HAL
open science

Maximizing Communication Overlap with Dynamic Program Analysis

Emmanuelle Saillard, Koushik Sen, Wim Lavrijsen, Costin Iancu

To cite this version:

Emmanuelle Saillard, Koushik Sen, Wim Lavrijsen, Costin Iancu. Maximizing Communication Overlap with Dynamic Program Analysis. International Conference on High Performance Computing in Asia-Pacific Region, Jan 2018, Tokyo, Japan. hal-01937407

HAL Id: hal-01937407

https://inria.hal.science/hal-01937407v1

Submitted on 28 Nov 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Maximizing Communication Overlap with Dynamic Program Analysis

Emmanuelle Saillard
INRIA / LaBRI, University of Bordeaux, INP, France
emmanuelle.saillard@inria.fr

Koushik Sen
University of California, Berkeley, USA
ksen@cs.berkeley.edu

Wim Lavrijsen
Lawrence Berkeley National Laboratory, USA
wlavrijsen@lbl.gov

Costin Iancu
Lawrence Berkeley National Laboratory, USA
cciancu@lbl.gov

ABSTRACT

We present a dynamic program analysis approach to optimize communication overlap in scientific applications. Our tool instruments the code to generate a trace of the application's memory and synchronization behavior. An offline analysis determines the program optimal points for maximal overlap when considering several programming constructs: nonblocking one-sided communication operations, nonblocking collectives and bespoke synchronization patterns and operations. Feedback about possible transformations is presented to the user and the tool can perform the directed transformations, which are supported by a lightweight runtime. The value of our approach comes from: 1) the ability to optimize across boundaries of software modules or libraries, while specializing for the intrinsics of the underlying communication runtime; and 2) providing upper bounds on the expected performance improvements after communication optimizations. We have reduced the time spent in communication by as much as 64% for several applications that were already aggressively optimized for overlap; this indicates that manual optimizations leave untapped performance. Although demonstrated mainly for the UPC programming language, the methodology can be easily adapted to any other communication and synchronization API.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**; • **Software and its engineering** → *Software notations and tools*;

KEYWORDS

Dynamic Analysis, Optimization, One-sided communication, UPC

1 INTRODUCTION

Communication optimizations are important for the scalability of scientific applications. While for MPI codes most transformations optimize for bandwidth, the advent of one-sided communication shifted the emphasis from bandwidth related to latency hiding [13, 30] optimizations. These are performed using nonblocking primitives and overlapping communication with other independent work. Overlap is currently attained with manual transformations [13, 30, 31] at the application level or by compilers [18, 25] using static program analysis. Our work was motivated by the insight that current latency hiding optimization techniques may leave performance on the table. Due to parallel data domain decomposition and software engineering constraints, or because of practical limitations of static analysis, most transformations are localized to a single module. Furthermore, it is hard to retrofit these transformations to take advantage of novel runtime Application Programming Interfaces (API) or functionality at a lower level of abstraction.

We are interested in two types of transformations: 1) providing maximal overlap with respect to the data dependencies within a single thread of execution; and 2) providing maximal overlap with respect to multi-threaded data dependencies. The former is designed to augment the optimizations performed by developers manually at the application level, usually within the scope of a single function or library. The transformation tries to delay the completion of a nonblocking communication operation right before the first `load/store` data dependence, subject to the memory consistency imposed by synchronization operations. The latter, if at all, has been exploited in codes only in a trivial manner due to logical and functional "semantic" challenges. We provide optimizations to delay completion of communication across collective operations or inside the implementation of the collectives themselves. Another flavor of optimization delays completion of communication across bespoke[1] application level synchronization primitives and patterns, such as Producer-Consumer. We provide a program transformation tool to explore the space of these optimizations. Since we employ a dynamic analysis approach, the tool can reason about the entire execution of a program that uses multiple programming languages (C, C++, Fortran, UPC) and across multiple layers of abstraction for synchronization operations (UPC [2], GASNet [1], application level). Note that these combinations of languages and layers with different semantics pose challenges to existing optimizers based on static program analysis.

We use the LLVM [8] compiler infrastructure to instrument all data access operations in the code, `load/store` instructions and data transfer intrinsics. This is augmented with a link time interposition layer to trace communication and synchronization operations.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2018 Copyright held by the owner/author(s).

[1] Arbitrary user level implementations of synchronization and exchange operations.

In the first stage we run the program and collect memory, communication and synchronization trace files. An offline phase examines the data dependencies, runs the specialized analyses and determines the opportunity for each transformation. The main analysis goal is determining the furthest program point where the completion of a particular communication operation can be delayed to. The results of this stage consist of suggested program points and their associated transformations. These are presented to the application developers, followed by a last stage that performs the transformations they deemed correct. The transformations rewrite part of the original call sequence and insert new communication completion management code at the proper program points. These operations are supported by a lightweight runtime that manages and associates outstanding communication operations.

We evaluate the tool on several benchmarks written in the Unified Parallel C (UPC) programming language, running on top of the GASNet communication library. With this combination we cover a large portion of the design spectrum for one-sided communication paradigms and languages. *As the baseline for our comparison, we start with overlap optimized versions of all codes, each considered as the state-of-the-art in optimization for that particular code.* The benchmarks themselves cover multiple code optimization and design paradigms: regular domain decomposition, all-to-all communication, irregular domain decomposition combined with either global synchronization or hand optimized synchronization patterns. The evaluation is conducted on two systems with completely different communication characteristics: Cray Aries and InfiniBand.

When starting with unoptimized codes, we can match the performance of manual optimizations, and we observed improvements as high as 58% at 2,048 cores. When starting with well optimized codes, the transformation to provide maximal overlap with respect to single thread dependencies leads to improvements in communication time up to 26%, which translate into end-to-end execution time improvement of up to 7%. Attaining maximal overlap is also required as an enabler for the transformations across synchronization operations and across-threads dependencies, which are challenging to developers. We have observed improvements in communication time as high as 64% when running already optimized code at concurrency up to 2,048 cores. This translated into another 6% end-to-end performance improvement. Another lesson learned the hard way during this effort is that in order to attain these improvements we had to specialize the critical path inside the GASNet communication runtime itself, using the same dynamic analysis approach. This is further explained in Section 5 and 7 and illustrates again the scenario when internal or lower level runtime APIs mismatch the semantics of the higher application level interfaces.

Our tool can be used in practice in multiple ways: 1) as a standalone optimizer it is able to improve code performance beyond the abilities of any developer; 2) it provides a fast way to prototype and specialize transformations to the idiosyncrasies of networks and compositions of runtime APIs; and 3) it provides upper bounds on potential performance improvements for manual communication optimizations.

Overall, we believe that the methodology we employ can be easily generalized and ported to other one-sided communication paradigms or languages[2]: OpenSHMEM [32], MPI 3.0 RMA [28], X10 [3], Chapel [7]. To our knowledge, the optimizations to extend overlap across nonblocking barriers or bespoke optimization patterns have not been attempted yet in codes and we provide quantitative evidence of their efficacy. The transformation to delay completion inside the implementation of barrier operations and the specialization across multiple independent runtime layers showcase the ability of dynamic analyses to perform optimizations across multiple independent layers of abstraction within complex software stacks. As these are usually beyond developer ability, we believe this last line of research to hold great promise.

2 BACKGROUND AND MOTIVATION

One-sided communication exploits networking hardware support for Remote Direct Memory Access (RDMA). Library based approaches (uGNI/DMAPP [11], IBVerbs, OpenSHMEM [32], MPI RMA [28]) expose a communication interface for *Put/Get* primitives, augmented with *(try_)wait* completion operations, and with synchronizations such as *barriers*, *fences*, etc. Languages (UPC [2], X10 [3], Chapel [7]) add support in the type system for building a Global Address Space and memory consistency models.

Our target language is Unified Parallel C (UPC), a good representative of the family of Global Address Space (GAS) languages. In UPC, memory is divided into thread private and global spaces. The global address space is further partitioned logically into portions that are local to a task. Thread private data can be accessed only through proper C pointer data types. All global data can be accessed through "pointers to shared" type extensions. This implies that communication optimizations need to reason about all memory accesses in the program.The language implements Single Program Multiple Data (SPMD) parallelism.

Communication in UPC is one-sided, and can be either implicit or explicit. Implicit communication happens whenever variables of type "pointer to shared" appear in an expression. Optimizing implicit communication in UPC, and all other GAS languages, is outside programmer control and requires compiler support. The language also exposes blocking *Put/Get* primitives for explicit communication, e.g. `upc_memput(shared void *dst, void *src, size_t n)`, where `*dst` encodes both a thread identifier and a memory address. A blocking operation returns only when the data has arrived and has been written to the memory of the remote node. Both implicit and explicit blocking communication are subject to the language memory consistency model and can be optimized by compilers. These features are common to other GAS languages such as X10 and Chapel, but it appears that existing production compilers do not perform any aggressive communication optimizations.

Communication Overlap: To facilitate manual optimizations, library extensions for nonblocking one-sided communication have been adopted. These take the form `{init_put/get(); ... wait();}`, where `init` initializes a nonblocking operation in hardware and `wait` checks for its completion. Nonblocking one-sided communication opens the possibility of latency hiding optimizations and communication overlap. In the optimal case, the application never waits for the completion of any communication operation: it executes

[2]Incidentally, all have an implementation running on top of GASNet.

independent code after `init` that takes longer than the duration of communication. Thus, it is key to delay both communication completion and synchronization until the furthest logically correct point in the program. For the explicit primitives mostly this is manually attained by interposing independent code between the `init-wait` pair of calls. Previous work described compiler optimizations for overlapping implicit communication, but to our knowledge these optimizations are not deployed in production compilers, because of engineering challenges. Furthermore, most are local in scope, both in terms of the domain decomposition and code module, and are encapsulated in specialized transfer functions for each domain boundary. This is further detailed in Section 3. Localization and modularity results in conservative optimizations. For example, in complicated code bases it is hard for developers to associate synchronization operations (e.g. barriers) exactly with the transfers whose consistency they are introduced to maintain. This results in conservative design decisions and possible loss of communication overlap. We refer to this as logical semantic mismatch. Our tool can answer the question whether manual optimizations are sufficient and how much performance is untapped due to inability to optimize across multiple software layers.

Synchronization Optimizations: The UPC language provides synchronization primitives to explicitly enforce inter-task data dependence. Operations like `barriers` provide collective synchronization; in UPC these are either blocking or non-blocking. Nonblocking collective operations [23] are now available for MPI and are currently discussed for adoption in the OpenSHMEM standard.

Unlike two sided MPI Send/Recv communication, one-sided transfers decouple data transmission from inter-thread synchronization. Only the sender can verify the completion of the RDMA operation and a further, programmatic, synchronization between sender and receiver is needed as a means to signal this completion.

Thus, one-sided paradigms tend to provide a flexible set of point-to-point synchronization primitives such as locks, semaphores, atomics etc. Some are language primitives, some are library extensions, some are user defined. Primitives such as locks are supported by the UPC language and compilers could conceivably optimize around them. To allow for a richer set of synchronization patterns, library extensions have been introduced. One example is the Berkeley UPC [6] extension which introduces semaphores to build custom point-to-point synchronization patterns. The semaphores are plain counting semaphores with a `signal() <-> wait()` interface and can be used to efficiently build complex Producer-Consumer relationships. While nonblocking communication primitives are standardized in the UPC specification as a runtime library API, these third party synchronization primitives may be redefined with ad-hoc semantics across implementations.

To our knowledge, very little work has been performed in developing optimizations, either manual or automated, for non-blocking collectives. Furthermore, we are not aware of any automated optimization techniques for applications containing ad-hoc synchronization patterns. Our tool can answer the question whether these can be deployed in existing applications without restructuring and if they improve performance.

Breaking Abstractions: Usually (P)GAS languages have multiple runtime implementations. Vendor compilers such as IBM X10 or Cray Chapel have runtime implementations targeting directly the system native communication API, PAMI and DMAPP respectively. They also provide MPI or GASNet based implementations for portability. The UPC compiler we use runs on top of GASNet. GASNet is a lightweight portability layer across all the low level system communication APIs used in supercomputers. It is the case that any low level communication API provides a much richer and flexible set of interfaces than exposed at the language or application level. We refer to this as a functional semantic mismatch. Our tool provides a way to exploit this low level functionality directly at the application level without non-portable changes to the source code.

In the rest of paper, for brevity we will use the terms[3] local dependence and global or inter-thread dependence. Local refers to any data dependence between communication (e.g. Put), local memory accesses (e.g. store) and inter-thread synchronization operations. In the "local" view we do not distinguish between the targets of synchronization and any synchronization operation introduces a data dependence. This is similar to what static program analysis can accomplish. In the "global" view, we take into account the program dynamic behavior and add dependencies only between communication and synchronization operations with the same task. To address the perceived limitations of existing approaches we develop an optimization methodology to:

- Provide maximal overlap with respect to local and inter-thread dependencies and synchronization.
- Perform global optimizations across modules at same level of abstraction and bridge any logical semantic gap.
- Perform global optimizations across modules at different levels of abstraction and bridge functional semantic gaps.
- Allow for easy experimentation with new primitives (e.g. nonblocking collectives) and transformations.

3 TRANSFORMATIONS TO MAXIMIZE OVERLAP

We will use the simplified 2-D stencil example in Figure 1 to illustrate the target transformations. In each computation step boundary data are exchanged with `pack_send_boundary`, then tasks synchronize (`5:`) and proceed to update the local data domain at `6:`. In some codes communication initiated in `send_domain` is blocking and completed in place. This was the case in our HipMer benchmark. Most common optimization is to overlap the transmission with the boundary processing and have all communication be completed at `3:` before returning from `pack_send_boundary`. This was the case in most of our benchmarks optimized by third party developers: NAS BT, SP, IS, FT and miniGMG. More ambitious optimizations may delay and complete all communication initiated in any `send_subdomain` call at the program point `4:`. As this requires tracking per operation state across procedure or library boundaries, it is seldom encountered in codes. We have performed this manual optimization in miniGMG.

[3] Another taxonomy can be based on static and dynamic information.

```
  pack_send_boundary(neighbor) {
    foreach subdomain(neighbor)
1:      pack_subdomain() //local work
2:      send_subdomain() // init, wait pair
3: // can wait on all puts for this neighbor
}
...
{   foreach neighbor (E, W, N, S)
        pack_send_boundary(neighbor)
4: // can wait for all puts for all neighbors
5:  sync_with_neighbors(E, W, N, S) //semaphore, barrier ..
6:  update_domain()}
```

Figure 1: Overlap example.

From this discussion, it becomes apparent that most codes are likely to provide room for more overlap optimizations: the ideal behavior has the completion for any communication operation delayed until the data involved is touched locally. For example, completion for a *Put* operation can be delayed until the first store in the execution that overwrites any bytes involved in the payload. The goal of our first optimization is to transform codes for maximal overlap and delay communication completion based on the dynamic data dependencies. For this we need to reason about the data accesses during runtime, combined with a framework to automatically track outstanding communication operations and associate their completion with the correct program scope or point in the control flow graph.

3.1 Complex Overlap Transformations

1: Although cumbersome, sometimes developers may be able to attain maximal overlap with respect to local dependencies. Our next transformations are likely outside the reach of manual optimization, even in small codes. Going back to Figure 1, the insight is that completion of operations initiated in `send_domain` may be delayed either within `sync_with_neighbors`, or across it and until within `update_domain`. For some algorithms, `sync_with_neighbors` can be a proper barrier operation. Here the optimization requires delaying completion of communication within the implementation of a runtime library function. Alternately, it can implement ad-hoc synchronization algorithms. Our transformations can handle both scenarios.

2: Next, consider the case where threads synchronize using barriers. GASNet provides support for split-phase[4] barriers. A split-phase barrier comprises two phases: *barrier_notify* and *barrier_wait*. In the former, a thread notifies others that it is ready for the barrier. In the latter, a thread waits for others to be ready. Our optimizer always treats a `upc_barrier` as its constituents and attempts several transformations:

- Move notify and wait operations as far apart as possible, subject to data dependencies. This overlaps barrier latency with other work and illustrates a scenario where code inside library modules is split and mixed with application code.
- Move completions of communication that precedes barriers in between the notify/wait calls. This enhances the communication overlap and illustrates a transformation where application level code is moved inside runtime libraries.
- Move completion of communication operations across multiple barriers, subject to data dependencies.

The first two transformations are examples of optimizations that address functional semantic mismatches between layers of the software stack, while the last addresses logical mismatches.

3: For codes using one-sided communication, one common [24, 31] transformation for performance is replacing collective or group based synchronization with point-to-point communication and synchronization. In this case, the transformed code ends up implementing a Producer-Consumer pattern using ad-hoc synchronization primitives. This can be abstracted as follows. On the left hand side we present the base case, while on the right hand side we present the hand optimized pattern, where all communication and synchronization operations are grouped together. This pattern which we implemented for miniGMG occurs also in Particle-In-Cell codes, as well as the code described in [31].

```
wait_one(empty)                          wait_all(empty)
//produce                  OR            //produce
put ..                                   put ..
put ..                                   put ..
..                                       ..
signal_one(full)                         signal_all(full)
```

The transformation we perform in this case is moving the completion of communication operations across these ad-hoc synchronization primitives, subject to data dependencies. In our study we selected codes using the Berkeley UPC (BUPC) semaphore library extension. In order to support these transformations, the tool-chain needs to provide support for marking and reasoning about "arbitrary" portions of code.

4 IMPLEMENTATION OF DYNAMIC ANALYSIS TOOLCHAIN

We implement a toolchain for guided code transformations comprised of three components. First we do compile-time instrumentation of the code to collect data and control dependence information about the executing program. The output of a run is recorded in per thread trace files. A second pass merges the trace files and performs the optimizations. As maximal overlap with respect to local thread data dependence is an enabler for the other optimizations, it is performed first. For each operation of interest, e.g. `init, wait, barrier_notify`, this stage associates to the original location in the source file with the list of source locations across all monitored threads where the operation can be moved to. The output of this stage is presented to the application developers for validation. Finally, the third component can automatically generate the code for the transformations desired by the application developers. The input to this stage is a list of associations between source locations where the operations of interest should be moved to.

Dynamic analysis approaches cannot guarantee soundness of results across all program inputs and concurrencies. The transformations proposed by our tool are correct across the observed inputs and for safety we require the developer to instruct the tool which

[4]This permeates up to the UPC language level so our approach has been trivially extended to handle codes with explicit split-phase barriers.

transformations to perform. The trace analysis pass can be also used to assess the correctness of any transformation.

4.1 Trace File Generation

We use the LLVM [8] compiler infrastructure for instrumentation. For each bitcode file we instrument the LLVM IR nodes of interest, which are either instructions or a priori selected function calls. We track any instruction accessing memory (e.g. load or store) and intrinsic synchronization operations. We can also capture synchronization functions related to the programming model or the underlying communication runtime, for the cases where we decide they have generic enough semantics. Examples include SPMD barriers, fence or primitives for point-to-point synchronization using semaphores. Each operation of interest is surrounded by tracing calls. This allows us to not only track state, but to split operations into their components. For example for explicitly blocking communication, e.g. `upc_memput`, we need to generate the `init_put`, `wait` components. Similarly barriers are split into notify/wait pairs in the trace file. The tool allows to easily extend the set of tracked operations using link time interposition.

As tracing introduces significant runtime overhead, we use optimization passes to reduce the number of load and store instructions that are instrumented. We use the alias analysis as well as heuristics related to the UPC programming model. For example, automatic variables can be ignored, or we try to ensure that only loads and stores on local memory addresses involved in a data transfer (e.g. Put) function are recorded. Note that this last goal motivated our design decision to instrument certain functions at compile time, rather than having any runtime call instrumented though link time interposition.

4.2 Trace File Analysis

An analysis pass processes the trace files to determine the optimal placement for the operations of interest, for the three target transformations. In order to assess the profitability of a transformation, the analysis also computes metrics based on the number of tracked operations observed between two events of interest. For brevity, we give only an intuitive description of this stage.

Maximal Overlap Analysis. Consider a snippet of the trace containing the events e_i e_{i+1} ... e_j. Assume event e_i is a *Put* and the next event e_{i+1} corresponds to its completion as presented in Figure 2. Assume that e_j is the first event after e_i which requires that communication initiated in e_i is completed in order to maintain consistency. For Puts, these events are illustrated in Table 1: i) the transfer must complete before the destination memory is read and the source memory is written; ii) the transfer must complete before a synchronization operation. The analysis follows `use-def` chains across synchronization operations. Some applications we tested contain redundant barriers and the analysis can delay communication completion before the last redundant barrier.

Optimizing Across Barriers and Bespoke Synchronization. Consider a barrier operation and its `notify/wait` components. Any `notify` can be moved higher after the last read/write on a shared variable (PUT, GET, CPY, LOAD or STORE) or a `wait`. Similarly, any `wait` can be delayed before a read/write on a shared variable or a `notify`. This transformation is presented Figure 3. We also take into account the control flow of the program and rules to conform to the UPC language specification, which for example prohibits nested barriers. For instance, the following sequence: `notify notify wait wait` is illegal. We also provide an analysis to move communication completion across barriers while ensuring there are no data races. This occurs when the code contains redundant barriers or it is over-synchronized. In this case to account for control divergence, an `init` call is completed after the minimum number of barriers observed across all its occurrences in all traces. This is a conservative heuristic that simplifies the need for complex control flow information. Custom synchronization patterns are optimized in a similar manner, our analysis for semaphores is omitted for brevity. The gist of the optimization is that we associate all communication and synchronization operations with the same rank/thread and then delay completion of communication across all other independent synchronization, subject to data dependencies.

Table 1: Event association: sa = **shared address,** la = **local address,** t = **time stamp,** tid = **thread,** loc = **source location**

e_i	e_j
$PUT(sa, la_i, t_{e_i}, tid_{e_i}, loc_{e_i})$	$GET(la_i, *, t_{e_j}, tid_{e_j}, loc_{e_j})$, $STORE(la_i, t_{e_j}, tid_{e_j}, loc_{e_j})$, $CPY/GET(*, sa, t_{e_j}, tid_{e_j}, loc_{e_j})$, $LOAD(sa, t_{e_j}, tid_{e_j}, loc_{e_j})$ $BARRIER(t_{e_j}, tid_{e_j}, loc_{e_j})$ or $SYNCHRO(t_{e_j}, tid_{e_j}, loc_{e_j})$
$GET(la_i, sa, t_{e_i}, tid_{e_i}, loc_{e_i})$	$PUT/CPY(sa, *, t_{e_j}, tid_{e_j}, loc_{e_j})$, $STORE(sa, t_{e_j}, tid_{e_j}, loc_{e_j})$, $LOAD(la_i, t_{e_j}, tid_{e_j}, loc_{e_j})$, $PUT(*, la_i, t_{e_j}, tid_{e_j}, loc_{e_j})$, $BARRIER(t_{e_j}, tid_{e_j}, loc_{e_j})$ or $SYNCHRO(t_{e_j}, tid_{e_j}, loc_{e_j})$
$CPY(sa, sa', t_{e_i}, tid_{e_i}, loc_{e_i})$	$PUT/CPY(sa', *, t_{e_j}, tid_{e_j}, loc_{e_j})$, $STORE(sa', t_{e_j}, tid_{e_j}, loc_{e_j})$, $CPY/GET(*, sa, t_{e_j}, tid_{e_j}, loc_{e_j})$, $LOAD(sa, t_{e_j}, tid_{e_j}, loc_{e_j})$, $BARRIER(t_{e_j}, tid_{e_j}, loc_{e_j})$ or $SYNCHRO(t_{e_j}, tid_{e_j}, loc_{e_j})$.

Before optimization	**After optimization**
$e_i = PUT(\ldots)$	e_i
$e_{i+1} = SYNC(e_i, t_{e_i}, tid_{e_i}, loc_{e_i})$	⋮
⋮	e_{i+1}
e_j (see Table I)	e_j

Figure 2: Code transformation for maximal overlap with respect to local dependencies.

5 CODE GENERATION AND RUNTIME SUPPORT

The analysis output is presented to developers, which can direct a code generator to perform the code transformations. We try to minimize the changes to the original code, and also try to make it easy to revert to the non-optimized behavior at runtime. This makes

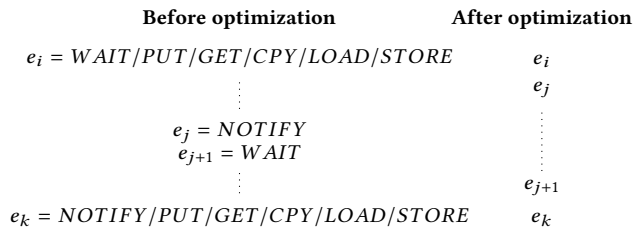

Figure 3: Code transformation for barriers optimization.

debugging easier, as well as being able to revert non profitable optimizations during runtime.

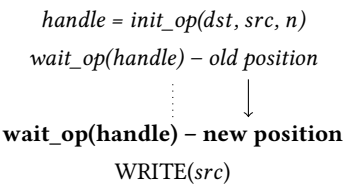

Figure 4: Communication overlap code generation. *wait_op* **needs to move from the original source position to the new position before** *WRITE*.

For example, consider the snippet in Figure 4. Communication completion can be moved right before the *WRITE*. We opted for a strategy where we leave the original call in place, transform it into a *noop* at runtime and insert a new call before the *WRITE* in order to complete the target communication operation. Thus we can control at runtime in what position completion actually occurs.

Calls to our runtime API are compiled into the client code into the positions specified by the analysis stage. Calls are also placed after each one-sided operation, and before each synchronization operation (barrier, fence, etc.) and load/stores deemed relevant by the analysis. After any one-sided operation we call `internalize` which takes a reference to the returned handle, a target description, a barrier skip count, and context markers (source location). The latter two come as input from the analysis. The call internalizes the handle, by keeping a local copy and setting the original variable to "complete." This way, the original `wait` calls become *noops*. For performance reasons, within `internalize` we implement flow control and complete a machine dependent number of operations in one step, if necessary.

The remaining API calls are for completion of outstanding handles, with specializations for barriers and the target thread(s) of a synchronization operation. The barrier specialization completes only those handles that have skipped the requested number of redundant barriers as determined by the analysis. The target specialization, which takes a thread id as an argument, finishes all communication to that particular rank. Synchronization before load/stores takes a marker from the analysis as input and completes the handles that were internalized with that marker. A final call completes any outstanding communication and is used for fences.

To support the specializations, handles are stored in a buffer per target in the runtime. Each buffer is circular and new handles are stored at a position relative to the current index, equal to the number of redundant barriers that need to be skipped. On each barrier, the current index is increased, and all handles that have come due are completed. The circular buffer is processed en-bloc on a target synchronization, as are all buffers of all targets on a full synchronization. Markers are associated to handles and are using double referencing.

The runtime takes maximum advantage of the increased overlap: it first checks completion of all relevant handles, collecting only the unfinished ones. If no unfinished handles remain, polling is avoided. Otherwise, the network card is polled until the remaining handles complete.

6 EXPERIMENTAL RESULTS

We experiment on two supercomputers with different functionality and performance characteristics: Edison and Shepard. Edison (Cray-XC30) is deployed at NERSC [5]. Its nodes contain two 12-core Ivy Bridge processors running at 2.4 GHz and are connected using the Aries [10] network. Every four nodes are connected to one Aries network interface chip. The Aries chips form a 3-rank dragonfly network. Shepard [9] is deployed at Sandia National Laboratory [9] and its nodes have two 16-core Haswell processors at 2.3GHz. The interconnect is Mellanox InfiniBand FDR. We use LLVM 3.7.1 and Berkeley Unified Parallel C, v. 2.22.0 [14]. Experimental results are averaged from five runs.

6.1 Benchmarks Description

We select applications with different communication characteristics to showcase the impact of our approach. miniGMG [37] is a grid based code that performs nearest (6 or 26) neighbor communication on a 3-D processor grid. HipMer [19] implements a De Novo Genome Assembler and exhibits highly irregular message patterns that vary both the number and the destination of messages in each communication step. We also evaluate our approach on the UPC NAS [12] Parallel Benchmarks v2.3, using the BT, SP, IS, FT and LU applications. For miniGMG and the NAS parallel versions we have implementations using blocking communication written in stock UPC, as well as hand optimized versions with communication overlap using non-blocking extensions. HipMer uses only blocking communication. All benchmarks were developed and optimized by third parties and obtained from public domain. miniGMG is written in UPC+OpenMP and can be configured as SPMD or hybrid parallelism. HipMer is written in UPC and runs only in SPMD mode. For brevity we do not describe the NAS Parallel Benchmarks, for details please consult [12].

miniGMG. Multigrid is a linear-time approach for solving elliptic PDEs expressed as a system of linear equations ($Lu^h = f^h$). miniGMG is developed to proxy the geometric multigrid solves within the Adaptive Mesh Refinement (AMR) MG applications [37]. Geometric multigrid (GMG) is a specialization of multigrid in which the PDE is discretized on a structured grid. At each successively coarser level during the recursion, the computational requirements drop by factors of 8×, but the communication volume falls only by 4×. miniGMG is designed mostly for weak scaling. We evaluate two very different algorithmic implementations, `Original` denotes the traditional one where each task communicates with six neighbors,

`Comm. avoiding` denotes a communication avoiding algorithm where tasks communicate with 26 neighbors. Each algorithm has two versions, each with two implementations, blocking or optimized for overlap. The aggregate (AGG) version is optimized for bandwidth and sends only one message per peer. The immediate (IMM) version is written in the one-sided spirit and sends a message for any chunk of boundary data. All these six variants use barriers for inter-thread communication. In addition, in collaboration with the miniGMG authors, we have developed another highly optimized version of the benchmark that uses double-buffering for boundaries in order to reduce the synchronization requirements in half and replaces barriers with bespoke point-to-point Producer-Consumer synchronization between threads. In this version, denoted by PCDB, we use the BUPC semaphore extensions.

Meraculous Genome Assembler. De novo genome assembly reconstructs a genome from a set of multiple, overlapping and potentially erroneous sub-sequences or reads. Georganas et al [19] present a parallel implementation with the best scalability to date. At the core of their implementation is a parallel de Bruijn graph construction and traversal algorithm, which uses a distributed hash table. Most of the communication is spent in hash table operations and the execution exhibits a random communication pattern. For performance reasons, during construction eight byte hashing operations are aggregated into larger messages, whose granularity is configurable. This application captures the characteristics of many other large scale graph algorithms. HipMer is a snapshot of Meraculous used in system procurements by the NERSC supercomputer center.

6.2 Maximal Overlap Results

In this section, `BLOCKING` denotes the version of a benchmark that uses blocking communication, `NONBLOCKING` denotes the hand optimized version, if any available, and `OPTIMIZED` denotes the version optimized by our tool.

Case 1: When starting with a blocking implementation the tool always found potential for optimization across all benchmarks. The improvements in end-to-end application performance are good and match or exceed the performance of hand optimized codes. The optimizations always reduce significantly the time spent in communication by the application. These trends are illustrated in Figures 5, 6 and 7. Note that for HipMer we have available only the blocking implementation and we were able to reduce communication time by 40% with 1,040 threads.

Case 2: When starting with a hand optimized implementation the tool found new optimizations for BT, SP, LU and miniGMG. All these applications were optimized using a similar strategy, which incidentally occurs in most SPMD optimized codes. The code is written such that each task performs in one logical step domain boundary exchanges with all its neighbors, followed by synchronization with all neighbors. Any operations for a single boundary are overlapped for latency hiding, but due to code complexity exchanges for multiple boundaries are not. The tool was able to overlap all communications across all boundaries and reduce both time spent in communication and end-to-end execution. For example, when looking at BT communication time in Figure 6, our optimized version takes less time from 576 threads and is faster by 26% with 784 threads. When looking at end-to-end execution in Figure 5 we observe 7% when using 900 cores.

When examining all the results, one trend becomes notable. At high concurrency our optimizations always help. At low concurrency, providing maximal overlap sometimes hurts performance. With strong scaling, messages become large at low concurrency. As indicated by Luo et al [27] on the networks we use (Aries and InfiniBand) issuing simultaneously a large number of large messages degrades performance.

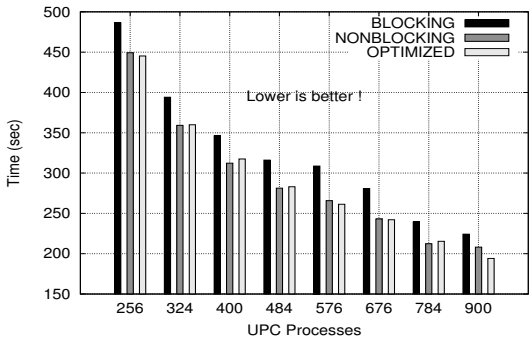

Figure 5: Execution-time for blocking, nonblocking and optimized versions of NAS BT, Class D (strong scaling). Results obtained on Edison.

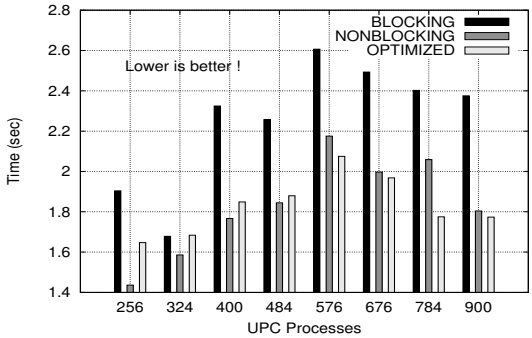

Figure 6: Time spent in communication for NAS BT, Class D (strong scaling). Results obtained on Edison.

6.3 Optimizing Across Barriers and Bespoke Synchronization

As all benchmarks use barrier synchronization, our tool was able to find places to exploit the GASNet split-phase barrier. This is an example of cross module optimization spanning levels of abstraction that is beyond the reach of application developers. In BT, SP, FT, IS and LU the transformations did not lead to performance improvements, as the analysis found this potential in the initialization, tear-down and performance instrumentation code. In all benchmarks we found multiple opportunities to overlap `malloc`

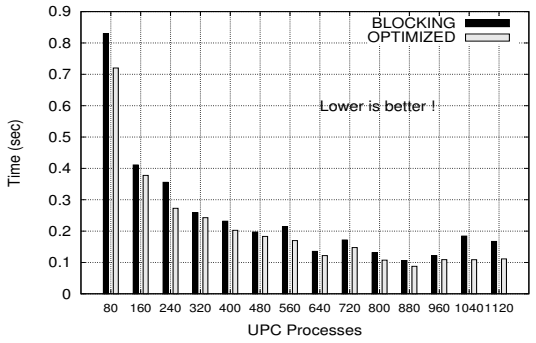

Figure 7: Time spent in communication for blocking and optimized versions of HipMer. The input is the human genome chromosome 14. Results obtained on Edison.

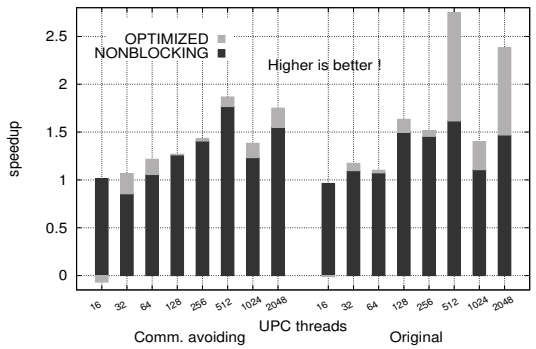

Figure 8: Speedup in communication when using nonblocking communication and our optimized version delaying synchronizations into nonblocking barriers of miniGMG, IMM version, (weak scaling). Results obtained on Edison.

with barriers, overlap `printf` with barriers, etc. In all cases, our performance model based on trace weights was useful to determine the importance of a particular transformation.

In miniGMG, the tool found opportunity to delay performance critical code inside barriers. Figure 8 shows the miniGMG speedup when delaying communication completion into the implementation of the barrier call. Except for 16 threads, using our tool improves the performance of this already optimized code. We obtained up to 64% performance improvements in communication time when compared to the nonblocking version (512 threads). This translates into 6% improvement in the end-to-end execution time. The performance improvements get more pronounced at high concurrency. Besides transforming blocking communication into overlapped nonblocking communication, our tool detected redundant barriers in HipMer. It also advised to delay communication completion before the last redundant barrier. Again, this would be a very complex transformation to perform manually.

When considering optimizations across bespoke synchronization patterns, again our optimizer is able to improve performance. Figures 9, 10 and 11 present the results for the AGG and IMM versions of miniGMG respectively. Figures 9 and 11 show the results obtained on Edison whereas Figure 10 shows the results obtained on Shepard. The gray bars represent the speedup we gain in addition to the version of the algorithm that implements a Producer-Consumer pattern. Note this is the most aggressively hand optimized implementation. After applying our tool we observe performance improvements as high as 29% (with 1,024 threads for the IMM version on Edison). These improvements come from the ability to mix communication completion with independent ad-hoc (semaphores) synchronization operations and translate into 6% end-to-end performance improvement. Note that Shepard does not have enough nodes to get the benefit we have on Edison.

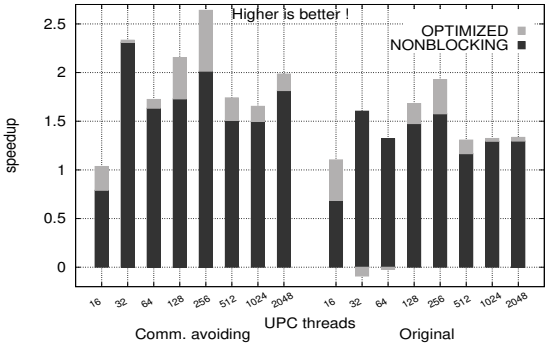

Figure 9: Speedup in communication when using nonblocking communication and our optimized version that delays synchronizations across semaphores of miniGMG, AGG version PCDB (weak scaling). Results obtained on Edison.

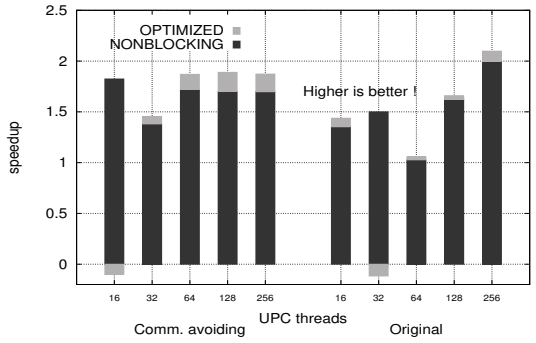

Figure 10: Speedup in communication when using nonblocking communication and our optimized version of miniGMG, AGG version PC (weak scaling). Results obtained on Shepard.

6.4 Statistics on Optimizations

Table 2 presents the number of optimizations returned for the NAS Benchmarks. *Total* depicts the number of optimizations found, *Performed* is the number of optimizations actually performed after verification and profitability analysis. Some optimizations were

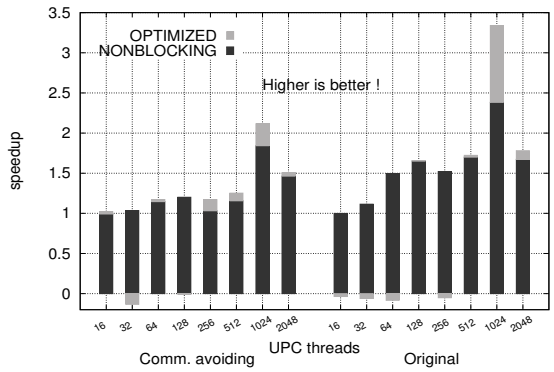

Figure 11: Speedup in communication when using nonblocking communication and our optimized version of miniGMG, IMM version PCDB (weak scaling). Results obtained on Edison.

considered as not interesting (e.g. blocking communication immediately followed by a barrier). The last column shows the number of optimizations found for nonblocking barriers (where to move `notify` and `wait`). As illustrated, for some benchmarks the number of possible placements is as high as 98 (27+61 for SP). The large number of available optimizations, combined with the fact that some span procedure boundaries argues for automatic approaches like ours.

Table 2: Number of one-sided and barriers optimizations returned by the tool for the UPC NAS.

Benchmark	One-sided Optimizations		# Possible Barriers
	Total	Performed	Optimization
BT	18	18	49
SP	27	12	61
IS	6	6	20
FT	1	1	19
LU	14	4	53

6.5 Practical Usage Considerations

Table 3: Overhead induced by the tool to generate a trace.

Benchmark	Execution Time	
	without the tool	with the tool
miniGMG IMM	12.517 sec	> 3h
NAS BT	0.09 sec	2.87 h
NAS SP	0.086 sec	29.87 min
NAS IS	0.01 sec	8.63 sec
NAS FT	0.063 sec	4.08 min
NAS LU	0.058 sec	19.92 min
HipMer	15.934 sec	> 3h

The tool proposes optimizations based on a single input. Since we cannot guarantee soundness across other inputs we return the possible list of optimizations only as advice to developers. For the SPMD applications tested this proved to be sufficient. In the current prototype, the runtime tracing and storage overhead is high and we had to trace on small inputs and generalize. All NAS benchmarks except LU (4) have been launched with 16 processes and Class S (small inputs). miniGMG was launched with 32 threads while HipMer was launched with 48 threads. Table 3 shows this runtime overhead. To improve scalability, we know how to engineer techniques that provide dynamic control over instrumentation code to exploit the iterative nature of most scientific applications. As explained, the good news is that for the programs considered a single run is enough to infer most of the information needed for optimizations. We believe this happens for most SPMD or hybrid parallelism SPMD+X codes.

7 DISCUSSION

The tool is useful in practice and feedback from scientific application developers indicates that it covers the important aspects of their code development. As an optimizer it provides performance unattainable through manual transformations due to the ability to cross procedure boundaries across levels of abstraction of independently developed user or runtime libraries. This is illustrated by our optimization to specialize barriers into their split-phase components. The tool can be also used in an exploratory manner for new transformations or experimentation with new APIs as it can be easily retargeted. This is illustrated by the transformations for bespoke synchronization using user defined semaphores. Finally, the tool provides an upper bound on the attainable performance improvements for aggressive communication optimizations.

When presented with a choice for placement, developers can rapidly assess its correctness due to code structure, but they have challenges tracking the state associated with individual operations. Using the analysis stage we can also provide feedback about the correctness of a choice, as performing the transformations already requires data race detection. The code generation stage can be used stand alone for experimenting with code transformations: generating a transformation just requires passing in a list of LLVM IR markers.

A precondition to obtaining all the performance results was specializing the call path for non-blocking operations inside GASNet. Each GASNet operation (init, wait, barrier, etc) will call eventually a function `AMPoll` to ensure progress of asynchronous activities. When performing aggressive overlap, we ended up with cascades of such calls which ended up degrading performance. We had to dynamically specialize the critical call path inside GASNet to avoid polling when our optimizations were active. Furthermore, it turned out that the specialization strategy depends on the application and we had to use different approaches for pure UPC SPMD code when compared with hybrid parallelism UPC+OpenMP code. These insights are useful to any other communication runtime developers.

The runtime API we use is of interest to other practitioners of communication optimizations. As shown, performance improvements depend on concurrency and problem size. A code generation strategy able to flip easily between transformations based on runtime data is beneficial for performance. We are already considering

the principles to guide enabling and disabling our optimizations during runtime.

We believe that our approach can easily support other programming models and runtime libraries. Most important, we believe we can easily extend it to support MPI 3 RMA (one-sided) operations. As there is no feasible compiler based approach to transform MPI codes across multiple programming languages (C, C++, Fortran), the only option left to developers is manual optimization. As there are not many RMA based applications available yet, we have talked to developers of several large scientific applications engaged in code modernization. We have compiled a list of applications that can benefit from our optimizations when transformed for RMA. Climate codes such as Model for Prediction Across Scales (MPAS) [4] or Particle-In-Cell codes use point-to-point communication and synchronization across the domain decomposition. Thus they have to implement Producer-Consumer synchronization. Other codes, such as the Cyclops [33] tensor contractions engine, MFDn [34] or BigStik for nuclear configuration interaction have formulations that leads to optimizations using non-blocking collective operations.

8 RELATED WORK

There exists a large body of work using static program analysis for communication optimizations in parallel programs. All are limited to intra-procedural analysis. The subject has been studied extensively in the context of data parallel languages [20, 21, 26, 35]. These projects tend to focus on array optimizations for bandwidth, such as communication vectorization. In addition, the work by Chakrabarti et al [16] uses redundancy elimination and a global communication optimization algorithm for reads, which identifies the earliest and latest safe position to issue the communication. In this case global denotes optimizations across data parallel statements contained within a single function.

In the context of explicitly parallel languages, Hendren and Zhu propose optimizations for latency hiding through communication overlap in parallel C programs [39]. Their analysis identifies the earliest point that a remote read can be issued, and applies either pipelined or blocking (coalesced) communication based on heuristics. Work by Chen et al [18] proposes more general techniques and applies them in the context of the UPC language. Both analyses handle only the implicit communication generated through scalar variable assignments, rather than the explicit Put/Get operations.

Work by Iancu et al [25] presents a static analysis that targets both bandwidth and latency hiding optimizations. For a loop nest written in UPC, they combine message vectorization with a technique to decompose and software pipeline the transmission.

Latency hiding optimizations for Send/Recv communication have been explored for MPI based codes using asynchronous task parallelism. HCMPI [17] introduces a runtime library where Send or Recv calls are spawned as stand alone tasks. HCMPI provides only the mechanisms, attaining optimal overlap is left to the application developer. Nguyen et al [29] introduce an alternate approach where they retrofit a tasking model onto the MPI runtime itself. Theirs is a `#pragma` based approach where communication and the related computation are marked in the source code. Due to static scoping and nesting restrictions for their pragma, they can handle only code within a single function. UPC++ [38] combines tasking with one-sided communication, but similar to HCMPI it provides only the basic mechanisms The optimization burden falls to the users of the infrastructure.

Dynamic program analyses have been only recently considered for communication optimizations in scientific codes. Chabbi et al [15] present an analysis able to detect and speculatively elide redundant barriers at program runtime. The analysis is implemented for a code using an underlying communication library very close in spirit to GASNet.

Another area relevant to our effort, is manual code optimization for scientific computing. Here, most optimizations target MPI Send/Recv message aggregation for bandwidth purposes. The wider adoption of one-sided communication opened the way for latency hiding optimizations [13, 30] and customized point-to-point synchronization patterns [31]. Nonblocking MPI collectives have been proposed [23] but not many experiences are available, due to implementation challenges. Sudarsan et al [36] describe the algorithmic challenges to optimize with overlapping collectives in a cosmic microwave background analysis code.

Recently Hayashi et al [22] describe LLVM extensions to support the implementation of PGAS languages. They extend the LLVM IR to represent global and local address spaces and with intrinsics for blocking communication. They present results for simple communication aggregation optimizations and have not considered yet representing or optimizing non-blocking communication.

9 CONCLUSION

In this paper we describe an advice and transformation tool to assist developers that would like to optimize their code using one-sided communication primitives. We have considered blocking or non-blocking point-to-point (Put/Get) and both collective and point-to-point synchronization operations. One optimization attempts to provide maximal communication overlap. In this case our experiments showed that the transformed code matches or exceeds the performance of manually optimized code. Another class of optimization combines communication overlap with synchronization optimizations. This type of transformation is beyond the ability of manual optimization and we were able to obtain good performance improvements over already optimized code.

Most large scientific computing applications are modular and compose multiple solvers. We believe that the main value of our approach comes from its ability to optimize across boundaries of software modules or libraries, while specializing for the intrinsics of the underlying communication runtime: our dynamic analysis approach is probably the only practical solution able to deal with complexities of real codes which combine programming languages and multiple third party libraries. Furthermore, the tool allows for easy prototyping of new transformations and communication APIs, as well as providing upper bounds on attainable performance improvements through manual optimization. Although demonstrated mainly for the UPC programming language, the methodology is programming language independent and can be easily adapted to the semantics of other communication and synchronization API.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their useful feedback. Support for this work was majorly provided through the X-Stack program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under collaborative agreement numbers DE-SC0008699.

REFERENCES

[1] *GASNet Communication System. http://gasnet.lbl.gov.*
[2] *UPC Home Page. http://upc-lang.org.*
[3] *X10: Performance and Productivity at Scale. http://x10-lang.org.*
[4] *The Model for Prediction Across Scales (MPAS)*, 2013. https://mpas-dev.github.io.
[5] *Edison*, 2016. http://www.nersc.gov/users/computational-systems/edison/.
[6] *Berkeley UPC User's Guide v. 2.22.2*, 2017. http://upc.lbl.gov/docs/user/.
[7] *The Chapel Parallel Programming Language*, 2017. http://chapel.cray.com/index.html.
[8] The LLVM Compiler Infrastructure, 2017. http://llvm.org.
[9] *Shepard*, 2017. http://www.sandia.gov/asc/computational_systems/HAAPS.html.
[10] B. Alverson, E. F. L. Kaplan, and D. Roweth. *The Cray XC Network.*, 2012. http://www.cray.com/sites/default/files/resources/CrayXCNetwork.pdf.
[11] B. Alverson, E. F. L. Kaplan, and D. Roweth. *CrayÂõ XCTM Series Network*, 2012.
[12] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks – summary and preliminary results. In *Supercomputing*, 1991.
[13] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing bandwidth limited problems using one-sided communication and overlap. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, 2006.
[14] The Berkeley UPC Compiler, 2002. http://upc.lbl.gov.
[15] M. Chabbi, J. M. Crummey, K. Sen, W. de Jong, W. Lavrijsen, and C. Iancu. Barrier Elision for Production Parallel Programs. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2015*, 2015.
[16] S. Chakrabarti, M. Gupta, and J. Choi. Global communication analysis and optimization. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 68–78, 1996.
[17] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. CavÃĭ, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan. Integrating asynchronous task parallelism with MPI. In *IEEE Parallel and Distributed Processing (IPDPS)*, 2013.
[18] W.-Y. Chen, C. Iancu, and K. Yelick. Communication optimizations for fine-grained UPC applications. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, 2005.
[19] E. Georganas, A. BuluÃğ, J. Chapman, L. Oliker, D. Rokhsar, and K. Yelick. merAligner: A fully parallel sequence aligner". In *Proceedings of the 2015 IEEE International Symposium on Parallel&Distributed Processing*, IPDPS '15, 2015.
[20] M. Gupta, S. Midkiff, E. Schonberg, et al. A HPF compiler for the IBM SP2. In *Supercomputing 1995*, November 1995.
[21] M. Gupta, E. Schonberg, and H. Srinivasan. A unified framework for optimizing communication in data-parallel programs. *IEEE Transactions on Parallel and Distributed Systems*, July 1996.
[22] A. Hayashi, J. Zhao, M. Ferguson, and V. Sarkar. LLVM-based communication optimizations for PGAS programs. In *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, LLVM '15, 2015.
[23] T. Hoefler, P. Kambadur, R. L. Graham, G. Shipman, and A. Lumsdaine. *A Case for Standard Non-blocking Collective Operations.* 2007.
[24] P. Husbands and K. Yelick. Multi-threading and one-sided communication in parallel LU factorization. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC '07, 2007.
[25] C. Iancu, W. Chen, and K. Yelick. Performance portable optimizations for loops containing communication operations. In *Proceedings of the 22nd Annual International Conference on Supercomputing*, 2008.
[26] M. Kandemir, P. Banerjee, A. Choudhary, J. Ramanujam, and N. Shenoy. A global communication optimization technique based on data-flow analysis and linear algebra. *ACM Transactions on Programming Languages and Systems*, 21(6):1251–1297, 1999.
[27] M. Luo, D. K. Panda, K. Z. Ibrahim, and C. Iancu. Congestion avoidance on manycore high performance computing systems. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, 2012.
[28] *The Message Passing Interface Standard*, 2016. http://www.mpi-forum.org/.
[29] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden. Bamboo: Translating MPI applications to a latency-tolerant, data-driven form. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.
[30] R. Nishtala, P. H. Hargrove, D. O. Bonachea, and K. A. Yelick. Scaling communication-intensive applications on BlueGene/P using one-sided communication and overlap. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, 2009.
[31] H. Shan, S. Williams, Y. Zheng, A. Kamil, and K. Yelick. Implementing high-performance geometric multigrid solver with naturally grained messages. In *Proceedings of the 2015 9th International Conference on Partitioned Global Address Space Programming Models*, 2015.
[32] Man page collections: Shared memory access. http://www.cray.com/craydoc/20/manuals/S-2383-22/S-2383-22-manual.pdf.
[33] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IPDPS '13, 2013.
[34] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, and H. V. Le. Accelerating configuration interaction calculations for nuclear structure. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, 2008.
[35] E. Su, A. Lain, S. Ramaswamy, D. J. Palermo, E. W. H. IV, and P. Banerjee. Advanced compilation techniques in the PARADIGM compiler for distributed-memory multicomputers. In *9th ACM International Conference on Supercomputing*, pages 424–433, July 1995.
[36] R. Sudarsan, J. Borrill, C. Cantalupo, T. Kisner, K. Madduri, L. Oliker, Y. Zheng, and H. Simon. Cosmic microwave background map-making at the petascale and beyond. In *Proceedings of the 25th International Conference on Supercomputing*, 2011.
[37] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker. Optimization of geometric multigrid for emerging multi- and manycore processors. In *Proc. of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12. IEEE Computer Society Press, 2012.
[38] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick. UPC++: A PGAS extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, 2014.
[39] Y. Zhu and L. J. Hendren. Communication optimizations for parallel c programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation(PLDI)*, pages 199–211, 1998.