Ownership Passing: Efficient Distributed Memory Programming on Multi-core Systems

Andrew Friedley
Indiana University
Bloomington, IN
afriedle@indiana.edu

Torsten Hoefler
ETH Zurich
Zurich, Switzerland
htor@inf.ethz.ch

Greg Bronevetsky
Lawrence Livermore National Laboratory
Livermore, CA
bronevetsky@llnl.gov

Andrew Lumsdaine
Indiana University
Bloomington, IN
lums@indiana.edu

Ching-Chen Ma
Rose-Hulman Institute of Technology
Terre Haute, IN
mac@rose-hulman.edu

Abstract

The number of cores in multi- and many-core high-performance processors is steadily increasing. MPI, the de-facto standard for programming high-performance computing systems offers a distributed memory programming model. MPI's semantics force a copy from one process' send buffer to another process' receive buffer. This makes it difficult to achieve the same performance on modern hardware than shared memory programs which are arguably harder to maintain and debug. We propose generalizing MPI's communication model to include ownership passing, which make it possible to fully leverage the shared memory hardware of multi- and many-core CPUs to stream communicated data concurrently with the receiver's computations on it. The benefits and simplicity of message passing are retained by extending MPI with calls to send (pass) ownership of memory regions, instead of their contents, between processes. Ownership passing is achieved with a hybrid MPI implementation that runs MPI processes as threads and is mostly transparent to the user. We propose an API and a static analysis technique to transform legacy MPI codes automatically and transparently to the programmer, demonstrating that this scheme is easy to use in practice. Using the ownership passing technique, we see up to 51% communication speedups over a standard message passing implementation on state-of-the art multicore systems. Our analysis and interface will lay the groundwork for future development of MPI-aware optimizing compilers and multi-core specific optimizations, which will be key for success in current and next-generation computing platforms.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming

Keywords Ownership Passing; Distributed Memory; Shared Memory; Message Passing; Multi-core

1. Introduction and Motivation

The most commonly used programming model for large-scale parallel applications is the Message Passing Interface (MPI [13]). This model generally assumes a one-dimensional distribution of P processes, where each process has its own (private) memory space and data is solely exchanged through explicit messages.

PPOPP '13 February 23–27, Shenzhen, China

The message passing style of programming enables easy abstraction and code composition. Its *shared nothing* semantics and the SPMD programming simplify reasoning about the program's state and avoid complex problems that are often encountered in shared memory programming models [10]. Composition is achieved through communication contexts (called *communicators* in MPI) that enable multiple parallel libraries or objects to be combined into a single program without interference [8]. Those features have made MPI the predominant programming model for parallel scientific applications. However, this abstraction comes at a cost: all message transmissions have copy semantics, that is, the implementation requires a single copy from a buffer at the source process to a buffer at the destination process. Typical MPI libraries even require more copies, either through intermediate shared memory buffers or for the serialization and deserialization of complex data structures.

All modern parallel computing systems consist of network nodes with multiple processing elements (or cores). Processing elements (PEs) on a single node commonly have access to a cache-coherent hardware shared memory system. MPI was originally designed for distributed memory computers with either single-core or small SMP nodes. On today's architectures, the current copy-based message-passing model is suboptimal in terms of *memory* (send and receive buffers), *energy* (data movement consumes most energy [2]), and *time* (busses are used twice which reduces performance). To avoid those issues, many software developers switched to hybrid programming techniques, combining MPI for inter-node communication with shared memory programming models such as OpenMP [20] for intra-node communication. However, achieving the same level of performance is a tedious and complex task and often requires major code restructuring to work in the shared memory world [21].

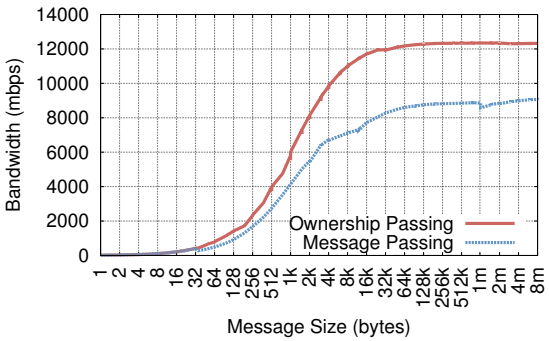


Figure 1: Bandwidth of Ownership Passing vs. Message Passing.

In this work, we use an ownership passing technique to easily and safely transform message-passing parallel applications to utilize shared memory hardware more efficiently. Instead of re-writing existing applications, we simply change them to pass a pointer from the sender to the receiver instead of copying the data. In fact, the production and consumption of buffers in our system automatically aligns in a pipelined fashion so that both stages can overlap.

As a motivating example, we show the effective bandwidth when passing a memory buffer instead of copying its contents on a modern HPC architecture in Figure 1. The measurement was done with the well-known NetPIPE [23] ping-pong benchmark on the Cab system (described later). To ensure fair comparison we extended NetPIPE to read the received data, thus accounting for the cost of transferring the data using ownership passing. We see that the standard copy approach is limited by the memory copy bandwidth and synchronization costs while ownership passing essentially requires only synchronization and reading from (potentially remote) memory. Thus, ownership passing is usually significantly faster than standard message passing on today's multicore systems.

Our approach is true zero-copy (zero-touch, in fact) because the buffer contents are neither read nor written during the communication. We develop a novel memory pooling technique to re-use communication buffers and avoid synchronization.

The main contributions of this work are:

- We design an interface for ownership passing that is compatible with MPI and allows for easy porting from MPI codes.
- We propose a static analysis technique to transform codes automatically to use ownership passing.
- We analyze the performance of ownership passing in realistic environments.
- We demonstrate practical results of important HPC micro-applications and application kernels that have been improved with our technique.

In the next section, we describe our Hybrid MPI implementation that spawns MPI processes as threads, creating the shared address space necessary for ownership passing. Section 3 describes the ownership passing technique and introduces our Ownership Passing Interface (OPI) library and API. We describe and give examples for applying ownership passing to point-to-point message passing codes in Section 3.3 and collective communication in Section 3.5. We propose a technique for automatically transforming parallel codes to ownership passing in Section 4. Finally, Section 5 provides a performance analysis of ownership passing using a micro-benchmark and several applications.

2. Hybrid MPI—A Threaded MPI Implementation

Before any shared memory communication can be introduced alongside MPI, processes need a method to directly access each others' memory. We use a thread-based MPI approach [16, 22] which replaces the common process-based rank design with a thread-based design (i.e., an MPI process is a thread). All MPI processes (threads) on a node are grouped into one operating system process[1]. Hereafter, we use the term *rank* to refer to an MPI process as defined by the MPI standard, which may be an operating system process or thread depending on the MPI implementation. We use the terms *process* and *thread* as defined in the context of the operating system.

The benefit of a thread-based MPI implementation is portability—no additional support is required from the operating system. However, the application must be made thread-safe due to sharing of a single address space (global variables become shared by all MPI ranks). Solutions exist to perform privatization of global variables automatically [14, 16], minimizing the required developer effort. Such thread-based hybridization approaches are indeed used routinely in practice, for example in CHARM++'s AMPI [14].

We have developed a portable library called Hybrid MPI (HMPI) that implements the thread-based rank design on top of any standard MPI library. HMPI intercepts messages destined for ranks within the same node and uses a faster single memory copy communication path (MPI often performs two copies via shared memory segments), while utilizing an existing MPI for inter-node communication. Although we use HMPI in this paper to demonstrate results, our proposed ownership passing optimization would work with any thread-based MPI library or other scheme enabling direct memory access between MPI processes, such as XPMEM [25].

3. Ownership Passing

MPI's distributed memory design ensures that only one rank can access a buffer (any arbitrary memory region). That is, exactly one rank *owns* a buffer, and that is the only rank that may read or write that buffer. Using a shared memory technique with MPI (explained in Section 2), ownership can be *passed* from one rank to another via message passing. When a rank gives away ownership of a buffer, that rank can no longer access that buffer. Likewise, taking ownership of a buffer enables exclusive read and write access.

Ownership passing reinterprets the concept of distributed memory in a way that retains its simplicity while taking advantage of shared memory hardware. Traditional distributed memory partitions the application's address space into static private memory blocks. In contrast, ownership passing allows this partition to be dynamic, with memory regions moving from one private space to another while maintaining the invariant that each memory region is privately owned by exactly one thread of execution. The resulting flexibility makes it possible for message passing applications to utilize shared memory hardware in ways similar to native shared memory applications but without concern about data races and other complications of the shared memory model. Not only is our approach beneficial on cache-coherent architectures (e.g., commodity x86), it also works on non-cache-coherent and non-uniform memory architectures (NUMAs).

3.1 Ownership Passing Programming Interface

Transferring ownership of a buffer only requires sending a pointer instead of the entire message data. When the new owner of a buffer begins reading the message from its original location, the shared memory hardware will begin to stream data from the sender rank's cache and/or memory to the receiver. Standard architectural CPU features such as snoop caches and memory prefetching improve the performance and enable efficient communication/computation overlap in hardware. Since the communication library never accesses the buffer data, true zero-copy communication is achieved.

We define a small extension API, referred to as the Ownership Passing Interface (OPI), to simplify the use of ownership passing in applications. The C interface is shown in Figure 2 and Figure 3 shows a MPI simple example and its OPI counterpart. All routines are thread-safe with respect to one another.

`OPI_Alloc` and `OPI_Free` allocate and deallocate new communication buffers, calling `malloc` and `free` and performing additional management of buffer memory pools, as described in Section 3.2. Ownership passing is performed using the `OPI_Give` and `OPI_Take` routines, which are analogous to `MPI_Send` and `MPI_Recv` (nonblocking versions are also available in analogy to nonblocking MPI calls). `OPI_Give` consumes the provided buffer. If the destination rank is in the same address space (on the same

[1] The term "MPI Process" is abstractly defined in the MPI standard and does not necessarily mean operating system process.

`OPI_Alloc(void** ptr, size_t length)`	Allocate a communication buffer of some length.
`OPI_Free(void** ptr)`	Release a buffer allocated by `OPI_Alloc`.
`OPI_Igive(void** ptr, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm, MPI_Request req)`	Pass ownership of a buffer to another MPI rank.
`OPI_Itake(void** ptr, int count, MPI_Datatype datatype, int rank, int tag, MPI_Comm comm, MPI_Request req)`	Receive ownership of a buffer from another MPI rank.

Figure 2: Nonblocking Ownership Passing Interface (OPI).

MPI Code	OPI Code

double buf[...];
if(rank==0)
  MPI_Send(buf, 1, ...)
 	
else if(rank==1)
  MPI_Recv(buf, 0, ...)
	double* buf;
	if(rank==0) {
	  buf=OPI_Alloc(...);
	  OPI_Give(&buf, ...);
	} else if(rank==1) {
	  OPI_Take(&buf, ...);
	  OPI_Free(&buf);
	}

Figure 3: Example of MPI to OPI conversion

node), then the source rank synchronizes with the destination and passes the buffer. If the destination is in a different address space, then the source rank invokes a normal `MPI_Send` call with the given arguments and returns the buffer to the buffer pool after the send completes. `OPI_Take` returns a new buffer to the receiver. If the buffer comes from a rank in the same address space, then it will simply return the pointer to the buffer. If the source is a rank from a different address space then it allocates memory of the required size, invokes `MPI_Recv` on this buffer, and returns the buffer upon completion of the remote receive. Figure 4 depicts the ownership passing mechanism, flow control, and buffers.

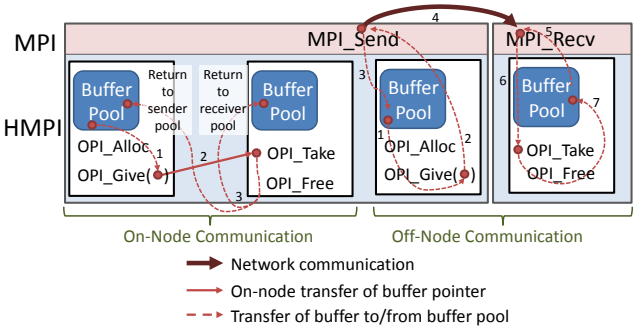


Figure 4: Flow of Control and Buffers in Ownership Passing.

Note that `OPI_Alloc` and `OPI_Take` introduce new buffers, while `OPI_Give` and `OPI_Free` relinquish ownership of a buffer. For safety and to promote the *ownership* concept, the latter two routines clear the provided pointer to `NULL` before returning.

3.2 Communication Buffer Management

Once a rank has taken ownership of a buffer and consumed its contents, that buffer must be disposed of. We could simply free the buffer back to the heap, but this is not ideal. `Malloc` and `free` must be implemented in a thread-safe manner, which in our case implies a lock shared between all ranks on a node. Furthermore `malloc` and `free` are costly library calls, and ownership passing encourages allocating a new buffer for every message sent.

We can alleviate the costs of `malloc` and `free` by caching buffers in a memory pool. When allocating, we search the pool for the first buffer large enough for the requested size and reuse it. If no such buffer exists, a new one is allocated. When freeing a buffer, we return it to a memory pool instead of the heap.

Using one memory pool per node would require a lock shared between all ranks on a node, which is not an improvement over using the heap. Instead, we maintain one memory pool per rank. Now, a choice must be made—buffers can be returned to either the sender's or the receiver's memory pool. Returning buffers to the sender's pool requires a lock, since multiple ranks may simultaneously return a buffer to the pool, perhaps also while the sender is allocating. However, this approach distributes contention over many locks rather than one, yielding an improvement.

On the other hand, no lock is required if we return buffers to the receiver's local pool—each rank only accesses its own memory pool. However if one rank receives more messages than others, buffers accumulate in one memory pool and never get reused, wasting memory. To solve this problem, we introduce a check when a buffer is added to a memory pool. If the number of unused buffers exceeds a threshold, some buffers are freed back the heap. This memory will eventually be reused in later `malloc` calls, potentially on other ranks.

To evaluate the performance of these different buffer management schemes, we measure the time to perform the code "`OPI_Free(OPI_Alloc(8))`" (8 byte buffer size), averaged over 5,000 runs. The results, shown in Figure 5 demonstrate that returning buffers to the receiver's pool has the lowest cost—this solution has no synchronization between ranks, and reduces the frequency of expensive `malloc` and `free` calls. We use this scheme for all experimental results shown later in the paper.

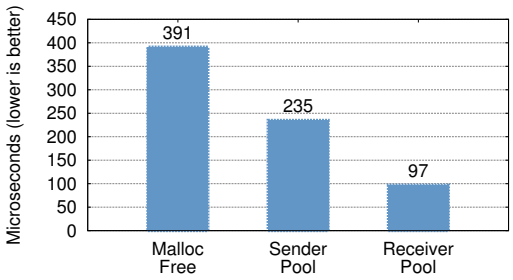


Figure 5: Average time of the different buffer management schemes to allocate and free eight bytes.

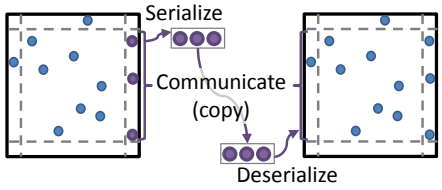
3.3 Point to Point Ownership Passing

Transforming an application to use ownership passing consists of three steps:

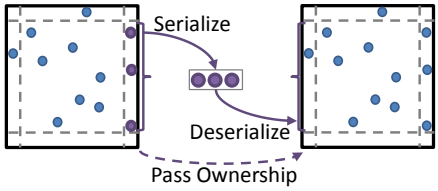
1. Replace `MPI_Send` and `MPI_Recv` and related communication functions with `OPI_Give` and `OPI_Take`, respectively. OPI makes this trivial; except for the additional referencing in the first argument (buffer pointers) `OPI_Give` and `OPI_Take` accept the same arguments as `MPI_Send` and `MPI_Recv`.
2. Insert a call to `OPI_Alloc` before packing a communication buffer for sending. Since giving away ownership consumes the communication buffer, a new one must be allocated every time a message is sent.
3. Insert a call to `OPI_Free` after receiving and unpacking a communication buffer. Since taking ownership produces a new communication buffer, every received message must be freed.

Although communication buffers can be allocated at any time before they are packed and can be freed any time after they are unpacked, the best buffer reuse is achieved by allocating send buffers as 'late' as possible in the application (immediately before they're packed), and freeing received buffers as 'early' as possible (immediately after they're unpacked).

To illustrate the changes needed to perform ownership passing with MPI, we present a two-dimensional molecular dynamics (MD) example, where space is divided into regions and each processor is responsible for computing forces on and positions of particles within its region. Particles on the boundary of each processor's region are communicated to processors responsible for adjacent space regions. Figure 6(a) shows how this is performed using MPI for a single boundary exchange (the same is done with other neighbors). Boundary particles are serialized into a buffer, which is then sent via MPI (copied) and deserialized on the receiver. Ownership passing, shown in Figure 6(b), speeds up the process by replacing the MPI copy with a transfer of ownership of the packed buffer.



(a) MPI



(b) Ownership Passing

Figure 6: Molecular dynamics overlap communication. The boundary particles must be serialized into contiguous buffers.

Figure 7 presents pseudo-code for the MPI and the ownership passing implementation. The skeleton and code flow (computation of the directions and the computations) are identical in both codes. The ownership passing version allocates a buffer from the local memory pool just before packing, regardless of whether the destination is local or remote. Ownership is passed if the neighbor rank is local; otherwise the message is passed as would normally be done with negligible overheads. After unpacking, the received buffer is returned to the receiver's memory pool.

3.4 MPI Datatypes

MPI datatypes allow strided sequences of elements or arbitrary memory layouts to be sent and received. The ownership passing principle can be adapted to communication of disjoint sets of elements. Since MPI datatypes must be specified at both the sender and the receiver, ownership to the elements specified by the datatype can be easily transferred as long as the datatypes on both sides have the same memory layout. In cases where the receiver only consumes part of a buffer or when the sender wishes to reuse the data after passing ownership, the receiver can pass ownership back. Such an approach is analagous to protecting access to the buffer with a mutex. Compiler support for this technique is part of our future work. MPI also allows applications to provide different datatypes on the sender and receiver that place data in memory in different orders (e.g. matrix row on the sender and matrix column on the receiver). Copying is required to support this use-case and is the most efficient way to provide this specialized functionality.

3.5 Collective Ownership Passing

Although this paper specifically focuses on ownership passing for point-to-point communication, it can also be used effectively for collective communication. Here, we discuss how ownership passing can be used to implement scatter, gather, and all-to-all. Further

MPI

```
pack_particle_buffer(send_buffer, particle_data);

MPI_Isend(send_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[0]);

MPI_Irecv(&recv_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[1]);

MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);

unpack_particle_buffer(recv_buffer, particle_data);
```

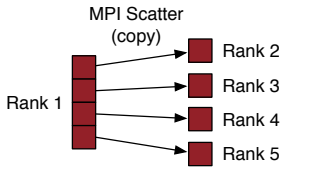
Ownership Passing

```
OPI_Alloc(&send_buffer, max_particles);

pack_particle_buffer(send_buffer, particle_data);

//Pass ownership of our send buffer.
OPI_Igive(&send_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[0]);

//Take ownership of a new receive buffer.
OPI_Itake(&recv_buffer, count, datatype,
    neighbor_rank, TAG, MPI_COMM_WORLD, &reqs[1]);

MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);

unpack_particle_buffer(recv_buffer, particle_data);

//Always return the receive buffer.
OPI_Free(&recv_buffer);
```

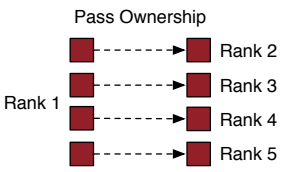
Figure 7: Boundary exchange communication between a pair of neighbors.

techniques are possible, such as allowing read access for multiple ranks, though are not the focus of this paper.

First, consider the MPI scatter operation in which applications allocate and pack into one send buffer, with respective portions to be scattered to each rank. Ownership of this buffer can be passed to the receive ranks as a collective, but this approach raises the question of which rank should release the buffer, and when. Synchronization (e.g. a barrier) is required to solve this problem, but negates the performance benefits of ownership passing.



(a) MPI



(b) Ownership Passing

Figure 8: Scatter collective communication. MPI copies out of one buffer, while ownership passing gives separate buffers to each rank.

A better solution is to allocate a separate buffer for each destination rank, as illustrated in Figure 8. `OPI_Igive` is used to pass ownership of each buffer to its respective rank. Each receiver can then return its buffer to the sender's memory pool without a global synchronization.

Figure 9 demonstrates ownership passing for scatter communication in code form. Note that we have used point-to-point communication, although ownership could also be transferred using an `MPI_Scatter` routine or OPI equivalent. Gather operations are performed in a similar manner to scatter; the root rank gathers an array of buffer pointers to take ownership. When finished with the data, the root can release each buffer back to the respective memory pools. An ownership passing all-to-all can be constructed by

combining scatter and gather. This work focuses on point-to-point ownership passing; more advanced collective techniques for collective ownership passing will be investigated in the future.

```
if(my_rank == root) {
  for(int i = 0; i < mpi_size; i++) {
    //Allocate a buffer and pack data for rank i
    OPI_Alloc(&buffer, buffer_size);

    pack_buffer(buffer, i);

    //Pass ownership of the buffer.
    OPI_Igive(&buffer, count, datatype,
        i, TAG, shared_mem_comm, &reqs[i]);
  }
}

OPI_Itake(&buffer, count, datatype,
    root, TAG, shared_mem_comm, &recv_req);

//Wait to take ownership or receive from the root.
MPI_Wait(&recv_req, MPI_STATUS_IGNORE);

unpack_buffer(buffer);

//Always return the buffer to where it came from.
OPI_Free(&buffer);

//Complete the send requests.
MPI_Waitall(mpi_size, reqs, MPI_STATUSES_IGNORE);
```

Figure 9: Ownership passing in scatter collective communication.

4. Compiler Analysis

To simplify the deployment of OPI in legacy applications, we have designed a novel compiler analysis. Our analysis detects code patterns to which OPI is applicable and automatically transforms the code to use the OPI extensions. We have implemented our analysis using the ROSE framework [19], although several analyses we depend on are not yet available in ROSE. Our implementation functions on simple codes and will support complex codes when the analyses we depend on are available in ROSE.

We examined the NAS Parallel Benchmarks (NPB) for applicability of OPI. Almost all point-to-point communication fits the OPI pattern and can be converted by our analysis. The exception is CG's row reduction, which reads the send buffer immediately after `MPI_Send`. Many NPB codes communicate different non-overlapping regions of the same buffer, requiring an analysis that identifies these regions and presents them to our analysis as separate buffers. BT uses pointer-based data structures that require a simple points-to analysis to disambiguate.

Our analysis works in two phases. First, it analyzes the code around each `MPI_Send`, `MPI_Recv` and related calls via a backwards data flow analysis to determine if the application accesses each buffer used in these operations in a way that is compatible with ownership passing. Then, assocations between `MPI_Send` and `MPI_Recv` operations are made using either previously published analyses [3] or user annotations. If a given operation and all of its possible assocated operations are OPI-Compatible, we use a second forward analysis to identify the points where OPI operations need to be inserted and original operations need to be removed. Since each MPI send must match at least one MPI receive, every given buffer must be taken by its receiver.

4.1 Outline of Analyses and Transformations

The sender transformation is illustrated in Figure 10. The left graph describes the pattern of operations that may be performed on an OPI-compatible buffer as a control flow graph and the left code example provides an example of matching code. The right graph and code example describe how such code is transformed, with dashed arrows identifying locations in the original code where the new operations or replacements are inserted. The graphs correspond to just the operations that refer to a single buffer and may be interleaved with operations that refer to other memory regions. After a buffer is allocated (e.g., static buffer declaration or dynamic `malloc` or `new`) the application must pack and send that buffer zero or more times before it is deallocated (e.g., scope exit of static buffer or dynamic `free` or `delete`). During packing, the application must overwrite the buffer without reading its prior contents; reads are allowed as long as they are preceded by a write to the same location. The buffer is sent via `MPI_Send` or an equivalent function. Code that follows this pattern is transformed as follows: (i) the buffer's allocation is replaced with an assignment to a uniquely named integer that stores the buffer's size, (ii) `OPI_Alloc` is inserted at CFG locations where control transitions from Allocate or Send to Pack and takes the buffer's size variable as input, (iii) `MPI_Send` is replaced with `OPI_Give`, and (iv) the buffer's deallocation is removed.

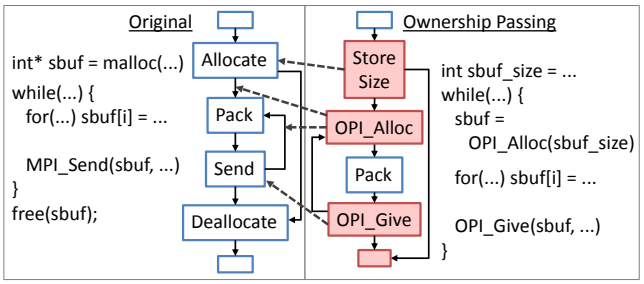


Figure 10: Sender code pattern and transformation

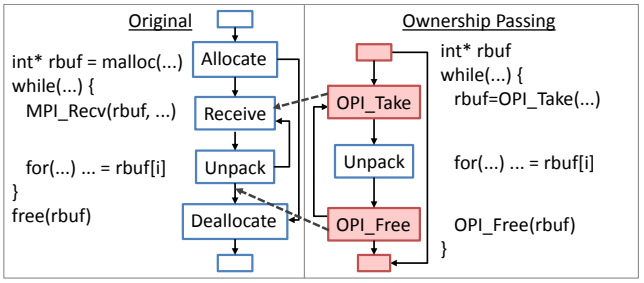


Figure 11: Receiver code pattern and transformation

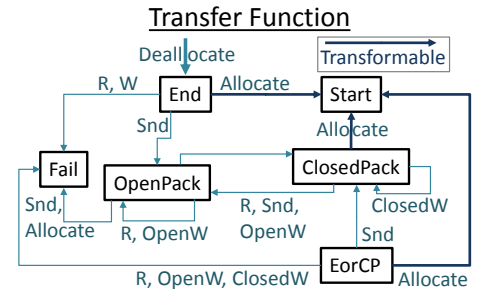
Figure 11 illustrates the receiver transformation. After a buffer is allocated, the application must receive and unpack it zero or more times before it is deallocated. The buffer's data is received via `MPI_Recv` or an equivalent function, which overwrites its contents. During unpacking the application may read from or write to the portions of the buffer overwritten by `MPI_Recv`. Matching code is transformed as follows: (i) the buffer's allocation is removed, (ii) `MPI_Recv` is replaced by `OPI_Give`, the return value of which is assigned to the buffer's pointer, (iii) `OPI_Free` is placed at CFG locations where control transitions from Unpack to Receive or Deallocate, and (iv) the buffer's deallocation is removed.

Both analyses operate by maintaining at each location in the application's control flow graph (CFG) a mapping from live memory regions to one of the states of a finite automaton that captures the patterns shown in Figures 10 and 11. The backward analysis also includes state Fail, indicating that the buffer's use does not fit the OPI pattern. This mapping can use information from any alias or points-to analysis [1] that indicates whether the buffers referred to by two pointers are always-same (must-equal) or never-same (not may-equal). If at any CFG node there exist two buffers that are

not always-same or never-same (e.g., one pointer refers to different buffers in different executions), their state is set to Fail. This conservatively handles cases where one portion of the OPI sender or receiver pattern holds for a given buffer and another portion holds for a different buffer (e.g., one buffer is packed but another is sent but in both cases the same pointer variable is used).

4.2 Pattern detection analyses

Figures 12 and 14 present the transfer and meet functions used by the backwards analyses that determine whether sender and receiver code fits the OPI pattern. Transfer functions are shown as finite automata where nodes correspond to the possible states of a buffer during the analysis and edges indicate how these states are transferred through operations. Each edge represents an operation relevant to the analysis. When the buffer is in the edge's source state and an operation is encountered, the buffer's transitions to the edge's destination state. Snd and Rcv edges denote `MPI_Send` and `MPI_Recv` of the buffer, respectively (or equivalent operation). Allocate and Deallocate correspond to the buffer's allocation and deallocation points. R identifies buffer reads and W buffer writes.



Meet Function

End	End			
OpenPack	Fail	OpenPack		
ClosedPack	EorCP	OpenPack	ClosedPack	
EorCP	EorCP	Fail	EorCP	EorCP
	End	OpenPack	ClosedPack	EorCP

Figure 12: Sender pattern detection backward analysis

Our analysis relies on an external array region analysis to indicate if the buffers adhere to the following required properties:

1. In the sender code pattern, all reads in the Pack must be preceded by prior writes to the same buffer location (i.e., reads cannot be upwardly-exposed), which includes the reads from `MPI_Send`.
2. In the receiver pattern, the Unpack code can only read the subregion of a buffer that was overwritten by the prior `MPI_Recv`.

This information is represented by replacing R and W with more focused operations. In the sender transfer function, OpenW indicates a write that is followed by some upwardly-exposed reads along the path between the write and the following `MPI_Send` (including the reads by `MPI_Send`. ClosedW denotes a write not followed by such reads. In the receiver transfer function, InR and InW denote a read or write to a buffer region overwritten by the prior `MPI_Recv` (in-buffer) and OutR and OutW indicate a read and write to a region not overwritten by the `MPI_Recv` (out-of-buffer).

Since the pattern detection analyses work backwards through the application's CFG, they begin to consider a given buffer at the point immediately before its deallocation in state End. The sender analysis transitions from End to OpenPack if it observes a send operation, to Send if it observes Allocate and to Fail if it observes reads or writes after the final send. Once in OpenPack, the analysis stays in this state until it observes a ClosedW operation. Whan that occurs, it transitions to ClosedPack, indicating that the code region between this code location and the next send is well-formed and has no upwardly-exposed reads. It returns from ClosedPack to OpenPack on any R or OpenW operations. If the analysis observes a Snd while in OpenPack state it transitions to Fail since the pack code between two adjacent sends is not well-formed. Snds observed while in ClosedPack state fit the OPI pattern and return the analysis to the OpenPack state. Finally, if Allocate is observed while in ClosedPack state, the analysis transitions to the Start state, indicating that the buffer's use fits the OPI pattern. If Allocate is observed in other states, the analysis transitions to Fail.

The meet function takes as input two states from two different control-flow paths that converge at a given CFG node and outputs the state at this meet point. It is a table with the input states on the horizontal and vertical axes and the output state at their intersection. The Fail state is omitted since the meet of any state with Fail is Fail. The meet of any state with itself is itself. The first property of the meet function is that the meet of OpenPack with either OpenPack or ClosedPack is OpenPack since this means that the meet point is followed by upwardly-exposed reads. Further, the meet of End with ClosedPack produces a new state EorCP, which captures the fact that a buffer fits the OPI pattern if it is either communicated according to the pattern or not communicated at all. On Allocate operations EorCP transitions to Start, indicating the buffer's use fits the OPI pattern.

Figure 13 shows an example of the sender analysis operating on two code examples. The left example has two statically distinct buffers, one of which follows the OPI pattern and one that does not. This example shows how the analysis state evolves to conclude that `buf1` fits the pattern. Starting at the deallocation of `buf1` and `buf2` it proceeds backwards around the `while` loop (steps 2, 3 and 4) to reach state ClosedPack for `buf1` (the buffer is fully packed before being sent) and OpenPack for `buf2` (the buffer is sent without being packed). At the loop's entry (step 5) the states evolve to EorCP for `buf1` (it is either packed/sent correctly or not used) and Fail for `buf2` (its communication in the loop doesn't follow the OPI pattern). At the end `buf1` transitions from EorCP to Start at its allocation site, indicating that it fully fits the OPI pattern, while `buf2` remains at Fail, indicating that it does not. In the code example on the right, the identity of the buffer pointed to by p is not statically unique. As discussed in the last paragraph of Section 4.1 if there exists ambiguity about the identity of buffers involved in OPI operations (reads, writes, sends, receives) their state is set to Fail. This is done at the `MPI_Send` operation where the referent of p is either `buf1` or `buf2` but it is statically not known which one.

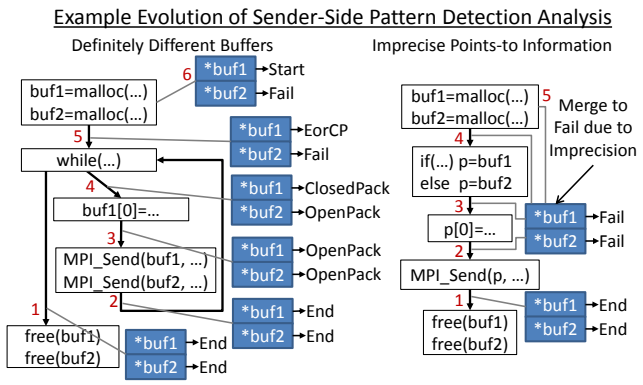


Figure 13: Example of sender pattern detection analysis

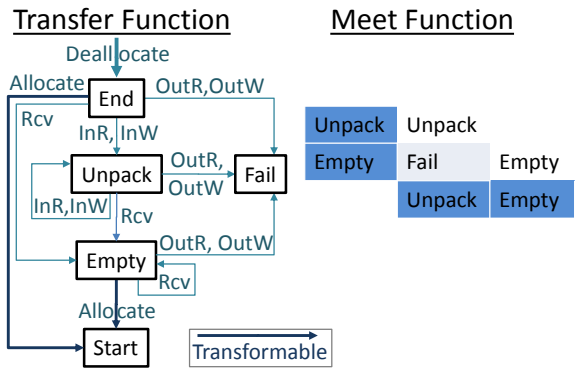
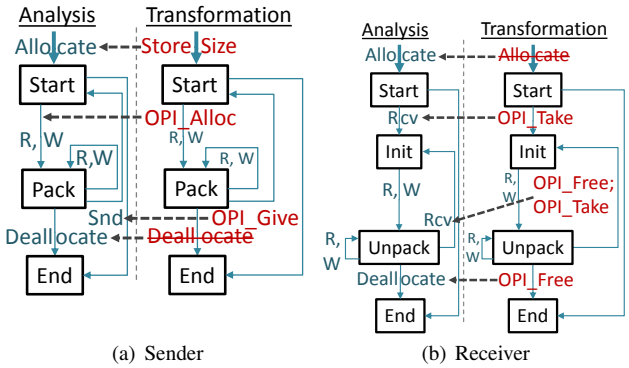


Figure 14: Receiver pattern detection backward analysis

Figure 14 presents the receiver analysis. Like the sender analysis, it starts from state End. When a receive occurs, it transitions to Empty to indicate an unused buffer. On Allocate it transitions to Start and on InR and InW to Unpack. The analysis stays in Unpack while it observes only InR and InW operations and stays in Dead while it observes receives. It transitions to Fail whenever OutR or OutW are observed in any state. This is a conservative decision; while some applications with such accesses can be made to use OPI (e.g. where communication is not inside a loop), this is too complex in general. If Allocate is observed in state Unpack, the analysis transitions to Start to indicate that the receiver OPI pattern holds for this buffer. Otherwise, if Allocate is observed while in another state, the analysis transitions to Fail. The meet of Unpack and Empty is Unpack since this corresponds to applications that stop unpacking on one side of a branch (looking forward in the code) and continue unpacking on another side.

4.3 Transformation Analyses

If the backward pattern detection analysis indicates that a buffer's use follows the OPI code pattern, we use a forward transformation analysis to identify the code locations that must be transformed to use OPI. The left parts of Figures 15(a) and 15(b) show the transfer functions of the sender and receiver analyses, respectively. Since these are forward analyses, they begin at each buffer's allocation and terminate at its deallocation. Transformations are performed when the transfer function transitions along an edge, as shown the graphs on the right sides of the figures. When the original operation must be removed, it is crossed out in each graph. When it is to be replaced with alternate code, the replacement code is specified. In the receiver transformation, transitions from Unpack and Init to Init correspond to replacing `MPI_Recv` with an `OPI_Free; OPI_Take` sequence.



(a) Sender (b) Receiver

Figure 15: Transformation forward analysis

5. Experimental Results

Experimental results were obtained using the LLNL Sierra and Cab systems. Sierra has two Xeon X5660 (six core, 2.8 GHz) processors (12 cores total) and 24 GiB of RAM, while Cab has two Xeon ES-2670 (eight core, 2.6 GHz) processors (16 cores total) and 32 GiB of RAM. MVAPICH2 v1.8 was used for all results. Since ownership passing is a shared memory optimization, we show performance results for varying numbers of ranks executed on a single node to avoid a network complicating the results. As discussed in Section 4, our compiler analysis only works on simple codes. All codes shown here were transformed manually.

5.1 Microbenchmark Analysis

We developed OPBench to analyze the performance characteristics of ownership passing (as implemented by the OPI interface) and compare them to MPI and HMPI. OPBench implements a simple nearest-neighbor stencil, performing the following steps:

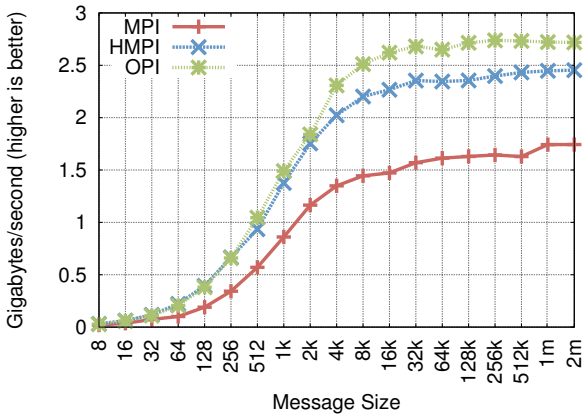
1. Computation time is simulated and measured by performing a simple calculation on each of the elements of an array.
2. A pack loop copies the data from the 'application' data structure to a communication buffer.
3. The communication buffers are exchanged between two ranks.
4. An unpack loop copies the data from the received communication buffer back to the application data structure.

For each iteration of the benchmark we perform step 1 once, then repeat steps 2-4 four times to simulate multiple neighbors. We ran our benchmark in two configurations on the Cab system: (i) ranks are located on different cores within the same processor and (ii) ranks are located on different processors. Each data point in the results is an average of 5,000 benchmark iterations, with timings acquired from both ranks.

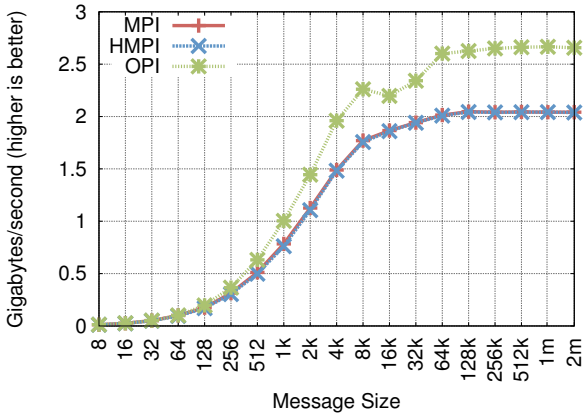
Figure 16 shows the bandwidth achieved when packing, communicating, and unpacking a message (the sum of time taken by steps 2, 3, 4 of OPBench) within and across processors. Figure 16(a) shows the total bandwidth achieved when packing, communicating, and unpacking a message (the sum of time taken by steps 2, 3, 4 of OPBench) on cores within the same processor. OPI (2.8 GB/s) significantly improves on HMPI (2.5 GB/s) and MPI (1.75 GB/s). Here, OPI performance is bounded by the memory bandwidth achieved during the pack and unpack phases.

Figure 16(b) shows the same measurement when both processes are on distant cores on different sockets. Here, we see that HMPI (2 GB/s) has a negligible benefit over MPI (2 GB/s) because MPI's pipelined copy is essentially using the on-board interconnect as well as HMPI can. MPI's bandwidth is slightly higher than in the previous case because the copy uses two NUMA domains and thus gets twice the write bandwidth. OPI (2.7 GB/s) improves the bandwidth significantly by streaming the data directly from the source buffer, avoiding the additional copy completely.

To understand the performance properties of OPI in detail, we measured cache behavior during OPBench execution. Figure 17(a) shows the number of L1 cache misses incurred while packing, communicating and unpacking a message. The data shows that MPI incurs several times more misses than HMPI or OPI for communicating the same message. This is because MPI must copy data across memory spaces, which involves the sender copying into a common shared buffer and the receiver copying back from this buffer. In contrast, HMPI and OPI only perform a single direct data transfer from one core to another. A key difference between the algorithms used by the approaches is their effect on the cache itself, demonstrated in Figure 17(b), which shows the number of L1 cache misses during the execution of the simulated application code (step 1 of OP-

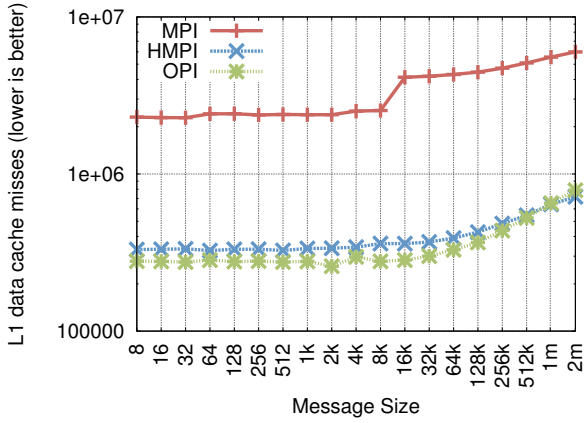


(a) Same-socket throughput

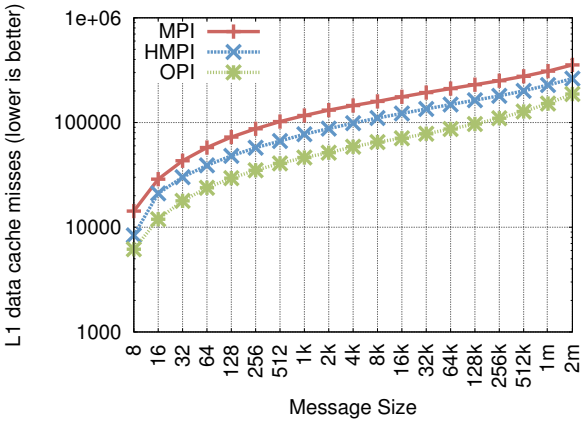


(b) Cross-socket throughput

Figure 16: Bandwidth for pack + communication + unpack on the Cab system.



(a) Total pack + communicate + unpack misses



(b) Application misses

Figure 17: L1 Cache misses for different components of OPBench on the Cab system.

Bench). Since HMPI must copy the entire buffer into the receiver's cache before allowing it to read the data, it can pollute the cache by evicting the application's state. During subsequent computation this state must be brought back into the cache, causing additional misses. In contrast, OPI interleaves application reads and transfers of message data, so it is less disruptive to the cache, resulting in fewer cache misses for OPI than for HMPI or MPI.

5.2 MiniMD

MiniMD is part of the Mantevo [7] mini-application suite, which consists of several mini-applications representing larger application classes. Such mini-applications are increasingly used in exascale research for their combination of simplicity and relevance. MiniMD is a molecular dynamics simulation that computes atom movement over a 3D space decomposed into a processor grid. The primary work loop performs the following steps every 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 from 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.

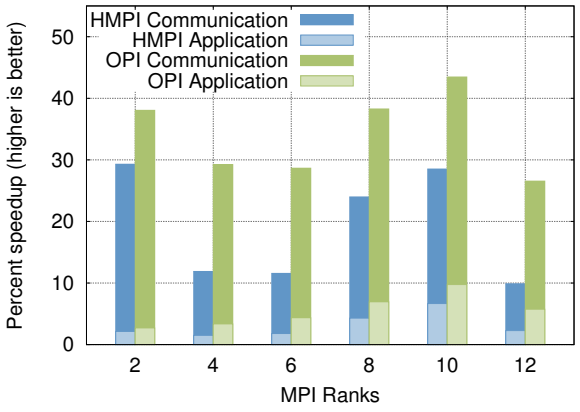
MiniMD is an example of a stencil communication pattern that exchanges irregular data with point-to-point messages during steps 2 and 4. We transformed these communication phases to use ownership passing in the same manner as was done for the two-dimensional stencil example described in Section 3.3.

Performance results are shown in Figure 18. Computation of forces between atoms dominates execution time, so optimizing communication has a smaller affect on overall application time. When the communication time alone is considered, significant speedups are observed—up to 43% on Sierra, and 51% on Cab.

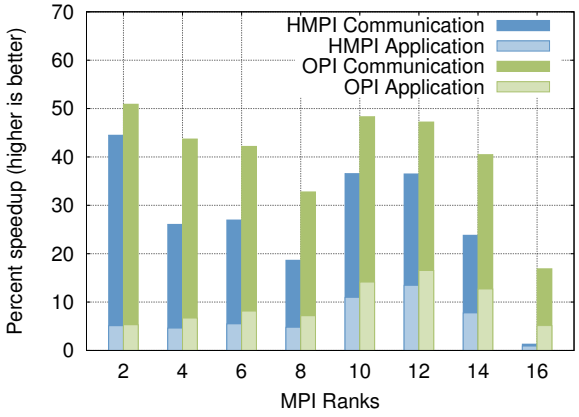
5.3 Fast Fourier Transform

Fast Fourier Transforms (FFTs) are among the most important operations in use today. Numerous algorithms and parallel applications use FFTs in their core computations [6, 9]. A one-dimensional FFT transforms 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* [17, §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 [6, 9].

We perform our experiments using a two-dimensional FFT kernel. The original 2D FFT code is implemented using MPI and transforms an $N_x \times N_y$ domain. The initial array is stored in x-major



(a) Sierra System



(b) Cab System

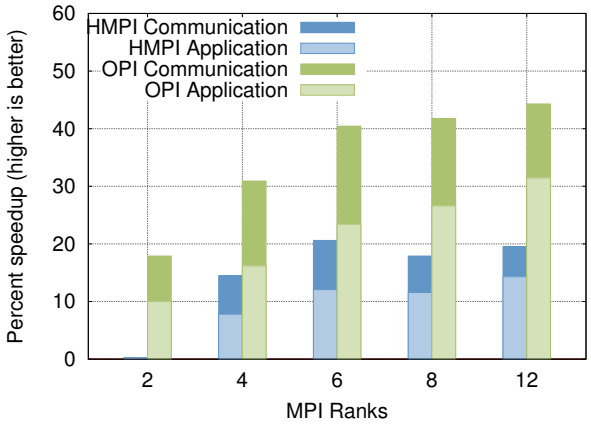
Figure 18: MiniMD speedup using ownership passing and HMPI, relative to MVAPICH2. A 4,000-atom problem size was used. Application times include communication time.

order and distributed in y-dimension such that each process has N_x/P y-pencils. The steps to perform the 2D FFT are:

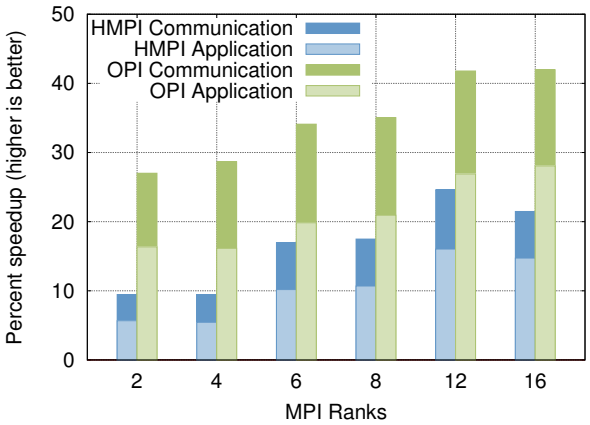
1. Perform N_x/P 1D FFTs in y-dimension (N_y elements each).
2. Serialize the array into a buffer for the all-to-all.
3. Perform a global all-to-all.
4. De-serialize the array to be contiguous in the x-dimension (each process now has N_y/P x-pencils).
5. Perform N_y/P 1D FFTs in the x-dimension (N_x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

The all-to-all communications in steps 3 and 6 make 2D FFTs an interesting application for ownership passing. We perform the all-to-all ownership passing transformation described in Section 3.5. Each sender has one memory pool, from which it allocates and packs one buffer for each other rank. As each rank is packed, ownership is transferred to the receiver. Buffers are unpacked as ownership control arrives from each other rank.

The 2D FFT execution time is dominated by all-to-all communication, which in turn is dominated by message copying overhead in MPI. Ownership passing eliminates this overhead, leading to large speedups. The remaining communication time is dominated by the



(a) Sierra System



(b) Cab System

Figure 19: 2D FFT speedup using ownership passing and HMPI, relative to MVAPICH2. A 6,144x6,144 problem size was used. Application times include communication time.

pack and unpack routines, which transpose the two-dimensional matrix of FFT data points. Figure 19 shows the results, with communication time speedups of up to 48% on Sierra and 35% on Cab.

6. Related Work

In Section 2, we introduced the concept of a thread-based MPI for enabling direct memory access between ranks. An alternative approach is to use virtual memory extensions to directly map memory from one process into another [24, 25]. The advantage of this approach is that the MPI process-rank design (i.e., a rank is a process) remains intact. However, extensions to the operating system are required and are not generally available on existing installations, limiting availability and portability.

The idea of ownership passing draws upon techniques that can be found in distributed shared memory (DSM) systems [18] and early cache coherence protocols [5]. In either case, *ownership* is defined in the same manner—only one process is entitled to read or write a particular block of memory (e.g., a page or cache line). In this work, we use the concept of ownership to present a clean interface to improved shared memory performance in the context of MPI's distributed memory model.

The Generic Message Passing Framework [11, 12] implements a message passing interface for C++ that is reminiscent of MPI, with an extension for doing ownership passing using the `auto_ptr` reference counting pointer class. Our approach integrates with ex-

isting MPI, making it possible to incorporate ownership passing into legacy applications written in FORTRAN, C, and C++.

Multi-Version Variables (MVVs) [4, §8] are a language extension to Co-Array FORTRAN for supporting a producer-consumer communication channel. A memory pool concept similar to our own is used to provide a form of streaming message passing in a partitioned global address space (PGAS) language. Though similar, our work describes a path for modifying legacy MPI applications for improved performance on shared-memory hardware.

Ownership passing has been used to speed up other parallel programming frameworks such as the actor-based framework *ActorFoundry* [15]. Significant performance benefits have been demonstrated in this context. However, C or Fortran with MPI codes have a more complex structure than *ActorFoundry* and complete static analysis is thus not always possible.

7. Discussion and Conclusions

We show how the principle of ownership passing can be used with MPI applications in order to utilize shared memory (multi-core) hardware more efficiently. This principle is often used implicitly in cache-coherency protocols and we extend it with a software interface to be used explicitly. Our ownership passing interface (OPI) is a simple extension to MPI and keeps MPI's ease of programming and abstraction (as opposed to shared memory programming with critical sections) while providing true zero-touch intra-node communication.

We address the challenge of returning the buffers with several pooling techniques. Our lock-free receiver pooling technique shows best results for practical applications where messages (buffers) are often passed symmetrically between MPI ranks.

Our interface allows the porting of legacy MPI applications to support "fat" shared memory nodes with simple transformations. Our examples show that the transformation is indeed simple. We provide a static compiler transformation that detects transformable code patterns and replaces them with appropriate OPI calls.

Our performance studies with microbenchmarks and real applications show that ownership passing is an effective technique for achieving better performance on shared memory hardware. Communication time speedups of up to 51% in a molecular dynamics application and 44% are realized in a two-dimensional FFT code.

Acknowledgments

This work was supported in part by the Department of Energy X-Stack program and the Early Career award program. It was partially performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. (LLNL-CONF-????)

References

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 2 edition, 2007.

[2] S. Borkar. Will interconnect help or limit the future of computing. Presented as the 19th IEEE Conference on Hot Interconnects, 2011.

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

[4] Y. Dotsenko. Expressiveness, programmability and portable high performance of global address space languages. Technical report, 2006.

[5] S. J. Frank. Tightly Coupled Multiprocessor System Speeds Memory-Access Times. *Electronics*, 57(1):164–169, Jan. 1984.

[6] X. Gonze et al. A brief introduction to the ABINIT software package. *Zeitschrift fr Kristallographie*, 220(5-6-2005):558–562, 2005.

[7] M. A. Heroux, D. W. Dorfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. 2009.

[8] T. Hoefler and M. Snir. Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions. In *18th European MPI Users' Group Meeting, EuroMPI, Proc.*, volume 6960, pages 345–355, Sep. 2011.

[9] 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.

[10] E. A. Lee. The problem with threads. *Computer*, 39(5):33–42, May 2006.

[11] L.-Q. Lee and A. Lumsdaine. Generic programming for high performance scientific applications. In *Proc. of the 2002 Joint ACM Java Grande – ISCOPE Conference*, pages 112–121. ACM Press, 2002.

[12] L.-Q. Lee and A. Lumsdaine. The generic message passing framework. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS)*, page 53, April 2003.

[13] MPI Forum. MPI: A message-passing interface standard. version 2.2, September 4th 2009.

[14] S. Negara, G. Zheng, K.-C. Pan, N. Negara, R. E. Johnson, L. V. Kale, and P. 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.

[15] S. Negara, R. K. Karmani, and G. Agha. Inferring ownership transfer for efficient message passing. In *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, pages 81–90, New York, NY, USA, 2011. ACM.

[16] M. Pérache, P. Carribault, and H. Jourdren. MPC-MPI: An MPI implementation reducing the overall memory consumption. In *Proc. of the 16th European PVM/MPI Users' Group Meeting*, pages 94–103, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[18] J. Protic, M. Tomasevic, and V. Milutinovic, editors. *Distributed Shared Memory: Concepts and Systems*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1st edition, 1997.

[19] D. J. Quinlan. Rose: Compiler support for object-oriented frameworks. *Parallel Processing Letters*, 10(2/3):215–226, 2000.

[20] R. Rabenseifner. Hybrid parallel programming on HPC platforms. In *In proceedings of the Fifth European Workshop on OpenMP, EWOMP'03*, Aachen, Germany, 2003.

[21] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 427–436, Washington, DC, USA, 2009. IEEE Computer Society.

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

[23] D. Turner and X. Chen. Protocol-dependent message-passing performance on linux clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, CLUSTER '02, pages 187–, Washington, DC, USA, 2002. IEEE Computer Society.

[24] S.-Y. Tzou and D. P. Anderson. The performance of message-passing using restricted virtual memory remapping. *Software - Practice and Experience*, 21:251–267, 1991.

[25] M. Woodacre, D. Robb, D. Roe, and K. Feind. The SGI Altix 3000 global shared-memory architecture. 2005.