Runtime Detection and Optimization of Collective Communication Patterns

Torsten Hoefler
Department of Computer Science
ETH Zurich
Switzerland
htor@inf.ethz.ch

Timo Schneider
University of Illinois
at Urbana-Champaign
Illinois, USA
timos@illinois.edu

ABSTRACT

Parallelism is steadily growing, remote-data access will soon dominate the execution time of large-scale applications. Many large-scale communication patterns expose significant structure that can be used to schedule communications accordingly. In this work, we identify concurrent communication patterns and transform them to semantically equivalent but faster communications. We show a directed acyclic graph formulation for communication schedules and concisely define their synchronization and data movement semantics. Our dataflow solver computes an internal representation (IR) that is amenable to pattern detection. We demonstrate a detection algorithm for our IR that is guaranteed to detect communication kernels on subsets of the graph and replace the subgraph with hardware accelerated or hand-tuned kernels. Those techniques are implemented in an open-source detection and transformation framework to optimize communication patterns. Experiments show that our techniques can improve the performance of representative example codes by several orders of magnitude on two different systems. However, we also show that some collective detection problems on process subsets are NP-hard. The developed analysis techniques are a first important step towards automatic large-scale communication transformations. Our developed techniques open several avenues for additional transformation heuristics and analyses. We expect that such communication analyses and transformations will become as natural as pattern detection, just-in-time compiler optimizations, and autotuning are today for serial codes.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*

General Terms

Performance

Keywords

Collective Operations, Dynamic Optimization, MPI

1. INTRODUCTION

Compiler transformations, such as loop tiling, hoisting, unrolling, or fusion, are well known to improve performance of serial codes.

Such transformations are important for performance portability to different architectures and work by transforming the dataflow graph (e.g., in a polyhedral model) without changing the program semantics. As applications move to hundreds of thousands of processors in the next generation computing systems, it is not sufficient to only optimize the serial parts of the computation. In fact, it can be expected that the time to solution will be dominated by communication and it will thus be most important to also optimize those critical communication parts of parallel applications.

Well-known optimizations at the message level, such as pipelining [30], remote direct memory access [24], or message coalescing, have a fundamental limitation: *No matter how powerful, they are no substitute for hand-optimized (collective) communication algorithms.* Consider a blocking broadcast to P processes. For large P, a less optimized binomial tree algorithm will always outperform a highly optimized (but asymptotically slower) linear broadcast.

Similar arguments have been well-studied for serial codes. For example, the performance of hand-tuned routines such as Goto's dense matrix multiplication [10] can hardly be achieved by automatic transformations of the simple three-loop formulation. Instead, kernel recognition is considered as generally beneficial [19]. The recognition is often performed on some representation of the dataflow graph [13] where semantic components are replaced with calls to the optimized kernel routine. *In this work, we extend this idea towards the recognition and replacement of parallel application communication patterns.*

Most of today's large-scale parallel codes are implemented in a message-passing programming model using the Message Passing Interface (MPI) standard [20]. MPI offers a large set of predefined communication patterns called *collective operations*. Many newer programming paradigms, such as PGAS languages [22, 33, 6, 5], declarative parallel languages [17], or annotation-based languages [35, 23] provide a conceptually simpler programming interface and rely on the compiler or the communication layer for optimizations. While some languages offer intrinsics or runtime libraries for several collective operations, many do not support such a specification **at all** moving the burden of pattern recognition to the compiler or forcing the user to hand-code non performance-portable algorithms. The following table shows an exemplary overview of parallel languages and environments and their support for collective operations:

MPI 2.2 [20]	17 collective library calls
UPC 1.2 [33]	8 collective library calls
CAF 2.0 [18]	9 collective intrinsics
CAF (Fortran 2008) [22]	no support for collectives
OpenMP [23]	only reductions
Chapel [6]	compiler support mentioned
XscalableMP [35]	compiler support mentioned

Many languages mention compiler support in publications but it is not implemented yet to the best of our knowledge. Others provide only the most commonly used collective operations. Even MPI which offers arguably the most extensive and mature set of collective operations may not support all variations. A good example is the recently introduced MPI_Reduce_scatter_block, which has been missing in MPI for 15 years and which offers additional optimization opportunities compared to MPI_Reduce_scatter. The optimization framework that we will present could have auto-detected this more restricted collective operation and replaced it with a hand-tuned kernel transparently.

Another issue with today's parallel programs is that many legacy codes use point-to-point patterns instead of collective communications because at some point programmers observed them to be faster than the corresponding MPI collective operation (in one particular MPI implementation). This problem is well known in the MPI community leading to efforts to define self-consistency performance metrics for MPI [16] to prevent it. Other applications implement hand-tuned pipelined communication patterns instead of using MPI optimized collective operations. The most prominent example here is the High Performance Linpack benchmark [8] which is used to rank machines in the top 500 list.

The lack of support for collective operations in many high-level programming models, legacy MPI codes using "manual" collectives, the potential for identifying new important collective communications, and the pipelined collective implementations motivate the *automatic detection and optimization of known communication patterns in parallel programs*. Such a detection scheme would allow all parallel languages (including Chapel and XalableMP) to take advantage of collective semantics and allow them to use highly tuned communication kernels or collective acceleration hardware (cf. Section 1.3).

Pattern detection and optimization could either be carried out during compile time or during run time of the application. Compile time, or static, optimizations have the advantage that their execution time does not influence the overall performance of the applications but it often suffers from *exponential state explosion* [34] and it cannot be applied if the communication structure is *input-dependent*. Run time, or dynamic, optimizations on the other hand can benefit from all information available at runtime. The overheads during runtime are usually offset by the repeated use of the optimized operation. We will show in our experimental section that the number of necessary repetitions needed to offset the optimization costs shrinks with the growing numbers of processes and is often small (e.g., less than 10). Thus, we focus on dynamic optimization in this work.

1.1 A Simple Internal DAG Representation

We use a simple, domain-specific directed acyclic graph (DAG) formulation to represent communications that we call *communication DAG* (cDAG). cDAG allows the DAG specification of operations such as sends, receives and local operations. All cDAG operations form a global graph with local dependencies (edges specifying the order of operations at each process) and remote dependencies (edges specifying dataflow and synchronization between processes).

In the following section, we show an example how well-known compiler transformations can modify a PGAS code to specify a cDAG IR during runtime. However, the focus of this work lies not on generating the IR from application codes (there are too many parallel libraries and languages in use today to tackle such a task in a single article) but rather how a cDAG just-in-time compiler and optimizer can detect communication primitives in the DAG IR and how it can transform the communication graph to use optimized kernels.

1.2 Extracting Communication Graphs

Communication graphs in MPI can be found in the parallel control flow graph of an MPI program (pCFG, [3]), or they can be specified directly as neighborhood collective communications, a functionality proposed for MPI-3.

To demonstrate the applicability to PGAS languages, we use an example from the FT code, the fast Fourier transform kernel from the NAS benchmark suite that was ported to UPC [9]. It includes the following code-fragment:

```
upc_barrier;
for (int i = 0; i < THREADS; i++) {
    upc_memget((dcomplex*)&dst[MYTHREAD].cell[chunk*i],
    &src[i].cell[chunk*MYTHREAD], sizeof(dcomplex) * chunk );
}
upc_barrier;
```

A parallel control flow graph (pCFG) as described in [3] could be used to track the dataflow from one image to another. Dataflow analysis identifies the buffer `dst[MYTHREAD].cell[chunk*i]` as target and `src[i].cell[chunk*MYTHREAD]` as source buffer. The analysis would then replace the data movement with send and receive calls which add edges to the communication DAG. This simple transformation would then result in the following cDAG code:

```
upc_barrier;
cDAG_graph g = cDAG_CreateGraph();
for (int i = 0; i < THREADS; i++) {
    int msgsize = sizeof(dcomplex) * chunk;
    cDAG_Send(g, &src[MYTHREAD].cell[chunk*i], msgsize, i);
    cDAG_Recv(g, &dst[MYTHREAD].cell[chunk*i], msgsize, i);
}
cDAG_Schedule sched = cDAG_CompileGraph(g);
cDAG_Run(sched);
```

We observe that the code's branch structure stays the same which simplifies the compiler analysis needed to perform this step. The function cDAG_CreateGraph() creates a DAG object to which send and receive operations and dependencies between them can be attached (cDAG_Send() and cDAG_Recv() do not communicate data themselves but add the specified communications to the communication graph). cDAG_CompileGraph() performs the online compilation, detection, and transformation of the graph as described later in this paper, and cDAG_Run() executes the optimized schedule. Just like a binary program, a cDAG schedule can be re-used with cDAG_Run() after it has been compiled, so repeated calls of the same loop would not need to rebuild the graph (the code for checking if the graph object already exists is trivial and has been omitted for brevity and clarity). We remark that cDAG is somewhat similar to MPI Datatypes [20] but targets communication patterns instead of memory accesses.

The described analyses and transformations into cDAG graphs use only well-known dataflow-techniques and simple loop splitting to transform the serial code and are outside the scope of this work. We also note that transformations to cDAG can always be done manually or with the help of refactoring tools.

1.3 Optimized Collective Operations

Numerous research groups demonstrated speedups up to several orders of magnitude using optimized algorithms and implementation strategies for collective operations in hundreds of publications, e.g., [31, 4, 25]. In addition to such highly optimized point-to-point patterns, nearly all modern supercomputers and high-performance networks such as Blue Gene/P [15], IBM PERCS [2], InfiniBand ConnectX2 [11], and Portals [29] provide specialized hardware support or even separate networks [15] for collective operations. Such *collective accelerators* provide significant speedup over implementations based on point-to-point messages and can easily be utilized once a supported collective operation is detected.

Implementing collective operations on shared memory systems is also a well-known research topic, e.g., [32]. The best implementations often depend on the details of the cache coherence protocol on a particular architecture and are often not performance-portable. Thus, machine-specific optimizations may provide benefits even for shared memory models such as OpenMP.

1.4 Related Work

Detecting collective operations in point-to-point patterns is an important problem. Most previous works performed the detection *post-mortem* by analyzing traces of the program run. This bears several problems: (1) real traces are often of unmanageable size, (2) on-node data-transformations (e.g., reductions) cannot be captured, and (3) the optimizations cannot be applied dynamically during the program run which is problematic if communication patterns depend on the input data. The fundamental difference between most trace-based analyses and our detection scheme is that we have exact data-dependency information, where trace-based analyses often resort to heuristic approches.

Knüpfer et al. proposed a scheme to detect alltoall, scatter, gather, and broadcast by analyzing point-to-point patterns in message traces [14]. This detection scheme can guide programmers to apply source-code changes. However, the authors mention that their analysis suffers from detecting "pseudo patterns" for some process counts but not for others. While programmers cannot replace such an operation with a collective call, our dynamic optimization scheme is able to optimize this case transparently without user intervention. In addition, Knüpfer's detection scheme is also not suitable to detect if data is forwarded through other processes as commonly done in manually optimized algorithms.

Kranzlmüller et al. investigate the detection of repetitive communication patterns in [26]. Like Knüpfer, they only look at single point-to-point operations in MPI traces such that forwarding of data through proxy processes cannot be detected in this scheme. The same authors discuss how to guide compiler transformations with the trace analysis techniques [28, 27], however, they leave the problem of detecting collective operations open.

We are not aware of any dynamic collective detection scheme. In this work, we propose a scheme that is *guaranteed* to detect collective operations on the full process set in a communication graph. That is, our scheme will detect collective operations, even after arbitrary transformations of the communication graph (e.g., data forwarding through proxy processes). Even though we focus on dynamic optimizations in this paper, the discussed techniques can also be applied statically (post-mortem, trace-based, assuming that the trace contains the whole dataflow including all buffer addresses). We presented some preliminary results of our work as a poster [12].

1.5 Terms and Notation

Let the complete communication and dependency graph of represented as $G = (V, E)$ where the vertex-set V contains the send and receive operations, $S \cup R$. The set of edges $E = D \cup M$ contains the communication edges (we call this *matching set* $M \subseteq S \times R$) and local dependency edges $D \subseteq V \times V$ (this set specifies the order of process-local send/recv operations). We use $|V| = n$, $E = M \cup D$, and $|E| = m$ in our analyses. The set of processes is denoted as P and $|P| = p$.

1.6 Limitations and Contributions

The proposed scheme is limited by the power of the transformations of a parallel code to a cDAG code. The symbolic analysis of pCFGs is often limited by the problem of static message matching [3] and fully automated transformations may not always be possible. Those static compiler transformations are outside the scope of this work. However, user-annotations or manual re-writing to cDAG can also be used to enable our optimizations. Once all concurrent communication operations are expressed in cDAG, our automatic detection and transformation scheme is guaranteed to find and replace all patterns that are semantically equivalent to predefined collective operations.

The key contributions of our work are:
- A specification of synchronization and dataflow semantics for communication DAGs.
- A dataflow propagation algorithm that transforms the communication DAG into an efficient internal representation in time $\mathcal{O}(n \log m)$.
- A classification of MPI collective operations with regards to their level of abstraction and composeability.
- An algorithm for detecting all collective operations in time $\mathcal{O}(n \log n)$.
- A proof that the problem of detecting several collective operations on process subsets in communication graphs is NP-hard.

In general, we demonstrate a flexible approach for dynamic communication optimization of parallel communication graphs. We accompany the theoretical discussion and analysis of the dataflow algorithms with a practical and complete open-source implementation of the analysis and transformation and demonstrate the benefits of applying our scheme to different parallel codes and architectures.

2. DYNAMICALLY DETECTING COLLECTIVE OPERATIONS

First we introduce our running example to explain our notation and detection scheme. Figure 1 shows an optimized alltoall implemented with a butterfly communication pattern [4] on four processes. All previous collective detection schemes would not be able to find the alltoall communication in this pattern. We will use this nontrivial example throughout the paper (Figures 1 to 5) to describe the dataflow propagation and other algorithms.
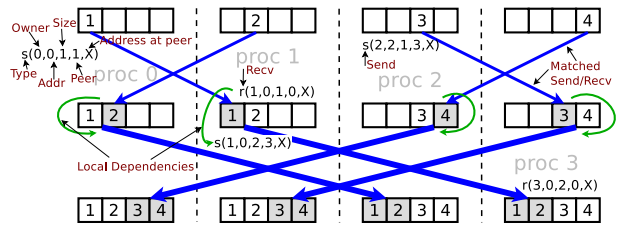

Figure 1: Alltoall with a Butterfly Pattern (p=4)

The send and receive sets (*S* and *R*) contain the send and receive operations and elements in S and R are named $s(...)$ and $r(...)$, respectively. Not all elements of those sets are shown in Figure 1

due to space constraints; Figure 2 contains all elements. Each element of those sets is a tuple with (owner, address, size, peer, address at peer). **The dependency set (D)** contains local dependencies (curved arrows) between operations. Dependencies are added by the transformations described in Section 1.2 if there are dependencies in the local control flow graphs between accesses (sends or local operations) to buffers that have been received before. This is illustrated in round 2 of the dissemination algorithm where the send on each rank reads data that was received before, i.e., the send *depends* on the completion of the respective receive. **The matching set (M)**, represented by straight arrows in Figure 1, is determined by the local order (specified by D) and send/recv pairs (S/R).

The dataflow propagation and all auxiliary algorithms are *serial*, i.e., we assume that the DAG is communicated (gathered) to a single process to perform the analysis and transformation and is scattered to all processes before execution. We will discuss the overheads of this approach in Section 3.

2.1 The Message Matching Problem

Many **message passing specifications** (e.g., MPI) use tagged communications (or named channels). User-selected tags restrict the valid executions such that only sends and receives with *matching* tags and source arguments generate a flow in the communication graph. Some messaging interfaces allow wildcard matching which can lead to non-deterministic [34] communications. A correct program must allow any matching order for non-determinstic matchings and our algorithm can match messages in one particular order to avoid *state explosion* (cf. [34]).

Applications using either the **one-sided PGAS or shared memory model** specify the remote buffer for each remote access directly and our transformation to cDAG generates a send/recv pair for each access. In this special case, the matching set M can be computed in $\mathcal{O}(1)$ by including all auto-generated send/recv pairs and the following matching algorithm is **not** necessary.

We now discuss an algorithm for generic dynamic matching, i.e., to construct the matching set M from the process-local sets R_i, S_i, and D_i for $0 \le i < p$. The algorithm essentially simulates a parallel execution of the schedule and needs to follow local and remote (send/recv) dependencies to ensure message matches in correct order.

Our algorithm starts to match send and receive operations that have no dependencies (in the Matching Queue MQ). Once a match is established, the operations and their dependencies are removed. Operations for which all dependencies have been removed may be and matched are thus added to MQ. This fix-point algorithm is repeated until it converges. This serial algorithm could be replaced by a symbolic parallel execution with zero-byte messages to record the matching set M, however, our algorithm performs significantly better than sending each message over the network. The complete pseudo-code is shown in the appendix in Algorithm 1.

An example for matching is shown in Figure 2 and 3. Figure 2 shows the initial state in the first iteration. All shaded tuples are in MQ waiting to be activated. Figure 3 shows an intermediate state of the algorithm where the first receive of process 0 is matched. After matching, the dependency is removed and the second send to process 2 is added to MQ (shaded grey). The algorithm continues by picking the next unmatched operation.

Time Complexity of Matching.
Each node is inserted in MQ exactly once and in each fixpoint iteration one node is removed from MQ, forcing convergence after n iterations. Finding the matching target for each node and updating the appropriate sets can be done in time $\mathcal{O}(n \log n)$ using red-black
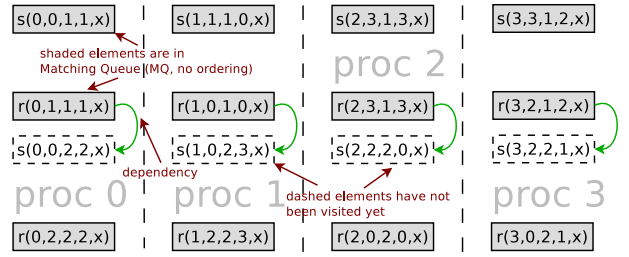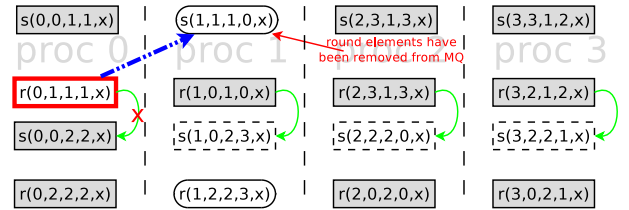

Figure 2: Matching Example, initialization (p=4)

Figure 3: Matching Example, after a few steps (p=4)

trees. While traversing the tree, each of the $|D| < m$ dependencies is removed once and each vertex is traversed once leading to an additional time of $\mathcal{O}(n + m)$. The total cost for the matching algorithm is thus $\mathcal{O}(n \log n + m)$.

2.2 Synchronization and Data Movement Semantics

Every communication has two separable side effects: *synchronization* and *data movement*. Synchronization semantics define for each process from which other processes it receives a message, i.e., it has to wait for. Data movement semantics define how memory locations are changed during the execution of a communication DAG. Both semantics are independent: synchronization can happen without changing remote memory (the best example is a barrier collective operation or simply zero-byte communications) and data movement can happen without synchronization (e.g., one-sided memory accesses).

2.3 Dynamic Dataflow Analysis

Our dataflow propagation computes both semantics: **Synchronization** is captured in the **wait set** W_i, which specifies the processes that process i needs to wait for (synchronize with) before exiting the communication. The wait set W_i for process i can be computed with a (reverse) traversal of the communication and dependency graph $G = (V, E = M \cup D)$ starting at i and adding all reachable processes to W_i. We omitted the detailed algorithm of this simple traversal due to space restrictions.

The more complex **data movement** semantics are expressed as a set of dataflow tuples from all **original sends** S^0 to all **final receives** R^∞. An original send is a send operation that specifies user memory that has not been received from another process and a final receive is a receive that specifies user memory that will not be overwritten as destination. The sets S^0 and R^∞ represents all dataflow semantics of a communication graph. In fact, they form an IR similar to the well-known Static Single Assignment form used for serial compiler transformations (every address is only written once). Thus, we call the resulting representation of the dataflow from S^0 to R^∞ Static Single Transfer (SST; each transfer from a source to a target buffer happens exactly once). The SST form expresses the overall data-movement abstractly and allows to detect (pattern-match) collective operations.

We use a dataflow propagation algorithm to transform our DAG IR into SST form. This fixpoint algorithms propagates the send buffer information along the matched send/receive edges in the graph until they reached the final receives. Like before, each send and receive is specified as a tuple of (owner, start address, size, peer (source or destination), start address at peer). Our algorithm works on a queue of independent sends (sends without incoming dependencies) and propagates the dataflow from the sends to the final receives. As receives are processed in the dataflow, their dependencies are marked as satisfied which may enable new sends to be added to the queue. Figure 4 shows the tuple transformations performed by the dataflow
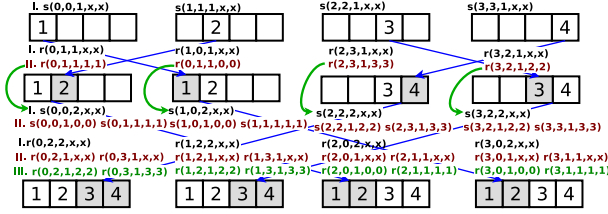

Figure 4: The Dataflow Propagation (p=4)

propagation algorithm. The original tuples are shown in the black lines, marked with *I*. the tuples that are transformed in the first iteration are shown in red and marked with *II*, and the tuples transformed in the second (last) iteration are shown in green, marked with *III*. The dataflow algorithm has to *split* send/recv matches in the second iteration to separate data flows from the local memory and data that was received in the first iteration. If a received buffer is not sent fully or sent to different destinations, we split the backwards dataflow into two separate flows (SST assignments, represented as tuples).

Figure 5 shows the split at process 0 in the first iteration of the dataflow algorithm. Since the two-element message from process 0 to process 2 sends data from local memory (original send) as well as data that was received from process 1, it needs to be split into two SST assignments.
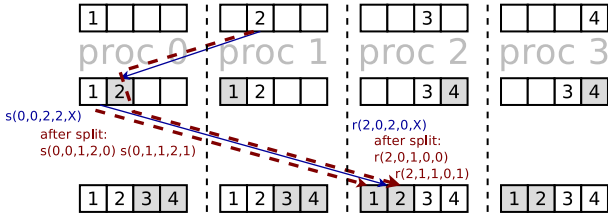

Figure 5: Split Example in Butterfly Alltoall (p=4)

In our example in Figure 4, the final set of receive tuples at process 0 is $\{(0,0,1,0,0), (0,1,1,1,1), (0,2,1,2,2), (0,3,1,3,3)\}$, which means that process 0 receives data from location $i \in \{1, 2, 3\}$ at process i into local location i. These SST assignments represent an alltoall pattern. We will discuss pattern detection in Section 2.4.

Algorithms 3 and 4 in the appendix show the detailed propagation and split algorithms that compute the SST for each process.

Time Complexity of Dynamic Dataflow Analysis.
We analyze the complexity by stating some invariants for the dataflow propagation. Every send or receive will be visited exactly once in the algorithm (data is propagated only forward). Finding a match in the ordered matching set M costs $\mathcal{O}(\log |M|)$. Each split can generate new send/recv vertices. However, a critical observation is that one recv in the input can result in a max-

imum of two new split send/recv pairs (cf. Figure 5). And every newly generated send/recv pair can again only generate a single split because the communicated memory is consecutive. Thus, there are a maximum of $\mathcal{O}(|V|)$ additional send/recv pairs generated in splits. The total runtime of the propagation algorithm is thus $\mathcal{O}(|V| \log |M|) = \mathcal{O}(n \log m)$.

2.4 Detecting MPI Collective Operations

Each collective data movement operation can be defined as a set of SST assignments (as returned from the dataflow propagation algorithm), for example a broadcast can be defined as $(i, o_i, l, j, o_j)\ 0 \le j < P, j \ne i$, where i is the root process, s is the length of the message, o_i is the offset at the root process, j are all other (receiving) processes, and o_j the local memory offset at process j.

2.4.1 Collective Data Movement Signatures in SST

We now discuss the signatures for all collective operations that move data. We use the SST tuple notation like before, i.e., (s, o_s, l, r, o_r) means that data of size l is moved from offset o_s at process s to offset o_r at process r. We use r to denote the root process of a collective (if applicable), j and i to denote source and target processes respectively, o_x to denote the local offset for communication originating or terminating at process x, and $o_{x,y}$ to denote local offset for communication from process x to y.

The following table lists the data movement signatures that are used to detect common collective operations:

bcast	$(r, o_r, l, j, o_j)\ 0 \le j < p, j \ne r$
gather	$(i, o_{i,r}, l, r, o_{r,i})\ 0 \le i < p, i \ne r$
scatter	$(r, o_{r,j}, l, j, o_j)\ 0 \le j < p, j \ne r$
allgather	$(i, o_i, l, j, o_{i,j})\ 0 \le i, j < p, j \ne i$
alltoall	$(i, o_{i,j}, l, j, o_{i,j})\ 0 \le i, j < p, j \ne i$

This list can easily be extended with any data movement signature. The last condition ($j \ne r$ or $j \ne i$) removes the send-to-self, i.e., local copy, from the detection.

2.4.2 Detection Hierarchy for Collective Operations

The data movement patterns show that MPI's collective operations form a strict hierarchy, i.e., some collective operations can be expressed as a set of others. For example, an alltoall can be expressed with p broadcasts, an allgather can be expressed as p gather operations, or a broadcast can be expressed as a scatter from the same memory location at the root. Collective operations that can be expressed as a composition of other operations offer a higher level of abstraction with higher optimization opportunities [14]. Thus, we establish the following order of collective operations from "highest" to "lowest" abstraction: **allgather** $\succ$ **alltoall** $\succ$ **broadcast** $\succ$ **gather, scatter**.

2.4.3 An Algorithm to Detect Collective Operations

We can now detect collective operations by searching for the data movement signatures from the most abstract to the least abstract operation as defined before. After finding an operation, we remove all tuples that matched this operation and insert a call to the detected operation instead (i.e., remove and replace the communication kernel). We then run the algorithm repeatedly until no operation is detected.

The detailed algorithm is shown in the appendix as Algorithm 5.

Time Complexity.
The number of SST assignments is limited by $\mathcal{O}(n)$ because each send and recv will result in one receive tuple and each send/recv pair can at most generate one new assignment in a split. The tuples
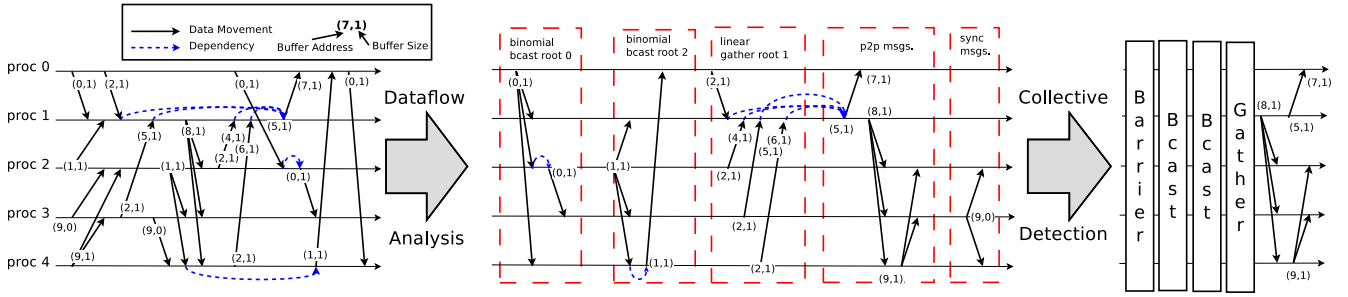
Figure 6: A more complex artificial test graph with 20 **messages among five processes**

are sorted in $\mathcal{O}(n \log n)$ before the detection. Assuming a constant number of message sizes, each tuple is only considered once in the detection of each algorithm. Thus, the overall time complexity of the detection algorithm is $\mathcal{O}(n \log n)$.

Operations with Noncontiguous Data Access.
Some communication layers (e.g., MPI) offer support for noncontiguous data layouts in send and receive memory. This allows to send data that is scattered in memory within the same collective operation call. The detection algorithm will detect one collective operation for each contiguous block (SST tuple). We can then simply combine all collective operations of the same *shape* (i.e., the same operation and the same root if applicable) into one collective operation with noncontiguous data layouts (e.g., MPI datatypes).

Detecting Vector Collective Operations.
We excluded the vector operations alltoallv, alltoallw, allgatherv, scatterv, gatherv from our previous classification because they can trivially match many SSTs, e.g., a simple send/recv pair can be expressed as gatherv with all counts but one set to zero.
Nevertheless, we can modify our detection algorithm to support vector collectives if we allow arbitrary data-sizes between pairs of processes and define a factor τ which specifies the desired *density* of the collective, i.e., the percentage of edges relative to the nonvector operation that need to exist in the detected vector collective. The vector collective detection is similar to the previous algorithm but only $\tau \cdot p(p-1)$ (allgather, alltoall) or $\tau \cdot (p-1)$ (broadcast, gather, scatter) need to be found to detect a collective vector operation.

Barrier Collectives.
Barrier collectives are an exception because data is neither communicated nor reduced. Thus, the detection based on dataflow cannot detect barrier calls. However, the barrier call will be detected by the synchronization detection and will result in complete wait sets for all processes. In general, any transformation must check if is fulfills the synchronization specified by the wait sets. If it lacks synchronizations, then they can simply be introduced by zero-byte send/recv pairs.

3. EXAMPLES AND EXPERIMENTS

Figure 6 shows a more complex example (one of our unit tests) to demonstrate the capabilities of our detection scheme. The left side shows a communication graph with five participating processes. Each solid arrow represents a message with a send (sendbuf, size) and a receive (recvbuf, size) at the beginning and end. The middle figure shows the result after our detection algorithm and the right shows the optimized graph.
We tested several different detection patterns in order to evaluate

correctness and performance of our implementation. We generated simple linear and complex (binomial tree, bruck, dissemination) broadcast, gather, scatter, and alltoall graphs and for different numbers of processes and added p random messages with dependencies. Our implementation detected all collectives and correctly isolated all operations that were part of it. In the following, we provide a performance analysis for detecting a broadcast, implemented as binomial tree on p processes.

3.1 Experimental Environment
Our optimization framework is implemented in C++ and uses MPI as transport layer. We analyzed the performance on multiple parallel systems and present the two systems: (1) Odin, an InfiniBand cluster with quad-core Opteron CPUs and (2) NERSC Hopper, a Cray XE6 with 2.1 GHz AMD MagnyCours CPUs. We used the default C++ compilers on both systems with highest optimization available (g++ 4.1.2 with -O3 on Odin and PGI 11.7 with -fast on Hopper). The complete source code and tests can be downloaded from http://unixer.de/research/cDAG.

3.2 Scalability and Performance Analysis
Figure 7(a) shows the runtime of the different parts of the collective detection on Hopper as described before. **Matching** implements a simple depth-first search (DFS) traversal as described in Section 2.1, **Dataflow** is the dataflow propagation algorithm, **Colldetect** is the detection and replacement algorithm, and **Scatter/Gather** represents the communication overhead to gather all the schedules to a single process and distribute (scatter) the optimized schedules back to all processes.
The figure shows measurement results as dots and our performance models as lines. For the broadcast problems, we have $n \approx p \approx m$. Thus, we use $a \cdot p \log(p)$ to represent the runtime of the Dataflow, Matching and Colldetect phases. The used scatter and gather implementations are linear (sending from the root to all processes). For relatively small process counts, the (≈ 100 Bytes) messages are communicated quickly into local eager MPI buffers. Beginning with ≈ 2500 processes, those buffers are exhausted and the curve starts the expected linear slope and we use the model $b \cdot p$ to represent the runtime. The performance for large process counts (> 2000) can be accurately represented with the following constants for a and b:

System	a (Dataflow)	a (Matching)	a (Colldetect)	b
Hopper	$0.59\ \mu s$	$0.19\ \mu s$	$0.15\ \mu s$	$39\ \mu s$
Odin	$0.85\ \mu s$	$0.15\ \mu s$	$0.13\ \mu s$	$21\ \mu s$

The b factor in the communication is dominating on both systems with $39\mu s$ and $21\mu s$ per process. For example, the detection and optimization scheme would take ≈ 15s on 300.000 cores of a petascale-class machine. The used memory for 300.000 processes is less than 85 MiB (283 Bytes per process). The whole optimization process can be offloaded to a separate core and optimize sched-
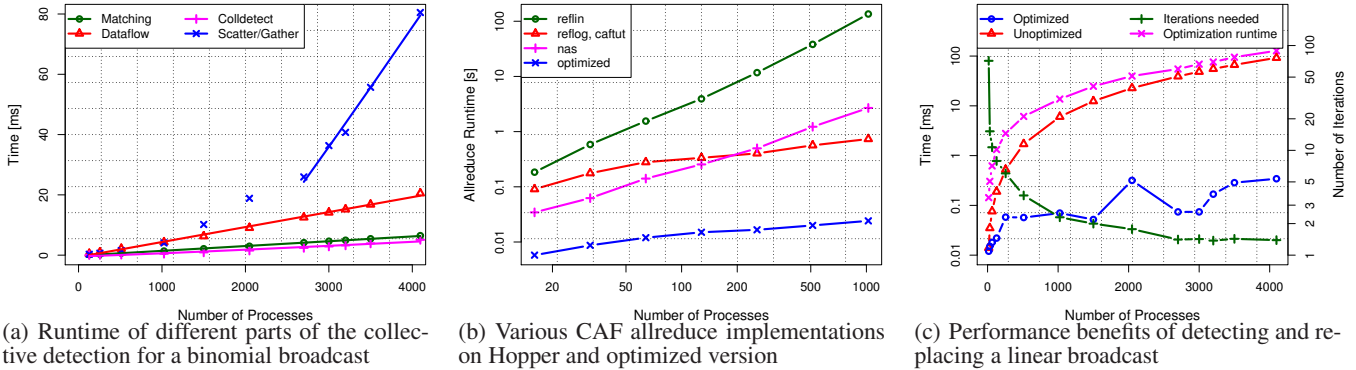
(a) Runtime of different parts of the collective detection for a binomial broadcast

(b) Various CAF allreduce implementations on Hopper and optimized version

(c) Performance benefits of detecting and replacing a linear broadcast

Figure 7: Experimental Results of the cDAG implementation with collective detection enabled

ules while the (unoptimized or partially optimized) communication takes place.

3.3 Microbenchmarks

We now discuss the possible performance gain by detecting and replacing known communication patterns in existing applications. We utilize the patterns that we found in the Co-Array Fortran NAS codes [7], the UPC NAS codes [9], and SPhot, an MPI application from the Sequoia benchmark suite.

First, we investigate the performance of logical allreduce patterns in Co-Array Fortran. We compare six different implementations in order to capture several versions of code that a typical user would implement. The first implementation ("reflin") is a linear reference algorithm from [22] (Section 2.5 "Summing over the co-dimension of a co-array"). The second algorithm ("reflog") is a more complex version with logarithmic complexity of the same algorithm [22] (Section 4.4). The third implementation ("caftut") has identical performance as "reflog" and was given as an example by Numrich in [21]. The fourth implementation ("nas") was extracted from the Co-Array Fortran version of the NAS benchmarks [7]. The last implementation ("optimized") represents an architecture-optimized call to MPI_Allreduce.

Figure 7(b) shows the time to perform 1000 allreduce calls with different number of processes on the Hopper system using the Gemini interconnect that is optimized for PGAS and Co-Array Fortran [1]. We used the Cray PE version 3.1.61 for all experiments. We observe that the different implementations have widely varying and scale-dependent performance characteristics. However, all user implementations are significantly slower than the well optimized "opt" algorithm (more than three orders of magnitude for the linear implementation).

As discussed before, the dynamic optimization comes with an overhead during runtime (cf. Figure 7(a)) and it is important to compare the runtime overhead to the gains of the optimized schedule. In Figure 7(c) compares the naive (as one would do in CAF for example) and the transformed execution of a broadcast on Hopper. It also shows the runtime of the complete detection and transformation (magenta, diagonal crosses) which is just slightly slower than a single linear broadcast. The green line shows how often a schedule needs to be re-executed before our scheme decreases the application runtime. We see that the necessary number of iterations is rapidly decreasing with scale and with $p > 1500$ even two iterations (a single re-use of the schedule) are sufficient to offset the costs of the optimization. On Odin two iterations are beneficial for $p > 1000$.

3.4 Application Benchmarks

In this section, we demonstrate the usefulness of our method on two representative applications, SPhot and NAS/FT. SPhot is part of the Sequoia benchmark suite. It contains a loop which performs a logical gather using point-to-point messages. Our cDAG scheme detected and optimized this communication pattern by replacing it with a call to a vendor-optimized gather call. The observed application improvement ranged from 0.3% to 14.1%.
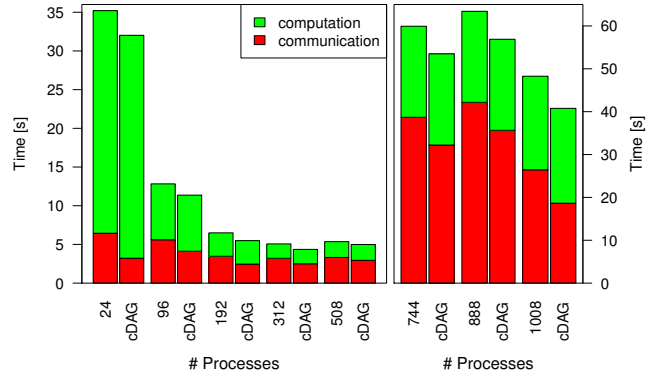

Figure 8: NAS/FT Performance on Hopper

To demonstrate the applicability to PGAS codes, we applied the cDAG scheme to the NAS codes that were ported to UPC [9]. While those codes offer several opportunities to detect and optimize collective patterns, we will focus on the FT (fast Fourier transform) benchmark in the following. The benchmark contains the exact loop and transformation that we showed in the example in Section 1.2. We used the class C FT benchmark up to 508 processes, to scale further we utilized class D (which could not be used for smaller numbers of processes due to memory requirements).

Our scheme replaces the communication in NAS/FT with an optimized version that uses Cray's DMAPP low-level communication API to perform the alltoall. We observe a 29% improvement of the raw communication performance which translated to 16% improvement in overall application performance at 1008 processes.

4. ON DETECTING COLLECTIVES ON PROCESS SUBSETS

We discussed fast algorithms for detecting arbitrary collective operations in arbitrary communication patterns. However, those algorithms only detect operations that are performed on the *complete set of processes*. The problem of detecting collective operations on arbitrary process subsets is also relevant. For example, a pat-

tern where every other process participates in a global broadcast or a pattern where all but one processes perform a global alltoall exchange.

This problem is much harder; we start by showing that the simplified problem of detecting maximum groups with barrier semantics is NP-complete. This makes the general case of detecting collective operations on maximum process subsets NP-complete.

4.1 Finding Maximum Barrier Groups

We say that a set $C \subset P$ has barrier semantics iff the wait sets W_i (cf. Section 2.3) of all $i \in C$ include all processes in C. We define the *single largest barrier* (SLB) problem as follows: Find the largest subset of $C \subset P$ that has barrier semantics and that cannot be extended by another process $p \in P$ such that the set $C \cup \{p\}$ has barrier semantics.

THEOREM 1. *The construction problem of SLB is NP-complete.*

PROOF. First, we show that SLB is in NP: Testing if a subset of processes C of a pattern has barrier semantics can be done by checking if every node in the wait set of all other nodes in C. The wait sets can be established in polynomial time as described in Section 2.3 and the check can be performed in $\mathcal{O}(|C|^2)$.

We show NP-completeness of the SLB problem by showing CLIQUE $\leq_P$ SLB. Let the undirected graph $G = (V, E)$ be the input for CLIQUE. We can now construct a cDAG communication graph $G' = (V', E')$ which has an SLB of size k iff G has a clique of size k. The mapping between elements in V and processes in the cDAG is bijective. For each edge $(u, v) \in E$ add a send and a receive operation to V' so that the process u' mapping to u sends data to the process v' mapping to v, the cDAG contains no dependencies and the matching set E' is such that $(u, v) \in E \Leftrightarrow (u', v') \in E'$. By construction, G' now has a SLB of size k iff G has a clique of size k. $\square$

The barrier problem is the simplest to show NP completeness since no data is communicated. However, adding dataflow in the form of SST does not make the problem simpler. Informally said, the detection of all-collectives (allreduce, alltoall, allgather, etc.) in our SST formulation is likely to be as hard as solving the SLB problem.

4.2 Finding Maximum Subsets of Rooted Collectives

The problem of finding maximum groups for rooted collectives (e.g., scatter, gather, bcast) can be solved in polynomial time in the SST form (Algorithm 3).

The detection algorithm is similar to the collective detection algorithm in Section 2.4.3. For example for broadcast or scatter it simply tests the number of flows of equal size from each process to other processes. The process with the highest number of flows to other processes is root of the largest broadcast or scatter subgroup. The matching nodes can be removed and the algorithm can be applied to find the next largest set. The time to perform this detection is $\mathcal{O}(n \log n)$. Algorithms for all other rooted collectives follow the same principle.

5. DISCUSSION AND FUTURE WORK

In this work we propose a dynamic transformation scheme for communication graphs as first steps towards communication- and network-centric compiler optimizations. We expect this area of research to grow quickly in importance as we are moving towards higher parallelism. Our transformations uses an abstract graph-based representation as input and we developed a set-based single static transfer (SST) representation that represents dataflows from source buffers to target buffers in a distributed memory model. We show that SST is ideally suited for analyses such as the detection of collective operations. We demonstrate fast algorithms for building the SST form and computing synchronization semantics of the input graph.

We introduced a hierarchy of collective operations based on their level of abstraction and composeability. We demonstrated that the algorithms are able to detect collective operations on the full process set on arbitrary subsets of operations in a graph. We proved that finding maximum synchronization groups (e.g., detecting barriers) on arbitrary process subsets is NP-complete. However, we also provide polynomial-time algorithms to detect rooted collectives on largest process subsets. Developing practical heuristics for detecting all-collectives on maximum process subsets is an interesting problem for future work.

Our approach for dynamic communication graph optimization is feasible in practice up to large numbers of processes. Our accurate performance model of the optimization time on 300.000 processes predicts $\approx 15s$ and a memory requirement of 283 Bytes per process.

We demonstrated that PGAS microbenchmark performance can be improved by an order of magnitude over an optimized textbook version and a practical implementation of the NAS benchmarks. We also measured performance improvements of up to 14% for the SPhot application from the ASC Sequoia Benchmark suite and up to 16% for the NAS FT benchmark.

Our work is an important step towards automatic dynamic and static optimization of communication patterns in large-scale computing. Detecting collective operations and replacing them with hardware-accelerated or optimized implementations is comparable to known techniques such as auto-vectorization. More advanced techniques that can be applied in addition to the optimizations proposed in this work, such as automated topology-aware graph tuning and transformations are being developed.

6. REFERENCES

[1] R. Alverson, D. Roweth, and L. Kaplan. The Gemini System Interconnect. In *18th IEEE Symp. on High Performance Interconnects*, 2010.

[2] B. Arimilli et al. The PERCS High-Performance Interconnect. In *Proc. of 18th Symp. on High-Performance Interconnects*, Aug. 2010.

[3] G. Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *Proc. of the 7th IEEE/ACM Intl. Symp. on Code Generation and Optimization*, pages 1–12, 2009.

[4] J. Bruck, C. T. Ho, S. Kipnis, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *6th ACM Symp. on Par. Alg. and Arch.*, pages 298–309, 1994.

[5] D. Callahan, B. L. Chamberlain, and H. P. Zima. The Cascade High Productivity Language. In *Intl. Workshop on High-Level Par. Progr. Models and Supportive Environments*, pages 52–60, April 2004.

[6] P. Charles, C. Grothoff, V. A. Saraswat, et al. X10: An

object-oriented approach to non-uniform cluster computing. In *Obj. Oriented Prog., Sys., Lang., and Apps.*, 2005.

[7] C. Coarfa et al. An evaluation of global address space languages: co-array fortran and unified parallel c. In *10th Symp. on Princ. and Practice of Par. Progr.*, 2005.

[8] J. J. Dongarra. Performance of various computers using standard linear equations software. *SIGARCH Comput. Archit. News*, 20:22–44, June 1992.

[9] T. El-Ghazawi et al. UPC Implementation of NAS Parallel Benchmark Kernels . Technical report, George Washington University, 2002.

[10] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35:4:1–4:14, July 2008.

[11] R. L. Graham et al. ConnectX-2 InfiniBand management queues: First investigation of the new support for network offloaded collective operations. *CCGrid*, pages 53–62, 2010.

[12] T. Hoefler and T. Schneider. Communication-Centric Optimizations by Dynamically Detecting Collective Operations. In *Proceedings of the 17th ACM PPoPP'12*, Feb. 2012. (poster paper).

[13] U. Khedker, A. Sanyal, and B. Karkare. *Data flow analysis: theory and practice*. Taylor and Francis, 2009.

[14] A. Knüpfer, D. Kranzlmüller, and W. E. Nagel. Detection of Collective MPI Operation Patterns. In *EuroPVM/MPI.*, volume 3241, pages 259–267. 2004.

[15] S. Kumar et al. The deep computing messaging Framework: Generalized scalable message passing on the BlueGene/P supercomputer. In *Proc. of the 22nd Intl. Conf. on Supercomputing*, pages 94–103, 2008.

[16] J. Larsson Träff, W. D. Gropp, and R. Thakur. Self-consistent mpi performance guidelines. *IEEE Trans. Parallel Distrib. Syst.*, 21:698–709, May 2010.

[17] H.-W. Loidl, P. W. Trinder, K. Hammond, A. Al Zain, and C. A. Baker-Finch. Semi-explicit parallel programming in a purely functional style: GpH. Dec. 2008.

[18] J. Mellor-Crummey, L. Adhianto, W. N. Scherer, III, and G. Jin. A new vision for coarray fortran. In *Proc. of the 3rd Conf. on PGAS Progr. Models*, PGAS '09, pages 5:1–5:9, 2009.

[19] R. Metzger and Z. Wen. *Automatic algorithm recognition and replacement: a new approach to program optimization*. MIT Press, Cambridge, MA, USA, 2000.

[20] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.2*, June 23rd 2009. www.mpi-forum.org.

[21] R. W. Numrich. A Co-Array Fortran Tutorial, 2001. Tutorial at ACM/IEEE Supercomputing 2001.

[22] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998.

[23] OpenMP Architecture Review Board. *OpenMP C and C++ Application Program Interface*, version 1.0 edition, 1998.

[24] S. Pakin. Receiver-initiated message passing over RDMA Networks. In *22nd IEEE Intl. Symp. on Par. and Distr. Proc., IPDPS 2008*, pages 1–12, 2008.

[25] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance Analysis of MPI Collective Operations. In *PMEO-PDS workshop*, 2005.

[26] R. Preissl et al. Detecting Patterns in MPI Communication Traces. In *Proc. of 37th Intl. Conf. on Par. Proc.*, 2008.

[27] R. Preissl et al. Using MPI Communication Patterns to Guide Source Code Transformations. In *8th Intl. Conf. on Comp. Science*, 2008.

[28] R. Preissl et al. Transforming MPI source code based on communication patterns. *Future Gener. Comput. Syst.*, 26:147–154, January 2010.

[29] R. E. Riesen et al. *The Portals 4.0 message passing interface*, April 2008. Sandia Natl. Labs, Tech. Rep. SAND2008-2639, April 2008.

[30] A. Rodrigues, K. Wheeler, P. M. Kogge, and K. D. Underwood. Fine-Grained Message Pipelining for Improved MPI Performance. In *Proc. of the 2006 IEEE Intl. Conf. on Cluster Comp.*

[31] P. Sanders, J. Speck, and J. L. Träff. Two-tree algorithms for full bandwidth broadcast, reduction and scan. *Parallel Comput.*, 35:581–594, December 2009.

[32] M. L. Scott and J. M. Mellor-Crummey. Fast, contention-free combining tree barriers for shared-memory multiprocessors. *Int. J. Parallel Program.*, 22:449–481, August 1994.

[33] UPC Consortium. UPC Language Specifications, v1.2. Technical report, Lawrence Berkeley National Laboratory, 2005. LBNL-59208.

[34] A. Vo et al. A scalable and distributed dynamic formal verifier for mpi programs. In *Proc. of Supercomputing 2010*, SC'10, pages 1–10.

[35] XcalableMP Specification Working Group. *Specification of XcalableMP, version 0.7a*, April 2010.

APPENDIX

A. DETAILED ALGORITHMS

The appendix lists all algorithms in pseudo-code to ensure reproducibility.

A.1 Matching Algorithm

This algorithm computes the matching set M from the sets R, S, and D. It assumes that the sets have been collected from all participating processes and are arranged in process-local sets R_i, S_i, and D_i such that $R = \bigcup_{i=1}^{p} R_i$, $S = \bigcup_{i=1}^{p} S_i$, and $D = \bigcup_{i=1}^{p} D_i$.

Algorithm 1: Matching Algorithm

Input: List of cDAG graphs G_i, grouped by process i: a set of receives (R_i), a set of sends (S_i), and a set of dependencies (D_i).
Output: Set of matches (M).

1 $MQ \leftarrow$ all independent nodes *// unsorted list of nodes*;
2 $\forall i : AQ_i \leftarrow \emptyset$ *// AQ is a sorted list of nodes*;
3 **while** $MQ \neq \emptyset$ **do**
4      $a \leftarrow pop(MQ)$;
5   *// invert type, swap peer and owner:*
    $y \leftarrow (\overline{a_{type}}, a_{peer}, a_{owner}, a_{size}, a_{tag})$;
6   **if** $y \in AQ_{a_{peer}}$ **then**
7          $AQ_{a_{peer}} \leftarrow AQ_{a_{peer}} \setminus \{y\}$;
8          $remove\_deps(a, D_{a_{owner}}, MQ)$;
9          $remove\_deps(y, D_{a_{peer}}, MQ)$;
10         $M \leftarrow M \cup \{(a, y)\}$;
11   **else**
12         $AQ_{a_{owner}} \leftarrow AQ_{a_{owner}} \cup \{a\}$;

The procedure $remove\_deps$ removed all dependencies to matched operations and adds the newly freed operations to the matching queue MQ.

Procedure remove_deps (*n, P, MQ*)

   Input: Node n, set of dependencies D, list MQ
1   **forall** $(n, x) \in D$ **do**
2      $D \leftarrow D \setminus \{(n, x)\}$;
3   **if** $\forall y : (y, x) \notin D$ **then** $MQ \leftarrow MQ \cup \{x\}$

A.2   Dataflow Propagation

The dataflow propagation algorithm works on the full DAG identified by the sets S, R, D, and M. It propagates the send buffers as sources to the receive buffers and uses splitting when incomplete buffers are sent. The result of the propagation algorithm is a static single transfer (SST) form (a set of SST assignments in form of tuples) where each memory location is written exactly once.

Algorithm 3: cDAG Dataflow Propagation Algorithm.

   Input: cDAG Graph as set of receives (R), set of sends (S), set of no-ops (N), set of matches ($M \subseteq S \times R$), and set of dependencies ($D \subseteq (\{S, R, N\} \times \{S, R, N\})$).
   Output: Updated set of receive dataflow tuples R (SST).
1   // recv and send tuples: (owner:start:size:peer:sstart);
2   set all $r \in R$, $s \in S$ to (owner:start:size:NIL:NIL);
3   $F_R \leftarrow \emptyset$; // finished receives;
4   $F_S \leftarrow \emptyset$; // finished sends;
5   $I_S \leftarrow \emptyset$; // independent sends;
6   find $I_S \subseteq S$ for $\forall s \in I_S \Rightarrow (x, s) \notin D$;
7   **while** $|I_S| \neq 0$ **do**
8   **foreach** $s \in I_S$ **do**
9   find $r \in R$ for $(s, r) \in M$; // or fail;
10   **if** $s_{sstart} = NIL$ **then**
11   update r dataflow tuple to (X:X:X:s.owner:s.start);
12   **else**
13   update r dataflow tuple to (X:X:X:s.peer:s.sstart);
14   remove $(r, *)$ from D;
15        $F_R \leftarrow F_R \cup \{r\}$; $F_S \leftarrow F_S \cup \{s\}$;
16   remove all no-ops $(n, *)$ from D for $(x, n) \notin D$;
17      $A_S \leftarrow \emptyset$; // active sends;
18   find A_S for $s \in A_S \Rightarrow s \in S \wedge (x, s) \notin D \wedge s \notin F_S$;
19      $I_S \leftarrow \emptyset$; // reset independent sends;
20   **foreach** $a \in A_S$ **do**
21        $O_R \leftarrow \emptyset$; // set of receive overlap regions;
22   insert each $r \in F_R$ and $r_{owner} = a_{owner}$ that overlaps with region (a_{start}, a_{size}) into O_R;
23        $S_S = \text{split}(O_R, a)$; // splits sends and matching receives;
24        $I_S \leftarrow I_S \cup S_S$;
25   **if** $(|F_R| \neq |R|) \wedge (|F_S| \neq |S|)$ **then**
26   return "unmatched Send/Recv or cycle!"
27   // collective detection can now be done for receive tuples;

A.2.1   Split

A split is performed when a part of a buffer that was received from a process is sent to another process. The split creates a new dataflow tuple (which will eventually be an SST assignment).

B.   COLLECTIVE DETECTION

The collectve detection algorithm finds collective operations in the SST form and removes all tuples that form the collective operation. It can then be replaced by a call to special hardware or an optimized algorithm.

Algorithm 4: Split.

   Input: set of overlapping receives O_R and single send s.
   Output: Set of split sends S_S, updated sets S, R, M, D as side-effect.
1   **if** $|O_R| = 1 \wedge r \in O_r : \wedge(r_{start}, r_{size}) = (s_{start}, s_{size})$ **then**
2      $s_{peer} \leftarrow r_{peer}$;
3      $s_{sstart} \leftarrow r_{sstart}$;
4   return $\{s\}$ // no split necessary (expected case)
5   sort O_R by start address;
6   find r for $(s, r) \in M$;
7   $riter \leftarrow r_{start}$;
8   dummyops $d_r \leftarrow d_s \leftarrow$ new cDAG no-op;
9   replace r with d_r in R, D; replace s with d_s in S, D;
10   remove (s, r) from M;
11   $siter \leftarrow s_{start}$;
12   **foreach** $ir \in O_R$ *(sorted)* **do**
13      $locrange \leftarrow \max(ir_{start}, siter)$ - siter;
14   **if** *locrange* > 0 **then**
15   // if we send something that wasn't received;
16   send $n_s \leftarrow (s_{owner}, siter, locrange, s_{owner}, siter)$;
17   recv $n_r \leftarrow (r_{owner}, riter, locrange, s_{owner}, siter)$;
18        $S \leftarrow S \cup \{n_s\}$; $S_S \leftarrow S_S \cup \{n_s\}$; $R \leftarrow R \cup \{n_r\}$;
19        $D \leftarrow D \cup \{(d_r, n_r)\}$; $D \leftarrow D \cup \{(n_s, d_s)\}$;
        $M \leftarrow M \cup \{(n_s, n_r)\}$;
20        $siter \leftarrow siter + locrange$; $riter \leftarrow riter + locrange$;
21      $remrange \leftarrow \min(ir_{start} + ir_{size}, s_{start} + s_{size})$ - siter;
22   send $n_s \leftarrow (s_{owner}, siter, remrange, ir_{sowner}, ir_{sstart})$;
23   recv $n_r \leftarrow (r_{owner}, riter, remrange, ir_{sowner}, ir_{sstart})$;
24      $S \leftarrow S \cup \{n_s\}$; $S_S \leftarrow S_S \cup \{n_s\}$; $R \leftarrow R \cup \{n_r\}$;
25      $D \leftarrow D \cup \{(d_r, n_r)\}$; $D \leftarrow D \cup \{(n_s, d_s)\}$;
      $M \leftarrow M \cup \{(n_s, n_r)\}$;
26      $siter \leftarrow siter + remrange$; $riter \leftarrow riter + remrange$;

Algorithm 5: Collective Detection.

   Input: All dataflow tuples R in STT form.
   Output: Detected collectives and remaining tuples in R.
1   **foreach** *size s in any tuple* $\in R$ **do**
2      $C \leftarrow \emptyset$; // detection set;
3   **foreach** $i = 0..p$ *and* $j = 0..p$ *and* $i \neq j$ **do**
4   find tuple $r \in R$ with size s, owner=i, and peer=j;
5   **if** r *exists* **then** $C \leftarrow C \cup \{r\}$
6   **if** $|C| = p(p - 1)$ **then**
7   **if** *input bufs for each peer are identical* **then** found alltoall of size s;
8   **else** found allgather of size s;
9        $R \leftarrow R \setminus C$; // remove tuples that formed collective;
10      $S \leftarrow \emptyset$; // scatter set;
11      $G \leftarrow \emptyset$; // gather set;
12   **foreach** r *is peer in any tuple of size s* **do**
13   **foreach** $i = 0..p$ *and* $i \neq r$ **do**
14   find tuple $r \in R$ with size s, owner=i, and peer=r;
15   **if** r *exists* **then** $S \leftarrow S \cup \{r\}$ find tuple $r \in R$ with size s, owner=r, and peer=i;
16   **if** r *exists* **then** $G \leftarrow G \cup \{r\}$
17   **if** $|S| = p - 1$ **then**
18   **if** *input buf for each owner is identical* **then** found bcast of size s;
19   **else** found scatter of size s;
20        $R \leftarrow R \setminus S$; // remove tuples that formed scatter;
21   **if** $|G| = p - 1$ **then**
22   found gather of size s;
23        $R \leftarrow R \setminus G$; // remove tuples that formed gather;