LLNL-TR-638557

Compiling MPI for Many-Core Systems

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

June 6, 2013

Compiling MPI for Many-Core Systems

Abstract

Processors with multiple (or many) cores and shared memory are becoming ubiquitous across the computing spectrum. MPI, the current de facto programming model for scalable parallel applications, enforces copies between source and target processes and thus can not 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 present a compiler-based transformation that transforms MPI processes into threads and 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. We show the utility of our transformation with two applications, a molecular dynamics code, MiniMD, and a two-dimensional parallel fast Fourier transform (FFT). Our benchmarks show that our loop fusion techniques reduce communication times up to 43% for MiniMD and up to 59% for the FFT on modern multi-core systems. Our techniques will enable the automatic transformation of existing MPI codes to take advantage of modern shared memory architectures. In the future, this approach will aid in the transformation of MPI codes to a hybrid communication model that achieves high performance on a wide range of systems ranging from individual nodes to very large clusters of many-core nodes.

1. Introduction

The Message Passing Interface (MPI) is the de facto programming model for High Performance Computing (HPC), providing developers with a portable communication API that allows them to precisely direct an application's inter-process communication and synchronization. However, MPI was motivated by and optimized for a system model of distributed memory hardware running processes with separate address spaces. With the rise of multi- and many-core computing, an increasing fraction of cores are connected via high-speed shared cache and main memory. While MPI implementations have been written to take advantage of such hardware [6], MPI's distributed memory heritage has resulted in semantics that inherently limit the performance of these solutions relative to native shared memory programming models such as OpenMP [20].

The requirements of MPI semantics can be seen in a point-to-point message between a sender and receiver where MPI requires that (1) the sender's data structure be serialized into a buffer or represented as a (serializable) MPI datatype, (2) each point-to-point message be matched to a receive operation on the receiver, and (3) the message be transferred into a serial buffer or a memory region represented as an MPI datatype. While this decoupling of sender and receiver is appropriate for distributed memory systems, it is inefficient on shared memory hardware, which allows the sender's data to be directly transferred to the receiver (or directly accessed by the receiver). On the other hand, the communicating sequential processes (CSP) model underlying MPI is a powerful means of organizing parallel codes and for expressing parallelism. Further, because MPI's distributed memory model requires developers to explicitly specify data transfers, it is more natural to write applications with good locality, which is critical for achieving high performance on modern hierarchical memory systems and networks. MPI's clear specification makes it possible to develop compiler analyses and optimizations to improve communication performance. With the use of such analyses, MPI codes can maintain the expressive power of the CSP model without the limitations of MPI's message-passing semantics.

In this paper, we present a novel optimization of MPI communication on shared memory hardware that replaces MPI calls with direct load/store transfers from sender to receiver data structures. It consists of a compiler transformation and a runtime support library. Our runtime library assigns ranks to individual threads and combines all ranks on each node into a single process, enabling faster communication through a shared address space. Our compiler analysis examines the code used to serialize and deserialize application data structures into a communication buffer. It leverages the relatively simple access patterns used for communication buffers to detect if the loops transfer data from the sender's to the receiver's data structures, eliminating the MPI calls and the associated communication buffers. The code to access the sender's and receiver's data structures is left as it is, allowing the analysis to support arbitrary application data structures. While such transformations are equally applicable to code that explicitly serializes and deserializes communication buffers and code that uses MPI datatypes, this paper focuses on the former because it represents a more general compiler analysis challenge (MPI datatypes have clear semantics) and because explicit serialization has a higher cost than an optimized MPI datatype implementation.

Our experimental results show that loop fusion is highly effective at improving communication performance in existing MPI applications. It reduces the communication time in the MiniMD molecular dynamics program by 43% and 42% when executed 16-core Opteron 8356 and 12-core Intel Xeon X5660 nodes, respectively. The optimization reduces communication time in a 2-dimensional FFT application by 52% and 59% on the same platforms.

This paper is organized as follows. Section 2 surveys related work. Section 3 describes the design of our compiler analysis and transformation. Section 3.3 discusses how transforms code on one rank to be executed by other ranks. We then describe in Section 3.4.1 a simple transformation to fuse code that serializes and deserializes buffers based on only the control-flow information. This transformation is then refined in Section 3.4.3 to consider and preserve dataflow ordering between expressions in the two code regions, ensuring that the semantics of the original application are preserved. Section 4 then discusses the runtime components of our approach, including implementing MPI ranks using threads and how our transformation can be applied to collective operations that span multiple nodes. Section 5 shows the improvements in application performance due to the fusion transformation.

2. Related Work

There has been prior compiler work on detecting a variety communication patterns. One example is alignment of barriers for APIs where barriers may be either textually aligned [14] or unaligned [11] [26]. Another example is Shao et al [22], which supports more complex patterns but can only match `sends` to `receives` at runtime, when the number of processes is known.

`MPI-CFGs` [23] are an extension of standard control-flow graphs (CFGs), with additional edges between `MPI_Send` and `MPI_Recv` operations. The analysis connects all `MPI_Sends` to all `MPI_Recvs` and then uses sequential information such as mismatched tags or datatypes to prune edges that cannot represent real matches. Bronevetsky [5] describes a more general analysis that represents all the possible behaviors of a parallel application using a parallel control-flow graph and extends traditional dataflow analyses over this graph. Further, Danalis et al. have worked on compiler techniques to improve the performance of MPI applications, focusing on improving communication/-computation overlap via sequential tiling [1] and code motion [7].

Finally, there has been extensive work on runtime mechanisms for enabling MPI ranks executing on the same node to communicate to each other using shared memory. This include work on both kernel-level extensions [4, 13, 17] as well as MPI implementations that use threads as ranks such as TMPI [25] and Tern [12].

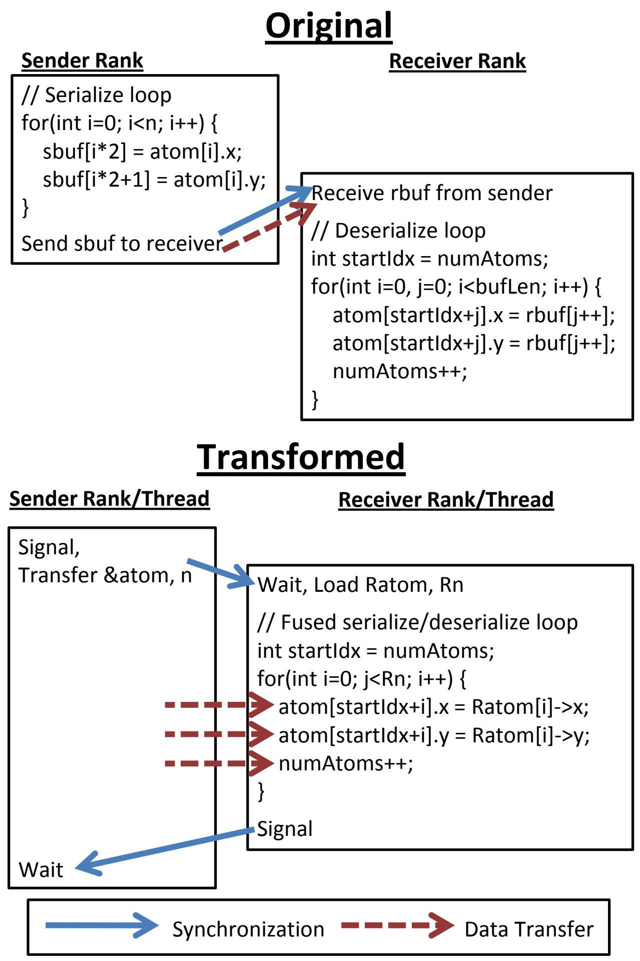
To our knowledge this paper is the first optimization for message passing applications that fuses code from multiple processes.

3. Compiler Analysis and Transformation

3.1 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 [2].

Figure 1 shows a motivating example for our transformation, extracted from the MiniMD benchmark (part of the Mantevo suite [16] and described in more detail in Section 5.1). 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.



```
// 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 1: Example of fusion serialize/deserialize code.

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 1, 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 for the fused loop to be 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. We use the above code sample from MiniMD as the running example through this paper.

3.2 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 exchanged 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 is described in Section 3.3. It 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.

The code fusion transformation is described in Sections 3.4. It 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.

Finally, Section 3.5 describes a few key normalization steps that support our analysis such as connecting matching sends and receives and identifying the serialization and deserialization loops themselves.

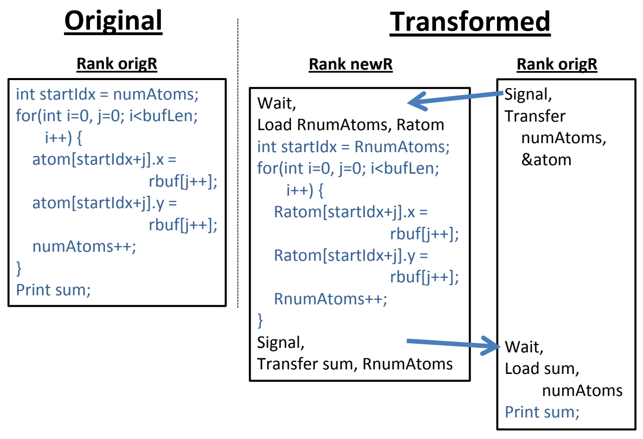
3.3 Rank Motion of Code

This transformation step ensures that both the serialize and deserialize code is executed on the same rank. It reduces the complexity of the compiler analysis because the code region becomes purely sequential. It takes code that is originally executed by rank origR and adjusts it so that it is executed by rank newR. The transformation uses liveness analysis [18] to identify the variables used in the migrated region and those modified by the region and used afterwards. It then transforms the application to utilize local copies of these used and modified variables. Finally, it adds additional communication to send the initial values of the variables used in the region from origR to newR before the migrated code executes and to send the variables modified in the region back when it finishes.

Figure 2 shows an example of this transformation, applied to the deserialization code from Figure 1. Here variables `numAtoms` and the pointer to `atom` are used and `sum` and `numAtoms` are modified. If MPI ranks on the same node are executed in a single process (runtime support for this is described in Section 4.1), they have direct access to each other's data structures. As a result, since our inserted communication transfers the initial value of `numAtoms` and a pointer to `atom`, the loop on rank newR is able correctly update the state of origR.

In general, only local variables that are used or modified in the migrated code region need to be explicitly transferred across ranks. There is no need to explicitly transfer heap memory because it is stored in shared memory and thus available to all threads from their existing pointers. The same is true for each thread's global variables because they must be stored in a heap-allocated table. Thus, by transferring local variables and accessing global variables through the table, the migrated code maintains the illusion that it is executing on rank origR. Note that although the migrated code also uses variables `rbuf` and `bufLen`, these are not transferred because they correspond to the original MPI communication and are therefore available at origR.

This transformation can be applied to migrate the deserialize code from the receiver to the sender rank or to migrate the serialize code from the sender to the receiver rank. Normally, reading is faster than writing, so when both the sender and receiver side perform the same number of memory accesses for serialization and deserialization, migrating serialization code to the receiver is preferred. However, the sender's serialization loop may have additional memory accesses (e.g., using a secondary array to look up array indices in the primary data structure), in which case moving the deserialization code to the sender will be preferred.



```
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++;
}
Print sum;
```

```
Wait,
Load RnumAtoms, Ratom
int startIdx = RnumAtoms;
for(int i=0, j=0; i<bufLen;
      i++) {
   Ratom[startIdx+j].x =
                     rbuf[j++];
   Ratom[startIdx+j].y =
                     rbuf[j++];
   RnumAtoms++;
}
Signal,
Transfer sum, RnumAtoms
```

```
Signal,
Transfer
   numAtoms,
   &atom

Wait,
Load sum,
      numAtoms
Print sum;
```

Figure 2: Example of code migrated across ranks.

3.4 Fusion of Serialize and Deserialize Code

After the above transformation both the serialization and deserialization code are executed by a single rank (either sender or receiver). The next compiler transformation fuses the serialization and deserialization code into a single loop, eliminating data transfer and serialized data buffers. Assuming that both loops iterate over the buffer in the same monotonic order, the transformation aligns the buffer writes and reads to directly transfer data from the sender's data structures into the receiver's using the application's own serialize logic but without any buffering.

The first component of the transformation simplifies the code. The code produced by Rank motion performs three copies. First, the serialization loop writes data to a communication buffer and may update a variable that holds the buffer's size. These variables come from the code on the sender. This data is then copied from the sender's variables to the corresponding buffer and size variable in the receiver's code, after which the deserialize loop is executed over these variables. The first transformation thus fuses the communication buffers and size variables into one, causing both loops to access the same memory.

The second transformation interleaves the iteration spaces of the two loops by managing dependencies between the writes of the serialize loop and the reads of the deserialize loop. The key intuition is that it is legal to move a given expression in the deserialization loop before one in the serialization loop if this does not change application semantics. For example, the size of the buffer is checked during every iteration of the deserialization loop to determine if it must exit. This check can only be moved into locations within the serialization loop (if any) where its outcome can never be affected. This is trivial if the buffer size is known at the start of the serialization loop but in many cases (e.g., our MiniMD example) it is computed during the loop. If this computation simply increases the size variable monotonically (true in almost every case) it is legal to move a check that the variable has reached a given value up to the point in the code where this value is first reached. For example, if the deserialize code checks whether `bufLen` $\geq$ `i` then it is legal to place this check immediately after the expression in the serialize loop that first sets `bufLen` to `i` or any larger value. Dependencies between communication buffer accesses are resolved similarly, where reads of buffer elements in the deserialization can be moved up to their last definition in the serialization loop.

A key challenge in this transformation is computing the mapping between iteration variables of the two loops to infer the ordering relationship between their expressions (e.g. mapping from expression `i` in MiniMD's serialization loop to expression `startIdx+j` in the deserialization loop). This can be done using any symbolic abstraction of application state. In this paper we focus on a linear abstraction, such as that used in polyhedral models, because it provides a good balance of simplicity and generality. As such we consider expressions such as `buf[ax+b]` or `x < ay + b` for any integer constants `a` and `b` and variables `x` and `y`.

Figure 3 provides an example of how expressions in sample executions of serialize and deserialize loops can be reordered while preserving application semantics. The serialize loop packs two elements into `buf` and increments `bufLen` to 2. The deserialize loop unpacks until its iterator variable `i` reaches `bufLen`. The loops have been unrolled to clarify presentation. The example shows that when `i=0` it is possible to move the conditional `i<bufLen` only upto the expression `bufLen++` that sets `bufLen` to 1 since this preserves the conditional's original outcome of True. This in turn ensures that the reads of `buf[bufLen]` in the deserialize code are only executed after this buffer location has been assigned. While this illustration focuses on the application's dynamic execution, the sections below explain how the compiler can perform the same type of reasoning symbolically.

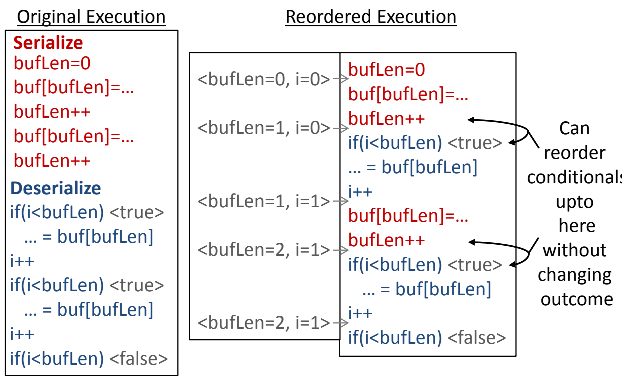


Figure 3: Example of a legal code reordering.

The algorithm will be presented in three steps. First, Section 3.4.1 presents a simple algorithm that fuses multiple CFGs based on their control structure but without taking the application's dataflow into account. Next, Section 3.4.2 summarizes symbolic dataflow analyses. Finally, Section 3.4.3 combines the two concepts into an integrated, dataflow-sensitive algorithm for fusing serialize and deserialize code.

3.4.1 Fusion of Finite Automata

The simplest way to fuse two pieces of code is to represent them as finite automata, the behaviors of which are defined by their control-flow graphs (CFGs). We then fuse the CFGs by directly extending the well-known algorithm for creating product finite automata [10]. Each node of the fused CFG is a pair of CFG nodes, denoted $\langle serN, deserN\rangle$, where $serN$ is in the serialize CFG and $deserN$ is in the deserialize CFG. There is a valid edge from one fused CFG node $\langle serN, deserN\rangle$ to another node $\langle serN', deserN'\rangle$ if there exists an edge from $serN$ to $serN'$ or $serN = serN'$, and the same for $deserN$.

The algorithm starts with a work list that contains CFG node $\langle serN_{start}, deserN_{start}\rangle$, where $serN_{start}$ and $serN_{start}$ are the starting nodes of the respective CFGs. At each step it takes a graph node that corresponds to some position in the two CFGs and identifies the states that can be reached if the application makes a single transition in either CFG. Consider the two loop CFGs shown in Figure 4, which are skeletons of the MiniMD loops in Figure 1, containing a loop entry point, a body and an exit point. If the application is at node $\langle A, \beta\rangle$ then either

- Loop 1 transitions from CFG node A to node B, in which case the fused application transitions from $\langle A, \beta\rangle$ to $\langle B, \beta\rangle$,
- Loop 1 transitions from A to C, and the fused application transitions to $\langle C, \beta\rangle$, or
- Loop 2 transitions from β to α, and the fused application transitions to $\langle A, \alpha\rangle$.

The process is repeated using a standard worklist algorithm until it generates all possible transitions in the fused CFG. Each edge in this CFG is annotated with the corresponding transition in the original CFGs. The fused CFG is synthesized into source code by treating it as a regular CFG where the edge annotations identify operations and the paths between them specify control flow.

Figure 4 shows these loops are fused. The fused loop begins at the entry node $\langle A|\alpha\rangle$. One possible transition is for either loop to enter its body, leading to nodes $\langle A|\beta\rangle$ and $\langle B|\alpha\rangle$. From there, if the other loop enters its body, the fused application reaches node $\langle B|\beta\rangle$, where both loops are executing their bodies. The only transitions from this node correspond to either loop returning to its entry point. Another option is for either loop to transition from its entry node (A or α) to the loop's exit node (C or γ), in which case the fused CFG transitions to the sub-graphs in the lower left and right corners. Once there, the completed loop stalls at its exit node until the other loop completes and the fused application reaches its final node $\langle C|\gamma\rangle$.

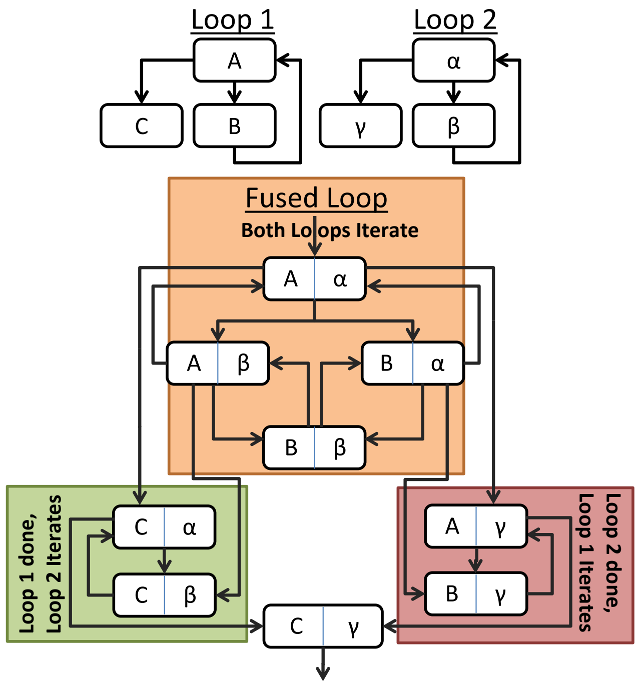


Figure 4: Example of fused control-flow graphs

3.4.2 Symbolic Dataflow

This section summarizes the key ideas of symbolic dataflow analysis, which will make it easier to understand how these concepts are applied in our analysis.

Dataflow analysis is a fixed-point iteration over a space of possible facts about each CFG node. The algorithm starts with no information about each node and iterates by accumulating all the assertions that may be true about the application's state at each node until it reaches a fixed point where no additional assertions can be discovered. The designer of a given dataflow analysis must specify an abstraction that captures assertions about the application state (denoted the "abstract state"). For example, an abstraction of a constant-propagation analysis contains either each variable's known constant value if it is initialized or a special symbol $\perp$ if it is not. Further, if the variable may have more than one value the abstraction contains special symbol $\top$. More sophisticated abstractions represent application state using predicate or first-order logic. The effect of expressions is modeled by a "transfer" function that maps the abstract state before any expression to the state after it. For example, if before expression `i++` it is known that `i==var` then afterwards it is known that `i-1==var`. Control flow is captured via a "meet" function that computes the union of abstract states along multiple control paths. For example, if at the end of one branch of a conditional it is known that `i==5` and after the other `i<10`, the strongest fact known immediately after the conditional is `i < 10`.

The dataflow algorithm monotonically increases the set of possible facts known at each dataflow node until it converges to the strongest set of facts that are true of all possible executions.

Figure 5 presents an example of symbolic dataflow analysis applied to the fused code from Figure 1. The abstraction application state maintains the linear relationship between each pairs of variables. The key parts of the CFG are shown in solid boxes and dashed boxes denote the dataflow states computed between each pair of nodes. Since these states evolve during the algorithm's execution, the CFG is shown twice and states computed later in the algorithm are shown in the bottom copy. Dashed arrows identify which states are used to compute which other states.

We know that `[i==0` $\wedge$ `startIdx==numAtoms]` after the intial node `A`. When this state is propagated through the loop's iteration condition it is combined with `i` $\geq$ `Rn` along the loop's exit edge or `i < Rn` along the loop body edge. When the latter is propagated through the body, it is updated from the state `[i==0` $\wedge$ `startIdx==numAtoms` $\wedge$ `i<Rn]` to `[i==1` $\wedge$ `startIdx== numAtoms-1` $\wedge$ `i<Rn-1]` by the increment operations. This state is propagated back to the top of the loop, where it is unioned with the previous state at this CFG location. This symbolic union is computed as follows. The union of `i=0` and `i=1` cannot be represented precisely as a single linear constraint and is conservatively approximated by `i` $\geq$ `0`. No relationship is known between `i` and `Rn` before the loop's start so nothing further can be inferred about their relationship. Finally, `startIdx==numAtoms-i` is true in both states, since `i=0` in the initial state above the loop. Thus, the resulting constraint is `[i` $\geq$ `0` $\wedge$ `startIdx==numAtoms-i]`.

This constraint is propagated again the CFG and it is easy to confirm that no additional facts can be derived from it. Therefore it is the fixed-point solution of the dataflow equations. We can infer that at the end of the loop `[startIdx==numAtoms-i` $\wedge$ `i` $\geq$ `Rn]` and thus `numAtoms-startIdx` $\geq$ `Rn`. Further, we can use the relationship between `numAtoms` and `i` to eliminate `i` from the loop and use `numAtoms` as the only iteration variable.

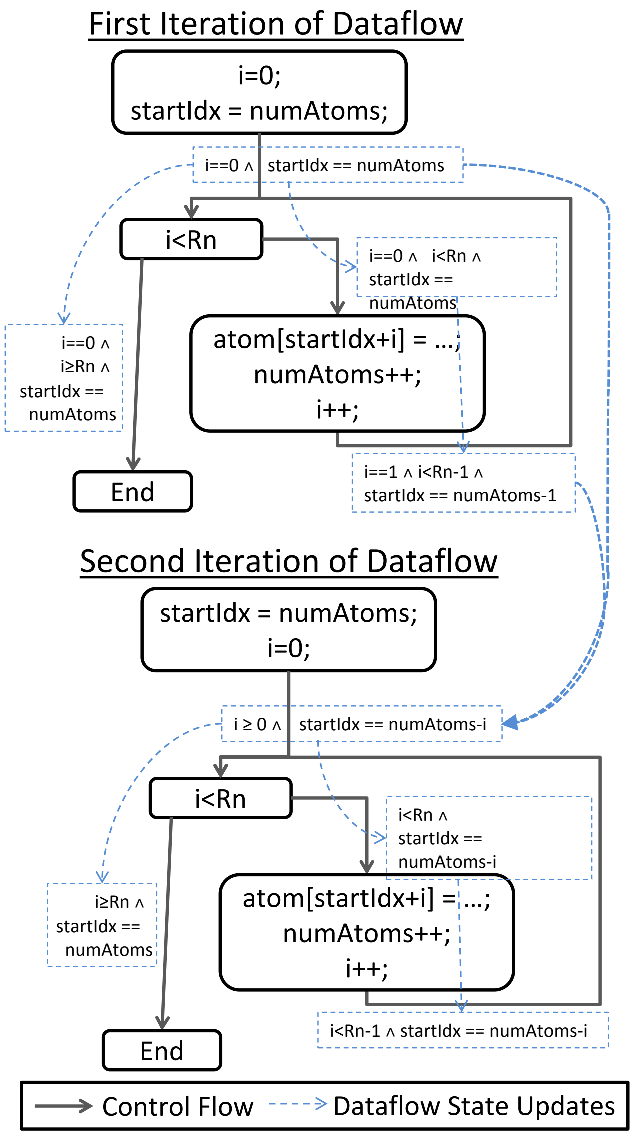

First Iteration of Dataflow
i=0;
startIdx = numAtoms;
i==0 ∧ startIdx == numAtoms
i<Rn
i==0 ∧ i<Rn ∧ startIdx == numAtoms
i==0 ∧ i≥Rn ∧ startIdx == numAtoms
atom[startIdx+i] = ...;
numAtoms++;
i++;
End
i==1 ∧ i<Rn-1 ∧ startIdx == numAtoms-1
Second Iteration of Dataflow
startIdx = numAtoms;
i=0;
i ≥ 0 ∧ startIdx == numAtoms-i
i<Rn
i<Rn ∧ startIdx == numAtoms-i
i≥Rn ∧ startIdx == numAtoms
atom[startIdx+i] = ...;
numAtoms++;
i++;
End
i<Rn-1 ∧ startIdx == numAtoms-i
Control Flow
Dataflow State Updates

Figure 5: Example of symbolic dataflow analysis.

3.4.3 Symbolic Fusion of Serialize and Deserialize Code

We now present the full algorithm to fuse serialize and deserialize loops. The above CFG fusion transformation generates illegal execution paths because it only considers control dependencies and ignores data dependencies. We prevent it from generating dependence-breaking paths by using a symbolic dataflow analysis to resolve these dependencies. The intuition is that unless otherwise known, the only safe way to generate the fused CFG is to first have it iterate over the serialize loop and then the deserialize loop. Thus, as the CFG fusion algorithm generates possible paths, it will only transition along a node within the deserialize loop's CFG is if the dataflow analysis can prove that this node's expression only depends on actions that the serialize loop has already performed. In other words, when it considers the fused node ⟨ `serN, deserN` ⟩, it generates an edge to ⟨ `serN`, `deserN`$_{\texttt{successor}}$ ⟩ if `deserN` is:

- A read access to an expression that is not changed by the serialize loop.
- A conditional that checks whether a monotonically increasing (or decreasing) variable is greater (or smaller) than some variable or quantity that doesn't change during the loop and it can be proven that this condition is satisfied at the current node in the serialization loop. Since the variable is changing monotonically, if such a condition is satisfied once, it will be satisfied for the remainder of the application's execution.
- A read access to an entry of the communication buffer and
  - The order in which the communication buffer is written is monotonic, and
  - All expressions that write to the buffer must to write either to earlier indexes (later, if monotonically decreasing) or exactly the same index (ensures that the serialization loop has filled enough of the buffer for this deserialization read to access the correct value), and
  - There is at least one write expression that targets the same index as the read (these expressions will be fused with the read expression).

Finally, if the fusion algorithm reaches a node where the dataflow state is provably inconsistent (e.g. the combination of conditionals around the node imply False), the node and its successors are not included in the fused CFG.

Once the fused CFG is generated, we fuse the read and write expressions that are proven to always access the same index of the communication buffer. For each set of such matching reads and writes we generate a temporary variable and replace buffer accesses with equivalent accesses to this variable. This copy through a register can be optimized into a direct copy by the back-end compiler.

The analysis has two pre-conditions. First, loop iteration variables and index expressions in accesses to the communication buffer must either increase or decrease monotonically. Second, there cannot be a path from one write access to a given communication buffer index and another write to potentially the same index (i.e. each index must be written at most once). These can be verified using a polyhedral analysis [3], combined with a simple reachability analysis.

```
if(is Sender) {
  // Serialize loop
  int len=0;
  for(int i=0; i<n; i++) {
    if(atom[i] is in boundary) {
      buf[len++] = atom[i].x;
      buf[len++] = atom[i].y;
    }
  }

  // Deserialize loop
  int Rj = 0;
  while(Rj<bufLen) {
      Ratom[RnumAtoms].x = buf[Rj++];
      Ratom[RnumAtoms].y = buf[Rj++];
      RnumAtoms++;
  }
```

Figure 6: Separate serialize/deserialize loops.

```
int len=0;
int j=0;
while(i<n) {
  if(Atom[i] not on boundary) {
    // Atom not on boundary, serialize loop
    // iterates, deserialize loop does not
    i++;
  } else {
    double T0 = atom[i].x;
    double T1 = atom[i].y
    Ratom[RnumAtoms].x = T0;
    Ratom[RnumAtoms].y = T1;
    i++;
    RnumAtoms++;
  }
}
// Both loops terminate, transfer complete
```

Figure 7: Fused serialize/deserialize loop

Figure 6 shows the serialize and deserialize loops of another loop from MiniMD, which is slightly more complex than the example in Figure 1. The serialize loop iterates over all the atoms at the sender and sends out just the atoms on the border of its space, computing the size of the buffer during the course of the loop. The figure shows the two loops after the code motion transformation and Figure 7 shows the results of fusing these two loops. The variables from the receiver rank have an R prepended to their names to clearly identify each loop. The fused loop iterates over the sender's atoms and copies each border atom into the next slot in the receiver's atom array, starting with index `numAtoms` and updating it after adding each new atom. Figure 8

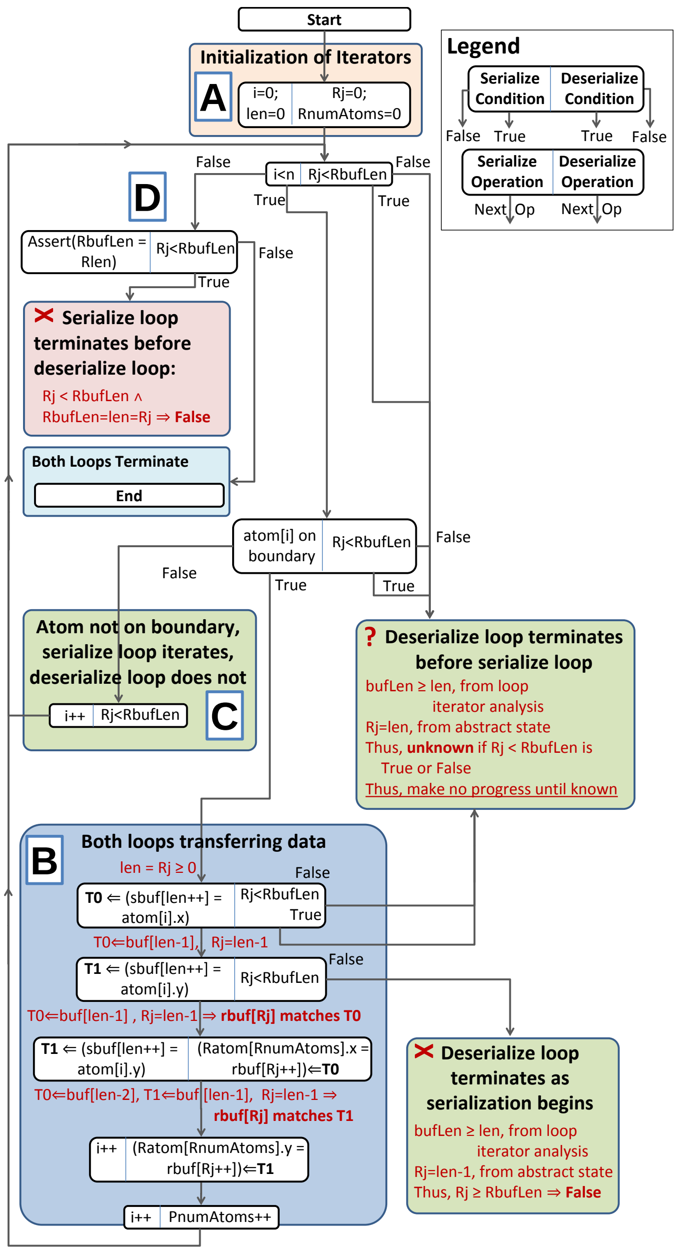

Start
Initialization of Iterators
A
i=0; len=0
Rj=0; RnumAtoms=0
Legend
Serialize Condition
Deserialize Condition
False
True
True
False
Serialize Operation
Deserialize Operation
Next Op
Next Op
False
i<n
Rj<RbufLen
False
D
True
True
Assert(RbufLen = Rlen)
Rj<RbufLen
False
True
Serialize loop terminates before deserialize loop:
Rj < RbufLen ∧ RbufLen=len=Rj ⇒ False
Both Loops Terminate
End
atom[i] on boundary
Rj<RbufLen
False
False
True
True
Atom not on boundary, serialize loop iterates, deserialize loop does not
i++
Rj<RbufLen
C
? Deserialize loop terminates before serialize loop
bufLen ≥ len, from loop iterator analysis
Rj=len, from abstract state
Thus, unknown if Rj < RbufLen is True or False
Thus, make no progress until known
B
Both loops transferring data
len = Rj ≥ 0
False
T0 ⇐ (sbuf[len++] = atom[i].x)
Rj<RbufLen
True
T0⇐buf[len-1], Rj=len-1
False
T1 ⇐ (sbuf[len++] = atom[i].y)
Rj<RbufLen
T0⇐buf[len-1] , Rj=len-1 ⇒ rbuf[Rj] matches T0
T1 ⇐ (sbuf[len++] = atom[i].y)
(Ratom[RnumAtoms].x = rbuf[Rj++])⇐T0
T0⇐buf[len-2], T1⇐buf[len-1], Rj=len-1 ⇒ rbuf[Rj] matches T1
i++
(Ratom[RnumAtoms].y = rbuf[Rj++])⇐T1
i++
PnumAtoms++
Deserialize loop terminates as serialization begins
bufLen ≥ len, from loop iterator analysis
Rj=len-1, from abstract state
Thus, Rj ≥ RbufLen ⇒ False

Figure 8: Example of fused serialize/deserialize loop

shows the fused loop's CFG, along with the dataflow state at each node.

The legend explains how each node's successors are displayed. For a given node ⟨ `serN`, `deserN` ⟩, if neither `serN` or `deserN` are conditionals the successor node corresponds to execution of either serN or `deserN`. Otherwise, the Figure shows the successors that correspond to the conditional's true and false outcomes (edges are marked "True" and "False", respectively). At key edges of the graph we show linear relationships inferred by the symbolic analysis.

The region labeled A corresponds the serialize and deserialize loops initializing their variables. Control then proceeds to node ⟨ `i<n`, `Rj<RbufLen` ⟩, which corresponds to the top of both loops. Because at this point in the code the serialize loop's condition `i<n` has not been resolved, the outcome of `Rj<RbufLen` is also unknown, meaning that the fusion algorithm can only make progress on the serialize loop. If the conditions `i<n` is "`atom[i]` is on the boundary" evaluate as true control enters region B, where the algorithm can safely make progress in the deserialize loop without violating dependencies. After this region of the fused CFG is computed it is possible to show that the write and read expressions marked with `T0` always access the same index of the communication buffer because `len` at the write is equal to `Rj` at the read. The same is true of the expressions marked with `T1`. These pairs of expressions will be fused and their data transferred through temporary variables `T0` and `T1` in the generated code shown in Figure 7. Note that the algorithm doesn't generate a path for the case where these conditions hold but the deserialize loop exits because this would imply an impossibility, that `j` $\geq$ `bufLen` when it is also known that `j` $<$ `bufLen`.

If the condition `i` $<$ `n` evaluates to true but "`atom[i]` is on the boundary" does not, this leads to region C. The only legal path from this region returns to the fused node at the top of both loops.

Finally, if condition `i` $<$ `n` evaluates to false, this leads to region D. The algorithm does not generate a path condition `Rj<RbufLen` evaluates to True because this is inconsistent with the fact that `j==bufLen` at the end of the serialize loop. As such, the only legal path is to the state where both the serialize and deserialize loops have exited.

3.5 Supporting Normalization Steps

Our analysis needs to know which send and receive operations in the source code must always match and how message data is serialized and deserialized.

A "send operation" (`MPI_Send`, `MPI_Isend`, etc.) matches a "receive operation" (`MPI_Recv` or `MPI_Irecv`) in the source code if in all possible executions the data in the send buffer will be communicated to the receive buffer. Non-blocking sends are considered to have occurred at the source code location of the `MPI_Isend`. Non-blocking receives are considered to have occurred at the source code location of its corresponding MPI_Wait or successful MPI_Test. The discovery of such information is orthogonal to this paper and has been addressed in prior work [5]. As such for the purposes of this analysis we assume that it is provided by such an analysis or via source-code annotations. Further, collective operations are interpreted as the equivalent set of sends and/or receives that would perform the same operation (e.g., MPI_Bcast on the root rank is a send, and on all other ranks is a receive).

To identify the application code that serializes and deserializes communication buffers our analysis needs to know the locations in the source code in which no MPI messages are posted (denoted "quiescence lines") and thus define boundaries between code regions. This information can also be obtained via prior work [5] or annotations. We use quiescence lines to partition the application code into regions that correspond to a single communication. Within each region we identify the receive operations and employ define-use analysis [18] to identify the reads from the receive buffer. We then denote all the operations between the receive operations and these reads as the "deserialize code". We then examine the region's send operations and use define-use analysis to identify the last writes to their send buffers. Again, we denote the operations between these writes and the send as the "serialize code". The serialize may write complex data into the send buffer that involves both copies and computations and the deserialize code may similarly incorporate the received data into its own state in complex ways. The only invariant that the subsequent analysis steps both require and verify is that both code regions access their respective communication buffers in *the same monotonic order*.

Finally, the analysis performs a code normalization step. If there are any conditionals that may lead control from the serialize code to anywhere other than the send operations or conditionals that may lead control from the receive operations to anywhere other than the deserialize code, these conditionals are expanded to include both the serialize code and the sends and/or the receives and the deserialize code. This is done to simplify control flow to enable the analysis to treat all the serialize and deserialize code as a simple code sequence.

4. Runtime Components

This section describes the runtime components that support our compiler transformations, enabling different MPI ranks to directly access each other's memory. We also describe how to handle collective communication that cross node boundaries in a portable way that avoids additional memory copies.

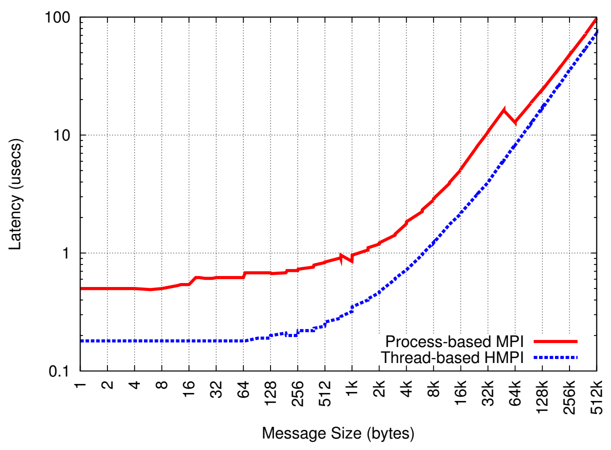
4.1 Hybrid MPI

Loop fusion requires MPI ranks to share a single address space so that they can access one another's data structures. However, MPI implementations normally assign each rank its own process, resulting in disjoint address spaces. Instead, we can assign each rank to its own thread and collect all ranks (threads) on a node into a single process with one address space. 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. Transforming a program to use HMPI can be done by the following simple steps:

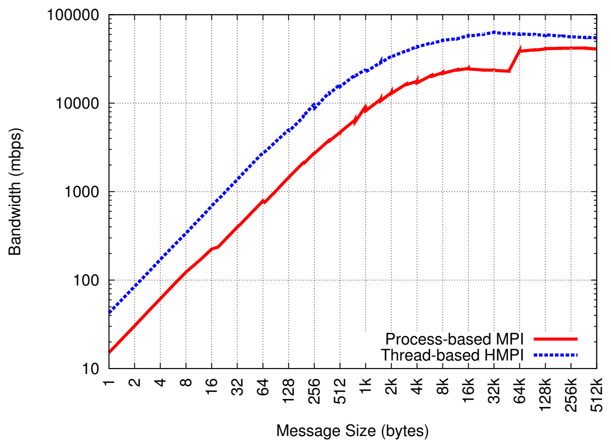
1. Rename the original main function to tmain and remove the call to MPI_Init{_thread}.
2. Create a new main function that calls HMPI_Init, which will in turn start threads using tmain.
3. Replace all MPI calls with calls to equivalent HMPI functions.
4. Privatize all global variables.

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 [19].

Figure 9 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.



(a) Latency



(b) Bandwidth

Figure 9: NetPIPE same-node latency comparison between MVAPICH2 1.6 and our thread-based Hybrid MPI library.

4.2 Collective Operations

Collective operations specify collective data movements (or reductions) across a set of processes. They present additional challenges to our framework. In the case of point-to-point operations communication is either within a node or across node. Loop fusion is used with any on-node communication where it is applicable and in all other cases we communicate using the native MPI implementation. In contrast, a single collective operation includes both communication within nodes and across nodes. To support general MPI applications it is necessary to implement collectives in a partitioned fashion, utilizing fused communication for the on-node portions of each collective and MPI operations for the remaining transfers.

This section explains how this is done for the representative case of alltoall operations. Without loss of generality, we assume that the number of ranks within all HMPI process is the same (e.g., nodes with identical multicore processors), with p total ranks t ranks sharing a memory domain. The HMPI transformation would run the p ranks with p/t MPI processes and t threads each. Each set of t serialize/deserialize loops on the p/t different processes is fused. After the fusion, a global alltoall on the p/t MPI processes permutes the data globally, excluding the pieces local to each node. The additional local copies are excluded by constructing an MPI datatype that omits the parts of the buffers that are local in the sending phase and re-orders the buffers correctly at the receiver for the overall alltoall communication among p/t processes. We assume that every process communicates a single element of a datatype with the same size. Multiple elements can be handled by representing them as a single contiguous MPI datatype.

At all senders in block `i` ($0 \leq i < p/t$), we create an hindexed type (`MPI_Type_create_hindexed`) with blocklength t that spans all t buffers (arrays local to threads). An hindexed type describes the data layout as a list of basic data blocks with a relative offset in bytes. We have to use an hindexed type because each sent element is in a different (separately allocated) array. We then call alltoallv with the correct offsets (to omit block `i` on process `i` as shown in Figure 10). At each receiver, the incoming data is transposed and written to the t receive buffers, which is done by receiving the data using an hindexed datatype with t blocks of size t and the correct buffer offsets. The hindexed type for receive is equivalent to a transposed version of the send type.

Figure 10 shows the created datatype and a subset of the communication of alltoall on 8 processes with two processes sharing a shared memory domain. The HMPI processes are shown at the top of the figure and the MPI processes on the bottom. The arrows indicate parts of the data movement and the crossed out ranks don't need to communicate in the MPI process.

Other collective operations can be handled accordingly in the transformation phase. Using datatypes and the vector variants of the collective operations, we minimize the internal data copies for off-node communication and effectively enable zero-copy mechanisms if supported by the hardware [9].

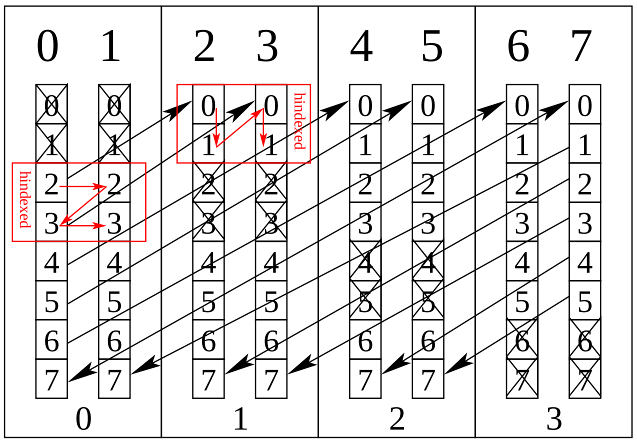


Figure 10: Example of fused Alltoall for $p = 8$ and $t = 2$.

5. Experimental Evaluation

To demonstrate the effectiveness of our loop fusion optimization, we have transformed two applications (MiniMD and FFT2D) by hand and measured the change in performance. The first step is to transform the applications to use our HMPI library using the steps listed in Section 4.1. Finally we applied loop fusion, and have shown that in general, it yields significant speedups.

Experimental results were obtained using the LLNL Hera and Sierra systems. Hera has 16 2.3 GHz Opteron 8356 cores and 32 GiB of RAM, while Sierra has 12 2.8GHz Xeon X5660 cores and 24 GiB of RAM. MVAPICH2 v1.6 was used for all results. All figures show performance for varying numbers of ranks executed on a single node.

5.1 miniMD

MiniMD is part of the Mantevo [16] mini-application suite, which consists of several important application kernels distilled into smaller benchmark-sized programs. It is a molecular dynamics simulation that computes atom movement over a 3D space decomposed into a processor grid. Molecular dynamics is an important application area for loop fusion because it simulates long-duration phenomena. Thus, the primary use for additional cores provided by future HPC systems will be to reduce the execution time of each time step for a fixed-size problem. This strong-scaling problem will be increasingly bound by communication costs.

The primary MiniMD work loop performs the following steps during each iteration:

1. Every 20th iteration, migrate atoms to different ranks depending on atom locations.
2. Exchange position information of atoms in boundary regions to neighboring ranks.
3. Compute forces applied to both local atoms and those in boundary regions from neighboring ranks.
4. Exchange force information of atoms in boundary regions to neighboring ranks.
5. Update local atom velocities and positions.

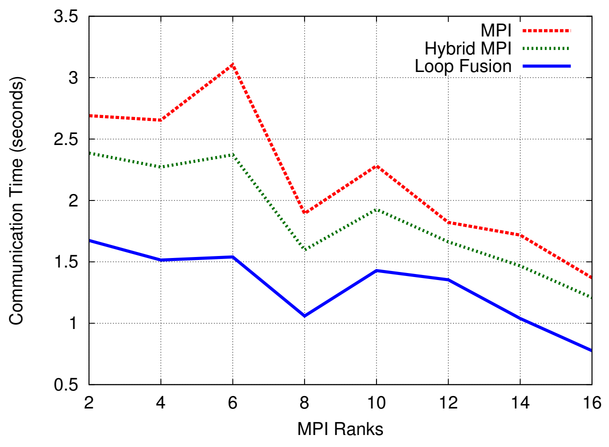
We have applied the loop fusion transformation to steps (2) and (4), which send and receive data between neighboring ranks. For our purposes these exchanges perform the same operation. Each rank serializes outgoing atom information into a buffer and sends it, repeating for each of six neighbors. Incoming atom information is received and then deserialized back into the main atom list. We eliminate the intermediate buffers and fuse the serialization loops together for neighbors residing on the same node.

When loop fusion is performed, we have the option of placing the fused loop at the sender or receiver. MiniMD, in particular, uses a secondary array of indices that identifies which elements out of its main atom list should be shared with a particular neighbor. In step (2), this means the serialization loop is a gather operation; the deserialization loop is a straightforward copy. Step (4) moves data in the reverse direction, so the serialization loop is the copy and the deserialization loop is a scatter operation. We found that the best performance is attained by placing the fused loop on the rank doing the scatter or gather operation. Thus, step (2) writes to the receiver while step (4) reads from the sender.

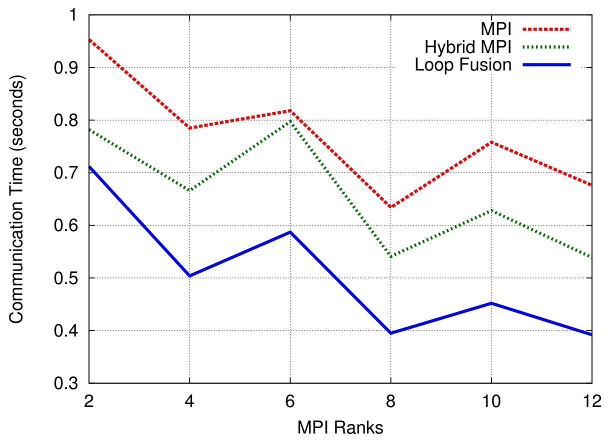
MiniMD's communication performance is shown in Figure 11. It presents the original MPI based code, a version modified to use HMPI, and the HMPI version with loop fusion. A fixed problem size of 20x20x20 was used for strong scaling, and we did 2,000 iterations per run. The HMPI code is comparable to the original MPI version, frequently performing better. Loop fusion provides varying but generally significant speedup in all cases – up to 43% reduction in communication time on 16 Hera cores and 42% on 12 Sierra cores.

5.2 Fast Fourier Transformation

Fast Fourier Transforms (FFT) are among the most important operations in use today. Numerous algorithms and parallel applications use FFTs in their core computations [8, 15]. A one-dimensional FFT trans-



(a) Hera System



(b) Sierra System

Figure 11: MiniMD communication time; lower is better.

forms a one-dimensional array of N complex numbers from real space to N complex numbers in frequency space. Such a one-dimensional FFT can be expressed in terms of multi-dimensional FFTs with additional application of *twiddle factors* [21, §12]. A multi-dimensional FFT with d dimensions can be computed by applying one-dimensional FFTs in all d dimensions. Multi-dimensional FFTs are very important in practice; image analysis often requires two-dimensional FFTs and transformations in real-space require three-dimensional FFTs [8, 15]. Since FFTs transfer large amounts of data, they are communication-bound and thus are an excellent candidate for the loop fusion optimization.

We perform our experiments using a 2-dimensional FFT kernel. The original 2D FFT code is implemented using MPI and transforms a $N_x \times N_y$ domain. The initial array is stored in x-major order and distributed in y-dimension such that each process has N_x/P y-pencils. The steps to perform the two-dimensional FFT are:

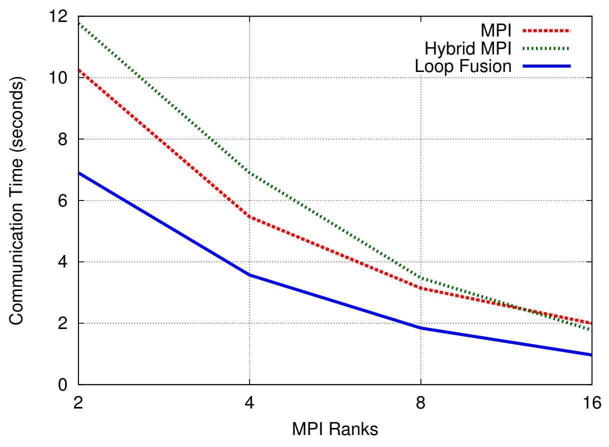
1. Perform N_x/P 1D FFTs in y-dimension (N_y elements each).
2. Serialize the array into a buffer for the all-to-all.
3. Perform a global all-to-all.
4. Deserialize the array to be contiguous in the x-dimension (each process now has N_y/P x-pencils).
5. Perform N_y/P 1D FFTs in the x-dimension (N_x elements each).
6. Serialize the array into a sendbuffer for the all-to-all.
7. Perform a global all-to-all.
8. Deserialize the array into its original layout.

Due to unsatisfactory performance in the MPI's datatype implementation, we have implemented our global alltoall manually instead of using the datatype-based approach explained in Section 4.2. However, we replicate what we would expect an optimized MPI implementation (using datatypes) would do:

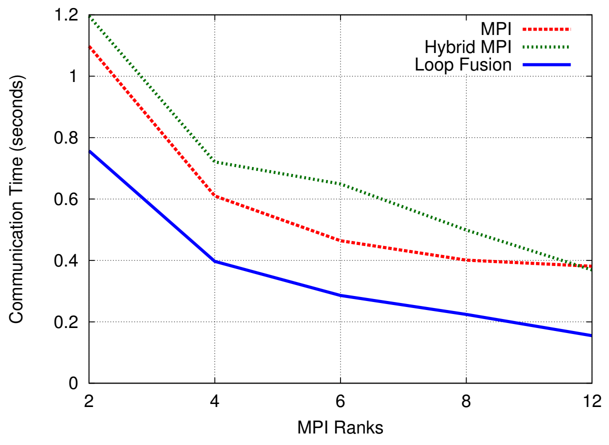
1. One rank on each node posts receives for one rank on every remote node in a round-robin fashion starting with the node after it.
2. Each rank on a node copies its send buffer into a shared send buffer using a stride to place the data in the correct location for each remote node.
3. One rank posts a send to each remote node in round-robin fashion using the respective portions of the shared send buffer.
4. Each rank copies (reads) the data it should receive from other local ranks, taking advantage of the shared address space.
5. One rank waits for the remote receive operations to complete, then signals other local ranks.
6. Each rank copies (reads) data from the shared receive buffer into its local receive buffer.

In original and HMPI versions of the FFT code presented here, serialization and deserialization are performed around the alltoall communication. For data that is destined for a rank on the same node, we can perform loop fusion so that each rank writes directly to the correct location in local ranks' memory. The fused serialization/deserialization loop replaces step (4) in the alltoall communication described above.

Two optimizations to the fused loop are possible. We avoid contention by having each rank write to other ranks in a round-robin fashion starting from itself, rather than having each rank first write to the



(a) Hera System



(b) Sierra System

Figure 12: FFT2D communication time; lower is better.

first thread on the node. Second, we observe that we are reading across a two-dimensional space. Improved cache locality is obtained by 'blocking' the fused loop in the same manner as is done for matrix multiplication. These optimizations can be done by traditional cache-oriented polyhedral transformations [24].

Figure 12 shows FFT communication performance for the original MPI-based code, a version modified to use HMPI, and then with loop fusion optimization. Strong scaling is shown; a problem size of $8{,}192^2$ was used on Hera while $6{,}144^2$ was used on Sierra. All intermediate buffers as well as the global alltoall operation are eliminated by loop fusion; the transposition of the data is performed in parallel across all threads at local memory copying speeds. The result is that reducing FFT's communication time on 16 Hera cores is reduced by 52% and on 12 Sierra cores by 59%.

6. Conclusions and Future Work

This paper presents a novel compiler-based optimization for MPI applications that converts MPI code that executes on shared-memory hardware to directly transfer application data structures from senders to receivers without the cost of message serialization, deserialization and storage. It leverages the fact while the application's data structures may be complex, the code to serialize and deserialize them typically uses a simple linear iteration through the communication buffer. This makes it possible to align the serialization writes in the sender's source code with the deserialization reads in the receiver's code to directly transfer the sender's data structures to the receiver's regardless of their complexity. We combined our transformation with an implementation of MPI that implements MPI ranks as threads, enabling different ranks on the same node to directly access each other's memory. Our experiments demonstrate that the optimization significantly improves the communication performance of the miniMD molecular dynamics benchmark and a 2D fast Fourier Transform benchmark, on two different node architectures. This shows both the utility of our particular transformation as well as the promise of using compilers to enable applications parallelized using MPI to take full advantage of multi- and many-core hardware.

Looking forward, this work points the way towards additional compiler tools for MPI. The greatest power of our approach is to enable application developers to use MPI's explicit parallel programming model to describe their problem decomposition and data motion, and rely on the compiler and the MPI runtime to cooperatively map this specification to complex hierarchical nodes and networks.

References

[1] L. Pollock A. Danalis, K. Y. Kim and M. Swany. Transformations to parallel codes for communication-computation overlap. In *ACM/IEEE Supercomputing Conference (SC)*, November 2005.

[2] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS High-Performance Interconnect. In *Proceedings of 18th Symposium on High-Performance Interconnects (Hot Interconnects 2010)*. IEEE, Aug. 2010.

[3] Bastoul. Improving data locality in static control programs. Technical Report PhD Thesis, University Paris 6, Pierre et Marie Curie.

[4] Ron Brightwell. Exploiting direct access shared memory for mpi on multi-core processors. *International Journal of High Performance Computing Applications*, 24:69–77, February 2010.

[5] Greg Bronevetsky. Communication-sensitive static dataflow for parallel message passing applications. In *International Symposium on Code Generation and Optimization (CGO)*, March 2009.

[6] Darius Buntinas and Guillaume Mercier. Implementation and shared-memory evaluation of mpich2 over the nemesis communication subsystem. In *Proceedings of the Euro PVM/MPI Conference*. Springer, 2006.

[7] A. Danalis, L. Pollock, M. Swany, and J. Cavazos. MPI-aware compiler optimizations for improving communication-computation overlap. In *International Conference on Supercomputing (ICS)*, June 2009.

[8] X Gonze, G Rignanese, M Verstraete, J Betiken, Y Pouillon, R Caracas, F Jollet, M Torrent, G Zerah, M Mikami, and et al. A brief introduction to the abinit software package. *Zeitschrift fr Kristallographie*, 220(5-6-2005):558–562, 2005.

[9] T. Hoefler and S. Gottlieb. Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 132–141. Springer, Sep. 2010.

[10] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

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

[12] Jian Ke and Evan Speight. Tern: Thread migration in an mpi runtime environment. Technical Report CSL-TR-2001-1016, Cornell Computer Systems Laboratory, November 2001.

[13] Hyun-Wook Jin, Sayantan Sur, Lei Chai, and Dhabaleswar K. Panda. LiMIC: Support for high-performance mpi intra-node communication on linux cluster. In *International Conference on Parallel Processing (ICPP)*, 2005.

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

[15] S. Kumar, C. Huang, G. Zheng, E. Bohm, A. Bhatele, J. C. Phillips, H. Yu, and L. V. Kalé. Scalable molecular dynamics with namd on the ibm blue gene/l system. *IBM J. Res. Dev.*, 52:177–188, January 2008.

[16] Sandia National Lab. Mantevo Benchmark Suite. https://software.sandia.gov/mantevo.

[17] Teng Ma, George Bosilca, Aurlien Bouteiller, Brice Goglin, Jeffrey M. Squyres, and Jack J. Dongarra. Kernel Assisted Collective Intra-node Communication Among Multicore and Manycore CPUs. Technical report, INRIA, December 2010.

[18] Stephen Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1st edition, 1997.

[19] Stas Negara, Gengbin Zheng, Kuo-Chuan Pan, Natasha Negara, Ralph E. Johnson, Laxmikant V. Kale, and Paul M. Ricker. Automatic MPI to AMPI Program Transformation using Photran. In *3rd Workshop on Productivity and Performance (PROPER 2010)*, number 10-14, Ischia/Naples/Italy, August 2010.

[20] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*. OpenMP Architecture Review Board, May 2008.

[21] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical recipes in C (2nd ed.): the art of scientific computing*. Cambridge University Press, 1992.

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

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

[24] H. Tang and T. Yang. Induprakas kodukula and keshav pingali and robert cox and dror maydan. In *ACM International Conference on Supercomputing (ICS)*, 1999.

[25] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *ACM International Conference on Supercomputing (ICS)*, pages 381 – 392, 2001.

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