# Communication-Centric Optimizations by Dynamically Detecting Collective Operations

Torsten Hoefler
Department of Computer Science
University of Illinois at Urbana-Champaign
htor@illinois.edu

Timo Schneider
Department of Computer Science
Chemnitz University of Technology
timos@cs.tu-chemnitz.de

## Abstract

The steady increase of parallelism in high-performance computing platforms implies that communication will be most important in large-scale applications. In this work, we tackle the problem of transparent optimization of large-scale communication patterns using online compilation techniques. We utilize the Group Operation Assembly Language (GOAL), an abstract parallel dataflow definition language, to specify our transformations in a device-independent manner. We develop fast schemes that analyze dataflow and synchronization semantics in GOAL and detect if parts of the (or the whole) communication pattern express a known collective communication operation. The detection of collective operations allows us to replace the detected patterns with highly optimized algorithms or low-level hardware calls and thus improve performance significantly. Benchmark results suggest that our technique can lead to a performance improvement of orders of magnitude compared with various optimized algorithms written in Co-Array Fortran. Detecting collective operations also improves the programmability of parallel languages in that the user does not have to understand the detailed semantics of high-level communication operations in order to generate efficient and scalable code.

***Categories and Subject Descriptors*** D. Software [*D.1. PROGRAMMING TECHNIQUES*]: D.1.3 Concurrent Programming, Parallel Programming

***General Terms*** Performance, Languages

***Keywords*** Collective Communication, Parallel Compiler Optimization, Parallel Dataflow

## 1. Introduction

Most of today's large-scale parallel codes are implemented in a message-passing programming model using the Message Passing Interface (MPI) standard. MPI offers a large set of predefined communication patterns as *collective operations*. However, parallel programmers sometimes use suboptimal point-to-point algorithms to implement collective semantics. One reason may be that the programmer found a particuler point-to-point algorithm to be faster than the collective implementation on a particuler machine. In addition, other programming paradigms, such as PGAS languages [5, 7] or declarative parallel languages provide a conceptually simpler interface and rely on the compiler or the underlying communication layer for optimizations.

In this work, we describe a detection scheme that can recognize arbitrary collective semantics in parallel codes. This scheme can be used to "search and replace" suboptimal implementations of collective operations during runtime. We use a parallel dataflow analysis to identify the collective semantics and outline automatic transformation schemes to dynamically optimize such detected operations. We utilize GOAL [3] as an intermediate representation (IR) for basic communication operations such as send and receive and the data dependencies between those operations. We show a detailed examples how such IR representations can be extracted from a PGAS code.

### 1.1 Related Work

Previous works that attempted to detect collective operations performed the detection *post-mortem* by analyzing traces of the program run. Knüpfer et al. proposed a scheme to detect alltoall, scatter, gather, and broadcast by analyzing point-to-point patterns in message traces [4]. Kranzlmüller et al. investigate the detection of repetitive communication patterns in [6]. 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.

In this work, we propose a scheme that is guaranteed to detect all collective operations on the full process set in a communication schedule.

## 2. An Example Transformation

To demonstrate the applicability to PGAS languages we use an example from the MG code—the multigrid kernel from the NAS benchmark suite that was ported to Co-Array Fortran (CAF) [2]:

```
! omitted initializations for brevity
if (this_image().eq.iimage) then
  ibuf(1:n) = ii(1:n)
  call sync_all()
else
  call sync_all()
  ii(1:n) = ibuf(1:n)[iimage]
endif
call sync_all()
```

The compilation of the pCFG, similarly as described in [1], would be used to track the dataflow from one image to another. A dataflow analysis identifies the buffer `ibuf(1:n)` on process `iimage` as source and the buffers `ii(1:n)` on all images as destinations of the flow. The compiler can now simply insert send and re-


*PPoPP'12,* February 25–29, 2012, New Orleans, Louisiana, USA.
ACM 978-1-4503-1160-1/12/02.

ceive statements at the synchronization points (`sync_all()`). This would then result in the following GOAL code[1]:

```
! omitted initializations for brevity
if (this_image().eq.iimage) then
  ibuf(1:n) = ii(1:n)
endif
call GOAL_Create(g)
if (this_image().eq.iimage) then
  do dst=0, num_procs-1
    if (dst.ne.iimage) call GOAL_Send(g, ibuf, n*8, dst, ierr)
  end do
else
  call GOAL_Recv(g,ii,n*8,GOAL_ANY_SOURCE,ierr)
endif
call GOAL_Compile(g, sched)
```

The serial compiler transformation to emit the communication schedule is outside the scope of this work. We now discuss how to recognize the logical broadcast and replace the messaging schedule in the GOAL compile step accordingly. The CAF NAS codes include numerous occurrences of logical broadcast and allreduce calls similar to the one demonstrated above because the presented code is a natural way to express data movement in PGAS languages.

### 2.1 Dataflow Analysis

The `GOAL_Compile()` step above is executed during runtime and allows for dynamic optimizations, similar to the well-known commit phase in MPI Datatypes. A dataflow analysis computes the flows from all *original sends* to all *final receives*. An original send is a send operation that specifies local memory that has not been received from another process and a final receive is a receive that specifies user memory as destination.

In the following, we assume that the transformed example CAF code is running with six processes. The compile call is collective across all images (processes) and collects the complete global communication graph. The left part of Figure 1 shows the global communication graph for the linear broadcast as collected in compile. Each arrow represents a send from the source buffer to the destination buffer. The source buffer and process for each final destination is encoded as a tuple and a simple pattern matching on this tuple determines that the pattern (or a subset) is a known collective operation. In this case, all processes receive data from a single buffer at rank `iimage`, i.e., `iimage` is source of a broadcast operation. Our scheme detects all collective operations defined in MPI.

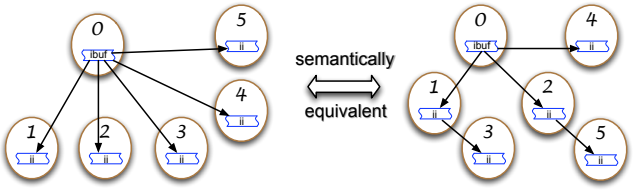


**Figure 1.** Example Schedules.

### 2.2 Schedule Transformation

The right part of Figure 1 shows a semantically equivalent communication graph in a tree shape. This graph was automatically transformed after the broadcast operation was detected by removing all communication edges of the broadcast and inserting a well-known binomial tree algorithm into the schedule.

[1] This code has been corrected after the original publication

### 2.3 Experimental Evaluation

We now compare the performance of the broadcast as shown in the CAF example code with an optimized implementation on a Cray XK6 system. We used the Cray 4.0.30 Programming Environment and the communicated data size was a single double value. Figure 2

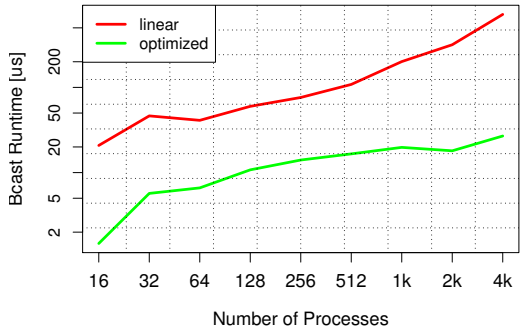


**Figure 2.** Naive vs. Optimized Performance in Cray XK6.

compares the performance of both implementations. The optimized implementation is up to a factor of 26 faster than the simple CAF implementation.

## 3. Discussion and Conclusion

We demonstrated novel dataflow techniques for the automatic and transparent detection and transformations of collective operations. These techniques can be used to detect arbitrary collective operations by pattern matching data-movement from source to destination buffers in arbitrary communication topologies. The techniques can easily extended to define semantic equivalence of other communication patterns and enable automatic transformations on the communication graph. For example, detecting collective operations on subsets of processes is a harder problem and left for future work.


### Acknowledgments

This work was supported by the DOE Office of Science, Advanced Scientific Computing Research, under award number DE-FC02-10ER26011, program manager Sonia Sachs.


## References


[1] 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*, CGO '09, pages 1–12, 2009.

[2] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda. An evaluation of global address space languages: co-array fortran and unified parallel c. In *Proc. of the tenth ACM SIGPLAN Symp. on Princ. and Practice of Par. Progr.*, PPoPP '05, pages 36–47. ACM, 2005.

[3] T. Hoefler, C. Siebert, and A. Lumsdaine. Group Operation Assembly Language - A Flexible Way to Express Collective Communication. In *38th Intl. Conf. on Par. Proc., ICPP'09*, 2009.

[4] A. Knüpfer, D. Kranzlmüller, and W. E. Nagel. Detection of Collective MPI Operation Patterns. In *Proc. of EuroPVM/MPI'04*, volume 3241 of *LNCS*, pages 259–267. 2004.

[5] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17:1–31, August 1998. ISSN 1061-7264.

[6] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. d. Supinski, and D. J. Quinlan. Detecting Patterns in MPI Communication Traces. In *Proc. of the 37th Intl. Conf. on Par. Proc., ICPP'08*, pages 230–237, 2008.

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