LLNL-TR-549171

# Compiled MPI: Cost-Effective Exascale Applications Development

G. Bronevetsky, D. Quinlan, A. Lumsdaine, T. Hoefler

April 17, 2012

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

# Compiled MPI: Cost-Effective Exascale Applications Development

Greg Bronevetsky, Daniel Quinlan, Torsten Hoefler and Andrew Lumsdaine

The complexity of petascale and exascale machines makes it increasingly difficult to develop applications that can take advantage of them. Future systems are expected to feature billion-way parallelism, complex heterogeneous compute nodes and poor availability of memory (Peter Kogge, 2008). This new challenge for application development is motivating a significant amount of research and development on new programming models and runtime systems designed to simplify large-scale application development. Unfortunately, DoE has significant multi-decadal investment in a large family of mission-critical scientific applications. Scaling these applications to exascale machines will require a significant investment that will dwarf the costs of hardware procurement. A key reason for the difficulty in transitioning today's applications to exascale hardware is their reliance on explicit programming techniques, such as the Message Passing Interface (MPI) programming model to enable parallelism.

MPI provides a portable and high performance message-passing system that enables scalable performance on a wide variety of platforms. However, it also forces developers to lock the details of parallelization together with application logic, making it very difficult to adapt the application to significant changes in the underlying system. Further, MPI's explicit interface makes it difficult to separate the application's synchronization and communication structure, reducing the amount of support that can be provided by compiler and run-time tools. This is in contrast to the recent research on more implicit parallel programming models such as Chapel, OpenMP and OpenCL, which promise to provide significantly more flexibility at the cost of reimplementing significant portions of the application.

We are developing CoMPI, a novel compiler-driven approach to enable existing MPI applications to scale to exascale systems with minimal modifications that can be made incrementally over the application's lifetime. It includes:

- New set of source code annotations, inserted either manually or automatically, that will clarify the application's use of MPI to the compiler infrastructure, enabling greater accuracy where needed
- A compiler transformation framework that leverages these annotations to transform the original MPI source code to improve its performance and scalability
- Novel MPI runtime implementation techniques that will provide a rich set of functionality extensions to be used by applications that have been transformed by our compiler
- A novel compiler analysis that leverages simple user annotations to automatically extract the application's communication structure and synthesize most complex code annotations

## I. CoMPI Runtime Infrastructure

Exascale systems will most likely consist of large multi-core nodes with hundreds to thousands of processors in a shared memory domain. This hybrid design allows us to take advantage of hybrid programming techniques. Many current codes are written in MPI-only style and cannot immediately take advantage of shared memory. In this section, we outline runtime techniques to take advantage of shared memory architectures. Another feature of exascale systems is the massive number of network endpoints. After we described how to transparently optimize MPI codes for hybrid architectures, we will show how those optimized codes can be adapted for large-scale networks by automatic detection and optimization of communication patterns.

## A. Hybrid MPI

We have developed Hybrid MPI (HMPI), a wrapper library that sits between the native MPI implementation and the application. HMPI performs the proposed mapping of ranks to threads and uses the shared address space to optimize point-to-point and collective communication bound for other ranks in the same node. A program is transformed to use HMPI can be done by following these steps:

1. Rename the original `main` function to `tmain` and remove the call to `MPI_Init` or `MPI_Init_thread`.
2. Create a new `main` function that calls `HMPI_Init`, which will start threads using `tmain`.
3. Replace all MPI calls with calls to equivalent HMPI functions.
4. Privatize all global variables.

Transformations (1)-(3) can easily be performed with a very simple compiler (replacing text statements only). Step (4), the privatization of global variables, can be done by annotating them as thread-private (e.g., `__thread` in GCC), which can also be automated with a compiler, as has been explored in prior work [6].

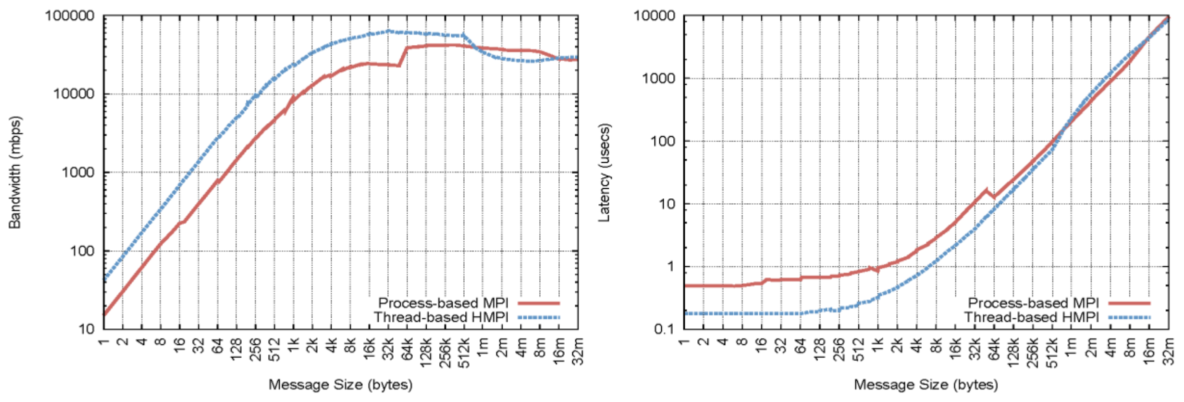


**FIGURE 1: BANDWIDTH AND LATENCY OF MVAPICH 1.6 VS HMPI ON ONE XEON X5660 NODE.**

Figure 1 compares same-node (i.e., shared memory) message passing performance of MVAPICH2 and HMPI. A shared address space permits simplified synchronization and requires only a single memory copy from the send buffer to the receive buffer (MPI normally requires two copies, which may be pipelined). 1-byte message latency is reduced from 0.50 microseconds for MVAPICH2 to 0.18 microseconds for HMPI on the LLNL Sierra cluster, which consists of dual-socket six-core Xeon X5660 CPUs. HMPI's peak bandwidth (63,168 mbps) is much higher than that of MVAPICH2 (41,996 mbps) due to one less memory copy. For messages larger than 512KB the bandwidth of HMPI and MVAPICH is the same. The next optimization shows how bandwidth can be improved for all message sizes.

## B. Sender-Receiver Synergistic Transfer

In HMPI, on-node data transfers are performed in two steps. First, the sender sends a pointer to a message buffer to the receiver. The receiver then copies the message sender's message buffer using `memcpy`. A single core cannot use all of a node's bandwidth, so it is possible to speed up such transfers by using multiple cores to perform them. We developed a technique where both the sender and receiver participate in the transfer of large messages (≥8KB). In this algorithm the transfer of the message is broken up into multiple calls to `memcpy`. If the sender polls its send request and sees that the receiver has matched but is still copying, it will start copying blocks as well. Figure 2 shows that this results in significantly higher bandwidth for messages ≥8KB on the LLNL Sierra cluster. This allows HMPI to outperform MVAPICH even for messages larger than 512KB.

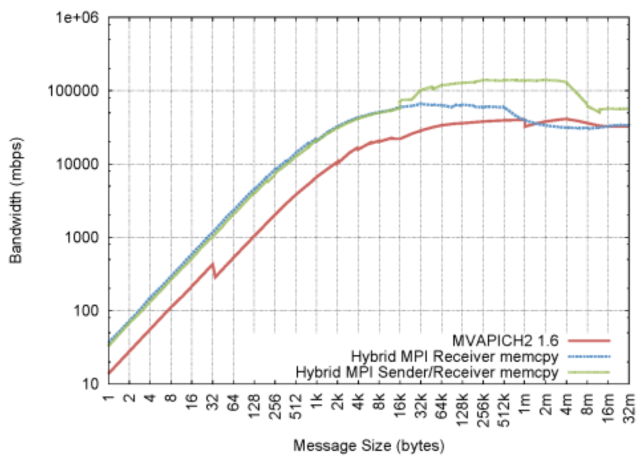


FIGURE 2: IMPROVED PERFORMANCE FROM SYNERGISTIC DATA TRANSFER ON XEON X5660 NODE.

## C. Ownership passing

The MiniMD benchmark, part of the Mantevo [3] benchmark suite, packs data into a buffer, sends it via MPI, then the receiver unpacks. Transferring via traditional MPI requires at least one memory copy. Since HMPI gives us a single address space for ranks on the same node, we can simply pass a pointer to the packed buffer from the sender to the receiver (using MPI). The receiver unpacks directly from the sender's buffer, then sends a 0-byte MPI message back to the sender when it is finished. Figure 3 shows the amount of time the MiniMD benchmark spends communicating on the LLNL Sierra cluster. Those shared memory optimizations will improve on-node communication performance and reduce energy usage. We will now discuss strategies to optimize off-node (network) communication performance by transparent communication pattern recognition and optimization.

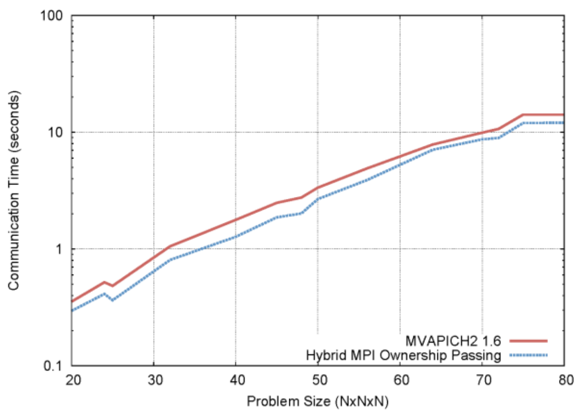


FIGURE 3: COMMUNICATION TIME IN MINIMD USING OWNERSHIP PASSING VS. MPI

## D. Multi-Version Variables

MVVs are a form of producer-consumer queue established prior to communication between two ranks. The producer obtains a buffer, fills it with data, and commits it to the MVV. The consumer retrieves the buffer and reads the data. Multiple buffers may be used inside the queue, but our results show that reusing a single buffer gives the best performance due to better locality. Normally, a flow control mechanism prevents buffers from being exhausted or overwritten at the receiver. However, some applications have a communication pattern (e.g. nearest-neighbor

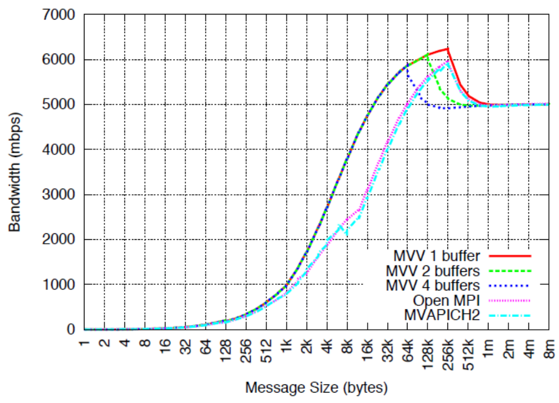


FIGURE 4: BANDWIDTH OF MVV VS MPI

stencils) that implicitly enforce flow control. In these cases, we can omit the flow control included in our protocols for better performance. Figure 4 shows the improvement in bandwidth that results from using MVVs with different numbers of buffers over traditional MPI implementations.

## E. Detection of Collectives with GOAL

Exascale systems will exhibit massive communication networks and optimizations of communication patterns will become mandatory. In addition, many parallel languages, such as Co-Array Fortran, do not offer high-level abstractions for collective communication primitives. Even if support for some collectives is provided, such support is never exhaustive, as new numerical methods possibly can make use of new collective communication patterns, such as neighborhood collectives. Using GOAL we are able to dynamically build the communication graph of a communication phase of an application. Such a communication graph contains all information needed to recognize the data-movement pattern of such a phase. We introduced a concise notation for such data-movement patterns which we call Single Static Transfer notation.

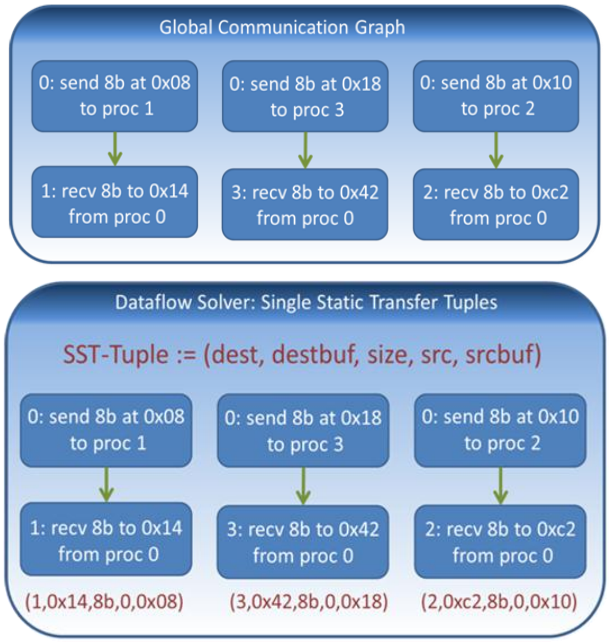

Global Communication Graph
0: send 8b at 0x08 to proc 1
0: send 8b at 0x18 to proc 3
0: send 8b at 0x10 to proc 2
1: recv 8b to 0x14 from proc 0
3: recv 8b to 0x42 from proc 0
2: recv 8b to 0xc2 from proc 0
Dataflow Solver: Single Static Transfer Tuples
SST-Tuple := (dest, destbuf, size, src, srcbuf)
0: send 8b at 0x08 to proc 1
0: send 8b at 0x18 to proc 3
0: send 8b at 0x10 to proc 2
1: recv 8b to 0x14 from proc 0
3: recv 8b to 0x42 from proc 0
2: recv 8b to 0xc2 from proc 0
(1,0x14,8b,0,0x08)
(3,0x42,8b,0,0x18)
(2,0xc2,8b,0,0x10)

**FIGURE 5: DETECTING COLLECTIVES**

After we have extracted the communication structure of a code we can compare the found structure with that of high-level communication primitives for which the current machine or software stack offers tuned implementations, such as many MPI collective functions. Replacing such codes with the appropriate calls to the optimized function at runtime can decrease the communication time by an order of magnitude. As Figure 6 shows, the performance benefits of this optimization pay off almost immediately: On a 4000 process allocation our optimization pays off after the second iteration through the communication phase of a Co-Array Fortran program implementing a Broadcast.

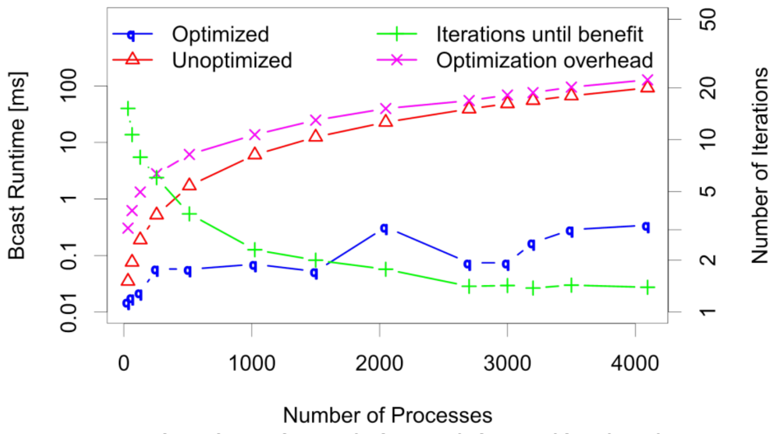


**FIGURE 6: PERFORMANCE OF TRANSFORMED COLLECTIVES**

## II. Compiler Infrastructure

### A. ROSE Dataflow

The ROSE compiler provides state of the art capabilities for analyzing and transforming applications written in the major DoE languages, including C, C++ and Fortran. Various tools successfully use the ROSE Abstract Syntax Tree (AST) to extract important information about the application structure and such analyses have been further simplified via additional features such as the system dependence graphs and def-use chains. However, while the analysis capabilities of ROSE have grown stronger over the last several years at the start of the CoMPI project ROSE did not include support for the dataflow framework [2] and the Static Single Assignment (SSA) representation [5], two fundamental tools for compiler analysis that were required for our work.

As part of the CoMPI project we are developing a new infrastructure that supports dataflow analyses and symbolic evaluation on top of Control-Flow Graphs (CFGs) in dense as well as SSA forms. This framework is designed to be generic: arbitrary intra- and inter-procedural analyses can be coupled to each other. Further, memory locations and functions are represented using abstractions that can be implemented with arbitrary degrees of precisions by external analyses. For instance, a pointer analysis may be able to infer that two pointers do not alias each other. This information is then abstracted behind our memory location interface and any other analysis can gain the benefit of increased precision from knowing that the targets of the pointers may never be the same, without explicitly considering how this information was computed. This powerful capability will make it possible to plug together different analyses with different precision/performance trade-offs without having to explicitly write them to be aware of each other.

While advanced analysis capabilities will make it possible to infer useful facts about applications and to select optimizations, transforming applications to implement these optimizations is difficult. This is because ROSE, being a source-to-source compiler, represents the application as an AST. While useful for producing source code, transformations of the AST can be difficult because they must account for many details of the source language. For example, given the expression `a=foo()+bar()*baz()`, to insert a to call `printf()` after the multiplication (and the side-effects of `bar` and `baz`) is done but before the addition (and the side-effects of `foo`) has started it is necessary to transform it into the code in Figure 7. We have developed a new ROSE transformation API that makes it possible for analyses to implement transformations on the higher-level CFG and have the results be directly applied to the AST. This enables analysis and transformation developers to reason about the application at a high level while still gaining the advantages of the ROSE compiler and source-to-source compilation in general.

```
tmp = bar()*baz();
printf();
a=foo()*tmp;
```

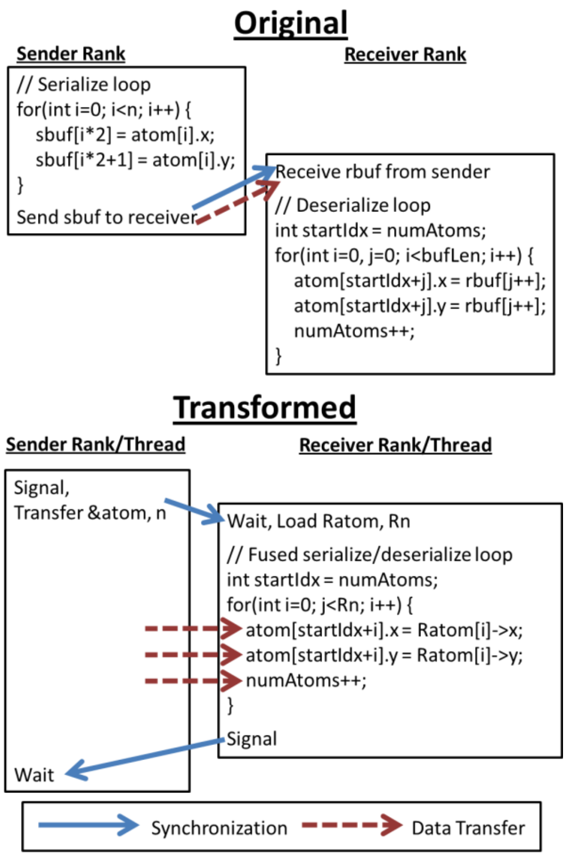
**FIGURE 7: TRANSFORMED CODE**

### B. Send-Receive Fusion

Although processors with multiple (or many) cores and shared memory are becoming ubiquitous, MPI enforces copies between source and target processes and thus cannot fully utilize shared memory and cache architectures of modern machines. To enable MPI-based programs to more fully exploit features of multi- and many-core architectures, we developed a compiler-based transformation that fuses message serialization and deserialization loops such that send and receive calls can be replaced by direct memory accesses. Our compiler replaces most of the MPI communication functions with direct load/store accesses and our runtime provides a threaded MPI implementation to handle the remaining functions.

## 1. Approach

Our compiler analysis operates on application code that explicitly serializes data into, and deserializes data from, a serial representation. The analysis detects serialization and deserialization by MPI ranks executing on the same node and fuses those operations into a single loop that directly transfers data from sender to receiver without the use of intermediate buffers. This optimization can significantly improve performance in the case where both ranks execute on the same node and also in cases where the interconnection network supports efficient fine-grained remote memory access [5].



```
// Serialize loop
for(int i=0; i<n; i++) {
   sbuf[i*2] = atom[i].x;
   sbuf[i*2+1] = atom[i].y;
}
Send sbuf to receiver
```

```
Receive rbuf from sender
// Deserialize loop
int startIdx = numAtoms;
for(int i=0, j=0; i<bufLen; i++) {
   atom[startIdx+j].x = rbuf[j++];
   atom[startIdx+j].y = rbuf[j++];
   numAtoms++;
}
```

```
Signal,
Transfer &atom, n

Wait
```

```
Wait, Load Ratom, Rn
// Fused serialize/deserialize loop
int startIdx = numAtoms;
for(int i=0; j<Rn; i++) {
   atom[startIdx+i].x = Ratom[i]->x;
   atom[startIdx+i].y = Ratom[i]->y;
   numAtoms++;
}
Signal
```

**FIGURE 8: FUSION OF SERIALIZE/DESERIALIZE CODE**

Figure 8 shows a motivating example for our transformation, extracted from the MiniMD benchmark. In the original code, a sender rank serializes an array of local atom records into buffer `sbuf` and uses MPI point-to-point communication to copy it to buffer `rbuf` on the receiver rank. This buffer is then deserialized into the receiver's `atom` data structure writing the incoming entries after the existing ones.

Since both loops iterate over the same sequence of buffer indices, the write to `sbuf[k]` by the sender corresponds to the read from `rbuf[k]` by the receiver. As such, the right-hand-side of the write to `sbuf[k]` produces the value that is ultimately copied to the left-hand-side of the read from `rbuf[k]`. Our transformation thus aligns the iterations of both loops to directly copy the data with no intermediate buffering.

This is shown in the Transformed code in Figure 8, where the serialize and deserialize loops have been fused to so that the receiver executes the entire data transfer in one pass. The new code also includes additional synchronization to ensure the data is delivered from the sender as well as transfers of the variables and pointers used in the sender's code to make them accessible by the receiver. It also valid if the fused loop is executed by the sender. The resulting code uses the original specification of parallelism from the MPI code but implements it in a way that is inherently suited to shared memory hardware.

## 2. Outline of Transformations

"Serialization code" is the code region that writes data into a buffer passed to a send operation (e.g. `MPI_Send`, `MPI_Isend`). "Deserialization code" is the code region that reads data from the receive operation (`MPI_Recv`, `MPI_Irecv`) that matches the send operation. Our work focuses on the common case where the serialization and deserialization loops iterate over the communication buffer in the same monotonic order. If the amount of data sent is computed during the serialization code, we assume that it is also sent in another message. Our algorithm for fusing serialize and deserialize code operates in two steps. First, the loop on one rank in the exchange is transformed so that it can execute on the other rank. This produces a single code region, executed by one of the ranks, that includes both loops and can be analyzed as a unit. Second, the control flow graphs of the loops are fused into a single graph that executes the statements of both loops in an order guaranteed not to violate the application's original data flow dependencies.

The transformation that enables code on one MPI rank to be executed by other ranks on the same node works by sending the initial values of live variables (those used in the migrated code) from the source rank to the destination rank, running the migrated code using these local copies of the variables and finally sending the result of the computation back from the destination rank to the source rank. Since the pointers used by the migrated code still refer to the same data structures in shared memory, it has exactly the same effect on these data structures regardless of which rank it is actually executed in.

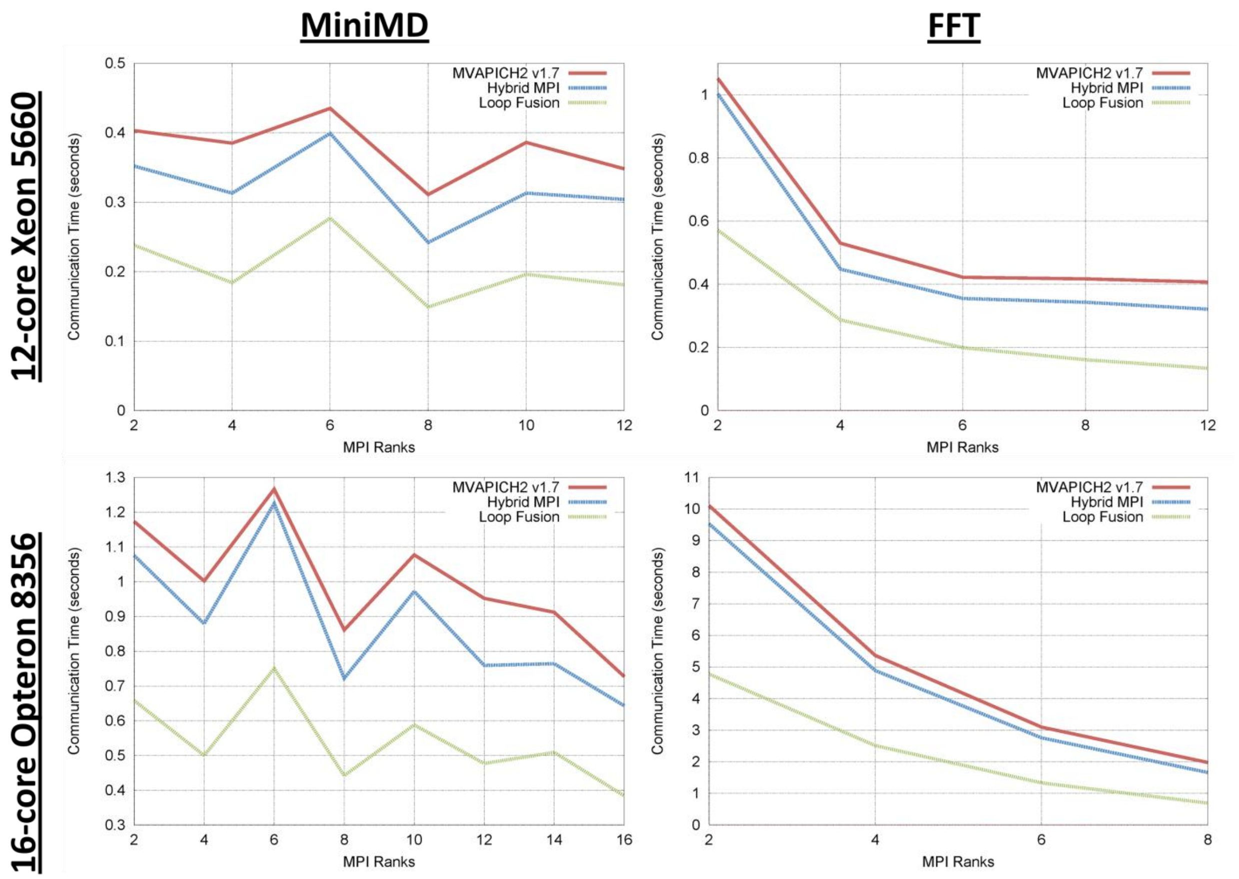


FIGURE 9: COMMUNICATION TIME OF ORIGINAL MPI AND HMPI CODE VS. HMPI+FUSION

The code fusion transformation takes the serialization and deserialization loops that are now both executed on either the sender or the receiver rank and fuses them into a single piece of code that transfers data from the sender's data structures to the receiver's. This transformation analyzes the linear expressions used by both code regions to index the send and receive buffers. It then moves expressions from the deserialization loop inside the serialization loop, while ensuring that each serialized value is consumed by the deserialization code after it is produced by the serialization code.

Figure 9 shows that loop fusion improves the performance of the MiniMD benchmark and an FFT kernel, showing the communication time of MVAPICH 1.7, HMPI and HMPI with loop fusion. Results are shown on the LLNL Sierra cluster (dual-socket six-core Xeon X5660 CPUs) and the Hera cluster (quad-socket four-core Opteron 8356 CPUs). At all scales HMPI performs better than MVAPICH and loop fusion is significantly faster than HMPI.

## III. Summary

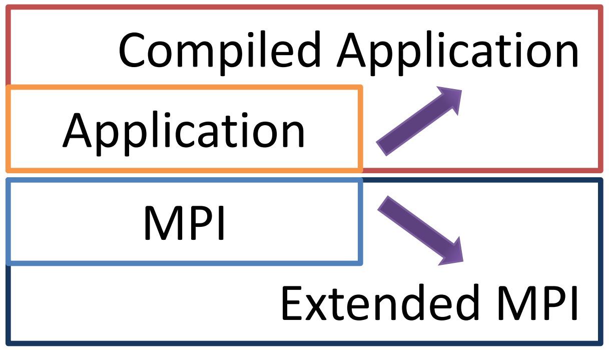
The goal of the CoMPI project is to help legacy MPI applications reach exascale performance while making only incremental changes to their source code}, thus gaining most of the benefits of implicit programming models without most of the costs. By providing developers with unprecedented levels of compiler and runtime support, CoMPI will enable them to focus on new science and to concentrate their reimplementation efforts on the few portions of their applications that require truly new numerical methods rather than new ways of expressing them.

We are planning to generate optimized HMPI code from our compiler passes. In addition to that, we plan to extract static communication regions from legqcy MPI codes and instantiate and optimize GOAL communication graphs. This will lead to automatic collective detection and communication optimizations.

## IV. Bibliography

1. B. Arimilli, R. A. (2010). The PERCS High-Performance Interconnect. *Symposium on High-Performance Interconnects.*
2. Kildall, G. (1973). A Unified Approach to Global Program Optimization. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL).
3. Michael A. Heroux, D. W. (2009). *Improving Performance via Mini-applications.* Sandia National Laboratory, SAND2009-5574.
4. Peter Kogge, e. (2008). ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems. Defense Advanced Research Projects Agency.
5. Ron Cytron, J. F., & Barry K. Rosen, M. N. (1991). Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems*, 451–490.
6. Stas Negara, G. Z.-C. (2010). Automatic MPI to AMPI Program Transformation using Photran. *Workshop on Productivity and Performance (PROPER).*

# Compiled MPI: Cost Effective Exascale Application Development



## Novel Ideas

**MPI is the dominant DoE programming model. Cost of revolutionary migration for MPI applications to Exascale prohibitively high! Need evolutionary path to Exascale performance for MPI applications.**

- **Developers describe parallel structure of application via annotations**
- **Used by compiler to implement transformations**
- **Extend MPI runtime to use compile-time information**
- **Compiler analyses automatically derive most complex annotations**

## Impact and Champions

**Billions of dollars and decades of effort invested in today's MPI applications. Project will enable DoE to leverage this investment on Exascale systems. We will use compiler analyses to capture the structure of MPI applications and eliminate the performance bottlenecks that constrain performance at large scales and many-core nodes. Approach will leverage explicit, locality-aware parallelism available in MPI applications to maintain high performance on future systems with reduced need for manual performance tuning.**

**Revolutionary performance, evolutionary development.**

**PIs: Greg Bronevetsky, Dan Quinlan, LLNL, Andrew Lumstaine, IU, Torsten Hoefler, UIUC**

## Milestones/Dates/Status

| | Scheduled | Actual |
|---|---|---|
| • Develop Prototype Dataflow Framework | May 2011 | May 2011 |
| • Develop Prototype Performance Optimizations | Sep 2011 | Sep 2011 |
| • Develop Formal Model of Parallel Protocols | Feb 2012 | Feb 2012 |
| • Develop Production Dataflow Framework (SSA, Polyhedral) | Sep 2012 | |
| • Develop Formal Validation Tools | Feb 2013 | |
| • Develop Parallel Dataflow Analysis Framework | Mar 2013 | |
| • Integrate Analysis and Runtime Optimizations | Jun 2013 | |

# Compiled MPI: Cost-Effective Exascale Application Development

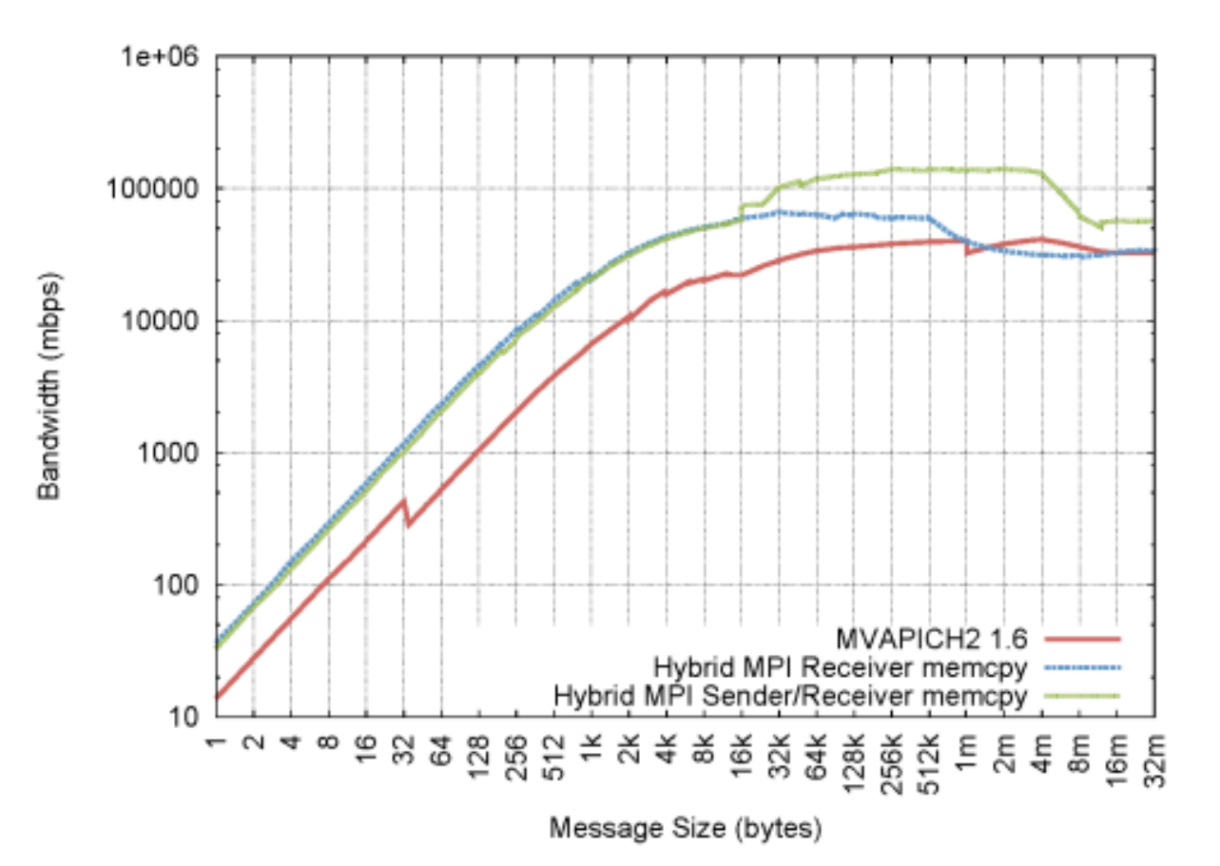
Daniel Quinlan, Greg Bronevetsky: Lawrence Livermore National Laboratory,
Andrew Lumsdaine, Indiana University and Torsten Hoefler, University of Illinois at Urbana-Champaign.

## Goals

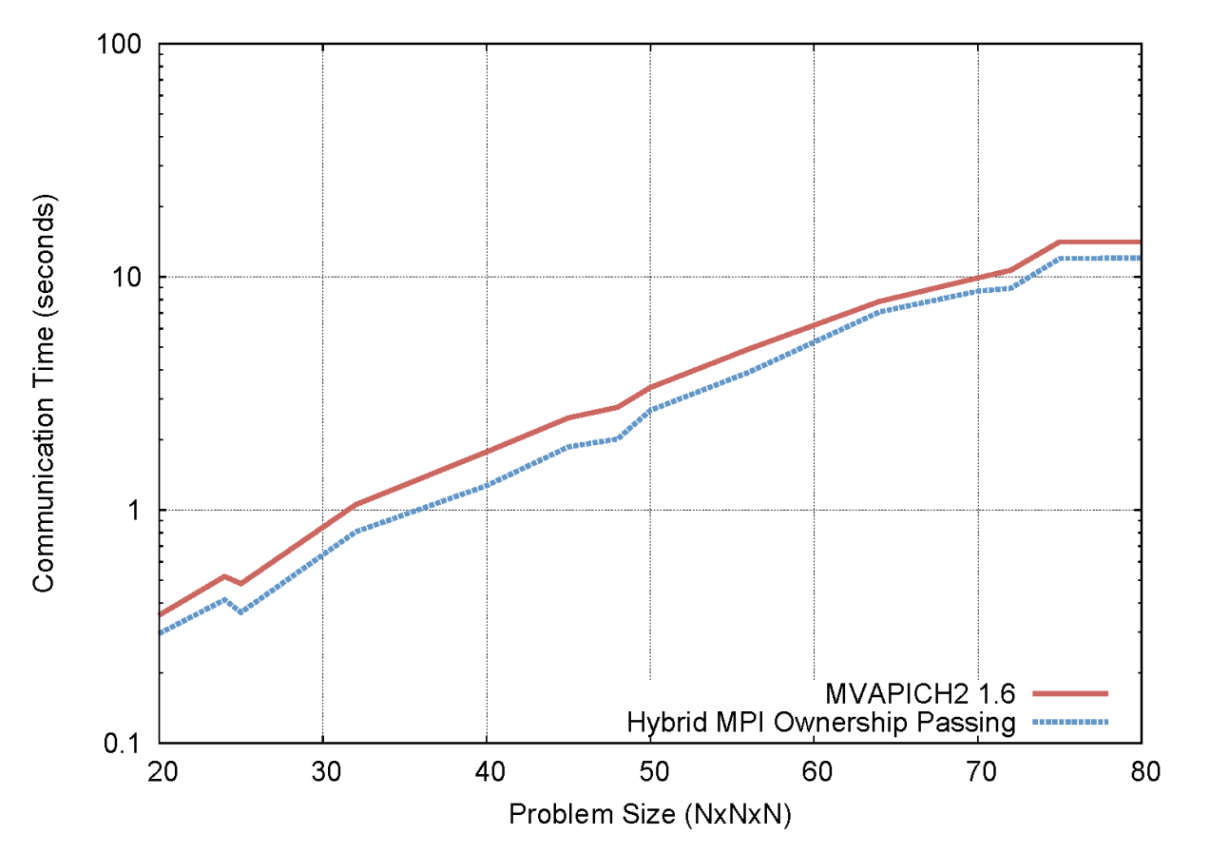
- MPI is the dominant DoE programming model, billions invested in applications
- Exascale systems pose new programming challenges
    - New programming models designed to reach Exascale performance
    - Cost of revolutionary migration is prohibitively high!
- Need evolutionary path to Exascale performance for MPI applications
- Approach
    - Developers annotate their code with description of parallel structure
    - Compiler uses annotations to implement transformations
    - Extend MPI runtime to take advantage of compile-time information
    - Compiler analyses to automatically derive most complex annotations

## Runtime: Communication Protocols

Our first strategy is to transform two-sided communication into more scalable alternatives. For instance, we optimize same-node communication by MPI ranks on each node as threads, enabling them to communicate more efficiently using hardware shared memory. First, we can transfer messages using one memory copy (MPI normally requires two). We can further improve performance for large messages by having both the sender and receiver perform parts of the copy (top). Second, we can replace message passing with ownership passing. Instead of copying the data, the sender passes ownership of its buffer to the receiver and avoids incurring the cost of copying (bottom).



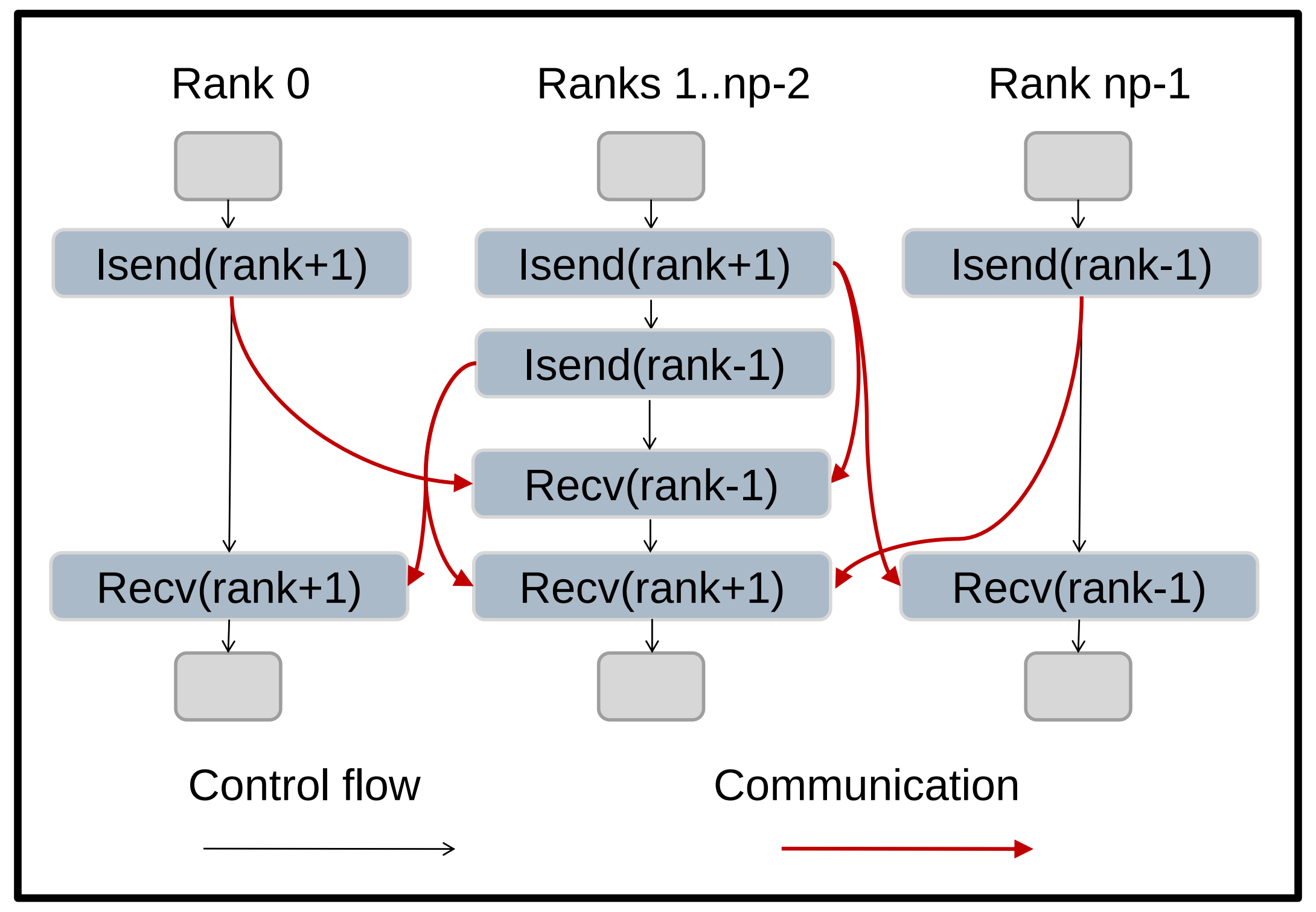
NetPIPE Bandwidth (higher is better)



Communication Time in miniMD

## Compiler: Communication Structure Analysis

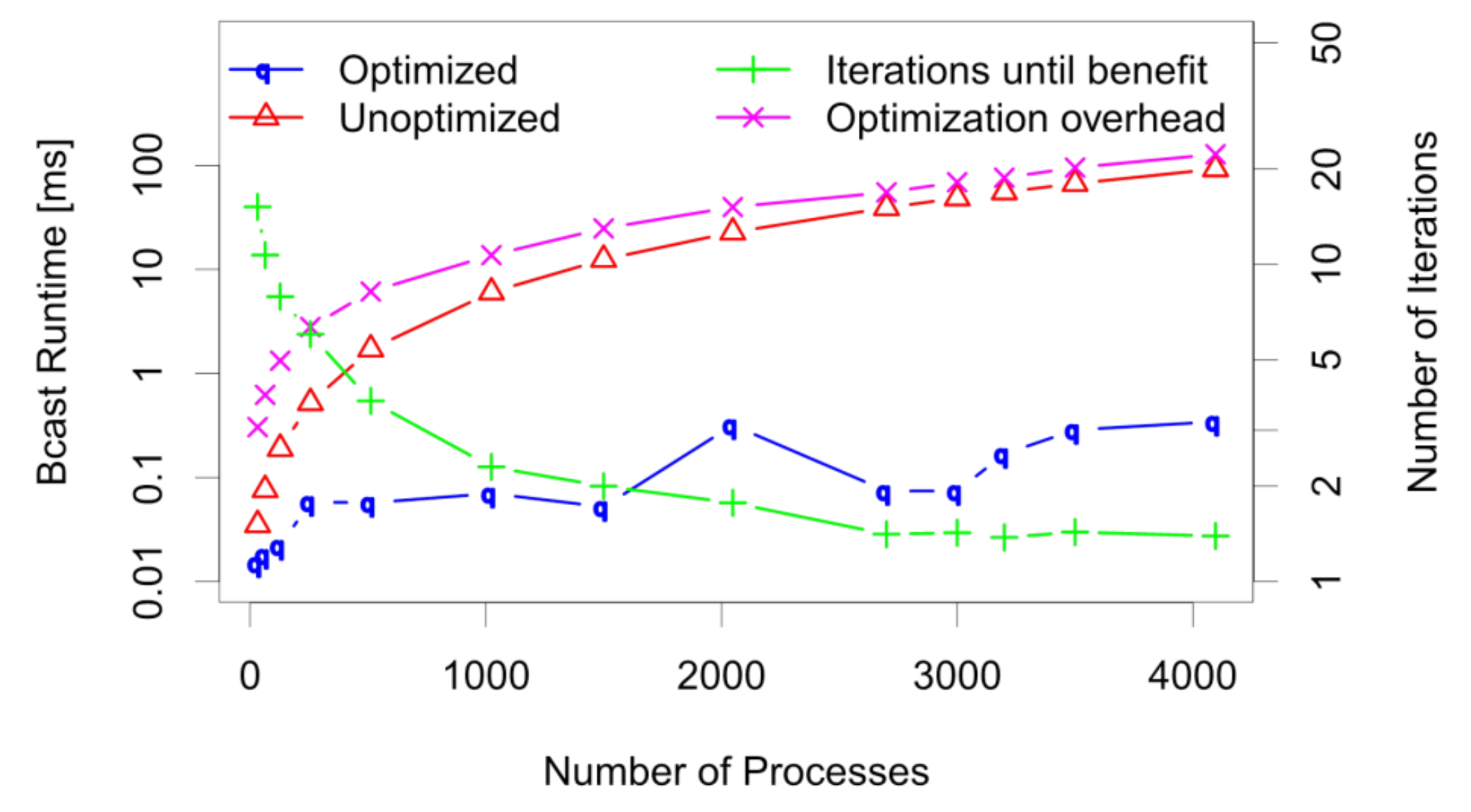
We are developing a compiler analysis infrastructure to statically match MPI send and receive operations. The analysis abstracts the behaviors of any number of processes into a few equivalence classes and symbolically matches send and receive expressions. It will be used to reduce the cost of receive matching and to replace collective communication patterns with optimized implementations.



If a send and receive operation are executed in the same shared memory domain, their data packing and unpacking code can be fused so that data is copied directly from the sender's to the receiver's data structure. We are developing a compiler analysis and transformation to perform this fusion.

## Runtime: Group Operation Assembly Language

We are also working to improve cross-node communication performance by using programmer annotations to identify and restructure the application's communication pattern. Using this Group Operation Assembly Language (GOAL) we are able to dynamically build the communication graph of a communication phase of an application. Such a communication graph contains all information needed to recognize the data-movement pattern of such a phase. Our experiments show the result of transforming a naïve point-to-point broadcast implementation into the equivalent collective on 4000 processors. The optimized version scales better, quickly amortizing the cost of detecting and optimizing the communication pattern.



# Compiled MPI: Cost-Effective Exascale Application Development

Daniel Quinlan, Greg Bronevetsky: Lawrence Livermore National Laboratory,
Andrew Lumsdaine, Indiana University and
Torsten Hoefler, University of Illinois at Urbana-Champaign.

The complexity of Petascale and Exascale machines makes it increasingly difficult to develop applications that can take advantage of them. Future systems are expected to feature billion-way parallelism, complex heterogeneous compute nodes and poor availability of memory. This new challenge for application development is motivating a significant amount of research and development on new programming models and runtime systems designed to simplify large-scale application development. Unfortunately, DoE has significant multi-decadal investment in a large family of mission-critical scientific applications. **Scaling these applications to Exascale machines will require a significant investment that will dwarf the costs of hardware procurement.** A key reason for the difficulty in transitioning today's applications to Exascale hardware is their reliance on explicit programming techniques, such as the Message Passing Interface (MPI) programming model to enable parallelism.

MPI provides a portable and high performance message-passing system that enables scalable performance on a wide variety of platforms. However, it also forces developers to lock the details of parallelization together with application logic, making it very difficult to adapt the application to significant changes in the underlying system.

Further, MPI's explicit interface makes it difficult to describe the application's synchronization and communication structure, reducing the amount of support that can be provided by the compiler and run-time tools. This is in contrast to recent research on more implicit parallel programming models such as X10, Chapel, OpenMP and OpenCL, which promise to provide significantly more flexibility at the cost of reimplementing significant portions of the application.

**Compiled MPI (CoMPI) is a novel compiler-driven approach to enable existing MPI applications to scale to Exascale systems with minimal modifications that can be made incrementally over the application's lifetime.** It includes:

- A new set of source code annotations, inserted either manually or automatically, that will clarify the application's use of MPI to the compiler infrastructure, enabling greater accuracy where needed,
- A compiler transformation framework that leverages these annotations to transform the original MPI source code to improve its performance and scalability,
- Novel MPI runtime implementation techniques that will provide a rich set of functionality extensions that will be used by applications that have been transformed by our compiler,
- A novel compiler analysis that leverages simple user annotations to automatically extract the application's communication structure and synthesize most complex code annotations.

CoMPI is based on the R&D 100 award-winning ROSE compiler and production Open MPI implementation of MPI, enabling it to target complex DoE Applications and become a practical tool.

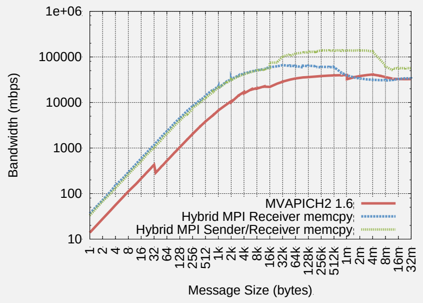
**The final deliverable of this project will be a system that will help legacy MPI applications reach Exascale performance while making only incremental changes to their source code**, thus gaining most of the benefits of implicit programming models without most of the costs. By providing developers with unprecedented levels of compiler and runtime support, this system will enable them to focus on new science and to concentrate their reimplementation efforts on the few portions of their applications that require truly new numerical methods rather than new ways of expressing them.

## Selected Research Thrusts

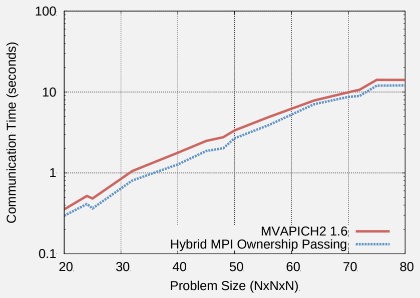
### Communication Protocols

Our first strategy is to transform two-sided communication into more scalable alternatives. For instance, we optimize same-node communication by MPI ranks on each node as threads, enabling them to communicate more efficiently using hardware shared memory. First, we can transfer messages using one memory copy (MPI normally requires two). We can further improve performance for large messages by having both the sender and receiver perform parts of the copy (left figure). Second, we can replace message passing with ownership passing. Instead of copying the, the sender passes ownership of its buffer to the receiver and avoids incurring the cost of copying (middle figure).

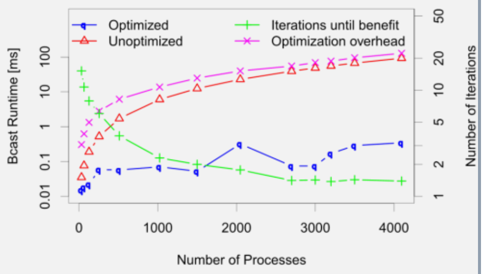
We are also working to improve cross-node communication performance by using programmer annotations to identify and restructure the application's communication pattern. Our experiments (right figure) show the result of transforming a naïve point-to-point broadcast implementation into the equivalent collective on 4000 processors. The optimized version scales better, quickly amortizing the cost of detecting and optimizing the communication pattern.



NetPIPE Bandwidth (higher is better)



MiniMD Communication Time (lower is better)

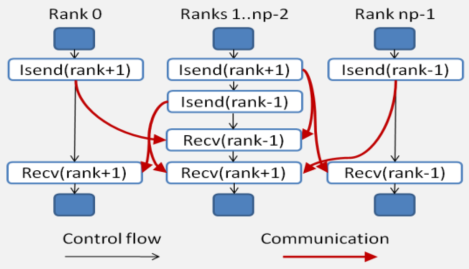


Performance Improvement from Restructuring

### Compiler Analyses and Transformations

Communication Structure Analysis:

We are developing a compiler analysis infrastructure to statically match MPI send and receive operations. The analysis abstracts the behaviors of an unbounded number of processes into a few equivalence classes and uses symbolic inference to connect matching send and receive expressions. This analysis will be used to reduce the cost of receive matching and to detect collective communication patterns and replace them with implementations optimized for the target platform.



Send-Receive Fusion:

If a send and receive operation are executed in the same shared memory domain, their data packing and unpacking code can be fused so that data is copied directly from the sender's to the receiver's data structure. We are developing a compiler analysis to perform this fusion.

| Original Code | | Transformed Code |
|---|---|---|
| for(i=0; i<numAtoms; i++)<br>  if(borderAtom(i)) {<br>    buf[len] = atoms[i];<br>    len++;<br>    totalAtoms--;<br>  }<br>MPI_Send(buf, len); | MPI_Recv(buf, status);<br>MPI_Get_count(status, &len)<br>for(j=0; j<len; j++) {<br>  atoms[totalAtoms] = buf[j];<br>  totalAtoms++;<br>} | for(i=0; i<numAtoms; i++)<br>  if(borderAtom(i)) {<br>    remote->atoms[remote->totalAtoms] = atoms[i];<br>    remote->totalAtoms++;<br>    len++;<br>    totalAtoms--;<br>  } |

Lawrence Livermore National Laboratory

# Compiled MPI: Cost-Effective Exascale Applications Development

Greg Bronevetsky, Daniel Quinlan,
Peter Pirkelbauer, Chunhua Liao,
Andrew Friedley, Andrew Lumsdaine, and
Torsten Hoefler

## Bringing MPI to the Exascale era

- Multi-billion dollar investment in MPI apps

  Highly effective at getting petascale performance
- MPI is generally considered Dead…
- MPI is well-suited to Exascale systems
  - Communication is explicit
  - Distributed memory enforces locality by default
  - Various features to implement asynchronous algorithms
- The problem is **Today's** MPI implementations

## Developing runtime and compile-time techniques to ensure high MPI performance

- Runtime Thrust
  - Hierarchical MPI
  - MPI Optimizations
  - Detection and Synthesis of Collectives
- Compiler Thrust
  - Send-Receive Fusion
  - Buffer Structure Analysis
  - Communication Pattern Detection