations, have become very popular. These applications are common on a wide variety of platforms and use popular APIs such as `BSD sockets`, `MPI` and `PVM`.

The past few decades have seen significant amounts of work on message passing optimization and applications. The bulk of it has focused on runtime and techniques that either improve the underlying hardware (e.g. network processors [12]), the software infrastructure (e.g. self-tuning `MPI` [6]) or use run-time information to analyze and tune application performance [11]. In contrast, work on static techniques to analyze and optimize such applications has been mostly limited to sequential analyses that either improve the performance of the sequential portions of these applications or improve parallel performance by looking at the sequential application properties [3] [21]. What is currently missing is work on static techniques to analyze the parallel properties of parallel applications, including their communication topologies and communication-induced dependencies.

This paper presents a novel compiler analysis framework that extends traditional dataflow analyses to message passing applications. This framework makes it possible to analyze the interactions between different processes and to detect the shape of the application's communication topology by statically matching pairs of `send` and `receive` operations that may communicate at runtime. The framework is defined for a communication model with an unbounded number of processes that communicate exclusively via `send` and deterministic `receive` operations. It is formulated as a dataflow analysis over the cartesian product of control flow graphs (CFGs) from all processes, which we refer to as a parallel control flow graph (pCFG). During its execution, the analysis symbolically represents the execution of multiple sets of processes, keeping track of any `send` and `receive` operations. These `sends` and `receives` are periodically matched to each other, establishing the application's communication topology.

Although intuitively the pCFG should be unbounded in size because the total number of processes used at runtime is unbounded, the framework groups processes into a finite number of sets, each of which features the same communication behavior. By defining the pCFG and dataflow equations over a finite number of process sets rather than the unbounded number of individual processes, the framework makes it possible to analyze applications with unbounded number of processes, number of messages and running time.

The proposed analysis framework can be instantiated by a variety of "client analyses" to provide useful information about parallel applications. Examples include,

- *Communication Optimizations*. The wide variety of available network topologies makes it difficult for programmers to optimize their applications, forcing them to forgo such optimizations or optimize for their current communication platform. If a compiler analysis can detect the application's communication pattern, it can maximize its performance on any network by tuning the underlying communication library [6] or transforming the application to target the network's properties.
- *Error Detection and Verification*. The analysis can detect potential bugs, including message leaks (messages that were sent but never received) or inconsistent types on the sender and receiver for the same message.
- *Constant propagation*. Distributed-memory applications can waste memory on multi-core hardware by having multiple processes keep private copies of identical data. By instantiating the framework with a traditional constant propagation and dependence analyses, we can reduce application

memory footprint by sharing common read-only data among different processes.
- *Information Flow*. The framework can be used to reason about information flows in concurrent programs, identifying privacy- or security-related data leak vulnerabilities.

For a concrete example of performance improvements, consider the pseudocode in Figure I, taken from the mdcask molecular dynamics code, from the ASCI Purple benchmark suite [10]. In this example process 0 sends a message to every other process and then exchanges a message with every other process. The fact that process 0 exchanges messages with every other process makes it both unscalable and a poor fit for sparse network topologies, such as the torus. However, we can significantly improve performance by condensing it into two broadcast operations and a gather, since native communication libraries provide very efficient implementations of collective operations. The application can be transformed further to employ compiled communication features of libraries such as CC-MPI [9], which expose low-level MPI implementation details, which enables compilers to orchestrate the details of communication at fine granularity.

```
if id == 0 then
   for(proc=1..np-1)
      receive <- proc
   for(proc=1..np-1)
      send -> proc
      receive <- proc
else
   send -> 0
   receive <- 0
   send -> 0
```

Fig. 1. Example communication pattern from mdcask

Prior work on static analyzes for message passing applications is summarized in Section II and the execution model assumed in this paper is presented in Section III. Section IV then outlines the overall structure of the analysis framework with Sections V and VI defining pCFGs and dataflow, respectively. Section VII instantiates the framework with a specific analysis that can detect simple communication patterns such as scatter-gather. This analysis is extended in Section VIII with more precise abstractions that are shown to be able to analyze a much wider variety of communication patterns, including patterns over cartesian communication topologies. Finally, Section X presents ongoing and future work.

II. Related Work

Prior work on static analyses for message passing applications has taken several directions. First, there is work on analyses that detect simple communication patterns. One example is work no detecting barrier operations that may match each other at runtime, for APIs where barriers may be either textually aligned [8] or unaligned [7] [22]. Another example is Shao et al [19], which supports more complex patterns but can only match sends to receives at runtime, when the number of processes is known.

MPI-CFGs[21] are an extension of standard control-flow graphs (CFGs), with additional edges between MPI_Send and MPI_Recv operations. Although this work shares similar goals with the analysis presented here, the MPI-CFGs approach is more sequential in that it first connects all MPI_Sends to all MPI_Recvs and then uses sequential information such as mismatched tags or datatypes to prune edges that cannot represent real matches. The two analyses are orthogonal and can be combined to improve overall precision.

Model checking has also been used to detect application communication patterns. MPI-SPIN [20], an extension of the SPIN model checker to MPI, and the more automated approach of Pervez et al [16] are two examples of this. While very accurate, model checking only supports analyses for a constant number of processes and exhibits poor performance when analyzing more than a few processes.

A number of general analyses for send-receive matching have been developed for functional languages. The analysis by Colby [1] for Concurrent ML uses call and fork path information to detect communication patterns between any number of tasks. Martel and Gengler [14] developed a related analysis for the π-calculus. While both analyses are quite capable, they are limited to a simpler variant of message-passing where communication only occurs along channels that are explicitly identified when a process is forked (i.e. communication is constrained by scoping rules). In contrast, many popular message passing models support communication among all tasks, allowing the use of complex expressions to identify communication partners.

III. Execution Model

The analysis assumes a standard message passing execution model. One or more processes execute in parallel and may exchange information using send or receive operations. Each process is identified by a unique number $\in [0..np-1]$, where np is the total number of processes. Each send and receive takes an argument that uniquely identifies the ID of its communication partner. No receive may match messages from more than one sender process (i.e. no wildcard receives).

For each pair of processes there exists one or more bi-directional communication channels. All messages sent along a channel are delivered in FIFO order. Sends are non-blocking and any number of messages may be in-flight at the same time. Receives block until the arrival of a message from their designated sender. Processes may read arbitrary input data and execute non-deterministic operations. Each process maintains two special variables:

- np: the number of executing processes
- id: each process' unique id $\in [0..np-1]$.

The above model captures the bulk of the MPI 1.1 specification and is sufficient for many popular benchmarks, including SMG2000 [10], HPL [15], UMT2K [10] and all the NAS Parallel Benchmarks [5] except LU. For simplicity in this paper we assume that there exists only one channel per process pair and that sends are blocking. While the first restriction is trivial to generalize, the second results in more complex dataflow formulas. This generalization is outlined in Section X.

An important property of this execution model is that no source of non-determinism depends on the application's communication pattern (i.e. it may only come from input files, checking the clock, etc.). As such, this model is *interleaving-oblivious*: *the execution of the application is completely independent of the interleaving of operations on different processes*. A proof of this is included in the Appendix.

IV. Outline of Analysis

The analysis framework works over parallel control-flow graphs (pCFGs), an extension of traditional control flow graphs

(CFGs) to parallel applications. Each pCFG node is a tuple of process sets and maps each set to the CFG node it is executing at. Process sets at a given node are not specified explicitly but are instead identified via unique IDs. It is left to specific dataflow analyses to associate actual process sets with these IDs. The starting pCFG node contains a single process set that is mapped to the starting CFG node. Each pCFG edge from node s to node t corresponds to

- Transitions by one or more process sets along the CFG,
- The split of some process set into two sets, with the different subsets making independent transitions, and/or
- The merging of two process sets into a single process set because they both reach the same CFG node.

pCFGs are defined more formally in Section V. The main idea of the analysis, defined formally in Section VI, is to extend traditional dataflow over CFGs to this new, more complex graph, defining rules for dataflow propagation, `send`-`receive` matching and process set splits and . Figure 2 shows how the analysis operates.

Figure 2(a) shows a code sample where processes 0 and 1 exchange a value that is initialized by process 0 to 5 and print the result. Figure 2(b) is the code's CFG and Figure 2(c) illustrates how a constant-propagation analysis works on the code's pCFG, showing the relevant portions of the pCFG and dataflow state. The analysis starts with process set $[0..np-1]$ (all processes) at the start of the application (CFG node A). Since this node is a conditional that depends on id, the process set $[0..np-1]$ is split into sets [0], [1] and $[2..np-1]$, each of which proceeds down its respective branch of the conditional. Process set $[2..np-1]$ advances to the end of the application, where it blocks until the end of the analysis. Process set [1] also blocks because it reaches the `receive` at node F, which needs to be resolved by a matching `send`. Meanwhile, set [0] transitions to node B, where it identifies that $x = 5$, before advancing to node C. At this point process sets [0] and [1] are at matching communication operations. Since our execution model guarantees that each communication operation has a unique match, we record this match of CFG node C to node F, propagate 5 (x's value) to variable y on process set [1] and allow both process sets to proceed to nodes D and G, respectively. Since these nodes also contain a matching `send` and `receive` pair, we record the $G \rightarrow D$ match, propagate 5 to variable y on process set [0], and allow both process sets to transition to their respective `print` statements. At this point the analysis has shown that both processes will print the value 5 by making the appropriate `send`-`receive` matches and propagating constant information accordingly. Furthermore, the analysis has identified the application's communication topology, shown in Figure 2(d). Note that neither task can be accomplished by traditional analyses that do not explicitly consider the application's parallel structure.

V. Definition of pCFGs

pCFGs extend traditional CFGs by representing all possible control-flow states and state transitions that may be performed by multiple sets of processes, all of which operate over the same CFG. While theoretically, pCFGs may be infinite in size (the number of possible processes is unbounded), our framework focuses on finite pCFGs that are bounded to represent only a finite number of process sets. This is sufficient for real-world applications, which typically divide processes into several groups, each of which performs a specific role or pattern of communication operations. For example, master-worker applications have two such roles and nearest-neighbor stencil applications have $2n+1$ roles, where n is the dimensionality of their grid.

Formally, given a CFG $\langle N_{CFG}, E_{CFG}\rangle$, where N_{CFG} and E_{CFG} are the sets of nodes and edges, respectively, the corresponding pCFG is $\langle N^p_{pCFG}, E^p_{pCFG}\rangle$, where

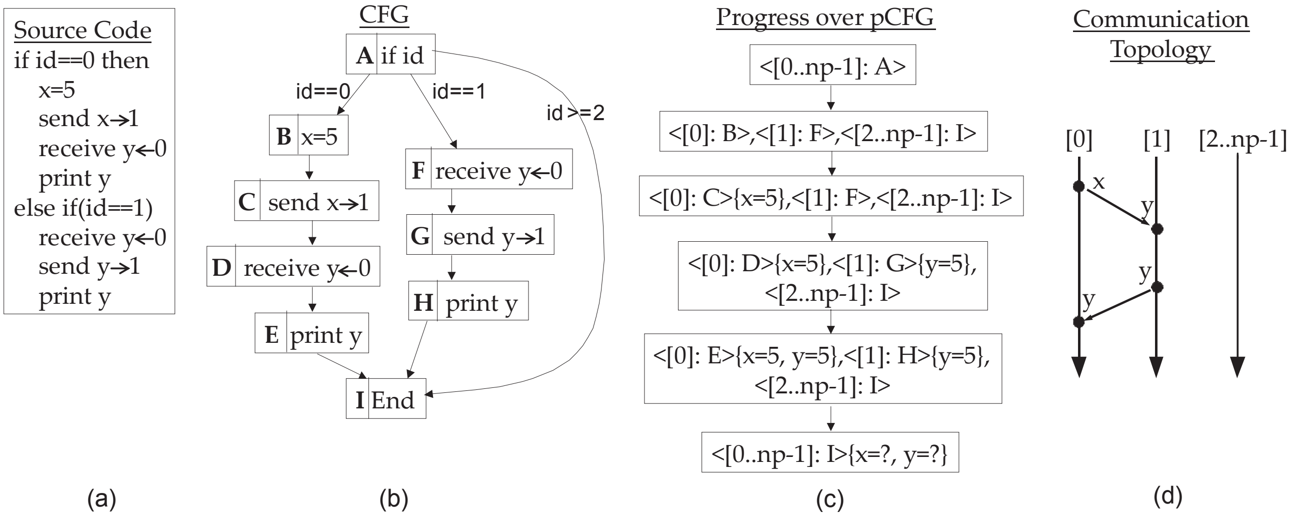
- p is the maximum number of process sets that may be represented at any node,
- Each pCFG node $\in N^p_{pCFG}$ is a tuple of up to p CFG nodes, each annotated with a unique process set id:
  $N_{pCFG} = \{\langle\langle id^1, n^1_{CFG}\rangle, ..., \langle id^q, n^q_{CFG}\rangle\rangle \mid$
  $q \leq p \;\wedge\; \forall j.\ n^j_{CFG} \in N_{CFG}\}$
- Each pCFG edge $\in E^p_{pCFG}$ corresponds to some process sets making a CFG transition, splitting or merging:
  $E_{pCFG} = \{\langle pred_{pCFG}, succ_{pCFG}, trans\rangle \mid$ such that
  The predecessor and successor are both pCFG nodes
  $pred_{pCFG}, succ_{pCFG} \in N_{pCFG} \wedge$
  $trans$ is a function that maps the id of each process set in $pred_{pCFG}$ to its respective id(s) in $succ_{pCFG}$
  $trans \in pred_{pCFG} \mapsto 2^{succ_{pCFG}} \wedge$
  All of $succ_{pCFG}$ is covered by $trans(pred_{pCFG})$
  $(\bigcup_{p \in pred_{pCFG}} trans(p)) = succ_{pCFG} \wedge$
  All process sets $\in pred_{pCFG}$ either made a CFG transition or did not transition
  $(\forall (p = \langle p_{id}, p_{CFG}\rangle) \in pred_{pCFG}, \langle s_{id}, s_{CFG}\rangle \in trans(p).$
  $\langle p_{CFG}, s_{CFG}\rangle \in E_{CFG} \vee p_{CFG} = s_{CFG}) \wedge$
  At least one process set transitioned, split or merged
  $(\exists (p = \langle p_{id}, p_{CFG}\rangle) \in pred_{pCFG}, \langle s_{id}, s_{CFG}\rangle \in trans(p).$
  $\langle p_{CFG}, s_{CFG}\rangle \in E_{CFG} \vee |trans(p)| > 1 \vee$
  $\exists p' \in pred_{pCFG}.\ |trans(p) \cap trans(p')| > 1)$

Because pCFGs are defined exclusively in terms of p and the application CFG, the pCFG is unique for each application and does not depend on the dataflow analysis used with it. Since it contains a node for each possible state of the parallel application and an edge for every possible process set transition, split or merge, it is also very large and dense. However, as discussed in Section VI, the *interleaving-obliviousness* property of the execution model leaves each dataflow analysis free to choose a particular sequence of transitions, splits and merges to analyze. This allows it to examine a small fraction of the graph and ignore the rest, which in practice is not even generated.

In the text below $\langle pred_{pCFG}, succ_{pCFG}, trans\rangle \in E_{pCFG})$ is abbreviated as $(pred_{pCFG} \overrightarrow{trans}\ succ_{pCFG})$. For example, the pCFG in Figure 2(c) contains the edge $\langle\langle id_0 : A\rangle\rangle \overrightarrow{trans} \langle\langle id_0, B\rangle, \langle id_1, F\rangle\rangle$, where $trans$ maps $\langle id_0 : A\rangle$ to both $\langle id_0, B\rangle$ and $\langle id_1, F\rangle$. This corresponds to the fact that a single process set splits into two sets, each of which makes a CFG transition. Similarly, in the edge $\langle\langle id_0, E\rangle, \langle id_1, H\rangle, \langle id_0, I\rangle\rangle \overrightarrow{trans} \langle\langle id_0, I\rangle\rangle$, $trans$ maps all three process sets to the same set $\langle id_0, I\rangle$, merging all three process sets into one when they transition to the same CFG node I.

VI. Dataflow over pCFGs

Dataflow over pCFGs is very similar to dataflow over CFGs. Dataflow state is associated with every pCFG node and these



```
if id==0 then
    x=5
    send x->1
    receive y<-0
    print y
else if(id==1)
    receive y<-0
    send y->1
    print y
```

Fig. 2. Constant Propagation Example

states are propagated along outgoing pCFG edges. The fixed-point solution of the dataflow equation over the pCFG represents a conservative estimate of the state of all process sets at each pCFG node. The interaction of `send` and `receive` operations is a key component of the pCFG dataflow formulas. Each of these operations has an arithmetic expression that identifies its communication partner process. Matching these `sends` and `receives` to each other enables the analysis to propagate dataflow information across process sets, although analyses may also infer constraints between variables on different process sets. The result is that the solution to the dataflow equations is much more precise than would be possible without this `send-receive` matching information [4] [13].

The dataflow state above pCFG node n_{pCFG} is:
$state[n_{pCFG}] = \langle dfState, pSets, matches\rangle$, where

- *dfState* - the dataflow state at n_{pCFG}; contains information about the variables on all process sets in this node,
- *pSets* - maps each element of the tuple n_{pCFG} to a description of the actual corresponding set of processes,
- *matches* - the matching between `send` CFG nodes and `receive` CFG nodes that has been established thus far.

Figure 4 presents the details of the analysis framework over pCFGs. Much like traditional dataflow, this is an analysis framework, rather than a specific analysis. Thus, it specifies the key analysis components required for dataflow. The details of maintaining and propagating dataflow states, the representation of process sets and the details of matching `send` and `receive` expressions are left to more specific "client" analyses that use the framework to derive specific information about the parallel application (this is the difference between dataflow theory and a specific dataflow analysis). References to such components are underlined in the dataflow formulas and Sections VII and VIII provide sample implementations of these operations that support a variety of communication patterns. While this paper focuses on how client analyses deal with pCFG dataflow, in practice these analyses will be combined with additional task-specific analyses such as constant propagation that solve specific dataflow analysis problems. The complexity of the pCFG portion of the client analysis depends on the application's communication pattern and is independent of the analysis problem in question.

At a high level, during each analysis step the framework propagates the dataflow state from one pCFG node to another, It uses the functions supplied by the client analysis to either split a process set at the node, merge multiple process sets or transition a single process set along a CFG edge. The transformed dataflow state is then propagated to descendant pCFG nodes. Each pCFG node may have many different outgoing edges for each of the different events that may happen at this node:

- any process set that is not blocked on a communication operation may make a transition along the CFG,
- sets blocked on `sends` and `receives` may be matched to each other and released to progress along the CFG,
- a process set may be split into two subsets because of a conditional that depends on *id* or because one subset's `send` or `receive` gets matched and the other's does not, and
- multiple process sets may merge into one because they transitioned to the same CFG node and/or because some of them were discovered to be empty and thus, deleted.

At the top of the pCFG the analysis state is initialized to $\langle defaultState, [id^1 \mapsto allProcsSet], \emptyset\rangle$, where the dataflow state is the default specified by the client analysis, there is one process set, which contains all processes and there are no `send-receive` matches. For other nodes n_{pCFG}, the state above n_{pCFG} is computed by looking at all the preceding pCFG nodes p_{pCFG}. For each predecessor we invoke the *propagate* function, defined in Figure 4, to transfer the state above p_{pCFG} through the client analysis' merge, match or transfer functions and along the edge $p_{pCFG} \overrightarrow{trans} n_{pCFG}$.

As indicated in Section III the execution model used in this paper is *interleaving-oblivious*, meaning that the execution of the application is completely independent of the interleaving of operations on different processes. *Since any valid interleaving of operations must produce exactly the same result, it is legal for the analysis to model any one such interleaving, while ignoring all others.* The framework uses this invariant to significantly speed up pCFG dataflow by allowing the client analysis to choose the interleaving to be modeled. Thus, many edges in the pCFG may not be used by a given analysis because these edges correspond to application steps not taken by the chosen interleaving. The formulas below model this by passing along such edges the special dataflow state ⊥, which carries no information. Real dataflow information is passed along edges that do correspond to the chosen interleaving.

The *propagate* function first calls *matchSendsRecvs*, which detects if there exist two process sets in p_{pCFG} that are currently at matching communication operations and makes the appropriate `send-receive` matches via the following procedure. Let *sender* and *receiver* $\in$ p_{pCFG} be two process sets, with the former blocked on a `send` and the latter blocked on a `receive`. *pSets*[*sender*] and *pSets*[*receiver*] are the actual sets of processes associated with *sender* and *receiver*, respectively (represented using the abstraction specified by the client analysis). *sender* and *receiver* match each other if there exist $sProcs \subseteq pSets[sender]$ and $rProcs \subseteq pSets[receiver]$ such that

- the `send` operation's expression surjectively maps *sProcs* onto *rProcs* (all processes in *rProcs* are mapped), and
- the composition of the `receive` and `send` expressions on the domain *sProcs* (the `send` expression is applied to *sProcs* and the `receive` expression is applied to the result) produces the identity function.

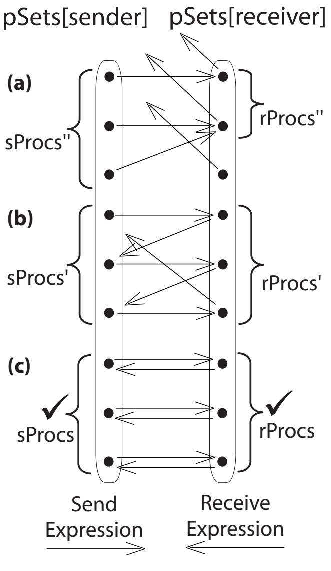


Fig. 3. Send-Receive matching

Figure 3 provides the intuition behind these conditions, showing possible matchings between the set of senders and receivers made by the `send` and `receive` expressions. Figure 3(a) shows a subset of senders being mapped onto a subset of receivers. However, this is not a valid match because (i) two senders are mapped to the same receiver and (ii) all the processes in the receivers set are mapped to some other set of processes. Figure 3(b) shows another subset of senders *sProcs*′ that is mapped onto *rProcs*′, a subset of receivers, via an isomorphism (each sender is mapped to a unique receiver and vice versa). Furthermore, *rProcs*′ is also mapped to *sProcs*′ via an isomorphism. However, because these isomorphisms are not each other's inverses (i.e. their composition is not the identity) the individual processes in these sets do not match each other, making this an invalid match. Finally, Figure 3(c) shows matching sender/receiver process sets. The `send` expression maps *sProcs* onto *rProcs* and the composition of the `receive` and `send` expressions is the identity function on the domain *sProcs*. Since each sender process is uniquely matched to a receiver process, the message sent by each sender must be received by the corresponding receiver.

If *sProcs* and *rProcs* include all processes in *pSets*[*sender*] and *pSets*[*receiver*], respectively, then both *sender* and *receiver* become unblocked and proceed along the CFG. Otherwise, *sender* and/or *receiver* must be split into the appropriate subsets: *sProcs* and *pSets*[*sender*] − *sProcs* for *sender* and *rProcs* and *pSets*[*receiver*] − *rProcs* for *receiver*. The matched subsets (*sProcs* and *rProcs*) become unblocked, while the unmatched subsets (*pSets*[*sender*] − *sProcs* and *pSets*[*receiver*] − *rProcs*) remain blocked on their respective communication operations.

For an example of `send-receive` matching, consider the case where process set $[0...r-1]$ performs `(send x` → `id+r)` and process set $[r...3r-1]$ performs `(receive y` ← `id-r)`. The `send` expression `me+r` maps $[0...r-1]$ to $[r...2r-1]$. The `receive` expression `me-r` maps $[r...2r-1]$ to $[0...r-1]$. Finally, $(\lambda\ id.\ id - r) \circ (\lambda\ id.\ id + r) = (\lambda\ id.\ (id + r) - r) = (\lambda\ id.\ id)$ is the identity function on all domains, including $[0...r-1]$. Thus, the entire set of senders $[0...r-1]$ is matched to subset of receivers $[r...2r-1]$. Therefore, the sender process set is unblocked, while the receiver process set is split into $[r...2r-1]$, which is released and $[2r...3r-1]$, which remains blocked.

Client analyses are responsible for providing a representation of process sets and the expressions used in `send` and `receive` operations. This makes it possible to extend the framework to identify and match very complex message expressions and communication patterns by developing appropriately robust and capable representations. *A key requirement on any such representation is that it must ensure that* `send`*/*`receive` *matching is done exactly, rather than approximately.* This is important because if some process is incorrectly matched or not matched, the dataflow analysis ceases to model a real application execution. However, the remaining application state may be represented via any conservative approximation. In future work this restriction will be removed by defining dataflow over a graph larger than the pCFG that can represent multiple alternative `send-receive` matchings. This is analogous to the approach discussed in Section X for supporting non-deterministic operations.

matchSendsRecvs returns a representation of the process sets that have been released or split as a result of the client analysis' `send-receive` matching. If the client analysis successfully matches process sets at node p_{pCFG}, the analysis framework only propagates real state along the outgoing edge that corresponds to this matching. All other edges are assigned the ⊥ state, which is special in that the union of any real state with ⊥ produces that state. If some process sets were split, the framework uses the function *splitpSets*, defined by the client analysis, to transform the dataflow state to account for the creation of new process sets the state of which is a copy of the old process sets.

It is possible for all process sets at a node to become blocked on either a communication operation or the end of the program. If all process sets are blocked on program end, then nothing further is propagated below the node. However, if some process sets are blocked on communication operations, the framework must make `send-receive` match to make progress. If the state representation or the inference power of the client analysis are not sufficient to make such a match or if such a match is impossible (possibly because of an application bug), the only way to make progress is to make a conservative approximation of the true matches. Since it is not possible to tell whether this situation is caused by a limitation of the client analysis or a property of the application, the analysis framework gives up by passing a ⊤ state down all descendant pCFG edges. ⊤ is special in that the union of any real state with ⊤ produces ⊤.

If no `send-receive` matching is performed and not all process sets are blocked at the node, the framework uses the transfer

$$state[n] = \begin{cases} \{\langle\langle defaultState, [n \mapsto allProcsSet], \emptyset\rangle\} & \text{if } n = entry_{pCFG} \\ \langle dfState, pSets, matches, \rangle\nabla & \\ \quad (\bigsqcup_{p_{pCFG} \xrightarrow{trans} n_{pCFG}} (\underline{rename}(p_{pCFG}, \overrightarrow{trans}, n_{pCFG}, & \text{otherwise} \\ \qquad propagate(state[p_{pCFG}], p_{pCFG}, \overrightarrow{trans}, n_{pCFG}))), & \\ \qquad pSets, matches) & \end{cases}$$

```
Propagates the given dataflow state through node $p_{pCFG}$ along the edge $p_{pCFG} \overrightarrow{trans}\ n_{pCFG}$,
returning the resulting dataflow state.
$propagate(predState, p_{pCFG}, \overrightarrow{trans}, n_{pCFG}) =$
    First, allow the client analysis to match sends to receives at node $p_{pCFG}$
    let $\langle released, split, matches\rangle = matchSendsRecvs(predState.pSets, p_{pCFG}, predState.matches)$ in
      If the released or split process sets were previously blocked on a communication operation and
      the splits/releases correspond to the $p_{pCFG}\ \overrightarrow{trans}\ n_{pCFG}$ transition
      if $(\forall\ p \in released.\ isCommOp(p.n_{CFG}) \wedge \langle p.n_{CFG}, trans(p).n_{CFG}\rangle \in E_{CFG}) \wedge$
         $(\forall\ p \in domain(split).\ isCommOp(p.n_{CFG}) \wedge (|trans(p)| = 2) \wedge$
             $(\exists\ pBlocked \in trans(p).\ p = pBlocked) \wedge$
             $(\exists\ pReleased \in trans(p).\ \langle p.n_{CFG}, pBlocked.n_{CFG}\rangle \in E_{pCFG})) \wedge$
         $(\forall\ p \in (p_{pCFG} - released - domain(split)).\ p = trans(p))$
      then
        If some process sets were released and no process sets were split
        if $released \neq \emptyset \wedge split = \emptyset$ then
          Allow the released process sets to advance without changing the dataflow state
          $predState$
        If some process sets were indeed split
        else if $split \neq \emptyset$ then
          The client analysis adjusts the dataflow state and the actual process sets to take
          the new process set split into account
          $\langle \underline{splitPSets}(split, trans, predState.dfState, predState.pSets), matches\rangle$
        If no blocked process sets were released or split and all process sets are blocked
        else if $(\forall\ p \in p_{pCFG}.\ p.n_{CFG} = End \vee isCommOp(p.n_{pCFG})) \wedge$
                $(\exists\ p \in p_{pCFG}.\ isCommOp(p.n_{CFG}))$ then
          $\top$
        If no blocked process sets were released or split and some process sets were not blocked
        else
          Apply the client analysis' transfer function to node $p_{pCFG}$
          $\langle [\![\underline{statement(p_{CFG})}]\!](predState.dfState, predState.pSets, p_{pCFG}, \overrightarrow{trans}, n_{pCFG}),$
                                          $predState.pSets, predState.matches\rangle$
      else
        $\perp$

True if $n_{CFG}$ contains a communication operation and false otherwise.
$isCommOp(n_{CFG}) = (statement(n_{CFG}) = (\texttt{send x} \rightarrow \texttt{sExpr})) \vee (statement(n_{CFG}) = (\texttt{receive x} \leftarrow \texttt{rExpr}))$

Tries to find a pair of process sets within $p_{pCFG}$ that are blocked on matching communication operations.
If such a pair is found, returns the process sets that are either fully released or are split into a subset that
is released and a subset that is not.
$matchSendsRecvs(pSets, p_{pCFG}, matches) =$
  if $\exists\ sender, receiver \in p_{pCFG}.\ sender.n_{CFG} = (\texttt{send x} \rightarrow \texttt{sExpr}) \wedge$
        $receiver.n_{CFG} = (\texttt{receive x} \leftarrow \texttt{rExpr}) \wedge$
    $\exists\ sProcs \subseteq pSets[sender], rProcs \subseteq pSets[receiver].$
        $(rProcs = sExpr.\underline{image}(sProcs)) \wedge$
        $(sProcs = sExpr.\underline{image}(rProcs)) \wedge$
        $(rExpr \circ sExpr).\underline{isIdentity}(sProcs)$
  then
    $\langle addSplitReleased(sender, sProcs, pSets, addSplitReleased(receiver, rProcs, pSets, \langle\emptyset, \emptyset\rangle)),$
      $matches \cup \{\langle sender.n_{CFG}, receiver.n_{CFG}\rangle\}\rangle$
  else
    $\langle \emptyset, \emptyset, matches\rangle$
```

Figure 4: Dataflow over pCFGs (a)

```
$addSplitReleased(proc, matchedProcs, pSets, \langle released, split \rangle)$ =
   If all the processes in $proc$ were matched, then $proc$ is released
   if $matchedProcs = pSets[proc]$ then $\langle released \cup \{proc\}, split \rangle$then
      If not all the processes in $proc$ were matched, then the matched processes within $proc$ are released
      while the rest remain blocked
   else $matchedProcs \neq pSets[proc]$ then
      $\langle released, split[proc \mapsto matchedProcs, proc \mapsto (pSets[proc] - matchedProcs)] \rangle$
```

Fig. 4. Dataflow over pCFGs (b)

function $[\![statement(p_{CFG})]\!]$, defined by the client analysis, to update the dataflow state based on the statements at node p_{pCFG}. This function is explicitly passed p_{pCFG}, $trans$ and n_{pCFG} so that it can take into account CFG transitions by process sets in p_{pCFG}.

propagate returns the state that results from either performing the `send-receive` matching, calling the transfer function, $\bot$ or $\top$. The function *rename*, defined by the client analysis, is then applied to this state to account for any changes in process set IDs between nodes p_{pCFG} and n_{pCFG} (recorded in *trans*). The new state above n_{pCFG} is its current state, widened with the union of states from all incoming edges. Widening (∇) is a special form of union that ensures that a given node's state can only make a finite number of transitions before reaching its maximal lattice value. Widening ensures convergence of dataflow for client analyses that represent their states using infinite lattices. Client analyses are responsible for defining the widening and union operations.

VII. Simple Symbolic Send-Receive Analysis

Having defined the parallel dataflow analysis framework, this section instantiates it with a specific client analysis that supports applications that use a number of non-trivial communication patterns.

A. State Analysis

The state analysis used in this paper is based on constraint graphs, which are suggested in the CLR [2] (4, Chapter 25.5, pp.539-543) and have previously been used in Shaham et al [18]. Constraint graphs are a representation of application state that is a conjunction of inequalities of the form $i \leq j + c$, where i and j are application variables and c is a constant. The primary difference between prior constraint-graph analyses and the variant used in this paper is that (i) the variables from all process sets are annotated with their process set id and (ii) each process set is assigned its own copy of the variable *id*. This use of different set-specific namespaces makes it possible to infer invariants on the states of variables from different process sets. Due to lack of space we refer readers to [2] and [18] for more details on the state analysis, which include definition of the key operations such as the transfer function, meet and widening.

B. Process Sets

Sets of processes are represented as a lower and upper bound of a range: $[lb..ub]$. The bounds themselves are sets of expressions (e.g. $var + c$) that they are equivalent to. Thus, if we have processes $[1, .., np - 1]$ and the state analysis determines that variable i is equal to 1, this is represented as $[(1, i)..np - 1]$.

Process set operations consider the known relationships between the lower and upper bounds of the input sets and assign the bounds of the output set accordingly. For example when computing $[lb..ub] - [lb'..ub']$, if it is known that $lb \leq lb' \leq ub \leq ub'$, then the upper edge of $[lb..ub]$ overlaps the lower edge of $[lb'..ub']$. Thus, their difference is $[lb..lb' - 1]$.

C. Message Expressions

The analysis supports message expressions of the form $expr(procID) \rightarrow var + c$, where c is some constant and var is optional. Although this is a very simple representation, it enables us to detect non-trivial communication patterns. Message expression operations are implemented as follows:

$$(expr(procID) \rightarrow [lb..ub]).isIdentity(pSet) = ([lb..ub] = pSet)$$

$$(expr(procID) \rightarrow [lb..ub]).image(pSet) = [lb..ub]$$

$$(expr(procID) \rightarrow [lb..ub]) \circ (expr(procID) \rightarrow [lb'..ub']) = [lb..ub]$$

D. Example: Exchange with Root

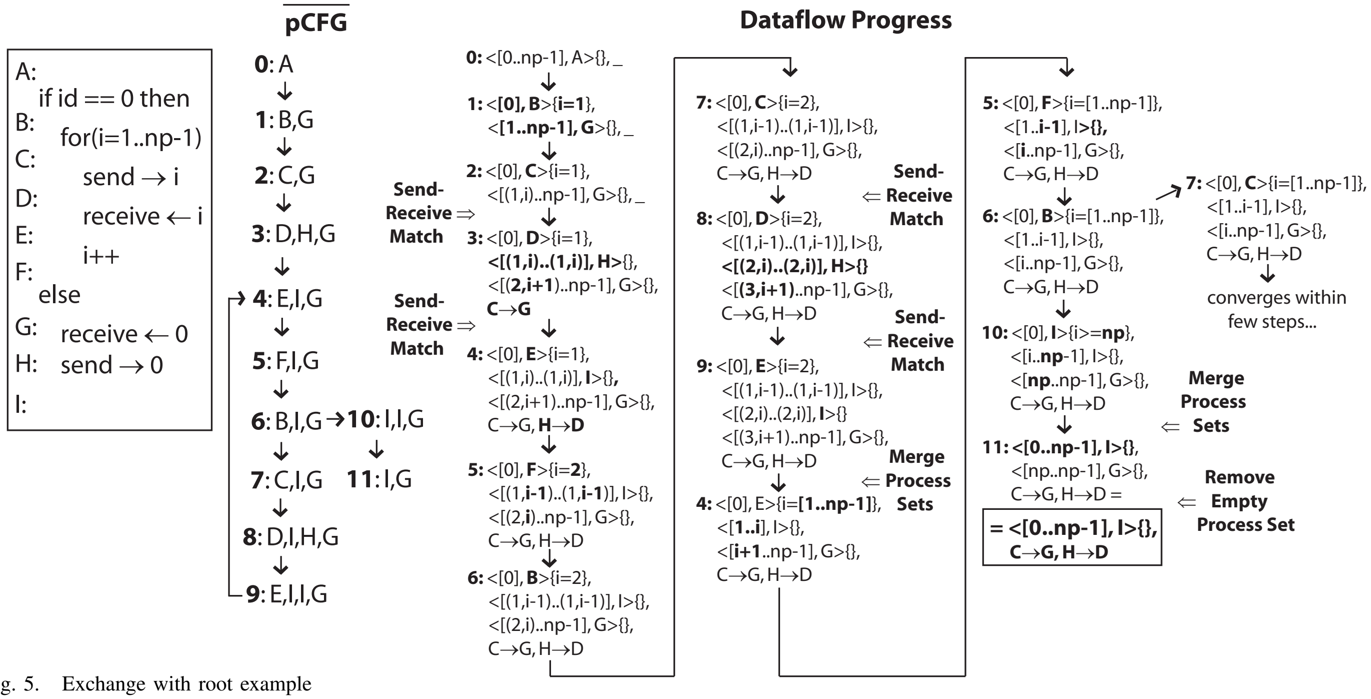
Figure 5 shows an example of how the analysis instantiated in this section can detect the communication topology of the `mdcask` code sample in Figure I. This sample consists of a gather to root pattern, followed by an exchange with root pattern. Due to space limitations, we show how the analysis operates on the second pattern, which is a more complex version of the first. Figure 5 contains the source code of the example (annotated with the CFG nodes), the portion of the pCFG relevant to the example and the actions of the dataflow analysis as it operates over the pCFG. For each step in the dataflow analysis we identify

- the pCFG node the analysis is currently operating on,
- the process sets in operation, including their current CFG nodes and relevant portions of their process states,
- the `send-receive` matches performed thus far.

We explicitly identify any dataflow transitions where a `send-receive` match or process set merging occurs and bold the parts of dataflow state that change from one transition to the next.

In this pattern process 0 executes a loop where it `sends` a message to and `receives` a message from every other process. Other processes first `receive` a message from process 0 and then `send` a message back. At the start of the analysis, all processes ($[0..np - 1]$) are at CFG node A and have no state. The analysis then processes `if id==0`, splitting process set $[0..np-1]$ into set $[0..0]$ (denoted by $[0]$ below), which transitions to the for loop entry point and $[1..np - 1]$, which transitions to the `receive` in the `else` block. Process set $[0]$ then transitions to its `send`.

Both process sets are now blocked on communication operations, with $[0]$ on `(send → i)` and $[1..np - 1]$ on `(receive ← 0)`. $(\lambda\ id.\ i)$ maps $[0]$ to set $[(i, 1)]$ ($i = 1$ during the first iteration) and the composition of the `receive` and `send` expressions $(\lambda\ id.\ 0) \circ (\lambda\ id.\ i) = (\lambda\ id.\ 0)$ is the identity function on domain $[0]$. The `send` on process set $[0]$ is therefore matched to the `receive` on process set $[1]$, causing $[0]$ to be released and $[1..np - 1]$ to be split into $[(1, i)]$, which is released and $[(2, i + 1)..np - 1]$, which remains blocked. The analysis thus transitions to pCFG node 3, where all process sets are again blocked on communication operations. Using the same reasoning as above the `(receive ← i)` on process set $[0]$ is matched to



```
A:
  if id == 0 then
B:  for(i=1..np-1)
C:    send → i
D:    receive ← i
E:    i++
F:
  else
G:  receive ← 0
H:  send → 0
I:
```

pCFG
0: A
1: B,G
2: C,G
3: D,H,G
4: E,I,G
5: F,I,G
6: B,I,G
10: I,I,G
7: C,I,G
11: I,G
8: D,I,H,G
9: E,I,I,G
Dataflow Progress
0: <[0..np-1], A>{}, _
1: <[0], B>{i=1}, <[1..np-1], G>{}, _
2: <[0], C>{i=1}, <[(1,i)..np-1], G>{}, _
Send-Receive Match ⇒
3: <[0], D>{i=1}, <[(1,i)..(1,i)], H>{}, <[(2,i+1)..np-1], G>{}, C→G
Send-Receive Match ⇒
4: <[0], E>{i=1}, <[(1,i)..(1,i)], I>{}, <[(2,i+1)..np-1], G>{}, C→G, H→D
5: <[0], F>{i=2}, <[(1,i-1)..(1,i-1)], I>{}, <[(2,i)..np-1], G>{}, C→G, H→D
6: <[0], B>{i=2}, <[(1,i-1)..(1,i-1)], I>{}, <[(2,i)..np-1], G>{}, C→G, H→D
7: <[0], C>{i=2}, <[(1,i-1)..(1,i-1)], I>{}, <[(2,i)..np-1], G>{}, C→G, H→D
⇐ Send-Receive Match
8: <[0], D>{i=2}, <[(1,i-1)..(1,i-1)], I>{}, <[(2,i)..(2,i)], H>{} <[(3,i+1)..np-1], G>{}, C→G, H→D
⇐ Send-Receive Match
9: <[0], E>{i=2}, <[(1,i-1)..(1,i-1)], I>{}, <[(2,i)..(2,i)], I>{} <[(3,i+1)..np-1], G>{}, C→G, H→D
⇐ Merge Process Sets
4: <[0], E>{i=[1..np-1]}, <[1..i], I>{}, <[i+1..np-1], G>{}, C→G, H→D
5: <[0], F>{i=[1..np-1]}, <[1..i-1], I>{}, <[i..np-1], G>{}, C→G, H→D
6: <[0], B>{i=[1..np-1]}, <[1..i-1], I>{}, <[i..np-1], G>{}, C→G, H→D
7: <[0], C>{i=[1..np-1]}, <[1..i-1], I>{}, <[i..np-1], G>{}, C→G, H→D
converges within few steps...
10: <[0], I>{i>=np}, <[i..np-1], I>{}, <[np..np-1], G>{}, C→G, H→D
⇐ Merge Process Sets
11: <[0..np-1], I>{}, <[np..np-1], G>{}, C→G, H→D =
⇐ Remove Empty Process Set
= <[0..np-1], I>{}, C→G, H→D

Fig. 5. Exchange with root example

the (send → 0) on set $[(1,i)]$, which allows the process sets to make progress and advances the analysis to pCFG node 4.

Process set [0] now iterates through its loop, incrementing i, while $[(1,i)]$ continues to the CFG end node, becoming $[(1,i-1)]$ as a result of the increment. At pCFG node 7 process set [0] is again at the (send → i), $[(1,i-1)]$ is at the end node and $[(2,i)..np-1]$ is blocked on (receive ← 0). The send on [0] and the receive on $[(2,i)]$ are matched as above, causing [0] to be released and splitting $[(2,i)..np-1]$ into $[(2,i)]$, which is also released and $[(3,i+1)..np-1]$, which remains blocked. As before, [0] and $[(2,i)]$ advance to their next communication operations. These are again matched and the analysis advances to pCFG node 9 where process set $[(2,i)]$ reaches the CFG end node, merging with set $[(1,i-1)]$ at the next pCFG node.

When the analysis returns to pCFG node 4 it has two representations of the process sets at this pCFG node. During the first time the analysis passed through this node the sets were [0], $[(1,i)..(1,i)]$ and $[(2,i+1)..np-1]$. However, this time around the sets are [0], $[(1,i-1)..(2,i)]$ and $[(3,i+1)..np-1]$. When the former is widened with the latter, the common portions are retained: [0], $[1..i]$ and $[i+1..np-1]$, which is the loop invariant. In other words, in every iteration process set $[1..np-1]$ is split into the set $[1..i]$ of processes that have performed the exchange and set $[i+1..np-1]$ of processes that have not. Because this is the fixed point, as the analysis passes through the loop again (pCFG nodes 6 → 7 → 8 → 9 → 4), it finds no new information.

When this state is propagated along the loop's exit edge, two things happen. First, process sets [0] and $[1..i-1]$ are now at CFG node I, enabling them to be merged into set $[0..i-1]$. Second, because $i = np$ is true along the loop's exit edge, these process sets evolve to $[0..np-1]$ and $[np..np-1]$, respectively. Since the latter set is $= \emptyset$, it is eliminated, with the final dataflow state being that all processes are at the end node and the set of send-receive matches is $C \to G$ and $H \to D$.

VIII. Cartesian Topology Analysis

The above analysis can detect communication patterns that use simple send/receive expressions like (send → var) or (receive ← 5). However, because it matches sends and receives by directly matching their expression variables and constants, it cannot match more complex expressions where the matching depends on the expressions' algebraic properties. For example, consider the code in Figure 6 from the CG benchmark from the NAS Parallel Benchmark suite [5]. This application runs on a 2-dimensional cartesian grid of processes where the number of rows and columns (nprows and npcols, respectively) is either equal or the number columns is twice the number of rows. In the sample code each process exchanges a value with the process in the transpose location in the grid.

```
assert(np = ncols * nrows)
if ncols == nrows then
   call send x → (id%nrows)*nrows + id/nrows
   call receive y ← (id%nrows)*nrows + id/nrows
else if ncols == nrows*2 then
   call send x → 2*((id/2%nrows)*nrows +
                     id/2/nrows) + id%2
   call receive y ← 2*((id/2%nrows)*nrows +
                     id/2/nrows) + id%2
end if
```

Fig. 6. Transpose example NAS-CG

Because the sends and receives use complex expressions composed of addition, multiplication, integral division and mod, the above analysis is not able to match them to each other. What is needed is an explicit representation of such expressions, as well as inference rules to derive relevant facts about mappings and identities. This section presents Hierarchical Sequence Maps (HSMs), an abstraction that extends the power of the analysis framework to communication over cartesian topologies. Section VIII-A presents HSMs and defines their basic operations. Section VIII-B then shows how they can be used to match the sends and receives in the above code segment. Finally, Section VIII-C explains how HSMs can also be used to detect n-dimensional nearest-neighbor communication patterns.

A. Hierarchical Sequence Maps

An HSM represents a hierarchy of sequences of integers with a known starting point and a constant stride. A simple example of an HSM is $[11 : 4, 5]$, which represents the sequence of length 4, that starts at 11 and contains integers that are a stride of 5 away from 11: $\langle 11, 16, 21, 26\rangle$. More generally, sequences may contain other sequences, such as $[[0 : 10, 1] : 3, 100]$, which represents the sequence $\langle 0, ..., 10, 100, ..., 110, 200, ..., 210\rangle$. In short, $[e : r, s]$ is the sequence where

- e - an HSM, an integral variable or a constant
- r - the number of times e is repeated ($r > 0$)
- s - the stride at which e is repeated ($s \geq 0$)

An HSM represents a function from one sequence of integers to another where the ith element in the domain sequence is mapped to the ith element in the HSM sequence. Formally, the HSM $[[e : r_1, s_1]... : r_k, s_k]$ defines the function:

$$index \mapsto e + \sum_{i=1}^{k} x_i \cdot s_i, \text{ where } x_i = (index/\prod_{j=1}^{i} r_j))\%r_{i+1}.$$

For a given expression executed by process set ps the corresponding HSM represents the mapping from IDs of processes $\in ps$ to the value of the expression on each process. First, the variable id is converted into an HSM where each process $\in ps$ is mapped to its respective id. Thus, if $ps = [l..u]$ then id is represented by HSM $[l : u - l + 1, 1]$, where the first process is mapped to l, the second to $l{+}1$, etc. and the last to u. Constants c and variables var that are the same on all processes are replaced with $[c : n, 0]$ and $[var : n, 0]$ (n is the size of the process set), which map each process to the same value. This procedure turns any expression into one that contains HSMs instead of constants and variables. Table I shows how the operations +, *, / and % are defined for HSMs. These are used to mechanically convert an HSM expression into a single HSM.

HSM addition is defined for equal-length sequences and corresponds to adding the values in both sequences together while preserving the length. In scalar multiplication all values in an HSM are multiplied by the same scalar value. HSM modulus is defined only for the case $[e : r \cdot r', s]\%(s \cdot r)$ where $s \cdot r$ divides e (expressed as $(s \cdot r)|e$). Since $(s \cdot r)|e$, $e\%(s \cdot r)$ is a sequence of 0's of the same length as e. The HSM $[e\%(s \cdot r) : r \cdot r', s]$ shifts this sequence of 0's over the range $[0..r \cdot s \cdot (r' - 1)]$, using stride s. Thus, its modulus by $r \cdot s$ is $[e\%(s \cdot r) : r, s]$ (ranges over $[0..r \cdot (s - 1)]$), repeated r' times. For example, $\langle 12, 14, 16, .., 38, 40\rangle\%6 = [12 : 15, 2]\%6 = [12 : 3{\cdot}5, 2]\%(2{\cdot}3) = [[[0 : 1, 0] : 3, 2] : 5, 0] = [[0 : 3, 2] : 5, 0] =$
$\langle 0, 2, 4, 0, 2, 4, 0, 2, 4, 0, 2, 4, 0, 2, 4\rangle$.

Finally, we define two types of HSM division. The first corresponds to the case $[e : r, s \cdot q]/q$, where $q|e$. As such, e is shifted using a stride that is divisible q and since $q|e$, every element in the sequence is divisible by q. Thus, $[e : r, s \cdot q]/q$ is equivalent to multiplication by $1/q$. Division is also possible if individual sequence elements are not divisible by q but the sequence is composed of sub-sequences in the range $[qk..q(k + 1))$, for some k. Dividing such subsequences by q causes all the elements to become k. For example,
$\langle 20, 25, 30, 35, 40, 45\rangle/10 = [20 : 6, 5]/10 = [20 : 2{\cdot}3, 5]/(2{\cdot}5) = [[20/(2 \cdot 5) : 3, 1] : 2, 0] = [[2 : 2, 0] : 3, 1] = \langle 2, 2, 3, 3, 4, 4\rangle$.

To put HSM operations in a larger context, the first communication expression in the code sample above is converted as follows (given the invariant $np = nrows \cdot nrows$):

$$\begin{aligned}
&(id\%nrows) \cdot nrows + id/nrows = \\
&= ([0 : np, 1]\%nrows) \cdot nrows + [0 : np, 1]/nrows \\
&= ([0 : nrows \cdot nrows, 1]\%nrows) \cdot nrows + \\
&\qquad [0 : nrows \cdot nrows, 1]/nrows \\
&= [[0 : nrows, 1] : nrows, 0] \cdot nrows + \\
&\qquad [[0 : nrows, 0] : nrows, 1] \\
&= [[0 : nrows, nrows] : nrows, 0] + [[0 : nrows, 0] : nrows, 1] \\
&= [[0 : nrows, nrows] : nrows, 1]
\end{aligned}$$

It can be easily verified that each process is mapped to its corresponding transpose process, with the k^{th} row $[k \cdot nrows, k \cdot nrows + 1, .., (k + 1) \cdot nrows - 1]$ mapped to the k^{th} column $[k + nrows \cdot 0, k + nrows \cdot 1, .., k + nrows \cdot (nrows - 1)]$.

Table I also defines two types of HSM equality rules: sequence-equalities, which assert that two HSMs represent the same sequence, and set-equalities, which asserts that the HSMs represent the same set of points in a potentially different order (sequence-equality implies set-equality). Proving that two HSMs are equal in either sense requires the repeated application of the appropriate rule(s) on one HSM to turn it into the other. It is mechanized by using heuristically guided search, a standard technique in automated theorem provers.

$[[e : r, s] : r', r \cdot s] = [e : r \cdot r', s]$ is a sequence-equality relation that asserts that a sequence of length $r * r'$ can be represented as a plain sequence or a hierarchical sequence of r' sub-sequences that each contain r elements. An example of this is $[[2 : 3, 2] : 2, 3 \cdot 2] = \langle 2, 4, 6, 8, 10, 12\rangle = [2 : 3 \cdot 2, 2]$. $[[e : r, r' \cdot s], r', s] \approx [e : r \cdot r', s]$ is a set-equality that says that an HSM that shifts $[e : r, r' \cdot s]$ by a stride s such that adjacent instances of $[e : r, r' \cdot s]$ interleave into each other can be reordered to have all the points line up in order using a stride s. For instance, $[[2 : 3, 2 \cdot 2] : 2, 2] = \langle 2, 6, 10, 4, 8, 12\rangle \approx \langle 2, 4, 6, 8, 10, 12\rangle = [2 : 2 \cdot 3, 2]$. Finally, the $[[e : r, s] : r', s'] \approx [[e : r', s'] : r, s]$ set-equality relation asserts that reordering the stride and repetition parameters in two adjacent levels of an HSM reorders the elements but doesn't change the actual set of values mapped by the HSM. This is illustrated by the following: $[[1 : 2, 1] : 3, 10] = \langle 1, 2, 11, 12, 21, 22\rangle \approx \langle 1, 11, 21, 2, 12, 22\rangle = [[1 : 3, 10] : 2, 1]$

B. `Send-Receive` Matching

The HSM $[[0 : nrows, nrows] : nrows, 1]$ represents the `send` and `receive` expressions in the `nrows==ncols` case. To show that these operations match when executed by process set $[0..np - 1]$, we must show that the (i) composition of the `send` and `receive` expressions is the identity map and (ii) there exists a subset of senders ($sProcs$) and a subset of receivers ($rProcs$) such that the `send` expression surjectively maps $sProcs$ onto $rProcs$. In this case $sProcs = rProcs = [0..np - 1]$.

1) Identity: An expression is the identity function on process set ps if the sequence defined by the corresponding HSM is itself equal to ps. Furthermore, the composition of two expressions $e_r \circ e_s$ on ps is the function that creates the HSM h_s from e_s by applying e_s to ps and then applies e_r to h_s (i.e. the operations from Table I are applied as normal on the `receive` expression, with the variable id replaced with h_s). This gives us a simple

Operations			
id	$id = [lb : (ub - lb) - 1, 1]$, ($lb$ and ub: lower and upper bounds of given process set)		
Constants	$c = [c : (ub - lb), 0]$		
Common Variables	$var = [var : (ub - lb), 0]$		
Addition	$[e : r, s] + [e' : r, s'] = [e + e' : r, s + s']$		
Multiplication	$[e : r, s] \cdot q = [e \cdot q : r, s \cdot q]$		
Modulus	$(s \cdot r) \mid e \;\Rightarrow\; [e : r \cdot r', s]\%(s \cdot r) = [[[0 : count(e), 0] : r, s] : r', 0]$		
Division	$q \mid e \;\Rightarrow\; [e : r, s \cdot q]/q = [e/q : r, s]$	$(r \cdot s) \mid e \;\Rightarrow\; [e : r \cdot r', s]/(r \cdot s) = [[e/(r \cdot s) : r, 0] : r', 1]$	
Definitions		Equalities	(= is sequence equality, ≈ is set equality)
Count	$count(c) = 1$	Adjacency	$[[e : r, s] : r', r \cdot s] = [a : r \cdot r', s]$
	$count([e : r, s]) = r \cdot count(e)$	Interleaving	$[[e : r, r' \cdot s], r', s] \approx [e : r \cdot r', s]$
Divisibility	$q \mid [e : r, s] = q \mid e \;\wedge\; q : s$	Reordering	$[[e : r, s] : r', s'] \approx [[e : r', s'] : r, s]$
Lower Bound	$lb(c) = c$ $lb([e : r, s]) = lb(e)$		
Upper Bound	$ub(c) = c$ $ub([e : r, s]) = ub(e) + (r - 1) \cdot s$		

TABLE I
HSM OPERATIONS AND DEFINITIONS

procedure for verifying whether given pair of `send-receive` expressions is the identity function on *sProcs*.

For example, consider the first branch of the NAS-CG example. The `send` expression $((id\%nrows) \cdot nrows + id/nrows)$ on process set $[0 : np, 1]$ corresponds to the HSM $[[0 : nrows, nrows] : nrows, 1]$, as shown above. When the `receive` expression $((id\%nrows) \cdot nrows + id/nrows)$ is then applied to this HSM, we infer the following:

$$\begin{aligned}
&([[0 : nrows, nrows] : nrows, 1]\%nrows) \cdot nrows+\\
&\qquad [[0 : nrows, nrows] : nrows, 1]/nrows =\\
&= [[[0 : nrows, 0] : nrows, 1] : 1, 0] \cdot nrows+\\
&\qquad [[[0 : nrows, nrows]/nrows : nrows, 0], 1, 1]\\
&= [[0 : nrows, 0] : nrows, 1] \cdot nrows+\\
&\qquad [[0 : nrows, 1] : nrows, 0]\\
&= [[0 : nrows, 0] \cdot nrows : nrows, nrows]+\\
&\qquad [[0 : nrows, 1] : nrows, 0]\\
&= [[0 : nrows, 0] : nrows, nrows] + [[0 : nrows, 1] : nrows, 0]\\
&= [[0 : nrows, 1] : nrows, nrows] \quad = \quad [0 : nrows \cdot nrows, 1]\\
&= [0 : np, 1]
\end{aligned}$$

This proves that the composition of the `send` function and the `receive` function is the identity function. The inference follows via sequence of HSM operations, followed by the application of the HSM adjacency equality rule.

2) Surjection: An expression is a surjective map from its domain onto a given set if the expression's range is equal to this set. This is proven for HSMs via the equality relations in Table I. To show that a given expression is a surjective map from process set ps_1 to process set ps_2, we need to show that applying the expression to ps_1 produces an HSM that is set-equal to ps_2.

The following inference uses these relations to prove that the `send` expression $(id\%nrows) \cdot nrows + id/nrows$ is a surjective map onto process set $[0..np-1]$, when applied to set $[0..np-1]$: $[[0 : nrows, nrows] : nrows, 1] = [0 : nrows \cdot nrows, 1] = [0 : np, 1] = [0..np - 1]$.

Looking at the more complex expression $2 \cdot ((id/2\%nrows) \cdot nrows + id/2/nrows) + id\%2$) used by the rectangular communication pattern ($nrows = 2 \cdot ncols$), we see that applying the expression to process set $[0..np - 1]$ produces the HSM $[[[0 : 2, 1] : nrows, nrows \cdot 2] : nrows, 2]$. The following proves that this HSM is a surjective map onto the set $[0..np - 1]$: $[[[0 : 2, 1] : nrows, nrows \cdot 2] : nrows, 2]) \approx [[[0 : 2, 1] : nrows \cdot nrows, 2]]) = [[[0 : 2 \cdot nrows \cdot nrows, 1]]) =$
$[0 : np, 1] = [0..np - 1]$

C. Nearest-neighbor Communication

Hierarchical sequence maps can be used to analyze applications that use the structured nearest-neighbor communication pattern commonly found in partial-differential equation codes. In this pattern all processes are embedded into a virtual n-dimensional cartesian mesh and the state of each process depends on the state of its nearest neighbors in each dimension. In the common case, where the edges of the mesh do not wrap around to the other side, this leads to a complex communication pattern that uses $2n + 1$ different types of processes. Each process type has its own configuration of neighbors, with central processes having $2n$ neighbors in all directions and corner processes having n neighbors, one for each dimension. This section shows how HSMs can be used to match `sends` and `receives` in the $n = 1$ case. While they also work with larger values of n, examples could not be included due to space constraints. Figure 7 shows a code sample for the heart of the $n = 1$ nearest-neighbor communication pattern: a shift of data along a single mesh dimension. Processes $[1..np - 2]$ receive a message from the left and send one to the right, while the edge processes either only send or only receive. This code thus features three process sets: $[0]$, $[1..np - 2]$, $[np - 1]$, which need to be matched as shown in Figure 8: $[0] \rightarrow [1]$, $[1..np - 3] \rightarrow [2..np - 2]$, and $[np - 2] \rightarrow [2]$.

```
if id==0 then
    call send x → id+1
else if id<np-1 then
    call receive x ← id-1
    call send x → id+1
else
    call receive x ← id-1
end if
```

Fig. 7. Data shift in 1D nearest-neighbor

We prove that expression $(id - 1) \circ (id + 1)$ is the identity on the three domains via the following inferences:

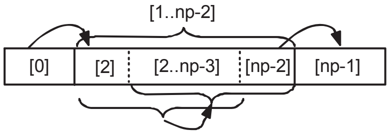


Fig. 8. Matching pattern for $n = 1$ nearest neighbor pattern

- $\underline{\text{domain} = [0] = [0 : 1, 0]}$: $([0 : 1, 0] + 1) - 1 = [0 : 1, 0] + [1 : 1, 0] + [-1 : 1, 0] = [1 : 1, 0] + [-1 : 1, 0] = [0 : 1, 0]$.
- $\underline{\text{domain} = [1..np - 2] = [1 : np - 3, 1]}$: $([1 : np-3, 1]+1)-1 = [1 : np-3, 1]+[1 : np-3, 0]+[-1 : np-3, 0] = [1 : np-3, 1]$.
- $\underline{\text{domain} = [np - 1] = [np - 1 : 1, 0]}$: $([np-1 : 1, 0] + 1) - 1 = [np - 1 : 1, 0] + [1 : 1, 0] + [-1 : 1, 0] = [np - 1 : 1, 0]$.

We prove that the `send` expression ($id + 1$) is a surjective map for all three cases as follows:

- $\underline{[0] \mapsto [1]}$: $[0 : 1, 0]+1 = [0 : 1, 0]+[1 : 1, 0] = [1 : 1, 0] = [1]$
- $\underline{[1..np - 3] \mapsto [2..np - 2]}$: $[1 : np-3, 1]+1 = [1 : np-3, 1]+ [1 : np - 3, 0] = [2 : np - 3, 0] = [2..np - 2]$
- $\underline{[np - 2] \mapsto [np - 1]}$: $[np-2 : 1, 0] + 1 = [np-2 : 1, 0] + [1 : 1, 0] = [np - 1 : 1, 0] = [np - 1]$

IX. Implementation

The analysis presented in Section VII has been implemented using the ROSE [17] compiler infrastructure. While the current implementation is still in the preliminary stages, it can already process MPI code examples such the one in Figure I and is being extended to support more complex expression representations such as HSMs. The current implementation takes 381 seconds to analyze a fan-out broadcast code sample on a 2.8Ghz Opteron processor with 2GB RAM. While this running time is still high, the primary cost of the analysis is not high asymptotic complexity but rather (i) the use of client analysis that is much richer and more complex than what is required for the job and (ii) very limited work on optimization. In particular, detailed performance profiles reveal that 92.5% of this time (351 sec) was spent keeping the dataflow state at each pCFG node consistent.

The constraint graph representation used in this analysis represents state as a graph of variables, with edges between variable pairs corresponding to known linear inequalities. When a new inequality is added to the graph, its implications must be propagated to other, potentially related inequalities via a transitive closure operation, which has $O(n^3)$ complexity in the number of variables. In this example transitive closure closure operation is executed 217 times, operating on an average of 52.3 variables, with a cheaper $O(n^2)$ variant called 78 times, operating on an average of 66.3 variables. The large variable count comes from the various auxiliary variables used by the analysis to represent process sets. In addition to computational cost, transitive closures incur high cache miss rates caused by large working sets. These miss rates were made worse by the fact that the implementation stores state using C++ STL containers rather than more efficient arrays.

Our development efforts are focused towards

1) Using a simpler dataflow state representation than constraint graphs
2) Reducing the number of individual variables needed to represent process sets,
3) Implementing dataflow state using efficient abstractions such as arrays instead of C++ STL container,
4) Using cache-conscious state traversal strategies, and
5) Using multi-threading to take advantage of multi-core processors.

The last point is important since pCFG-based analyses are naturally parallelizable since work on different portions of the pCFG may proceed independently.

X. Ongoing and Future Work

The work presented in the paper provides the potential to develop a wide variety of dataflow analyses for parallel message passing applications. However, as presented here, it has a number of limitations that we will address in future work. First, we are actively working to expand the range of communication patterns supported by the framework, including tree-shaped patterns and semi-structured meshes that cannot be handled by HSMs.

Second, the analysis presented in this paper is a simplified version that blocks on all `send` and `receive` operations, chosen for improved readability. While this is sufficient for many patterns, it fails to model operations such as `MPI_Isend` and `MPI_Irecv`, which do not block. The complication introduced by non-blocking operations is that the analysis may be required to process multiple such operations on a given process set before being able to match any of them. For example, in the all-to-all exchange pattern each process may execute a loop where it sends a message to every other process using a non-blocking `send`, followed by a loop where it uses a loop of blocking `receives` to get messages from all other processes. This code forces the dataflow framework to process the entire loop of `sends`, aggregating individual `send` expressions into a single abstraction that represents a potentially unbounded number of individual communication operations. The expressions in this abstraction must then be matched against individual `receive` expressions in the `receive` loop. The framework has already been extended with dataflow formulas that correctly capture the details of matching aggregated `send`/`receive` expressions. These extensions will be detailed in future publications.

Third, the current framework is intra-procedural and does not provide non-trivial support for recursion. The framework is currently being extended to support context-sensitive inter-procedural dataflow analyses. In short, we will compute a function summary for every pCFG node where some process set performs a function call to ensure that the communication operations performed inside the call will be correctly matched to communication operations on other process sets. The context-sensitivity will enable the entire pCFG analysis to be modular.

Finally, the execution model used by this analysis assumes that `receives` match `sends` in a deterministic FIFO fashion. This prevents the framework from supporting constructs such as `MPI_ANY_SOURCE`, which allows a single dynamic `receive` operation to match one of several `send` operations, or locks, since the `lock` operation may "receive" the lock from any unlock operation on any process. Because removing this constraint causes executions to no longer be interleaving-oblivious, this constraint is being overcome by defining dataflow over a richer variant of pCFGs where each node contains multiple non-deterministic options for the possible matchings. This graph has a property similar to interleaving-obliviousness, which will be exploited to bound analysis complexity. A similar approach is being taken to support inexact send-receive matching. In practice we expect that through the use of appropriate abstractions we will only need to consider a finite number of possible non-deterministic options at a time. A typical analysis scenario is applications where a single lock operation may acquire a lock from multiple possible unlock operations. In this case the analysis would explicitly represent each of these (finitely many) possibilities and analyze them independently. However, for various examples the use of appropriate abstractions results in no additional work for the analysis.

XI. Summary

This paper presents a novel compiler analysis framework that extends the traditional dataflow framework to deterministic message passing communication patterns. This framework defines parallel control flow graphs, an extension of control flow graphs that is the finite cross-product of the CFGs of all concurrent processes. We have defined dataflow analyses over these graphs, identifying both the core aspects of parallel dataflow and the extensions that may be instantiated by specific analyses. Specifically, we have provided two instantiations of the the framework that can analyze applications that use a variety of real-world communication patterns and provided examples of how these instantiations work with real patterns, including exchange with root, transpose and cartesian nearest-neighbor. To our knowledge, this is the first compiler analysis that analyzes the parallel structure and communication topology of parallel message passing applications, significantly extending the state of the art in this field.

References

[1] Christopher Colby. Analyzing the Communication Topology of Concurrent Programs. In *Symposium on Partial Evaluation and Semantics-Based Program Manupulation*, pages 202–214, June 1995.

[2] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press and McGraw-Hill, 1994.

[3] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. Implementing an Open64-based tool for improving the performance of MPI programs. In *Open64 Workshop*, 2008.

[4] Anthony Danalis, Ki-Yong Kim, Lori Pollock, and Martin Swany. Transformations to parallel codes for communication-computation overlap. In *IEEE/ACM Supercomputing Conference*, 2005.

[5] NASA Advanced Supercomputing (NAS) Division. NAS Parallel Benchmarks. http://www.nas.nasa.gov/Resources/Software/npb.html.

[6] A. Faraj, X. Yuan, and D.K. Lowenthal. STAR-MPI: Self tuned adaptive routines for mpi collective operations. In *International Conference on Supercomputing*, June 2006.

[7] T.E. Jeremiassen and S.J. Eggers. Static analysis of barrier synchronization in explicitly parallel programs. In *International Conference on Parallel Archtecture and Compilation Techniques*, 1994.

[8] Amir Kamil and Katherine Yelick. Concurrency analysis for parallel programs with textually aligned barriers. In *Workshop on Languages and Compilers for Parallel Computing*, October 2005.

[9] Amit Karwande, Xin Yuan, and David K. Lowenthal. CC-MPI: A Compiled Communication Capable MPI Prototype for Ethernet Switched Clusters. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 95–106, June 2003.

[10] Lawrence Livermore National Lab. ASCI Purple Benchmarks. http://www.llnl.gov/asci/platforms/purple/rfp/benchmarks.

[11] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[12] Panos C. Lekkas. *Network Processors : Architectures, Protocols and Platforms*. McGraw-Hill Professional, 2003.

[13] Glenn Luecke, Hua Chen, James Coyle, Jim Hoekstra, and Marina Kraeva Yan Zou. MPI-CHECK: a tool for checking fortran 90 mpi programs. *Concurrency and Computation: Practice and Experience*, 15(2):93–100, 2003.

[14] Matthieu Martel and Marc Gengler. Communication Topology Analysis for Concurrent Programs. In *International Workshop on SPIN Model Checking and Software Verification*, 2000.

[15] University of Tennessee. High Performance Linpack. http://www.netlib.org/benchmark/hpl/.

[16] Salman Pervez, Ganesh Gopalakrishnan, Robert M. Kirby, Robert Palmer, Rajeev Thakur, and William Gropp. Practical model-checking method for verifying correctness of mpi programs. In Franck Cappello, Thomas Hrault, and Jack Dongarra, editors, *PVM/MPI*, volume 4757 of *Lecture Notes in Computer Science*, pages 344–353. Springer, 2007.

[17] Dan Quinlan. Rose: Compiler Support for Object-oriented Frameworks. *Parallel Processing Letters*, 10(2-3):215–226, 2000.

[18] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. Automatic Removal of Array Memory Leaks in Java. In *International Conference on Compiler Construction(CC)*, March 2000.

[19] S. Shao, A. Jones, and R. Melhem. A compiler-based communication analysis approach for multiprocessor systems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, April 2006.

[20] Stephen F. Siegel. Model Checking Nonblocking MPI Programs. In *International Conference on Verification, Model Checking and Abstract Interpretation*, 2007.

[21] Michelle Mills Strout, Barbara Kreaseck, and Paul D. Hovland. Dataflow analysis for MPI programs. In *International Conference on Parallel Processing (ICPP)*, 2006.

[22] Yuan Zhang, Evelyn Duesterwald, and Guang Gao. Concurrency analysis for shared memory programs with textually unaligned barriers. In *International Workshop on Languages and Compilers for Parallel Computing*, October 2007.

Appendix

Appendix:
A key property of the execution model defined in Section III is that no source of non-determinism depends on the application's communication pattern (e.g. may only come from input files, checking the clock, etc.). This model is thus *interleaving-oblivious*: *the execution of the application is completely independent of the interleaving of operations on different processes*.

To see that this is true, consider an application that is executing on a given input and is using a given set of non-deterministic event outcomes. Assume that we have a prefix execution that completely depends on the input and the outcomes set (it is "determined"). We must show that the same is true for the next operation performed by any process. If a given process is not blocked on a receive, it can either execute a local operation or a send. In both cases the arguments of the operations are determined, meaning that the outcome of the operations is also determined. If the process is blocked on a receive, it can either remain blocked or receive a message. If the former, nothing changes. If the latter, we know that the FIFO channel chosen by the receive is determined and that the receive is matched to the next message on this FIFO channel. Further, the identity of the message is also determined because FIFO ordering means that a given process' current blocked receive of a given FIFO channel must return the data of the oldest unreceived send operation on the other side of this channel. Since both the previous sends on the sender process and the previous receives on the receiver process are assumed to be determined, the received data is also determined. Thus, the outcomes of all operations during the application's execution depends exclusively on the application's input and set of non-deterministic events outcomes.

One corollary is that *for a given input and given set of outcomes of non-deterministic events, there exists a unique matching between the* send *and* receive *operations performed by different processes*. Another corollary is that *all interleavings of application operations are equivalent*, in that they all produce the same sequence of state transitions on each process.