Winter 2015

Accelerating MPI collective communications through hierarchical algorithms with flexible inter-node communication and imbalance awareness

Benjamin Scott Parsons
Purdue University

Graduate School Form 30
(Updated 11/20/2014)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Benjamin Parsons

Entitled
ACCELERATING MPI COLLECTIVE COMMUNICATIONS THROUGH HIERARCHICAL ALGORITHMS WITH FLEXIBLE INTER-NODE COMMUNICATION AND IMBALANCE AWARENESS

For the degree of Doctor of Philosophy

Is approved by the final examining committee:

Vijay Pai

Milind Kulkarni

Mithuna s. Thottethodi

Samuel P. Midkiff

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Vijay Pai

Approved by: V. Balakrishnan 02/23/2015

Head of the Department Graduate Program Date

ACCELERATING MPI COLLECTIVE COMMUNICATIONS THROUGH HIERARCHICAL ALGORITHMS WITH FLEXIBLE INTER-NODE COMMUNICATION AND IMBALANCE AWARENESS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Benjamin S. Parsons

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2015

Purdue University

West Lafayette, Indiana

ACKNOWLEDGMENTS

I would like to thank the Information Technology Laboratory of the Army Corps of Engineers' Engineering Research and Development Center. They both granted generous access to their advanced infrastructure as well as aided in its use. I would also like to thank the Science, Mathematics, and Research for Transformation (SMART) Scholarship, for support of my graduate work..

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Figure Page

SYMBOLS

p	Number of processors
α	Latency component of the Hockney timing model.
β	Bandwidth component of the Hockney timing model.
γ	Reduction cost component of the Hockney timing model.
n	Number of processes
ns	Number of cores per node
a_i	Arrival time of i
e_i	Exit time of i
$\overline{a}$	Average arrival time
$\overline{e}$	Average exit time
δ_i	Arrival imbalance of i
$\overline{\delta}$	Mean process arrival imbalance
ω_i	Worst case process arrival imbalance of i
$\overline{\omega}$	Mean worst case process arrival imbalance

ABBREVIATIONS

COW	Copy on write
IO	Input/Output
MPI	Message passing interface
NUMA	Non-uniform memory access
OS	Operating system
PAP	Process arrival pattern
RDMA	Remote direct memory access
TLB	Translation lookaside buffer
TSC	Time stamp counter

ABSTRACT

Parsons, Benjamin S. PhD, Purdue University, May 2015. Accelerating MPI Collective Communications through Hierarchical Algorithms with Flexible Inter-node Communication and Imbalance Awareness. Major Professor: Vijay S. Pai.

This work presents and evaluates algorithms for MPI collective communication operations on high performance systems. Collective communication algorithms are extensively investigated, and a universal algorithm to improve the performance of MPI collective operations on hierarchical clusters is introduced. This algorithm exploits shared-memory buffers for efficient intra-node communication while still allowing the use of unmodified, hierarchy-unaware traditional collectives for inter-node communication. The universal algorithm shows impressive performance results with a variety of collectives, improving upon the MPICH algorithms as well as the Cray MPT algorithms. Speedups average 15x - 30x for most collectives with improved scalability up to 65536 cores.

Further novel improvements are also proposed for inter-node communication. By utilizing algorithms which take advantage of multiple senders from the same shared memory buffer, an additional speedup of 2.5x can be achieved. The discussion also evaluates special-purpose extensions to improve intra-node communication. These extensions return a shared memory or copy-on-write protected buffer from the collective, which reduces or completely eliminates the second phase of intra-node communication.

The second part of this work improves the performance of MPI collective communication operations in the presence of imbalanced processes arrival times. High performance collective communications are crucial for the performance and scalability of applications, and imbalanced process arrival times are common in these

applications. A micro-benchmark is used to investigate the nature of process imbalance with perfectly balanced workloads, and understand the nature of inter- versus intra-node imbalance. These insights are then used to develop imbalance tolerant reduction, broadcast, and alltoall algorithms, which minimize the synchronization delay observed by early arriving processes. These algorithms have been implemented and tested on a Cray XE6 using up to 32k cores with varying buffer sizes and levels of imbalance. Results show speedups over MPICH averaging 18.9x for reduce, 5.3x for broadcast, and 6.9x for alltoall in the presence of high, but not unreasonable, imbalance.

1. INTRODUCTION

For large-scale computing systems running MPI, optimizing collective communication operations is an important consideration for maximizing system performance. This is increasingly true as systems grow larger in both the number of nodes and the cores per node. Collective communications have traditionally suffered from scalability problems, making their study essential for future systems [1].

As the use of multicore processors in clusters has become standard, research has looked at improving MPI performance for these hierarchical architectures. One branch of this research aims at better mapping traditional algorithms to hierarchical topologies [2–4]. These works do not directly utilize the shared memory of multiprocessors, but use the understanding that intra-node communication is appreciably faster than inter-node. Another branch of research shows that directly using shared memory can offer further improvements with certain algorithms [5–10]. Nevertheless, Zhu et al. demonstrate that on a 16-node cluster of CMPs, the ideal speedup from utilizing hierarchical algorithms is greatly limited by the inter-node communication portion of the algorithm [11]. This severely hampers past works that create optimized hierarchical algorithms, as they are dependent on one inter-node communication algorithm.

This work proposes a general hierarchical algorithm that uses shared memory within a multiprocessor node but allows for independent inter-node and intra-node communication algorithms. This would allow an MPI implementation to choose these algorithms independently, based on the conditions of each portion of communication. A substantial amount of prior work is dedicated to the optimization of inter-node communications [12], as well as a newer body of work on intra-node collective algorithms for specific architectures. These intra-node algorithms account for features ranging from heterogeneous architectures [2, 13] to common NUMA systems [14]. Our system allows these and future works to be seamlessly incorporated into the

intra-node phases of the algorithm, without modification of the inter-node portion. Likewise, different inter-node communication algorithms can be used, allowing their continued evolution as network designs change [15].

Using these techniques, this study analyzes the performance of six collective algorithms over a variety of message sizes, and up to 65536 processes. This demonstrates that the new hierarchical algorithm utilizing MPICH algorithms for inter-node communication experience significant speedups averaging 6.7x for alltoallv, 22x for allgather, and 45x for reduce-scatter. To demonstrate the flexibility of this hierarchical algorithm, this work evaluates it with routines from the Cray MPT for inter-node communication. This shows that even the highly tuned Cray algorithms can see large speedups, averaging 21.7x for alltoallv, 29.9x for allgather, and 19.8x for reduce-scatter.

After introducing the new hierarchical algorithm, this study explores improving the inter-node communication further by utilizing multiple senders communicating from the same shared memory buffer. Prior works have observed that the network is not fully utilized with certain hierarchical algorithms, because only one process on the node participates in inter-node communication. These works have looked to mitigate this, but do so by reducing shared memory communication. These new algorithms modify traditional collective algorithms so multiple senders fully utilize the network from the same shared-memory buffer, thus avoiding the downsides of previous works and achieving speedups of 2.5x.

In addition to examining inter-node communication, the universal hierarchical algorithms flexibility is demonstrated using several intra-node communication algorithms. This work also explores several intra-node copy methods, which vary in effectiveness based on the buffer size. New special-case extensions are introduced that return shared memory buffers or use copy-on-write protection to reduce intra-node communication latencies when applicable.

The second portion of this work further improves collective communications in the presence of imbalanced process arrival times. This begins by studying the process

imbalance patterns for a modern supercomputer. Recent studies have shown that the assumption that all processes arrive at a collective operation at the same time are not true, and processes often suffer a larger synchronization delay than the collective latency [16]. Further improving collective algorithms with regard to process imbalance has been identified as a key improvement needed to move MPI into the exascale era [17]. These imbalance patterns have never been studied within a hierarchical cluster, and this report details characteristics of process imbalance that have not been reported. Further understanding this imbalance is key to creating algorithms to tolerate it.

The first contribution in this area is a study of process imbalance, revealing the nature of the imbalance in more detail than past works. The imbalance is examined on an inter- and intra-node level to understand how this imbalance impacts hierarchical collectives. This understanding of the arrival pattern is then used to develop imbalance-tolerant hierarchical algorithms. Because of the large speedup provided by hierarchical collectives, they are a natural starting point for imbalance-tolerant collectives. Reduce, broadcast, and alltoall are chosen for optimizations because they represent all-to-one, one-to-all, and all-to-all communication patterns. Expanding the principles of these algorithms to other collectives would be relatively straightforward.

The reduction and broadcast algorithms utilize a novel method known as *dynamic leader selection* to opportunistically select each node leader based on the imbalance pattern at every invocation of a collective. This method allows the early arriving processes to perform as much communication as possible before the later processes arrive. The alltoall algorithm utilizes multiple senders to avoid delays from late arriving leaders, and utilizes *opportunistic message fragmentation* to pre-send data from early arriving processes, without delaying for later ones.

These algorithms are tested on a Cray XE6 running up to 32k cores and show speedups over MPICH of 18.9x for reduce, 5.3x for broadcast, and 6.9x for alltoall when averaged across all buffer sizes, levels of imbalance, and numbers of processes.

These algorithms also out perform the default algorithms on the Cray system, with speedups averaging 35x for reduce, 5.3x for broadcast, and 2.1x for alltoall.

2. ACCELERATING MPI COLLECTIVE COMMUNICATIONS THROUGH HIERARCHICAL ALGORITHMS WITHOUT SACRIFICING INTER-NODE COMMUNICATION FLEXIBILITY

2.1 Introduction

For large-scale computing systems running MPI, optimizing collective communication operations is an important consideration for maximizing system performance. This is increasingly true as systems grow larger in both the number of nodes and the cores per node. Collective communications have traditionally suffered from scalability problems, making their study essential for future systems [1].

As the use of multicore processors in clusters has become standard, research has looked at improving MPI performance for these hierarchical architectures. One branch of this research aims at better mapping traditional algorithms to hierarchical topologies [2–4]. These works do not directly utilize the shared memory of multiprocessors, but use the understanding that intra-node communication is appreciably faster than inter-node. Another branch of research shows that directly using shared memory can offer further improvements with certain algorithms [5–10]. Nevertheless, Zhu et al. demonstrate that on a 16-node cluster of CMPs, the ideal speedup from utilizing hierarchical algorithms is greatly limited by the inter-node communication portion of the algorithm [11]. This severely hampers past works that create optimized hierarchical algorithms, as they are dependent on one inter-node communication algorithm.

In this study, we propose a general hierarchical algorithm that uses shared memory within a multiprocessor node but allows for independent inter-node and intra-node communication algorithms. This would allow an MPI implementation to choose these algorithms independently, based on the conditions of each portion of communication.

A substantial amount of prior work is dedicated to the optimization of inter-node communications [12], as well as a newer body of work on intra-node collective algorithms for specific architectures. These intra-node algorithms account for features ranging from heterogeneous architectures [2, 13] to common NUMA systems [14]. Our system allows these and future works to be seamlessly incorporated into the intra-node phases of the algorithm, without modification of the inter-node portion. Likewise, different inter-node communication algorithms can be used, allowing their continued evolution as network designs change [15].

Using these techniques, we analyze the performance of six collective algorithms over a variety of message sizes, and up to 65536 processes. We show that the new hierarchical algorithm utilizing MPICH algorithms for inter-node communication experience significant speedups averaging 6.7x for alltoallv, 22x for allgather, and 45x for reduce-scatter. To demonstrate the flexibility of this hierarchical algorithm, we also evaluate it with routines from the Cray MPT for inter-node communication. We show that even the highly tuned Cray algorithms can see large speedups, averaging 21.7x for alltoallv, 29.9x for allgather, and 19.8x for reduce-scatter.

After introducing the new hierarchical algorithm, we explore improving the inter-node communication further by utilizing multiple senders communicating from the same shared memory buffer. Prior works have observed that the network is not fully utilized with certain hierarchical algorithms, because only one process on the node participates in inter-node communication. These works have looked to mitigate this, but do so by reducing shared memory communication. In this work we modify traditional collective algorithms so multiple senders fully utilize the network from the same shared-memory buffer, thus avoiding the downsides of previous works and achieving speedups of 2.5x.

In addition to examining inter-node communication, we demonstrate the flexibility of our algorithm using several intra-node communication algorithms. We offer intra-node copy methods, which vary in effectiveness based on the buffer size. We also

implement special-case extensions that return shared memory buffers or use copy-on-write protection to reduce intra-node communication latencies when applicable.

2.2 Background

Collective communication optimization has been shown necessary for MPI scalability [1]. Traditional collective algorithms have been well studied, and for each type of collective many algorithms exist [12, 18]. Different algorithms are better suited for different message sizes, numbers of processors, and networks. MPI implementations, like MPICH [19], choose the most appropriate algorithm to use at run time. This section discusses collective optimizations that exploit features of multicore processors and nodes.

2.2.1 Hierarchical Optimizations

Hierarchical optimizations are useful in clusters of multicore processors as well as other hierarchical domains. Kielmann et al. created hierarchical algorithms for clustered wide-area systems with fast networks within a cluster and a slow network between them [20, 21]. Karonis et al. used a similar approach for several fan-in and fan-out algorithms, going a step further by automating the process within MPI for arbitrary levels of hierarchy [4]. Sanders and Träff presented one of the few works to target alltoall for hierarchical clusters, optimally scheduling the sequence of messages in rounds [3].

2.2.2 Shared Memory Hierarchical Optimizations

Sistare et al. developed collective algorithms for broadcast, reduce, allreduce, and barrier that use shared memory buffers for intra-node communication, noting that utilizing shared memory within a node provided additional performance improvements over hierarchical optimizations that only took the topology into consideration [9].

Tipparaju et al. created *Shared-Remote-Memory* collectives, which replace traditional point-to-point messages with intra-node shared memory communication and inter-node remote memory accesses (RMA) [8]. These collectives improve performance by utilizing intra-node shared memory for communication and avoiding extra copies to MPI buffers. This work targeted broadcast, reduce, allreduce, and barrier. While it depends on RMA, the hierarchical nature of the algorithms is similar to algorithms which use point-to-point communication. Mamidala et al. implemented allgather in a similar way, again using shared memory within a node and RMA between [6].

Mamidala et al. explore several multicore collective optimizations including a hierarchical allgather [5]. The allgather algorithm first performs a local allgather into a shared memory buffer, before having one designated process from each node (the *leader*) participate in an inter-node Allgather. This work is further explored by Kandalla et al. which uses multiple leaders per node to increase network performance [7]. This however, reduces the degree to which shared memory is utilized, because leaders on the same node communicate via point-to-point communication.

Cheetah is a hierarchical collective framework developed by Graham et al. for barrier and broadcast operations [10]. It takes advantage of shared memory for intra-node communications and uses either MPI point-to-point messages or specialized Infiniband capabilities for inter-node communication.

2.2.3 Architecture Dependent Optimizations

Several works have focused on improving collective communications taking place on a single SMP, some of which can also be used hierarchically in fan-in and fan-out algorithms [14, 22, 23]. Other works have proposed optimizations that target the unique communication features of specific processors, including Intel x86, AMD x86, Cell, and Intel Single-Chip Cloud processors [2, 5, 13].

2.2.4 Operating System Assistance

Several works develop specialized kernels to reduce communication time, for both point-to-point and collective communications. SMARTMAP [24] modified the kernel using a virtual memory technique that allowed processes to directly access each others memory. SMARTMAP also made specific optimizations for intra-node collectives [25], and used these optimizations in hierarchical versions for broadcast, reduce, and allreduce.

The KNEM kernel module also allows direct memory copies between processes [26, 27]. Ma et al. [28] use KNEM and add additional optimizations to their implementation for collectives, called HierKNEM. HierKNEM provides hierarchical algorithms for broadcast, reduce, and allgather that use message pipelining and KNEM memory copies to completely overlap the intra-node and inter-node communication latency.

Tang and Yang implement MPI using threads on nodes rather than processes, and include hierarchical optimizations for broadcast, reduce, and allreduce [29].

2.2.5 MPI Configuration and Collective Performance Model

MPI can be run with different configurations, but generally each *process* runs on one *processor*, and is numbered by its *rank*, ranging from 0 to the number of processors (p). Schedulers generally arrange MPI ranks onto systems in a consecutive fashion. Further discussion will assume MPI is running in this fashion. If the ranks were non-consecutive all hierarchical algorithms would need to be modified, with most only needing their buffers rearranged before or after the collective.

The Hockney cost model is a simple model for collective algorithms [30]. It models any messages sent between nodes as $\alpha + n\beta$. In this model α is the message start-up latency per message, β is the transfer time per byte, and n is the number of bytes. For reduction operations γ represents the computation time per byte. A detailed description and analysis this and move advanced cost models can be obtained from Pješivac-Grbović et al. [18].

2.3 Algorithms

2.3.1 Universal Hierarchical Algorithm

In this section we outline how our universal hierarchical algorithm operates and can utilize traditional MPI collective routines that are hierarchy-oblivious for inter-node communication. The fact that the inter-node algorithms have no knowledge that they are being used in this fashion is an important contribution because it allows arbitrary inter-node algorithms. Past works have shown that hierarchical optimizations can provide impressive speed-ups, but inter-node communication is still the primary contributor to overall latency [12].

Our algorithm aims to improve on past works in several key ways. First, our work completely decouples the choice of inter- and intra-node communication algorithms. This is done at the expense of overlapping intra-node copies with inter-node communication. We make this compromise to allow any inter-node communication algorithm to be used, knowing that inter-node communication dominates total latency. Another improvement over past works is that our system works with a wider variety of collectives. Past works have used shared memory to optimize allgather, as well as fan-in and fan-out algorithms like broadcast and reduce. We will show our method works for these collectives as well as the more complex alltoall. Additionally, we show our algorithm accommodates collectives with vector versions, e.g., alltoallv. These algorithms provide a greater challenge because of variable counts and input displacements. Figure 2.1 gives an overview of the algorithm.

The *ExchangeCntData* portion of the algorithm is only needed for vector versions of collectives without global knowledge of count arrays, including alltoall(v/w), scatterv, and gatherv. In these collectives, every rank can send and receive varied amounts of data, so the proper index into the shared memory buffer needs to be calculated. (allgatherv is the only vector collective with global knowledge of send counts, because all ranks have a receive count array.) An interesting feature of our algorithm is that the non-global data is never made global: it is only shared within a

```
function
    if (UnknownGlobalPattern) then
        ExchangeCntData(SharedCnts, *Cnts)
        Barrier(Intra_Node_Communicator)
        CopyIn(SharedBuff, PrivateBuff, SharedCnts)
        SetupInter(InterCnts, SharedCnts)
    else
        CopyIn(SharedBuff, PrivateBuff, *Cnts)
        SetupInter(InterCnts, *Cnts)
    end if
    Barrier(Intra_Node_Communicator)
    if (rank == Leader) then
        MPI_Collective(SharedBuff, *InterCnts, Leader_Comm)
    end if
    Barrier(Intra_Node_Communicator)
    CopyOut(PrivateBuff, SharedBuff, *Cnts)
end function
```

Fig. 2.1.: Universal hierarchical algorithm, function outputs in bold

node via a shared memory structure, *SharedCnts*. An important note with the vector collectives is because every rank transfers its data to and from shared memory using the displacement arrays, non-contiguous data becomes contiguous for the inter-node portion of the algorithm. This also means that the displacement arrays do not need to be communicated during the *ExchangeCntData* phase.

The *copy-in* and *copy-out* portions of the algorithm make up the intra-node communication, transferring data between private (*PrivateBuff*) and shared (*SharedBuff*) buffers. These functions are different for each collective. For broadcast, the entire buffer is copied in or out, but alltoall has to place data in a methodical manner, discussed in detail in Section 2.3.1. For reduction operations, this is the phase where the intra-node reduction is carried out, resulting in the shared buffer containing the reduced data from all the private buffers on the node.

Once the data is transferred to a shared buffer, the arguments for the inter-node collective need to be calculated. This is done by the *SetupInter* function, which produces *InterCnts*. This involves calculating the counts and the displacements vectors

needed as inputs to the MPI collective for the inter-node communication phase, which is completely isolated. This is what allows any routine to be used for inter-node communication. Past works have tried to overlap inter- and intra-node communication, which results in the algorithms being inherently intertwined.

The universal hierarchical algorithm uses several communicators in addition to MPI_COMM_WORLD. Each node has a communicator including all the ranks on that node, and is used for node synchronization. A communicator is also created that includes only the leaders, and is used to call the MPI collective in line 13 of Figure 2.1.

The following subsections will describe several algorithms selected from MPICH to provide a wide range of collective communication patterns [12, 19].

Allgather

Allgather is a collective communication routine similar to gather, except each process obtains a copy of the entire gathered buffer. A naive implementation would simply be a gather followed by a broadcast.

The *copy-in* portion of the allgather algorithm involves each process copying its private send buffer into adjacent locations in the receive buffer. The *copy-out* portion of the algorithm is a broadcast of the shared memory buffer to the private receive buffers. This can be implemented in different ways as described in section 2.3.3

Bruck's algorithm for allgather takes $\lceil \log p \rceil$ steps for p processes making it good for small message sizes [31]. At step k, each node i sends data to rank $((i+2^k) \bmod p)$ and receives data from rank $((i-2^k) \bmod p)$. After each step, it doubles the amount of data it sends and receives. If the number of processes is not a power of two, an extra step is used to perform a partial send, ensuring each process has all the data. The data in the receive buffer is then reordered so that the data from rank zero inhabits the first portion of the buffer.

The cost model for Bruck's algorithm is $T_{Allgather-Bruck} = \lceil \log p \rceil \alpha + \frac{p-1}{p} n\beta$. When using Bruck's Algorithm for the inter-node communication portion of our algorithm the number of senders involved is decreased, and the cost model is reduced to $T_{Allgather-Bruck-Hier.} = \lceil \log \frac{p}{ns} \rceil \alpha + \frac{p-ns}{p} n\beta$ where ns is the node size. This has the potential to dramatically reduce the α coefficient with large node sizes.

We also utilize the ring algorithm, which is preferred for large messages sizes. It has an advantage over Bruck's Algorithm because each process only communicates with its nearest neighbors, minimizing network contention. It performs poorly for small messages because it takes $p-1$ steps, where Bruck's only needs $\lceil \log p \rceil$. The cost model is $T_{Allgather-Ring} = (p-1)\alpha + \frac{p-1}{p} n\beta$. When using it in the inter-node portion of our algorithm it becomes $T_{Allgather-Ring-Hier.} = (\frac{p}{ns} - 1)\alpha + \frac{p-ns}{p} n\beta$. Like Bruck's algorithm the β coefficient is minimally changed and the α coefficient is reduced.

In several graphs and tables these algorithms will appear alongside two Cray allgather algorithms. We will refer to the graphs as allgather-*small* and -*large* for Bruck's and Ring respectively, and display each algorithm with the corresponding Cray algorithm.

Alltoall

Alltoall requires special attention because there is not a straightforward way to create the send buffers for the inter-node phase. Because the inter-node algorithm is not aware of the hierarchy, the send buffers need to be created so that the same semantics can be followed by the inter-node algorithm. To do this, each rank places messages destined for the same node in contiguous memory locations, as shown in Figure 2.2. The inter-node alltoall algorithm can then be called with the shared buffer as an argument, using the modified count and displacement vectors from *SetupInter*. Figure 2.2 also shows the *copy-out* data movement. Because the data is grouped systematically in the the *copy-in* phase, the *copy-out* phase just needs to transfer the

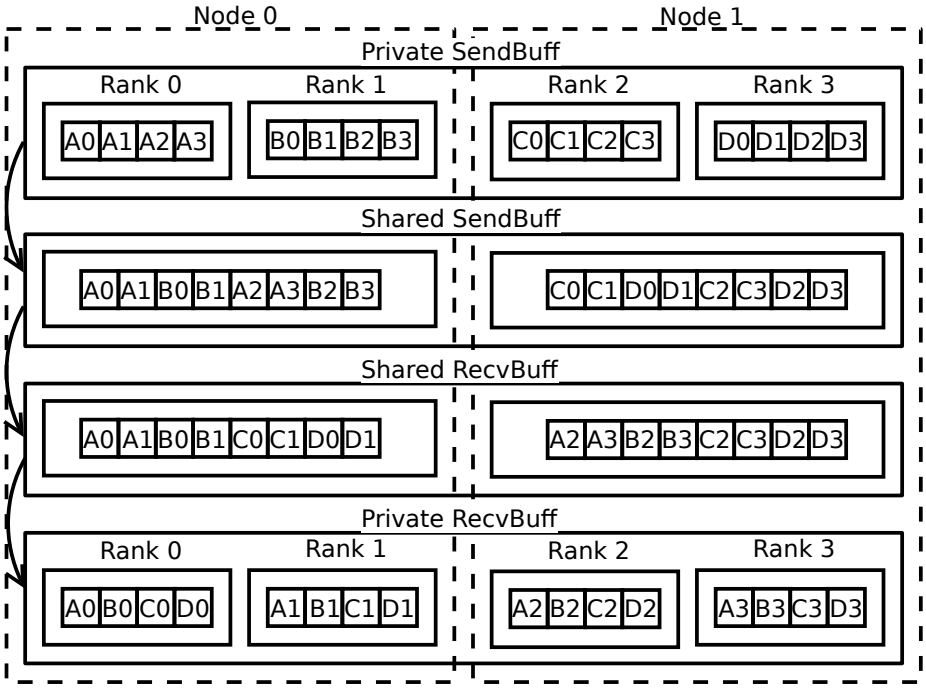


Fig. 2.2.: Alltoall buffer copy example

data into the correct private buffer. The alltoallv version uses the same method for the intra-node communication, except the buffers are not equal length, so the buffer offsets are calculated from the shared data structure.

For alltoallv, MPICH uses a series of individual messages between ranks. Because each rank sends and receives a potentially unique amount of data, optimizations for this algorithm are difficult. The cost model for this algorithm, if it had equal message sizes, is $T_{Alltoall} = (p-1)\alpha + \frac{p-1}{p}n\beta$, and when used for the inter-node portion of our algorithm it becomes $T_{AlltoallHier.} = (\frac{p}{ns} - 1)\alpha + \frac{p-ns}{p}n\beta$.

Scatter and Broadcast

For scatter and broadcast, we investigated both using the binomial tree algorithm. For scatter the algorithm starts by sending the second half of the buffer, size $\frac{n}{2}$ to rank $\frac{p}{2}$. Rank $\frac{p}{2}$ now has the data for ranks $i > \frac{p}{2}$ and rank zero is responsible for $i < \frac{p}{2}$. This continues recursively until all processes have data. For broadcast the same communication pattern is used, but the entire buffer is sent.

The *copy-in* portion of these collectives is simply the root node transferring its data to the shared memory buffer. The *copy-out* portion of scatter involves each rank transferring its piece of the shared memory buffer to private memory, and for broadcast each rank receives the entire buffer.

It is possible to create SMP-aware fan-in and fan-out algorithms using point-to-point MPI routines, by minimizing the inter-node transfers, as done by Karonis et al. [4]. These algorithms create a communication tree that ensure SMP-nodes map communication subtrees, so the first several levels of communication are intra-node. The MPICH binomial tree algorithm we evaluate is not SMP-aware, but we arrange it in a SMP-friendly fashion. When the root is 0, and node sizes are a power of 2, the binomial tree algorithm has a minimal number of inter-node transfers. We will use the term *SMP-friendly* to describe algorithms that are configured in this way, because they produce the same communication patterns are SMP-aware algorithms.

The cost model for the scatter algorithm is $T_{Scatter-Tree} = \lceil \log p \rceil \alpha + \frac{p-1}{p} n\beta$. When used as the inter-node communication portion of our algorithm it is reduced to $T_{Scatter-TreeHier.} = \lceil \log \frac{p}{ns} \rceil \alpha + \frac{p-ns}{p} n\beta$. The broadcast cost model is similar, $T_{Broadcast} = \lceil \log p \rceil (\alpha + n\beta)$ but reflects that more data is transferred. It is subsequently reduced to $T_{BroadcastHier.} = \lceil \log \frac{p}{ns} \rceil (\alpha + n\beta)$ when used as the inter-node communication portion of our algorithm. While both these algorithms enjoy a reduced α coefficient, the reduction does not realize the same performance benefits as other algorithms. The basic Hockney cost model we use does not take the difference of inter- and intra-node latencies into account. Because we run these two collectives in an SMP-friendly way, the last $\log ns$ rounds of communication are intra-node. This does not reduce inter-node communication like the other collectives, so we should not expect to see the same performance improvements, despite the cost model.

Reduce-Scatter

The reduce-scatter algorithm is similar to reduce, but the results scattered among all the processes. A naive implementation is simply a reduce followed by a scatter, but this can yield poor performance due to contention at the root. For long messages, MPICH uses a pair-wise exchange algorithm. This algorithm assumes that the reduction operation is both commutative and associative. For operations where this is not the case, like user-defined operations, different algorithms must be used. Kielmann et al. give a detailed description of hierarchical reduction algorithms and issues with non-commutative and non-associative operations [32].

Reduce-scatter is unique from the other *copy-in* algorithms in that it must perform the reduction operation on the data as it is transferred to the shared buffer. Our implementation does a parallel reduction in which each process reduces a n/ns sized portion of the buffer, taking ns steps. The *copy-out* algorithm is the same as scatter.

The cost model for this algorithm is $T_{Reduce-Scatter} = (p-1)\alpha + \frac{p-1}{p} n\beta + \frac{p-1}{p} n\gamma$ and when used for the inter-node communication portion of our algorithm it becomes

$T_{Reduce-ScatterHier.} = (\frac{p}{ns} - 1)\alpha + \frac{p-ns}{p}n\beta + \frac{p-ns}{p}n\gamma$. Like the prior collectives, this reduces the α coefficient.

Cray MPT

In addition to these algorithms from MPICH we also use the algorithms from the Cray MPT to perform the inter-node communication. This demonstrates how our algorithms can use any collective for the inter-node communication. The MPT documentation [33] gives general insight into the algorithms used for collectives, but not any source code. Allgather uses two algorithms: small messages use an optimized gather/broadcast algorithm, and large messages use an optimized ring algorithm. Environment variables allow us to adjust the message size cutoff and isolate these algorithms for our results, referring to them as the Cray *small* and *large* algorithms. The broadcast algorithm is an SMP-aware tree and the other algorithms are not specified. Additionally, Cray uses various optimizations on the collective algorithms, including "balanced injection" for the network interface and "non-default, architecture-specific algorithms".

2.3.2 Multi-Sender Inter-Node Communication

Past works have observed that hierarchical algorithms inefficiently utilize the network because they have only one process per node communicating [7]. Microbenchmarking showed us that increasing the number of senders between two nodes provided up to a 5x increase in bandwidth on the Cray XE6, and up to a 10x bandwidth increase over Gigabit Ethernet. In both cases, these speedups decreased as the message size increased and the network links became saturated. One solution from past work is to have only a portion of the processes on a node communicate through shared memory to their leader, with a variable number of leaders now possible on each node. This increases the network utilization, but at the expense of shared memory communication. Because only a subset of the ranks communicate through shared

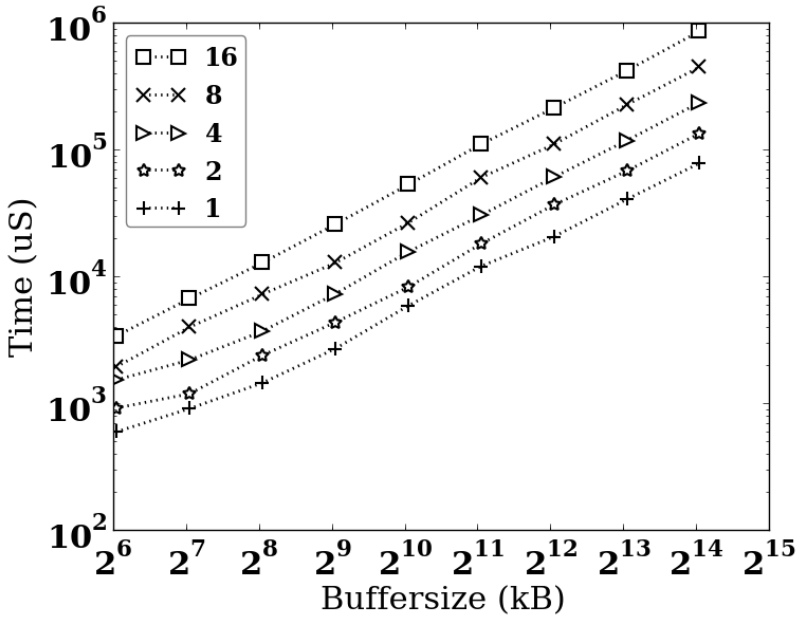


Fig. 2.3.: Allgather-Bruck latency vs buffer size with varying numbers of leaders, as proposed by prior work

memory to their leader, the ns term in the cost model becomes $ns/num\_leaders$. Figure 2.3 shows the latency of Bruck's algorithm with different numbers of leaders, all running on 4096 processors. It is important to note, that while our tests showed degraded performance when increasing the leaders, other networks could see an improvement due to better network utilization. Multi-ported networks without multi-rail MPI implementations are a prime example of this. Past works have show that using multiple leaders give speedups on smaller clusters, and placing one leader in each NUMA domain is an effective strategy. [7].

To avoid the performance degradation we observed on our system when using multiple leaders, we propose a series of algorithms that utilize multiple-senders communicating from the same shared memory buffer. This allows all ranks on a node to communicate through shared memory and multiple processes can communicate over the network. This leads to better network utilization, without the downside of prior work. Section ref gives these results, and shows how our new algorithm increases network utilization without the performance degradation.

One advantage of our universal hierarchical algorithm is that optimizations like this are possible and easy to implement without modifying the intra-node communication. We implemented several multi-sender variants of the MPICH algorithms described above. In order to use these algorithms within our universal hierarchical algorithm, line 12 of Figure 2.1 needs to be modified to allow all processes to call the multi-sender inter-node routine, which is multi-threaded. With these algorithms it is possible to vary the number of senders used. We found that as the number increased, the performance gains diminished, but was never degraded. All of our multi-sender results use all processes as senders.

Scatter

The multi-sender scatter algorithm is based on the binomial tree algorithm. To utilize multiple senders effectively we modified the MPICH algorithm to use a k-ary tree.

Broadcast

The multi-sender version of broadcast is again based on the binomial tree algorithm. Because broadcast retransmits data, we found the k-ary tree variant to be ineffective. While multiple senders increases the network bandwidth, it also increases the latency of any one of the k messages, and thus the latency before a receiver can begin retransmitting the data. To avoid this, the multi-sender version of broadcast is a pipelined variation of the binomial tree algorithm, where each sender is responsible for a segment of the total buffer.

Allgather

For Bruck's algorithm the multi-sender version uses a k-ary style modification, like the binomial tree algorithm for scatter. At each round of communication, instead of

communicating the buffer information to one other node, each sender transmits the buffer, reducing the number of rounds needed, and making the message size increase by k each round. The allgather ring algorithm is unique because it is designed to reduce network contention for large message sizes. The multi-sender version keeps this goal in mind, still using nearest neighbor communication, but with a two-way ring. This method only works with two senders per node, but keeps the same low-contention communication pattern as the original algorithm.

Reduce-Scatter

The multi-sender reduce-scatter utilizes multiple senders, performing the pair-wise exchange in parallel. The senders then place the received data into a buffer, and mark a flag for the reducer. The reducer takes the data from the temporary buffer and performs the reduction operation moving the data into the final receive buffer. Because we reserve one process for performing the reduction operation, this collective uses all but one of the processes as senders.

2.3.3 Intra-node copy optimizations

As this work targets large node sizes, memory copies into and out of shared buffers can be expensive. The following section details several copy methods we employ in our evaluation.

Memory Copies

The straightforward way to transfer the data from the shared buffers to private ones is a direct memory copy. This has very low latency for small messages, but memory contention can become an issue for larger ones. For allgather and broadcast we implement several *copy-out* methods, but all other copy routines use memory copies.

MPI Broadcast

Another method to transfer the shared buffer to all ranks on a node is to use MPI_Bcast. In our case this uses a binomial tree algorithm, which reduces the amount of data transferred between processor sockets, which have less bandwidth.

NUMA Friendly Hybrid

The NUMA Friendly Hybrid algorithm seeks to provide better memory bandwidth utilization on a generic four-region NUMA architecture. This algorithm begins with one rank from each NUMA region copying the shared memory buffer into its private buffer. Those ranks then use asynchronous sends to propagate the data to other ranks in the same NUMA region. Because our MPI implementation uses XPMEM for single copy communication on intra-node messages, we do not experience any performance degradation from double copies. Copy algorithms that take cache hierarchy into account could further improve performance, but this algorithm is designed to work well on most NUMA systems.

Shared Memory

In order to eliminate some intra-node memory transfer time we evaluate versions of the collectives that return a shared memory buffer to all ranks, instead of private buffer. Returning shared memory buffers instead of private buffers defies the MPI standard, but eliminates the *copy-out* time. It is a feasible solution when the buffer is only ever read after the collective, or the programmer is aware of shared memory and uses appropriate synchronization. The MPI 3.0 standard allows ranks to directly allocate shared memory, bringing shared memory into a traditional distributed paradigm [34].

Copy-on-Write

In addition to returning a shared memory buffer, we created versions of our algorithms that return copy-on-write protected buffers, which appear identical to private buffers. In order to create a copy-on-write buffer several costly system calls are needed, so it is only viable for large sized buffers. These buffers also have the potential to hurt program performance after the collective communication, if writes occur to the buffer and force memory page copies.

2.4 Experimental Testbed

All tests were run on a Cray XE6 with a dedicated Cray Gemini network. Each node on the system has two 16 core 2.5 GHz AMD 6200 Opteron processors, 64 GB of DDR3 memory, and runs the Cray Linux Environment 4.0.46. Jobs are scheduled using PBS Pro 10.4.7. To mitigate any result error from job placement or other network communication, multiple timing runs were used and the results averaged, which are the latencies we report. In practice, we found very little difference in the results when comparing different jobs from test to test. For all test sizes, multiple iterations were run, varying from fifty to several thousand, depending on the latency of the collective. This ensured that the small-sized tests received an adequate number of iterations for repeatable results, while still making large sized tests feasible.

The Cray Message Passing Toolkit 6.0 (MPT) was used for all MPI communication. It is the default MPI implementation for the Cray XE6, and is built upon MPICH. MPICH collective algorithms are built on point-to-point operations, so a minimally modified version of the MPICH 3.0.4 source code was run on the Cray system.

The buffer size for all graphs is given as the total buffer size of the largest buffer. For scatter, this is the send buffer, while for allgather, it is the receive buffer. The smallest buffer size we report is the minimum size for scatter, resulting in one integer per rank. Integers were used as the data type for all collectives, and the reduce-

Table 2.1.: Universal hierarchical algorithm speedups relative to base library, 32k cores, mean and max for all sizes

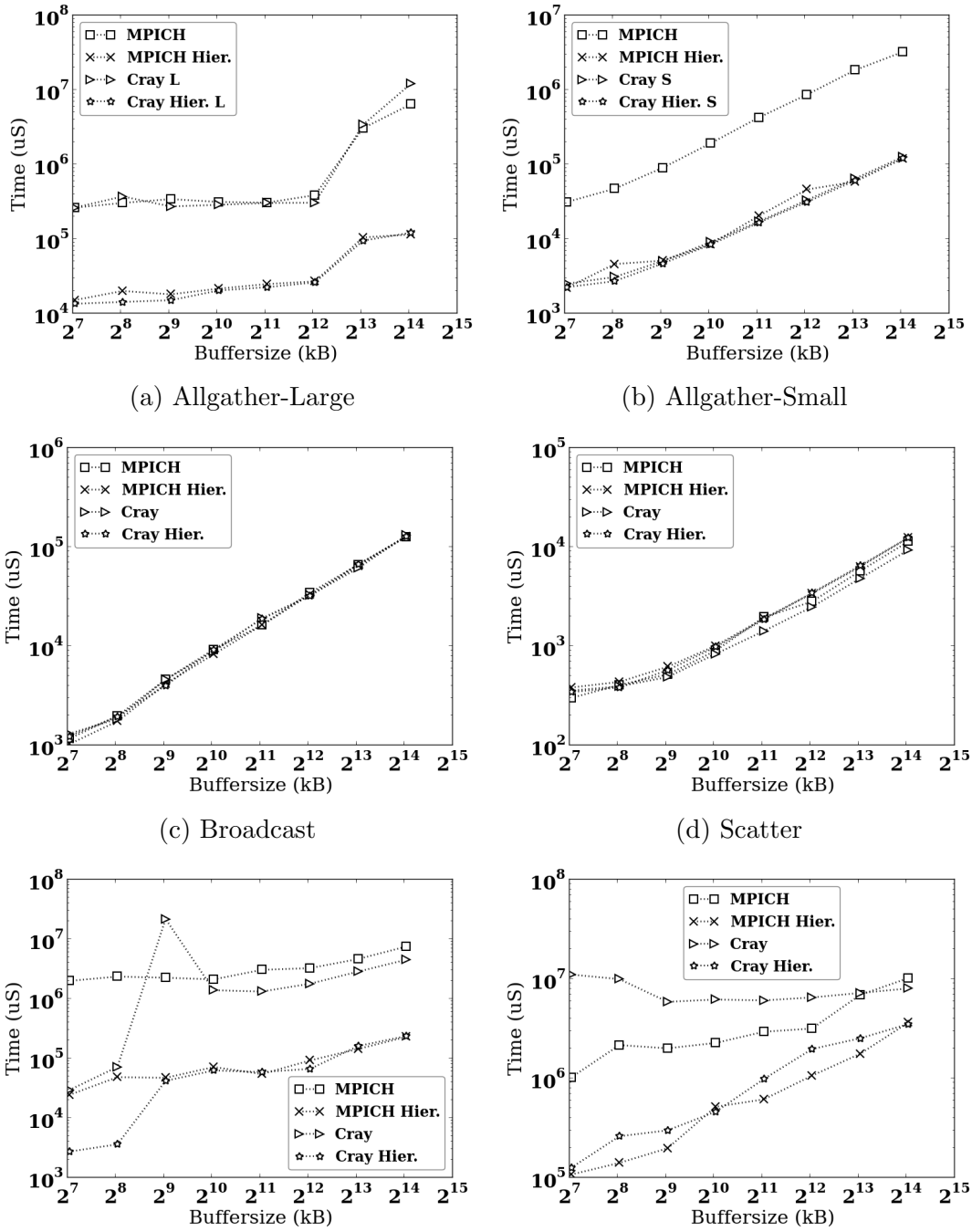
	Base Library			
	MPICH		**Cray**	
Collective	Mean	Max	Mean	Max
Allgather-Small	20.3	31.4	1.1	1.1
Allgather-Large	22.2	56.0	29.9	100.1
Alltoallv	6.7	15.3	21.7	87.8
Broadcast	1.1	1.1	1.0	1.2
Reduce-Scatter	45.5	81.8	19.8	26.8
Scatter	.9	1.0	.85	1.0

scatter reduction operation was a summation. For the alltoallv tests, every other buffer length was shortened by one integer. This allowed generally even latencies sending the buffers, without being so simple that using alltoall would suffice. The alltoallv algorithm was validated with more complex communication patterns.

2.5 Results

2.5.1 Universal Hierarchical Algorithm

Figure 2.4 compares the MPICH and Cray algorithms when used independently and as the inter-node communication portion of our algorithm. The graphs show varying buffer sizes for all collectives when run with 32768 cores. Table 2.1 gives the speedups over the same buffer size range. As described in Section 2.3, the Cray algorithms are closed source, but our algorithm allows the use of any traditional collective that performs the correct operation. The universal hierarchical algorithm performs very well on all collectives except the ones using fan-in and fan-out communication. As discussed in Section 2.3.1 these algorithms are SMP-friendly, and despite a better cost model, no inter-node communication is eliminated. The Cray algorithm for small allgather sizes also falls into this category, because it uses a gather/broadcast algorithm.



(a) Allgather-Large (b) Allgather-Small

(c) Broadcast (d) Scatter

(e) Reduce-Scatter (f) Alltoallv

Fig. 2.4.: Collective latencies vs buffer size, 32k cores

While the algorithm for Cray's reduce-scatter is not specified, the small message sizes likely use a reduce followed by a scatter. These algorithms are SMP-friendly

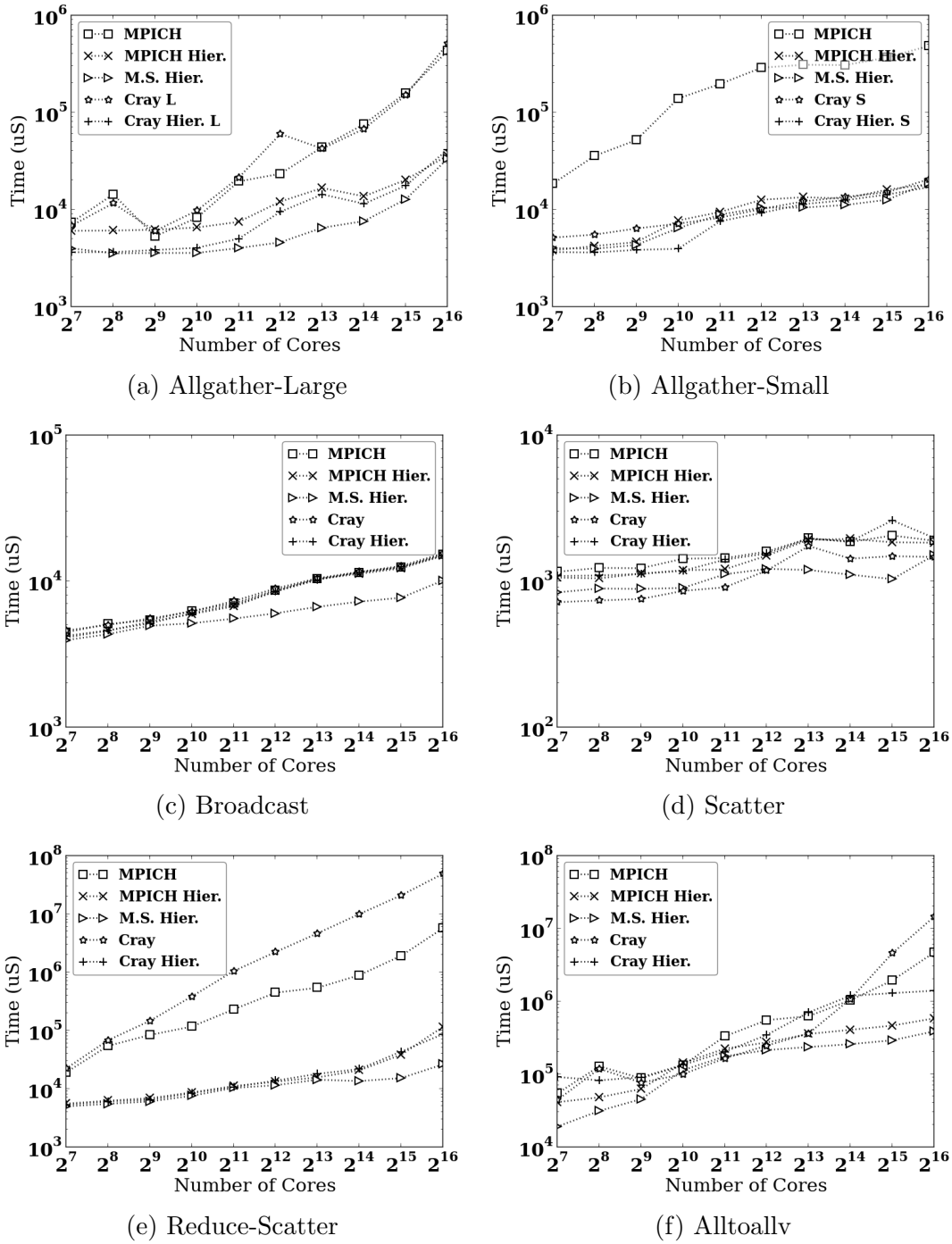


Fig. 2.5.: Collective latencies vs number of cores, 2 MB buffer size

so this would explain why the smallest two message sizes in Figure 2.4e have lower latency than the MPICH pairwise exchange algorithm. The large message sizes prob-

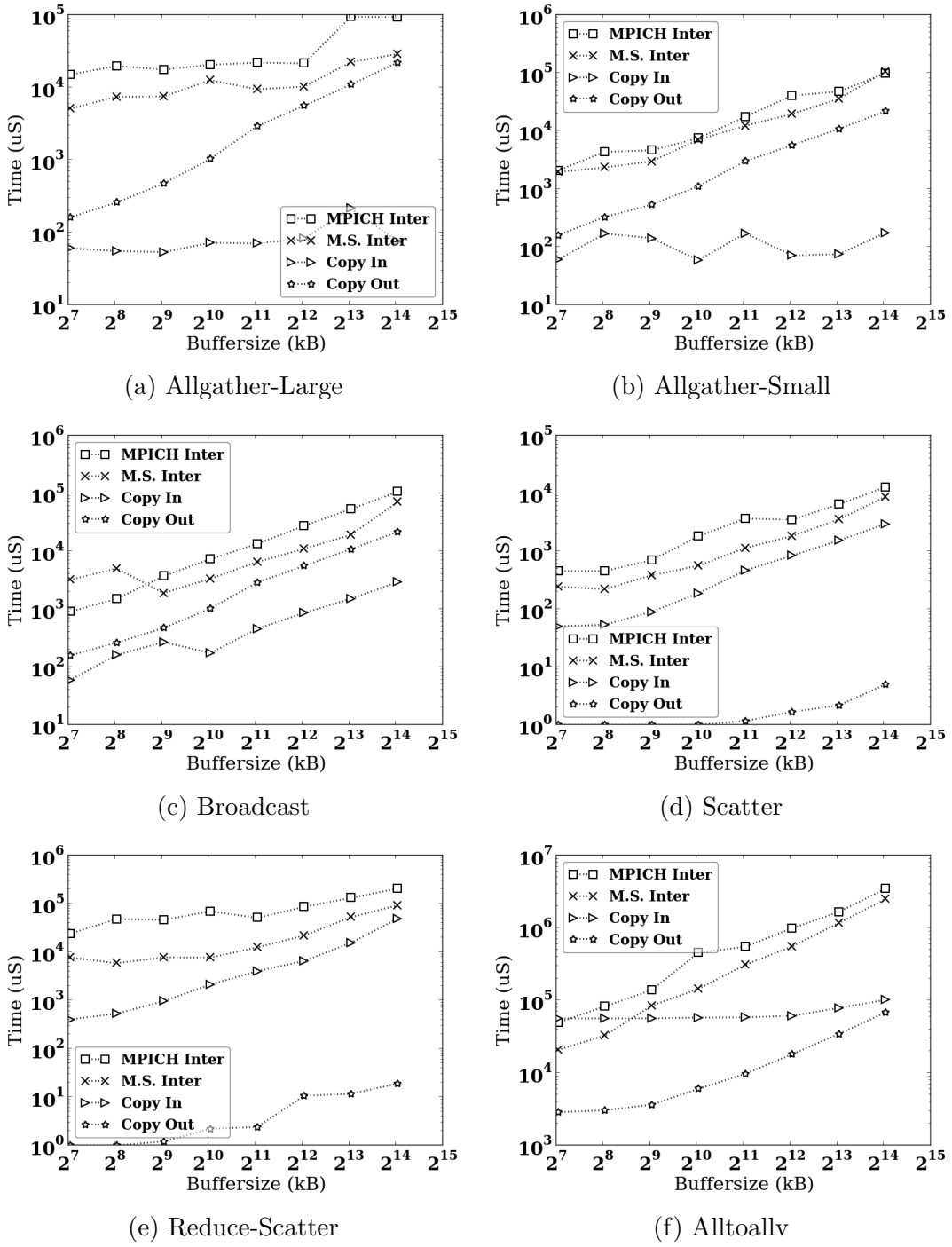
ably use the pairwise exchange algorithm, and have similar latency to the MPICH algorithm. The 512k buffer size experienced a particularly high latency, which was repeatable, but not characteristic of the other results. We excluded this point when calculating speedups of our technique.

The scalability of these algorithms is show in Figure 2.5, using a constant buffer size and varying the number of cores from 128 to 65536. The universal hierarchical algorithms show better scalability that their independent counterparts, because of the reduced inter-node communication. This again excludes the SMP-friendly algorithms, which have similar scalability across all sizes.

The choice of inter-node collective algorithm has a large impact on total latency. For Allgather using the smallest buffer size, Bruck's algorithm produces a 6.7x speedup over the ring algorithm when used in the universal hierarchical algorithm. As will be discussed in Section 2.5.3, this is a larger speedup than can be seen from ideal intra-node optimization. This demonstrates the impact that a choice of inter-node communication can have on collective performance. It also justifies not overlapping intra-node communication, since a clean separation allows the MPI implementation to select the best inter-node collective.

2.5.2 Multi-Sender Inter-Node Communication

Figure 2.6 shows the performance improvement that can be obtained in the inter-node communication portion of the algorithm by utilizing multiple senders. The inter-node communication latency is given as *MPICH inter* and *M. S. inter* for the inter-node communication latencies of the MPICH algorithms when used in the hierarchical algorithm, as well as the multi-sender versions respectively. Since this requires modifications to the source code, it is only applied to the MPICH algorithms. Having multiple processes communicate in parallel from the same shared memory buffer is a novel improvement over previous work, and overcomes problems of prior algorithms that reduced the number of processes communicating though shared memory in order



(a) Allgather-Large (b) Allgather-Small

(c) Broadcast (d) Scatter

(e) Reduce-Scatter (f) Alltoallv

Fig. 2.6.: Inter- and intra-node latency vs buffer size, 32k cores

to increase the number of senders on a node. Table 2.2 gives the speedup ranges for the multi-sender version, over the same size range as Figure 2.6.

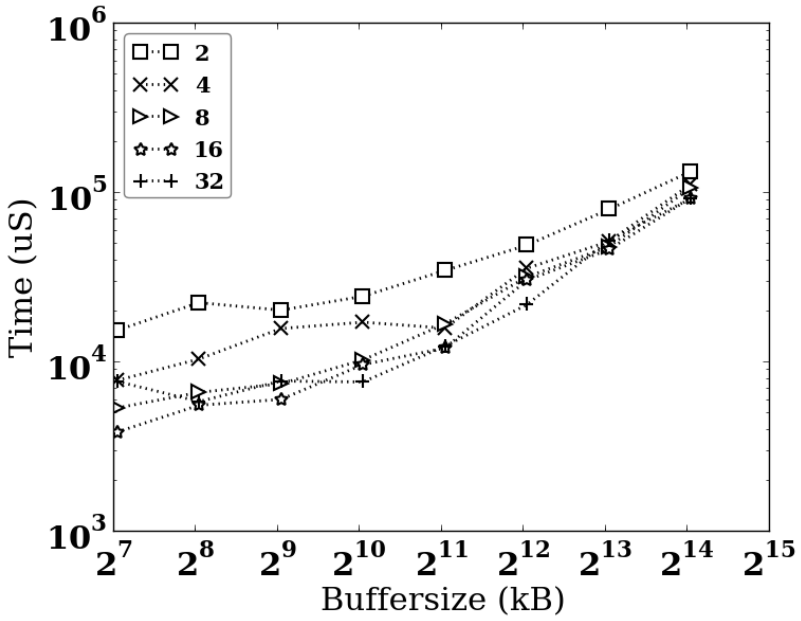


Fig. 2.7.: Reduce-Scatter latency vs buffer size with varying numbers of senders using multi-sender algorithm

Table 2.2.: Speedup of the multi-sender algorithm over corresponding MPICH algorithm for inter-node communication in universal hierarchical algorithm, 32k cores

Algorithm	Mean	Max
Allgather-Small	1.4	2.1
Allgather-Large	2.7	4.2
Alltoallv	2	3.1
Broadcast	1.7	2.8
Reduce-Scatter	4.9	9.1
Scatter	2.2	3.2

The multi-sender algorithm utilizes multiple senders communicating from the same shared memory buffer, unlike prior multi-leader algorithms. Figure 2.7 shows how increasing the number of senders improves latency, until the network become saturated. This is in stark contrast to the prior multi-leader strategy shown in Figure 2.3, which results in a performance degradation on our system due to reduced efficiency for intra-node communication, as described in Section 2.3.2.

The scalability of these multi-sender algorithms is shown in Figure 2.5, as *M. S. Hier*. On many algorithms the multi-sender versions show better scalability, because of more efficient inter-node communication. The broadcast algorithm shows minimal improvement using hierarchical algorithms, but the multi-sender variant shows large improvement as the cluster size grows. Some of this can be attributed to the fact that the broadcast algorithm is pipelined, as well as utilizing the multi-senders. Because the algorithm is pipelined, data can be retransmitted before the entire buffer is received.

2.5.3 Shared Memory Intra-Node Communication

Our hierarchical algorithms do not overlap inter- and intra-node communication, a compromise that allows us to convert any collective algorithm to a hierarchical one. Figure 2.6 shows the magnitude of the copy-in and copy-out portions of the algorithm, the intra-node communication, and the inter-node communication. This shows that the inter-node communication is generally the majority of the overall latency. The percent of total time spent in intra-node communication averages between 7% (reduce-scatter) and 21% (alltoallv). Based on Amdahl's Law, completely overlapping the intra-node communication would result in a speed up of 1.1x and 1.3x respectively. Additionally, the Cray Gemini high performance network minimizes inter-node communication time. If a slower network were used, the inter-node communication would be higher, making intra-node communication a smaller component of overall latency.

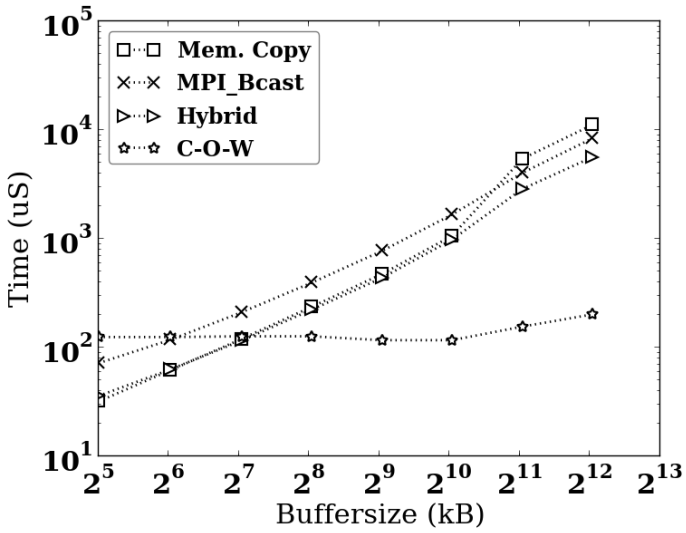


Fig. 2.8.: Copy-out latency vs buffer size, allgather and broadcast transfer algorithm, the shared memory variation is not shown, it is zero for all sizes

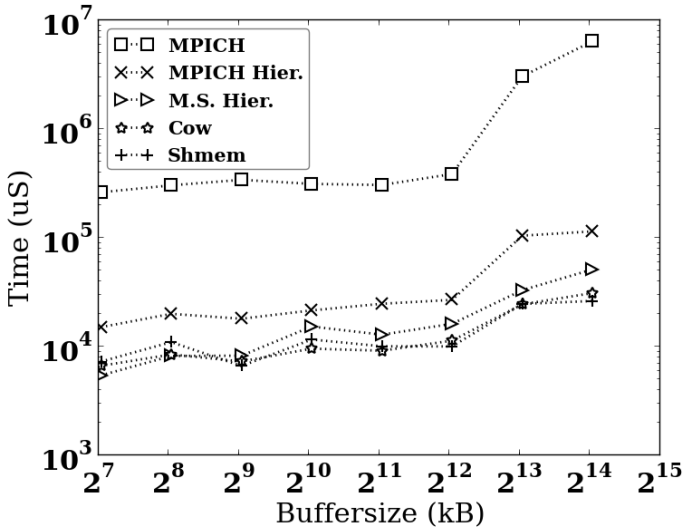


Fig. 2.9.: Allgather-ring latency vs buffer size, COW and shared mem. are used with the multi-sender hierarchical, 32k cores

Figure 2.8 shows the latency of the different copy-out methods discussed in section 2.3.3. Mem. Copy is a basic algorithm in which each process simultaneously copies the receive buffer from shared memory into its own private memory. This showed superior performance over MPI_Bcast for small message sizes but was infe-

rior for large sizes. Several versions were created which attempted to throttle the number of simultaneous memory copies, but all performed worse than the unthrottled version. The hybrid version is a generic NUMA-friendly combination of MPI point-to-point communication and memory copies. Returning a Copy-on-write buffer only shows reduced latency for large receive buffers due to the overhead of the system call. Returning a shared memory buffer is not shown, because it has no additional latency. These *copy-out* methods are also shown in Figure 2.9 used with the allgather ring algorithm. The shared memory and copy-on-write versions are used with the multi-sender inter-node algorithm. The small performance improvement they show again demonstrates the importance of inter-node communication optimization over intra-node optimization.

2.5.4 Number of Senders for Parallel Funneled Algorithms

The parallel funneled algorithms utilize multiple senders to communicate from the shared memory buffer, raising the question of the ideal number of senders. Figure 2.10 shows the results for the parallel funneled scatter algorithm, varying the number of senders from 2 to 32. As the number of senders increases so does the algorithm's performance, to a point at which it saturates. Adding more senders does not at any point hurt performance, so all tests were performed with the maximum number of senders. Test for the other collective communication algorithms showed results similar to Scatter.

These results indicate that for the high-performance network in this highly-scalable cluster, the benefit of funneling (compared to traditional, multi core-oblivious algorithms) comes primarily from the use of shared-memory buffers to avoid intra-node messaging rather than reducing the number of senders to avoid bandwidth contention. It is possible that lower-bandwidth networks would see a negative performance impact from additional senders because of increased bandwidth contention.

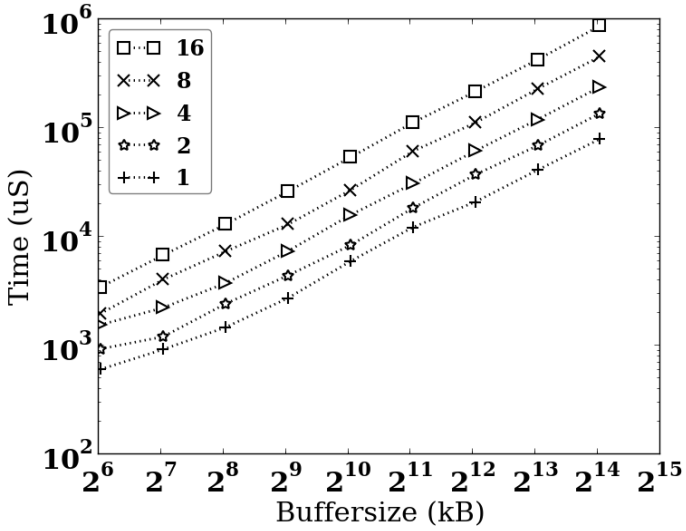


Fig. 2.10.: Parallel funneled allgather-Bruck with smaller node sizes

2.6 Conclusion

In this work, we present the universal hierarchical algorithm for MPI collectives that allows the utilization of hierarchy-unaware collectives for inter-node communication. To do this we sacrifice inter- and intra-node communication overlap that past works achieved, but the potential gains realized from having the ability to use any inter-node algorithm justify the sacrifice. We show that our algorithm works for a wide variety of collectives, including vector versions of collectives as well as alltoall, which has never been done hierarchically with shared memory. We test our algorithms with as many as 65536 cores and see speedups over the baseline averaging 14.2x for alltoallv, 26x for allgather, and 32.7x for reduce-scatter. We demonstrate the flexibility of our algorithm by proposing novel improvements for the inter- and intra-node portions of communication, and show these changes effortlessly plug into the universal hierarchical algorithm. We believe that our algorithm is an ideal solution for future MPI implementations. There exists a large number of existing collectives that are best used in different scenarios, and this algorithm allows them to be used in a hierarchical way without major modification. As new algorithms are designed for complex networks, or intra-node copies are optimized for specific architectures, the MPI implementation will be able to seamlessly integrate these features into the universal algorithm.

3. EXPLOITING PROCESS IMBALANCE TO IMPROVE MPI COLLECTIVE OPERATIONS IN HIERARCHICAL SYSTEMS

3.1 Introduction

For systems running MPI, optimizing collective communications is an important consideration for maximizing system performance moving into the exascale era [17]. Collective communications involve all process in a system, and have been show to suffer from scalability problems [1].

As the use of multicore processors in clusters has become standard, research has looked at improving MPI performance for these hierarchical architectures. Algorithms have been developed that utilize shared memory to optimize several collectives [5, 9, 10, 35, 36]. These algorithms can provide large speedups over their traditional counterparts, but do not address the issue of process imbalance, which has specifically been identified as an area of improvement for MPI on exascale machines [17].

Faraj et al. demonstrate that distributed memory systems suffer from large process imbalance, relative to collective latency, even with balanced applications [16]. This imbalance creates large synchronization delays for collective communications.

The first contribution of our work is a study of process imbalance, revealing the nature of the imbalance in more detail than past works. The imbalance is examined on an inter- and intra-node level to understand how this imbalance impacts hierarchical collectives. This understanding of the arrival pattern is then used to develop imbalance-tolerant hierarchical algorithms. Because of the large speedup provided by hierarchical collectives, they are a natural starting point for imbalance-tolerant collectives. Reduce, broadcast, and alltoall are chosen for optimizations because they

represent all-to-one, one-to-all, and all-to-all communication patterns. Expanding the principles of these algorithms to other collectives would be relatively straightforward.

The reduction and broadcast algorithms utilize a novel method known as *dynamic leader selection* to opportunistically select each node leader based on the imbalance pattern at every invocation of a collective. This method allows the early arriving processes to perform as much communication as possible before the later processes arrive. The alltoall algorithm utilizes multiple senders to avoid delays from late arriving leaders, and utilizes *opportunistic message fragmentation* to pre-send data from early arriving processes, without delaying for later ones.

These algorithms are tested on a Cray XE6 running up to 32k cores and show speedups over MPICH of 18.9x for reduce, 5.3x for broadcast, and 6.9x for alltoall when averaged across all buffer sizes, levels of imbalance, and numbers of processes. These algorithms also out perform the default algorithms on the Cray system, with speedups averaging 35x for reduce, 5.3x for broadcast, and 2.1x for alltoall.

The following Section, 3.2, will explain the background regarding process imbalance and MPI collective operations, including prior work in the area. While MPI collectives have been well studied, only a few works have considered imbalanced process arrival patterns. Section 3.3 will investigate process imbalance on hierarchical systems, giving background information on imbalance as well as measurements from our system. Using this information Section 3.7.4 will describe several algorithms designed operate in the presence of imbalance process arrival. Then Section 3.8 will outline the test system with Section 3.9.4 providing results on the developed algorithms.

3.2 Background

Early work that involved imbalanced process arrival patterns for parallel applications was focused on creating efficient barriers for shared memory machines. Software barriers for large scale shared memory machines were studied by Eichenberger and

Abraham, who showed that load imbalance was necessary when calculating the ideal degree of the combining tree used to implement the barrier [37]. They further showed that using fuzzy (non-blocking) barriers helped alleviate the synchronization delay incurred by the barriers, as long as the imbalance was low.

Mamidala et al. was one of the first works to create MPI collective routines that performed well in the presence of process imbalance [38]. This work created barrier and allreduce algorithms for Infiniband using a binomial tree to collect the information, and a hardware multi-cast to disseminate it. An adaptive version of both algorithms exists to deal with process imbalance. In this algorithm the root holds a token, but can pass the token to an adjacent process in the tree, moving the root closer the slowest process. The root, which is ideally the last process to arrive, is then responsible for performing the hardware multi-cast.

Faraj et al. produce a comprehensive study that investigated imbalance in distributed memory systems [16]. This work introduced several key ideas in the study of process imbalance, including the idea of the process arrival pattern and delay factor, terms that will be described in Section 3.3. The process arrival patterns of several macro benchmarks were studied on two machines, showing that the synchronization latency of a collective could often outweigh the communication latency of that collective. This work also created a micro benchmark that produced an *imbalanced* process arrival pattern with a *balanced* loop of array operations. This showed that the imbalanced arrival was not due to the network or any specialized libraries, but could be created with a *balanced* program. The micro benchmark will be used in this work as well, and is given in Listing 3.2.

Faraj et al. found that in the macro benchmarks the imbalance of a particular collective remained temporally correlated, and proposed a scheme that enabled them to change collective algorithms for future iterations based on the imbalance of past calls. Using the STAR-MPI framework, Faraj et al analyzed the performance of a variety of collectives when called with varying imbalance patterns. They found that particular collectives were more or less effective based on the amount of imbalance,

as well as the message size. STAR-MPI's run-time system was modified to take the imbalance pattern of a particular call-site into account when choosing a collective, and use that as a criteria for selecting a collective algorithm [39].

Patarasuk and Yaun continued this work and developed an efficient broadcast algorithm when used with an imbalanced arrival pattern [40]. This algorithm used control messages to notify the root when other processes had arrived at the collective. If multiple processes arrive simultaneously the root directs one of the processes to forward the data to the rest of the sub-group. This helps avoid serialization at the root if multiple processes arrive simultaneously.

Qian and Afsahi created process arrival pattern aware alltoall and allgather algorithms for RDMA over Infiniband [41, 42]. They use direct alltoall and allgather algorithms, where each process directly communicates its data to one other process. This makes all phases of communication independent from another. In order to notify one process of another's arrival, the RDMA control registers are monitored, so no additional messages need to be sent. The direct algorithms then simply schedule their RDMA accesses to ensure that they are communicating with processes that have already arrived at the collective. This method works well for these direct algorithms, but would require additional messages if not using the Infiniband RDMA.

3.3 Process Arrival Patterns

The *process arrival pattern* (PAP) of a collective operation is the set of times that each process arrives at the collective. Correlated to the process arrival pattern is the *process exit pattern*, the set of times that processes exit the collective. Given a world size of n there exist processes, $p_1, p_2, ...p_{n-1}$. Each process arrives at the collective at a unique time, $a_1, a_2, ...a_{n-1}$, and exits at a unique time, $e_1, e_2, ...e_{n-1}$, as shown by Figure 3.1.

The acknowledgment of this imbalance complicates how collectives are measured, so several terms are needed to characterize the collective. The average arrival time, is

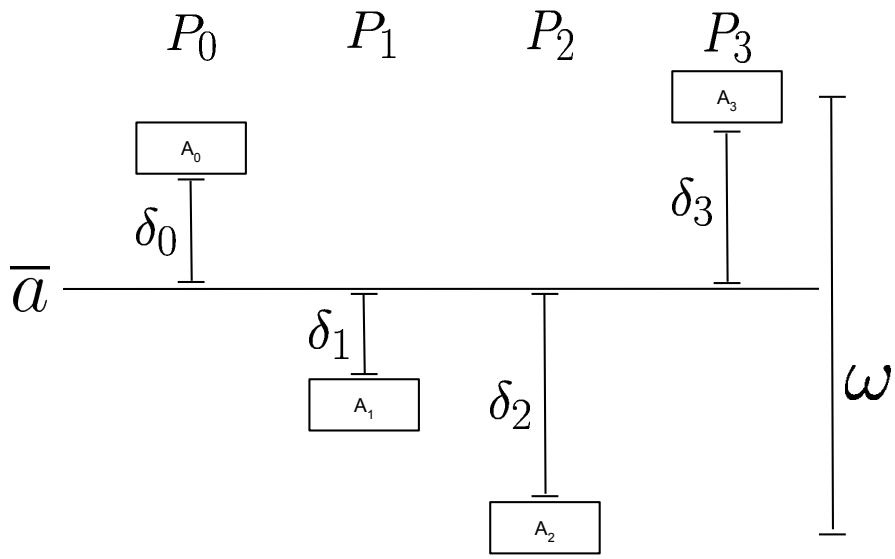


Fig. 3.1.: Process imbalance terms

given by $\overline{a} = \frac{\sum_{i=0}^{n-1} a_i}{n}$ and the average exit time is $\overline{e} = \frac{\sum_{i=0}^{n-1} e_i}{n}$. The *imbalance time* can be used to characterize the imbalance or a process compared to the $\overline{a}$, with the imbalance of each process being $\delta_i = |a_i - \overline{a}|$. The average imbalance can the be determined as $\overline{\delta} = \frac{\sum_{i=0}^{n-1} \delta_i}{n}$. The *worst case imbalance* is also a useful metric, and can is defined as $\omega = \max(a_i) - \min(a_i)$. It is useful to think about the imbalance times compared to the time it takes a system to send a message, Min_Lat. The average imbalance factor is defined as $\frac{\overline{\delta}}{Min\_Lat}$ and the worst case imbalance factor is $\frac{\overline{\omega}}{Min\_Lat}$ That means if the worst case imbalance factor is 100, a process could send 100 message between the first and last processes arriving.

In order to measure the impact of a collective, traditionally the collective's latency is used. This can be defined as the arrival time of all processes, assumed simultaneous, until the last process exits the collective. A balanced exit pattern is not assumed, and algorithms like the binomial tree broadcast can have highly imbalance exit patterns. This view of a collective's latency loses meaning with an imbalanced process arrival pattern, because the start time of the collectives is including latency for processes that have not yet arrived. A better why to measure the latency it the individual

time each process spends in a collective, and is given by $\iota_i = e_i - a_i$ and the average by $\bar{\iota} = \frac{\sum_{i=0}^{n-1} \iota_i}{n}$. This gives an idea of the average burden the collective places on each process, without measuring time for processes that are not in the collective.

3.4 Clock Synchronization

3.4.1 Introduction

Clock synchronization is essential when analyzing the process arrival patterns of collective communication operations. Modern systems have high resolution clocks making precise measurements possible, but understanding how these individual measurements relate to the measurements of other machines can be complicated. Two issues come it play when dealing with clocks in a distributed system. Firstly, the clocks are set to an arbitrary zero, so in order to compare the clocks to each other an offset is needed. Secondly, the rates at which clocks advance themselves are not uniform, so an understanding of how time varies for each clock is needed.

3.4.2 Background

Synchronization algorithms by Cristain [43] and the Tempo project [44] are some of the first works to look at clock synchronization. Cristain's algorithm is given by 3.1. It sends a series of messages between to nodes,A and B, recording the times that the messages are sent and received. Assuming a uniform round trip time the offset of the clocks can be calculated with $A_{Send} + \frac{A_{Recv} - A_{Send}}{2} - B_{Send/Recv}$.

```
1  for (j = 0; j < world_size; j++) {
2    for (i = 0; i < sync_iters; i++) {
3      if (rank == 0) {
4        A_Send = get_time(clock);
5        MPI_Send(&dummy, 1, MPI_DOUBLE, j, TAG, comm);
6        MPI_Recv(&B_SendRecv, 1, MPI_DOUBLE, j, TAG, comm, &status);
7        Recv_A = get_time(clock);
8        time_diff[i] = (A_Send+(A_Recv - A_Send)/2) - B_SendRecv;
9      }
10     if (rank == j) {
11       MPI_Recv(&dummy, 1, MPI_DOUBLE, 0, TAG, comm, &status);
12       B_SendRecv = get_time(clock);
13       MPI_Send(&B_SendRecv, 1, MPI_DOUBLE, 0, TAG, comm);
14     }
15   }
16   if (rank == 0) {
17     quick_sort(time_diff);
18     offset[j] = time_diff[sync_iters/2];
19   }
20 }
```

Listing 3.1: Clock synchronization algorithm

Further research has investigated clock synchronization as well as clock drift. Jones and Koenig create a run time system that uses linear offset interpolation for the correction of clock drift postmortem [45]. This will correct the drift so-long as it is linear. Non-linear clock drift will remain uncorrected. Becker et al. [46] Further studied clock drift and found that often it was not linear, but varied in an unpredictable way. To deal with this they propose resynchronizing the clocks after a period of time, and if possible piggybacking this synchronization on collectives to avoid extra synchronization delays. No solution currently exists to correct for clock drift without adding extra clock synchronization points within long running benchmarks.

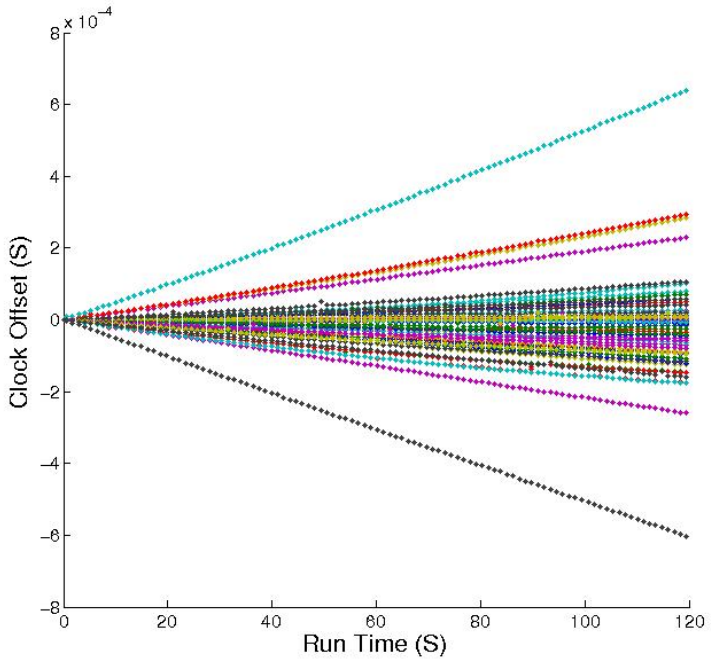


Fig. 3.2.: MPI_Wtime clock drift

3.4.3 Results

In order to properly measure the process arrival patterns on the test system, a study of the available clocks was necessary. The high performance system clocks as well as MPI_Wtime and the TSC register were compared, for precision and drift. MPI_Wtime as well as all system clocks seemingly used the same source, as the variations in clock drift were identical. The TSC register differed from the system clocks, but had higher drift. For this reason MPI_Wtime was chosen for further measurements.

The clock drift is shown in Figure 3.2 and is large enough that is will distort the process arrival time measurements if not corrected. In the micro benchmark was resynchronization is possible between tests, and periodic synchronization can contain the drift. Test results show that the drift of the fastest clock to the slowest can be as high as $3.5\mu S$ drift per second of run-time. By re-syncing every second this, limits

the imbalance due to clock drift to $1.75\mu S$, which is orders of magnitude less than the imbalance seen in the results.

3.5 Process Imbalance

This section analysis the processes arrival patterns of both macro and micro benchmark on our test machine in order to better understand the nature of the process imbalance.

3.5.1 Macro Benchmark Imbalance

This section shows the process imbalance for several benchmarks from the NAS Parallel Benchmarks [47]. These benchmarks were run with varying input sizes as well as a varying number of processors. Tables 3.1, 3.2, 3.3, and 3.4 give the average case and worse case imbalance for each benchmark, with varying world sizes and input sizes. These numbers average the imbalances of all collective calls with-in the computational loops of these benchmarks. When collectives are called concurrently the second collective has lower imbalance than the original, because the first collective synchronized the processes and eliminate imbalance. This lowers the averages, but even taking this effect into account the imbalance for all benchmarks is shown to be large.

The larger input sizes have more imbalance than the smaller ones, and smaller world sizes also have more imbalance. This is due to the fact that the imbalance is related to the run time, as will further be explored with the micro-benchmark. Some of the smallest imbalance results, e.g. MG B, run for less than a half second, which explains the low imbalance.

Table 3.1.: Average collective imbalance for all call sites in FT

Input Size	**1024**		**2048**		**4096**		**8192**	
	Avg.	W.C.	Avg.	W.C.	Avg.	W.C.	Avg.	W.C.
B	2051	11041	1391	7299	692	4427	1649	12206
C	711	3330	4795	26669	2797	19517	2346	14532
D	8170	52384	40321	529960	23943	346366	17037	232452
E	118854	1145400	60720	597955	234785	4489008	151553	2739416

Table 3.2.: Average collective imbalance for all call sites in IS

Input Size	**1024**	
	Avg.	W.C.
B	3132.86	26474.6
C	5911.18	42941.85
D	3722.49	31849.9

Table 3.3.: Average collective imbalance for all call sites in MG

Input Size	**1024**		**2048**		**4096**		**8192**	
	Avg.	W.C.	Avg.	W.C.	Avg.	W.C.	Avg.	W.C.
B	25	143	29	197	20	171	22	293
C	45	272	36	265	28	268	43	457
D	104	557	62	394	81	491	83	536
E	490	3330	253	1469	426	1929	310	1588

Table 3.4.: Average collective imbalance for all call sites in LU

Input Size	**1024**		**2048**		**4096**		**8192**	
	Avg.	W.C.	Avg.	W.C.	Avg.	W.C.	Avg.	W.C.
B	63	507						
C	72	542						
D	306	1866	204	1922	216	1669	238	1546
E	2072	11844	1226	9297	755	6878	614	6329

3.5.2 Micro Benchmark Imbalance

Our study of process imbalance will utilize a micro-benchmark proposed by Faraj et al. and given in Listing 3.2 [16]. This produces smaller imbalance than seen in appli-

cations, but is useful for inducing the imbalance in a reliable manner. This micro-benchmark can also be thought of as the minimum imbalance that programs will reasonably see, if their synchronization points are separated in time by an amount equal to the run time of the micro-benchmark. The benchmark has each process performing an equal amount of work on private arrays. Actual applications demonstrate more complex behavior, with network IO, library calls, and memory contention all contributing to the imbalance.

```
1
2  for (i=0; i < test_iters; i++){
3    MPI_Barrier(MPI_COMM_WORLD);
4    for(m=0; m<xtime; m++){
5      for (k=1; k <999; k++){
6        a[k] = b[k+1] - a[k-1] * 2;
7      }
8    }
9    //Take PAP timing here
10   arrival[i] = MPI_Wtime();
11   //If evaluating with imbalance
12   MPI_Collective()
13 }
```

Listing 3.2: Imbalance inducing micro-benchmark

Figure 3.3 shows the worst case and average imbalance factors for the micro-benchmark when run on a Cray XE6 with 8192 processes. The amount of time that the micro-benchmark runs is regulated by the *xtime* variable, which was set to produce a run time approximately equal to the listed computation time. It is important to note that the *xtime* is the same for all processes, and is not the source of the imbalance. In the results section the imbalance will be identified by the number of iterations of this benchmark, which is the value of *xtime*. These results confirm the imbalance seen by Faraj et al., which is quite high for a balanced benchmark [16].

Expanded results for the micro benchmark are given in Table 3.5. The columns labeled *all* show the basic results, and it is clear that the average and worst case

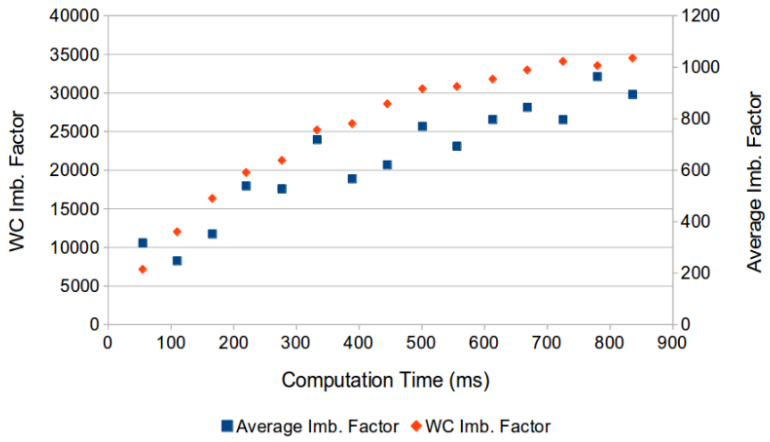


Fig. 3.3.: The average and worst case imbalance times for the micro-benchmark on a Cray XE6 running 8192 cores

Table 3.5.: Imbalance times for the micro-benchmark on 8192 Cores with varying computation times

	All		**First**		**Last**	
time (ms)	Avg.	W.C.	Avg.	W.C.	Avg.	W.C.
55	317.33	7160.2	245.93	1409.4	325.49	6305.5
110	247.72	12009	242.39	1420.2	535.09	10489
166	352.19	16342	295.54	2274.7	663.19	13774
221	538.57	19695	322.56	2507.5	828.8	16877
277	527.51	21263	355.47	2884.7	927.24	18046
334	718.68	25205	468.86	3719.5	1055.1	20969
389	566.17	26012	356.66	3016.9	1228.5	22624
445	620.96	28579	458.71	3731	1298.9	24444
501	770.03	30540	428.18	3670.6	1351.1	26286
556	692.95	30829	454.29	3787.5	1423.9	26533
612	796.74	31780	560.92	4367.7	1468.3	26782
668	844.16	32965	502.67	4049.2	1543	28186
724	796.11	34075	569.32	4683.6	1593.8	28752
781	963.35	33531	580.61	4636.2	1630.8	28047
836	893.8	34502	568.02	4571.9	1663.2	29175

imbalance are related to the time between collectives, *xtime*. The columns labeled *first* and *last* show the imbalance when only taking 1 process per node into account, either the first process to arrive, or the last. This shows imbalance on both sides of the arrival distribution. Understanding that the nodes themselves arrive with very large imbalances shows that the imbalance problem occurs between nodes, not just within them.

Figure 3.5 shows the heat map of the normalized process arrival times for 2048 processes run on a Cray XE6 with 32 core nodes. It can be seen that the imbalance pattern is random between nodes and within nodes. The imbalance is also distributed among all processes, with no particular process lagging from an issue such as unbalanced OS interference. Figure 3.4 shows the histogram of the arrival times, normalized to each nodes average arrival time. This pattern shows that most processes arrive in a group, with a longer tail of late arrivals. This histogram is produced with an *xtime* value of 40000. Smaller values produce a histogram with a similar shape, but with a

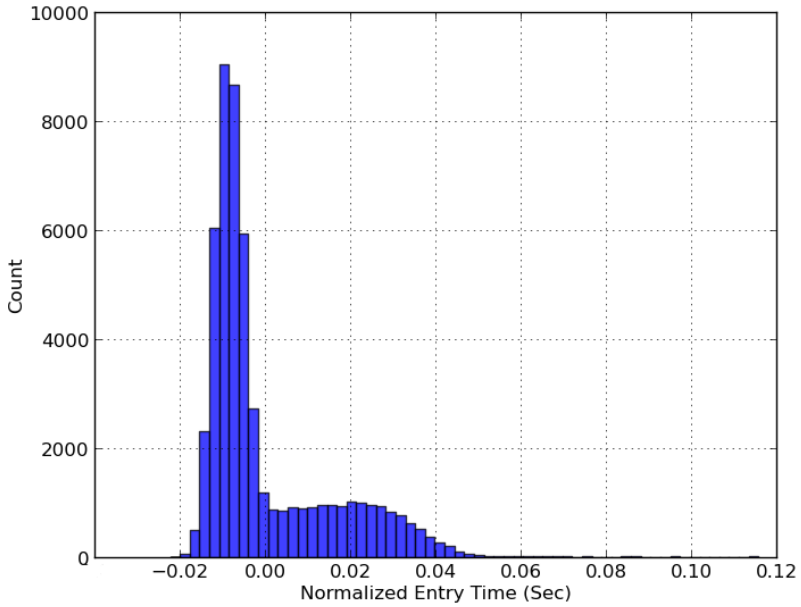


Fig. 3.4.: Normalized intra-node arrival time on a Cray XE6, 40k imbalance

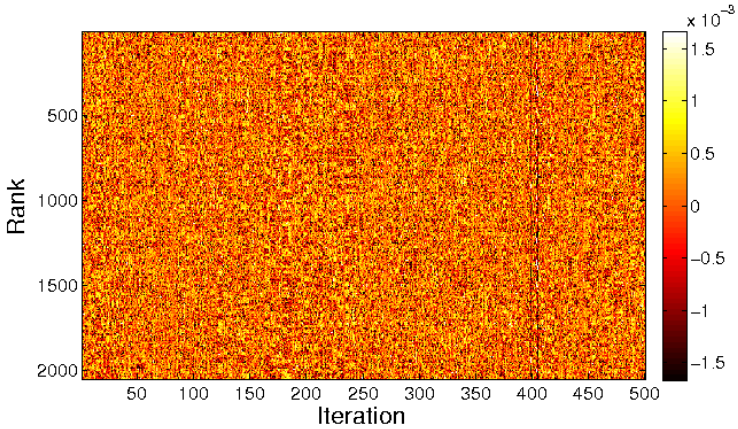


Fig. 3.5.: Entry times for micro-benchmark, 8192 processes, 50 mS computation time, color-bar represents the time in seconds

shorter tail and higher peak, and larger values produce a similar shape with a longer tail and shorter peak. Past work has examined imbalance in large scale systems and found that operating system interrupts cause imbalance, but these imbalances only effected certain processes [48]. This heat map and histogram show that this is not the case, and that the imbalance is random.

Figure 3.6 show the average imbalance time for each invocation of a micro benchmark run. This demonstrates that the imbalance remains fairly constant, and does not vary with time.

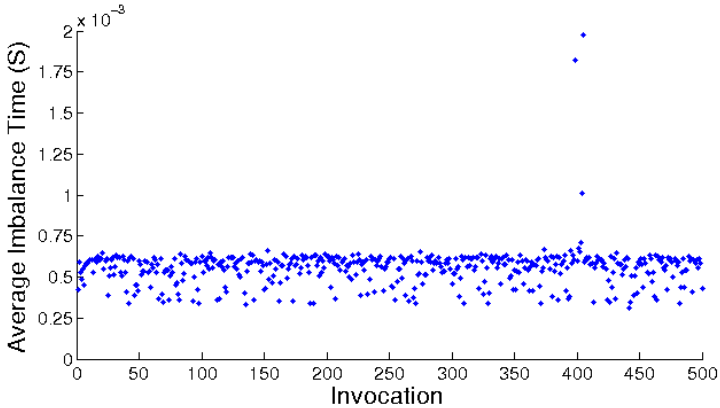


Fig. 3.6.: Average imbalance vs invocation for micro benchmark, 8192 processes, 50 mS computation time

This imbalance pattern is important to understand when designing imbalance tolerant collective algorithms. The observation that processes on the *same node* are just as likely as to be imbalanced as processes on *other nodes* is a new and unintuitive observation. This means that algorithms must deal with intra- and inter-node imbalance. If one process was routinely late, this history could be used to shape the communication algorithms for future invocations, but algorithms must tolerate any process arriving late.

This also means that our imbalance tolerant algorithms can focus on minimizing the average time processes spend in a collective, instead of being forced to focus on the latency of the slowest process. Due to this, our results will measure the average time that processes spend in the collective, instead of the latency of any given process.

3.6 Counters

To further explore the process imbalance created by the micro-benchmark, the Performance Application Programming Interface (PAPI) was used to measure several hardware performance counters while the micro-benchmark was run. These counters record the number of occurrences of processor events on a per processes basis. The value of these counters were then compared to the normalized arrival time of the processes to determine if there was any correlation between the value of the counter and the arrival time. Performance counters were observed for L1 misses, L2 misses, TLB misses, total instructions issued, hardware interrupts, stall cycles, branch mispredictions, and process migrations. The test system pinned processes to CPUs, and thus the process migrations were always zero, and will not be further discussed. Each of these counters also separately recorded counts for events that happened in user space as well as kernel space.

Figures 3.7 and 3.8 show the entry time versus the L1 and L2 chance miss counts. In both these cases the entry time seems to increase with the number of misses, but there are several distinct lines indicating that other system event also seem to contribute. The same can be seem with the hardware interrupts 3.9, stall cycles 3.10, total instructions 3.11, and kernel instructions 3.12. These results indicate that a variety of system features contribute to the imbalance, and not any particular thing. The total cycles of the CPU were also recorded and given in Figure 3.13, and show very high correlation, a predictable results because the test system does not use frequency scaling. This graph validates the profiler as well as counter data.

None of the counters produced any clear correlation. Each of the events do show some correlation, especially if the points are divided into several subsets. Because of this it seems that multiple factors are involved in the imbalance. A multivariate regression could provide further insight into what specific events are contributing to the delay and to what degree. A timing model that looks at the particular latency of each event could also be created. For the purpose of this work understanding that

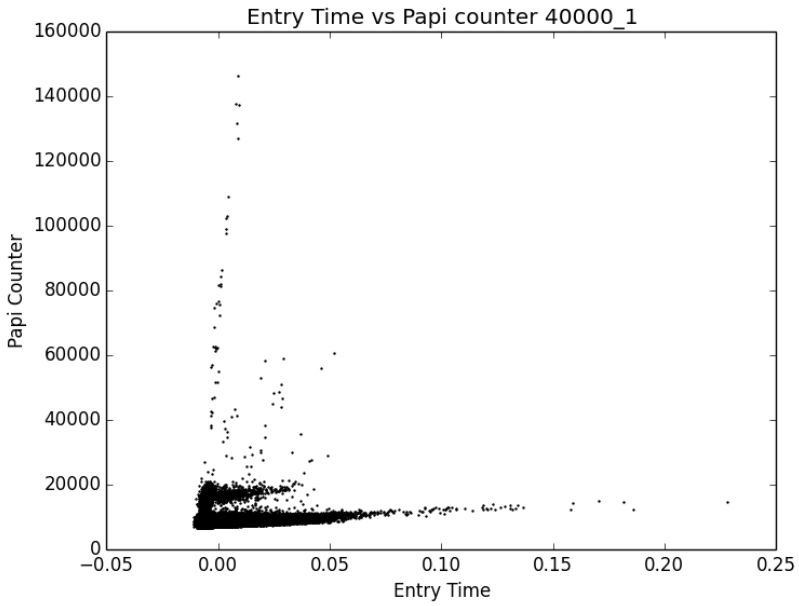


Fig. 3.7.: Normalized entry time vs L1 misses

the imbalance is not caused by a fixable event is satisfactory. If the delay was purely caused by operating system noise then the scheduler could be adjusted to reduce or eliminate the delay. This is clearly not the case however, and it appears the imbalance cannot be removed with a simple solution. The purpose of this benchmark is to evaluate the minimum amount of system noise that can reasonable by expected for a program running for the same amount of time after a synchronization event. Showing that the noise is not the results of a single fixable event validates this principle.

3.7 Process Arrival Aware Algorithms

The following section outlines several algorithms that are designed to perform well in the presence of imbalance. The characteristic of the imbalance is important in shaping the design of these algorithms. Because processes arrive at a given call site in a random order, there is no way to predict the arrival pattern from one collective invocation to another. If predictable early or late arrivers were know from past imbalance patterns, these algorithms would be designed to take advantage of that.

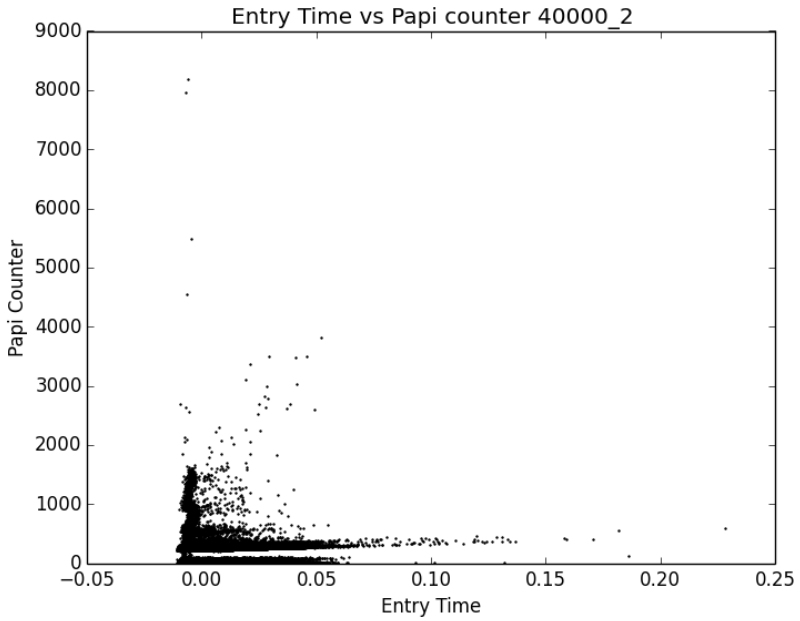


Fig. 3.8.: Normalized entry time vs L2 misses

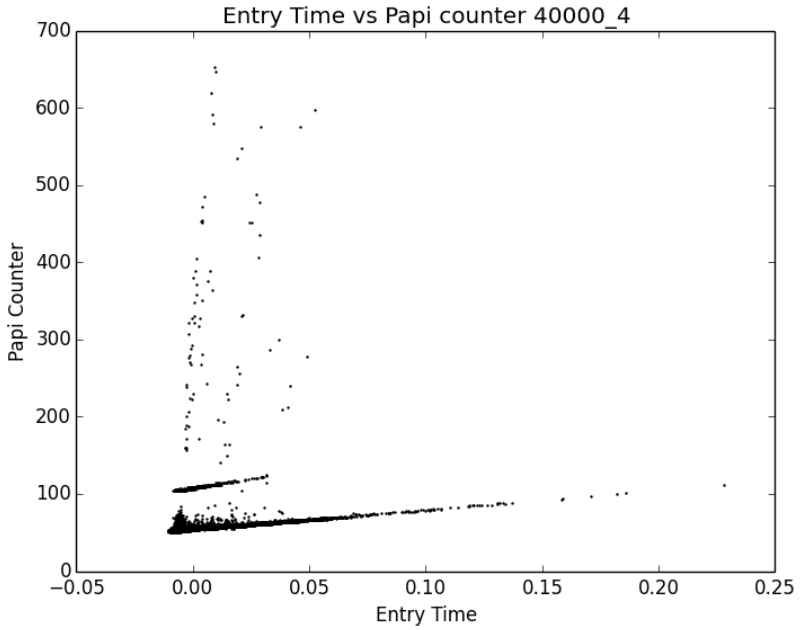


Fig. 3.9.: Normalized entry time vs hardware interrupts

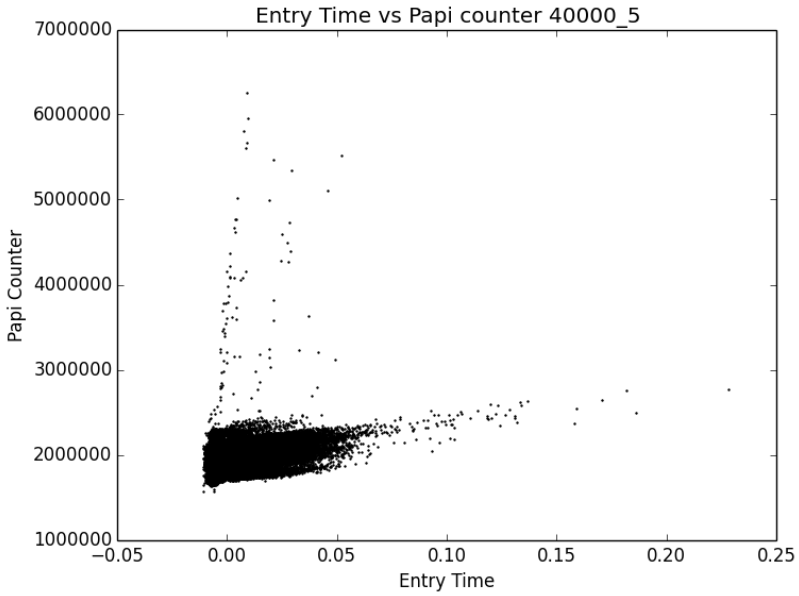


Fig. 3.10.: Normalized entry time vs stall cycles

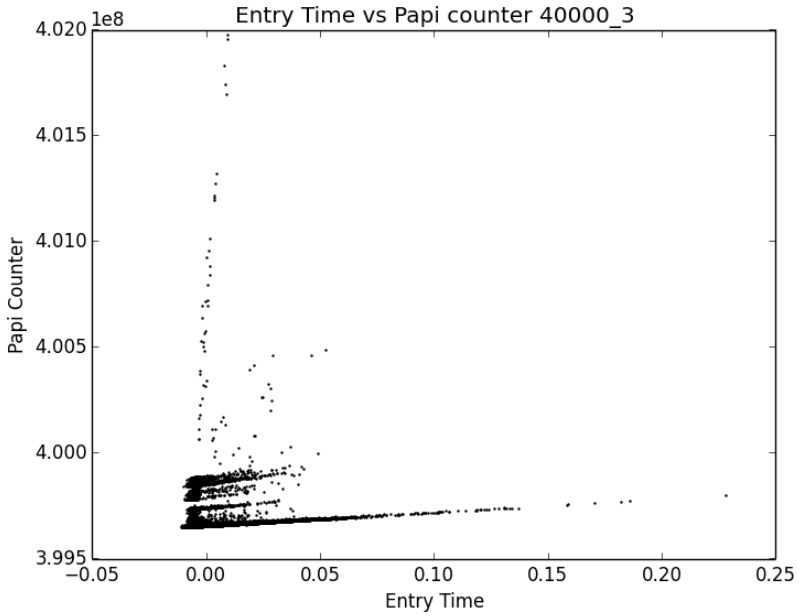


Fig. 3.11.: Normalized entry time vs total instructions

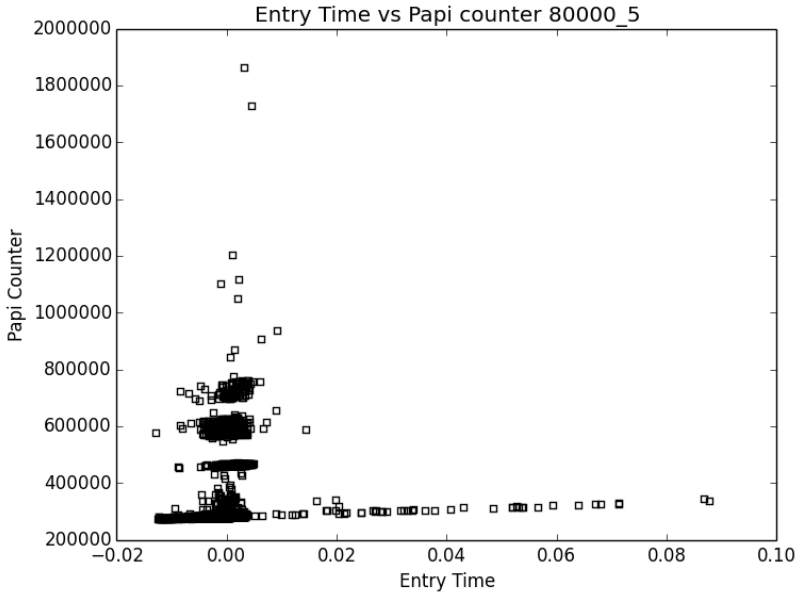


Fig. 3.12.: Normalized entry time vs kernel instructions

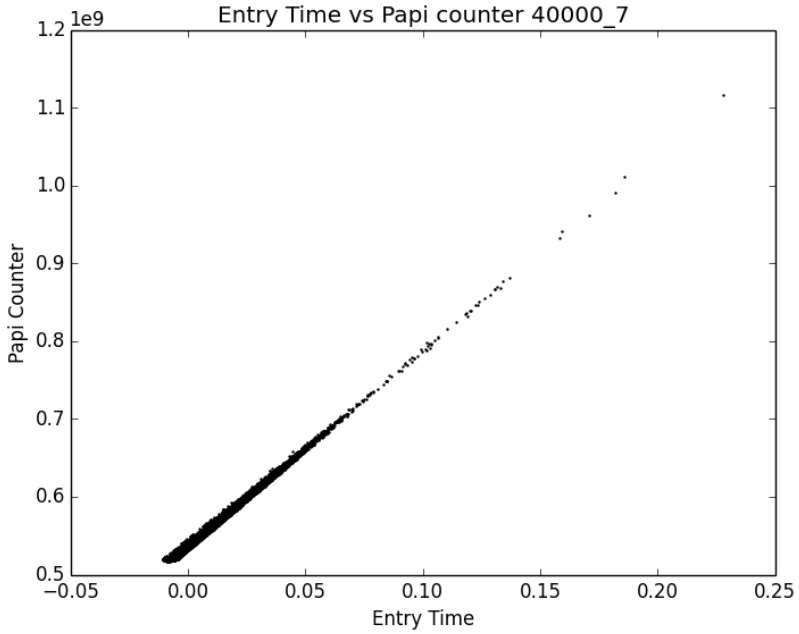


Fig. 3.13.: Normalized entry time vs total cycles

The principle of these imbalance aware algorithms is to tolerate imbalance by letting early arriving process communicate before the later ones arrive, and leave the collective call site as soon as possible. The goal is to reduce the mean time all processes spend in the collective, minimizing the impact of the call.

3.7.1 Reduction

The imbalance aware reduction algorithm is based on the MPICH binomial tree algorithm [19]. This algorithm was chosen because it had the best performance on our system for a wide range of message sizes, and in its default form does well in the presence of process imbalance. In MPICH, a reducescatter followed by a gather is the default for large messages, but this create an all-to-all data dependency that is not necessary, and our tests show the performance suffered in the presence of imbalance. The fan-in communication pattern of the binomial tree has a reduced number of processes that depend on data from one another.

The MPICH binomial tree algorithm is also evaluated in a hierarchical fashion, with shared memory buffers used for inter-node communication and a single leader performing the inter-node communication. The intra and inter-node communications are separated with a node-level barrier. This naive implementation performs well with no imbalance, using the default optimized MPI_Barrier, instead of implementing shared memory structures.

The intra-node barrier in the naive implementation adds a temporal dependency between every process on a node, when the dependency is really only needed between the leader and all other processes. This extra dependency leads to unnecessary synchronization in the presence of imbalance. To reduce this, a MPICH hierarchical reduce is developed with optimized intra-node communication. Several intra-node algorithms have been developed to take advantage of process arrival imbalance. The intra-node algorithms can take advance of shared memory buffers, which enables one sided communication. A process can reduce its private data into the shared mem-

ory buffer, without any synchronization required. To prevent data races two locking schemes have been developed. A serial locking scheme has one process lock the entire buffer and then perform the reduction. A parallel version only locks a segment of the buffer, so multiple processes can perform reductions simultaneously. The parallel scheme reverts to the serial scheme if the buffer size is less than a cache line to prevent false sharing. These algorithms are also useful when used with inter-node algorithms utilizing dynamic leader selection. Instead of having a static leader perform inter node communication, the last process to arrive at the collective becomes the leader, to prevent synchronization delay. Several intra-node point to point algorithms are also evaluated.

In order to avoid delays while the node leader waits for all processes on a node, a method called *dynamic leader selection* is employed. When a process enters the collective, its buffer is immediately reduced into the shared memory buffer. Unless the process is the root or the last process to arrive on its node, it can exit the collective. The last process per node and the root are responsible for the inter-node communication. This results in lower synchronization cost than traditional hierarchical collectives that use fixed leaders which must block waiting for slow processes. Extra overhead is incurred by the dynamic leaders, due to control messages used to discover other leaders. The node-to-node communication pattern remains the same as static leaders, a binomial tree in this case, but at each invocation a different set of leaders performs the reduction. This algorithm is given in Figure 3.14.

The control messages are needed for the parent nodes to identify the node leader on the child nodes, with parent and child referring to the position in the binomial tree. Theses algorithms are implemented entirely on MPI primitives, requiring the leader on the child node to identify a Rank for the MPI_Send. If an MPI implementation were to implement these algorithms at a lower level, special primitives could be envisioned that send a message to a particular node, and allow any process on that node to match the send. The leaders of the parent nodes do not suffer this problem, thanks

```
1:  function
2:      LOCAL_REDUCE(Shared_Buff, Private_Buff)
3:      if (Last_Process or Root) then
4:          SEND_ID_MESG(Children)
5:          while (Need_data) do
6:              RECV(Temp_Buff)
7:              LOCAL_REDUCE(Shared_Buff, Temp_Buff)
8:          end while
9:          RECV(Parent_ID_Mesg)
10:         SEND(Shared_Buff)
11:         CANCEL(Extra_ID_Mesgs)
12:     end if
13: end function
```

Fig. 3.14.: Dynamic leader reduction algorithm, outputs in bold

to MPI_ANY_SOURCE, which allows a receive to match a send from an unknown source.

Maintaining the binomial tree pattern between nodes is done to maintain performance under low imbalance situations, but leaves an unnecessary dependency between non-root nodes. This could be eliminated using a serial inter-node reduction. While this would perform well in high imbalance cases it would lead to contention at the root when the imbalance is low. Figure 3.15 shows an example of the dynamic leader timing and message pattern. Dotted arrow represent control messages and solid arrows represent data messages. While P0 can be viewed as the root, in the actual implementation control messages are eliminated from the root, so the diagram actually represents a subset of a larger tree.

Figure 3.14 uses MPI_Cancel calls to remove extra control messages from the system, which is the version is evaluated in the results section. The MPI_Cancel calls can be eliminated by leaving the control messages unreceived until the next function invocation, at which point they are matched before the collective starts.

The control messages can be eliminated all together by using the one-sided communication interface. To do this the one-sided communication window is created on top of shared memory, so that any process can access the memory region. Then the

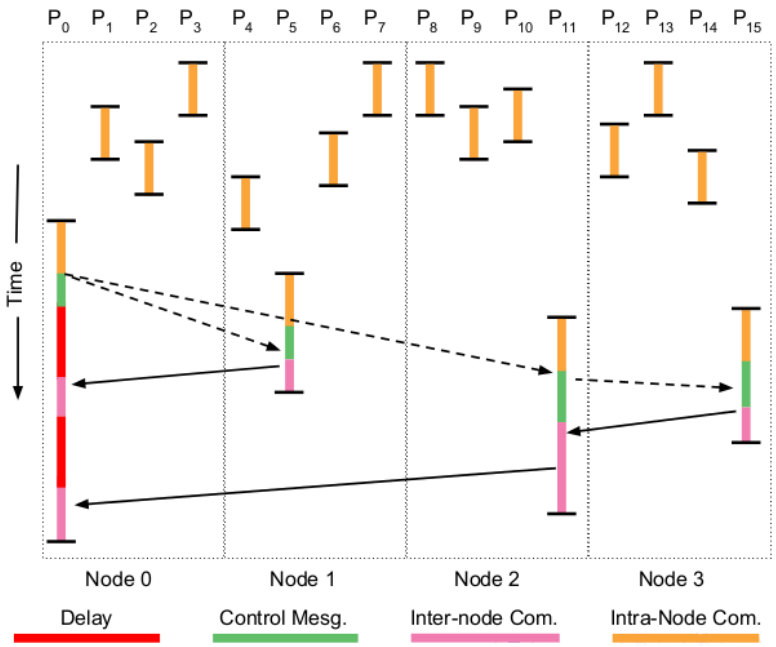


Fig. 3.15.: Timing example for dynamic leader selection with reduce

the dynamic leader uses an MPI_Accumulate to place the data onto the receiving node, and mark a data ready flag. The leader on the receiving node can spin on the flag, because it is in shared memory, without the sender knowing who the receiver is.

3.7.2 Broadcast

The broadcast algorithm is again based on the MPICH binomial tree algorithm. MPICH also uses a scatter followed by an allgather for large messages, but this suffers the same drawbacks as the reducescatter/gather reduce described in Section 3.7.1. The binomial tree algorithm is also evaluated in a naive hierarchical fashion using intra-node barriers to separate the inter and intra-node communication. To avoid this unnecessary synchronization an *intra-node optimized* version is evaluated. This version uses a shared memory structure that tracks which processes have arrived and allows early arriving processes to copy out the data and leave the collective before later arriving processes.

```
function
    if (First_Process or Root) then
        if (!Root) then
            SEND_ID_MESG(Parent)
            RECV(Shared_Buff)
        end if
        for all (children) do
            RECV(Child_ID_Mesg)
            SEND(Shared_Buff)
        end for
    else
        RECV(Child_ID_Mesg) //Unused message
    end if
    WAIT_FOR(Shared_Buff_Valid)
    MEM_COPY(Shared_Buff, Private_Buff)
end function
```

Fig. 3.16.: Dynamic leader broadcast algorithm, outputs in bold

The broadcast algorithm using dynamic leader selection is given in Figure 3.16. In this collective all processes must wait for the root, which has the potential to incur a high synchronization cost if the root arrives late, which was not a problem for reductions. For all other nodes, the **first** process to arrive at the collective call site on a particular node is selected as the leader, opposite of the reduction algorithm. This allows communication to take place before any of the other processes on a node arrives, moving the data onto nodes faster than would otherwise be possible.

Once the data is received on a given node, then the data is placed into a shared memory buffer. This allows early processes to copy out the data and leave the collective call site before later ones even arrive, reducing the synchronization delay. Unlike the reduction algorithm the extra control messages can be received by the non-leader processes, because these processes arrive after the leader, thus eliminating the need for MPI_Cancel. For this collective the control messages are used in the reverse direction, with child nodes sending ID messages to the parents. This is due to the reverse direction of the data flow, and the need for the parents to have a MPI_Send destination. Like the reduction, these messages could be eliminated with a lower level primitive to send a messages to a certain node, or using the one-sided communication interface.

The intra-node potion of the algorithm is simply a memory copy from a single shared memory buffer. A NUMA or architecture optimized version like past hierarchical works was avoided due to potential synchronization delays from late processes. A NUMA optimized version could be created using a per NUMA leader for the intra-node portion of the algorithm, but because the intra-node latency is much less than the inter-node it would have little effect on the total collective latency. Figure 3.17 shows the timing example for broadcast, in the same fashion as Figure 3.15 does for reduce.

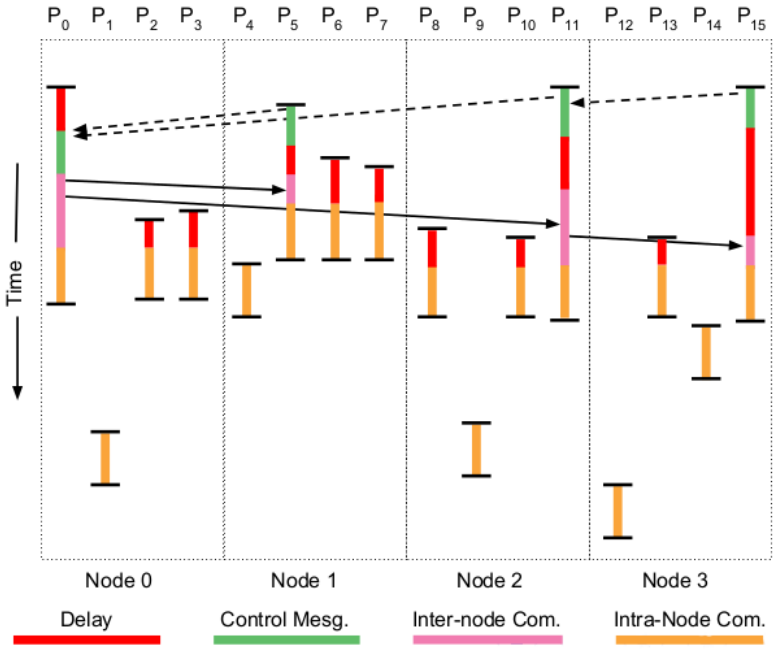


Fig. 3.17.: Timing example for dynamic leader selection with broadcast

3.7.3 Alltoall

The imbalance aware alltoall algorithm is approached differently than the broadcast and reduce because there is an inherent alltoall data dependency among the processes. Unlike the prior algorithms, there is complete synchronization so removing excess synchronization is not an option. There is also no opportunity for early arriving processes to exit the collective before late arriving ones enter, no process can exit until all processes have reached the call site.

The starting point for the imbalance aware algorithm will be the MPICH Isend alltoall algorithm. This algorithm uses a series on non-blocking sends and receives which send messages starting at the next higher rank. The pairwise exchange algorithm was also evaluated, but only had better performance on a few test cases. Bruck's algorithm performed better than the Isend algorithm for small messages, but when a hierarchical version of the Isend algorithm was used, the performance difference was eliminated. Bruck's algorithm is a store and forward algorithm that sends data to process $rank + 2^i$ in the ith communication round, and therefore maps very well to clusters whose node size is a power of 2, much like the binomial tree algorithm. Because of this Bruck's algorithm does not see the same speedups when used hierarchically.

The imbalance aware alltoall algorithms deal with process imbalance in two different ways, one for the imbalanced processes arriving within a node and another for communicating with processes on other nodes. The dynamic leader algorithms used control messages between nodes in order to establish the leader identity. This was found to have a higher overhead for alltoall due to the increased paths of communication, needing $NumNodes(NumNodes - 1)$ control messages as opposed to $NumNodes - 1$ for the binomial tree. Our experimentation found this overhead to be high enough that sending control messages was not a feasible solution. Instead of using control messages, different processes within a node were statically assigned to communicate with processes on other nodes. If multiple processes on a given node can

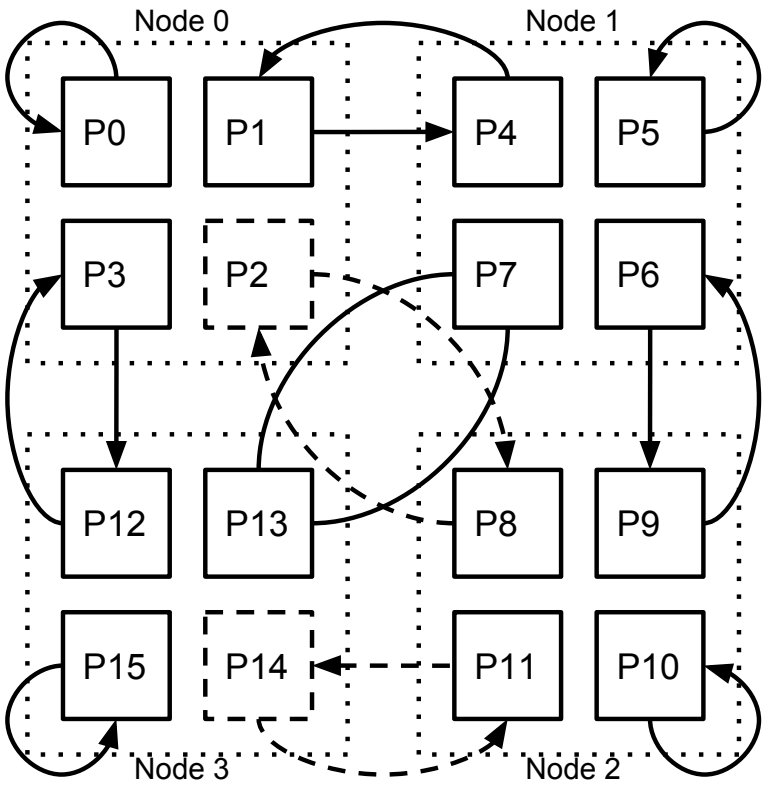


Fig. 3.18.: Alltoall communication example, arrows represent messages, dashed processors and arrows represent messages that cannot yet be set because the processes is delayed

communicate, having several of those processes arrive late would still allow the others to communicate data early. Only one process from a given node communicates with any other node, so the benefits of message aggregation from hierarchical algorithms is still preserved. Figure 3.18 shows how this method still allows communication with nodes that have late processes. In the actual implementation the senders and receivers are separate processes, to avoid cache contention with the send and receive buffers.

There is also imbalance involving the data being copied into the intra-node shared memory buffers, caused by imbalanced process arrival within a node. With the multi-sender method described above, one processes is responsible for sending all the data from its node to another. If all the data is sent in one message, no data can be sent

until all processes have arrived. This severely limits the amount of data that can be pre-sent. In order to avoid this limitation we introduce a method called *opportunistic message fragmentation*. When the intra-node processes copy data into the shared memory buffers they mark a flag for each segment of data. These flags are monitored by the sending processes and used to select the largest available blocks of data to send. The following preferences, outlined below, are used to send subsets of the total buffer.

1. Maximum length blocks
   (a) The entire buffer
   (b) A piece of the buffer between blocks that have already been sent or the buffer edge
   (c) Blocks going to the lowest node number higher than sender's
2. Largest available block
   (a) Blocks adjacent to sent blocks or buffer edge
   (b) Blocks going to the lowest node number higher than sender's

These preferences are used to select the largest blocks of available data, and in the event of equal amounts of data, blocks are select in a way that reduces the fragmentation of the buffer. If multiple blocks fit that criteria the lowest numbered node higher than the senders node number is chosen. Parameters can also limit the minimum message sizes, to avoid sending many small messages, which in the worse case would revert to the same message pattern as the Isend algorithm.

3.7.4 Binomial Tree

Both the reduce and broadcast are compared to the MPICH 3.0.4 implementation of the binomial tree algorithm [19]. MPICH uses this algorithm for broadcasts and reductions with small buffer sizes. For larger sizes a reduce-scatter followed by a

gather is used for reductions, and a scatter followed by an allgather is used for broadcast. These variations were not further evaluated because they performed poorly in the presence of imbalance, due to the all to all data dependence that they create.

When the binomial tree is evaluated, the test system uses XPMEM to optimize intra-node communication with memory copies. This allows, direct memory copies from one private address space to another, eliminating extra copies. The combination of XPMEM and a SMP-aware binomial tree results in the point-to-point implementation performing nearly identically to a shared memory hierarchical implementations when used without imbalance, so only the binomial tree will be evaluated.

3.8 Experimental Testbed

All tests were run on a Cray XE6 with a dedicated Cray Gemini network. Each node on the system has two 16 core 2.5 GHz AMD 6200 Opteron processors and 64 GB of DDR3 memory. Jobs are scheduled using PBS Pro 10.4.7. Runs with different job placements seemed to have minimal impact on the results. For all test sizes, multiple iterations were run, varying from fifty to several thousand, depending on the latency of the collective. This ensured that the small-sized tests received an adequate number of iterations for repeatable results, while still making large sized tests feasible.

The Cray-mpich/6.3.0 module was used for all MPI communication. It is the default MPI implementation for the Cray XE6, and is built upon MPICH. MPICH collective algorithms are built on point-to-point operations, so a minimally modified version of the MPICH 3.0.4 source code was run on the Cray system, which is what this paper refers to as the *MPICH* algorithms. Asynchronous communication is enabled for all tests, though it seemed to have a minimal impact on collective latency. It was originally speculated that this feature might help imbalance unaware collectives communicate before all processes arrived.

The buffer sizes for all results are given as the total buffer size. For broadcast and reduce all input and output buffers are the same size and equal to the message size.

For alltoall the per processes message size is the total buffer divided by the number of processes. The smallest buffer size we report for alltoall tests is the minimum size for each particular number of processes. Integers were used as the data type for all collectives, and the reduction operation was a summation. The results are also analyzed with different levels of imbalance, which is given in units of loop iterations. This is a reference to the micro-benchmark given in Listing 3.2, and described in Section 3.3.

In order to measure the process imbalance as well as the latency of collectives, the mpiP profiler was modified to used high resolution timers to record information about the entry and exit times of all processes [49]. In order to record the imbalance patterns across all process a globally synchronized clock is needed. Cristian's algorithm was used to compute the local clock offsets, with resynchronization every second to prevent clock drift [43]. To reduce the run-time of the testing program, imbalance was generated and recorded with high resolution timers. This trace was then used to reproduce the imbalanced arrival with high resolution sleep calls. The modified profiler showed that this produced identical imbalance as the micro-benchmark, but saved run-time by eliminating the need to run the micro-benchmark to generate imbalance for every collective call.

The test system uses XPMEM to optimize intra-node communication with memory copies. This allows, direct memory copies from one private address space to another, eliminating extra copies. When running the binomial tree algorithm a root of zero is always used. This makes the binomial tree ideally map to the shared memory nodes, because the cores per node is a power of two. The combination of XPMEM and ideal tree mapping results in the point-to-point implementation performing nearly identically to shared memory hierarchical implementations when used without imbalance, explaining the high performance of the MPICH implementations of these two collectives. If the MPICH binomial tree was naively run with a different root, or if the test system did not have highly optimized intra-node point-to-point communication, the latency would be higher.

3.9 Results

This section gives on overview of the results for the process imbalance algorithms. Further results can be found in Appendix A,B, and C.

3.9.1 Reduction

Table 3.6 gives the speedups for the reduction variations over the MPICH Isend algorithm for various numbers of processes. Figures 3.19 and 3.20 show the mean latency of the reduction variations versus the process imbalance and world size respectivly. The imbalance is given in loop iterations, as discussed in Section 3.3. The hierarchical version of MPICH does worse than the original in the presence of imbalance due to the intra-node barriers between the intra and inter-node communication. This added synchronization causes the drastic increase in latency when the imbalance is greater than zero. Using intra-node optimizations to remove these barriers produces speed-ups over the original version.

The dynamic leader version further reduces the latency, more-so as the imbalance increases. This shows that the speedup from utilizing the *dynamic leader selection* provides additional improvements over simply optimizing the intra-node portion of the algorithm. The dynamic leader with one-sided communication version further reduces latency. This version eliminates control messages, but speedups could also be due to difference in the underlying one-sided vs point-to-point implementations.

The default Cray algorithms were omitted from the reduction results. For buffer sizes less than 2048 the results were within 5% of MPICH, and larger buffer sizes showed a slowdown. All tests were run with rank 0 as the root, leading to the binomial tree mapping ideally to the 32 core nodes. If an arbitrary root was used the performance of the MPICH algorithms would be much worse, because it would naively send unnecessary messages across node boundaries. This is discussed further in Section 3.8.

Table 3.6.: Reduction mean latency speedups over MPICH, averaged over all buffer sizes and levels of imbalance

Number Cores	Hier. MPICH	Intra-Opt. H. MPICH	Dyn. Leader	D.L. One-Sided
512	0.34	5.67	18.72	23.95
1024	0.34	5.70	20.13	27.08
2048	0.34	5.73	21.13	28.98
4096	0.33	5.67	20.89	30.33
8192	0.34	5.71	20.40	29.11
16384	0.34	5.68	18.21	25.75
32768	0.34	5.66	13.35	22.47

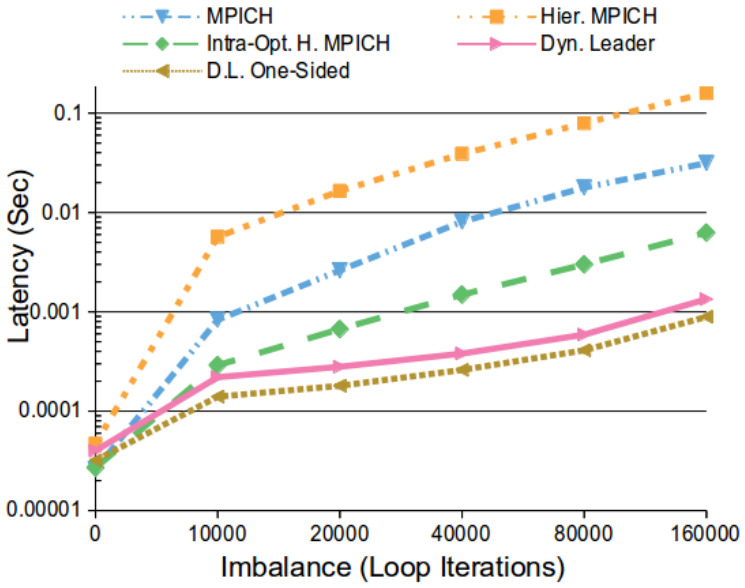


Fig. 3.19.: Mean latency for reduce vs process imbalance, 8192 cores, 2048 Byte buffer

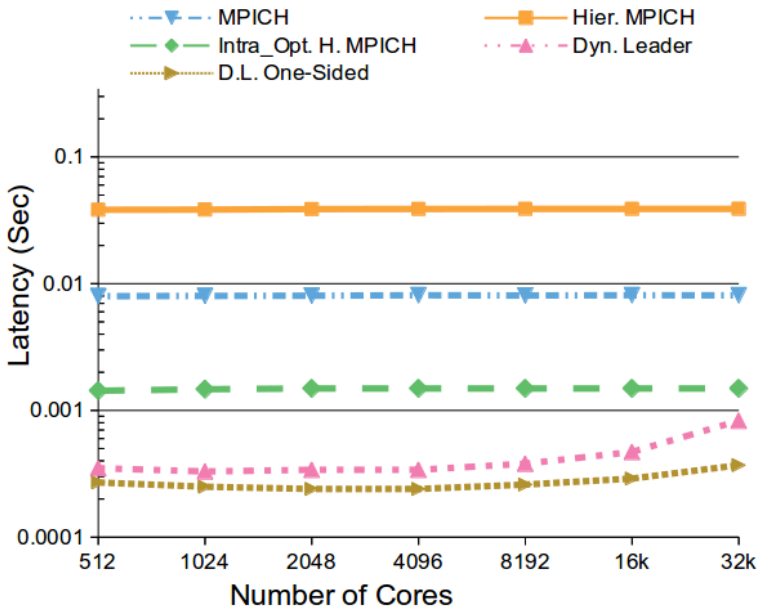


Fig. 3.20.: Mean latency for reduce vs world size, 40k iteration imbalance, 2048 Byte buffer

Table 3.7.: Broadcast mean latency speedups over MPICH, averaged over all buffer sizes and levels of imbalance

Number Cores	Hier. MPICH	Intra-Opt. H. MPICH	Dyn. Leader
512	0.49	1.66	7.53
1024	0.49	1.53	7.12
2048	0.51	1.43	6.15
4096	0.50	1.35	5.34
8192	0.57	1.27	4.72
16384	0.57	1.24	3.64
32768	0.54	1.16	2.52

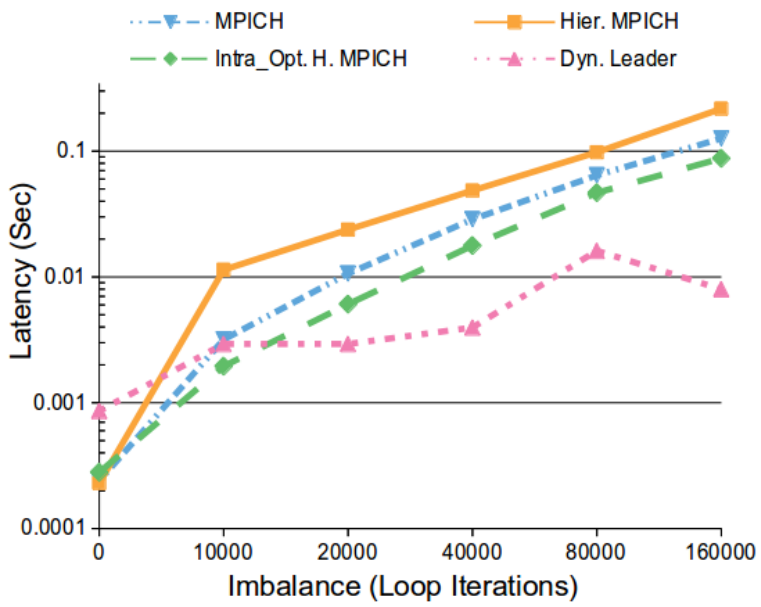


Fig. 3.21.: Mean latency for broadcast vs process imbalance, 8192 cores, 32k Byte buffer

3.9.2 Broadcast

The speedups for broadcast over the MPICH algorithm are given in Table 3.7 and the mean latency for 8192 cores vs the imbalance is given in Figure 3.21. Figure 3.22 shows the latency versus world size. The hierarchical MPICH algorithm again is slower than the default, especially in the presence of imbalance. Like the reduction

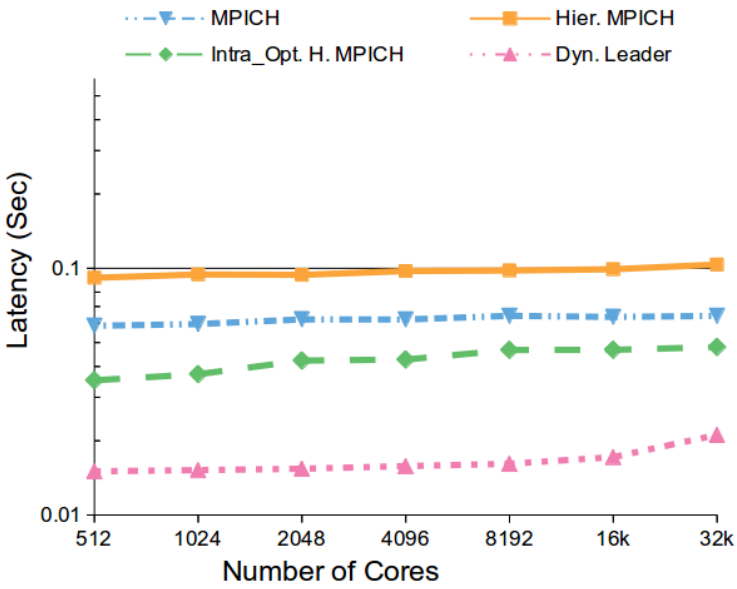


Fig. 3.22.: Mean latency for broadcast vs world size, 80k iteration imbalance, 32k Byte buffer

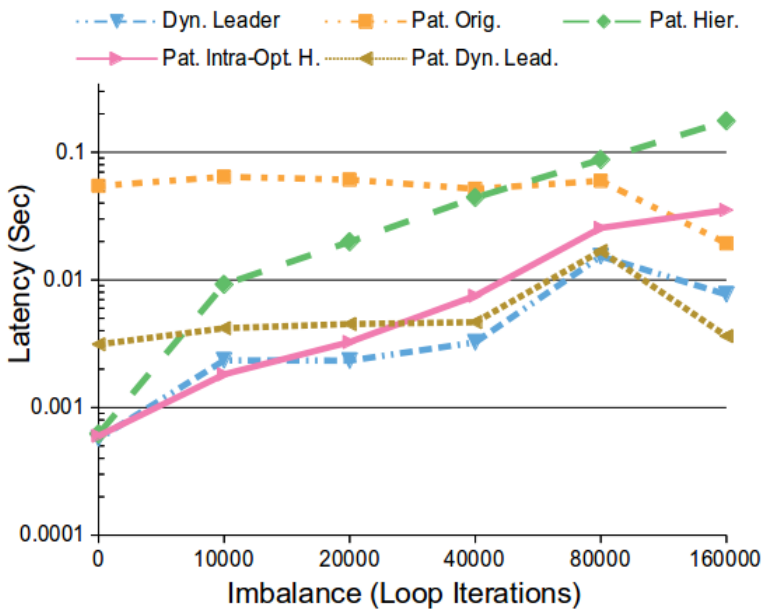


Fig. 3.23.: Mean latency for broadcast vs process imbalance, comparing to prior work, 8192 cores, 2k Byte buffer

this is due to the extra synchronization caused by the node level barriers. Removing these barriers in the intra-node optimized version leads to speedups over the original. Adding the dynamic leader algorithm to these intra-node optimizations gives even higher speedup in the presence of imbalance, but causes a slowdown in the zero imbalance case due to the extra control messages. In both cases the speedups are lower when the number of processes increases, due to the communication cost increasing while the imbalance remains constant.

A one sided communication version of the dynamic leader algorithm was tested but performed poorly. It is suspected that the one sided communication system handles multiple simultaneous messages out of the same node poorly. This would explain why the one-sided version did well for the reduction, since one message is sent from a node at a time. The Cray default algorithms were omitted from these graphs and tables because in all tests the results were within 5% of the MPICH algorithm.

Figure 3.23 compares the dynamic leader broadcast algorithms to the imbalance aware broadcast developed by Patarasuk and Yuan discussed in Section 3.2 [40],

as well as some improvements to it. This algorithm sends control message to the root, which then responds with the data, or directs other processes to forward the data in the case of simultaneous arrival. *Pat. Original* is our implementation of the original version of this algorithm. It uses the variation with a serial subgroup broadcast, but with the group size limited, in order to prevent complete serialization at low imbalance. It performs poorly for most cases on our large scale system, due to contention at the root, which is eased at higher imbalances. Using the Patarasuk algorithm on top of a hierarchical intra-node algorithm leads to better performance at low contention, but the extra synchronization from the intra-node barrier hurts high imbalance cases, as show by *Pat. Hier*. Using the intra-node optimizations helps the high imbalance cases, but further improvement can be had by using the Patarasuk algorithm for the inter-node communication with dynamic leader selection, given as *Pat. Dyn. Lead*. This performs worse than our dynamic leader algorithms, which uses a binomial tree pattern between nodes, at low imbalance. As the imbalance increases the performance gap narrows until the highest imbalance case where the Patarasuk algorithm with dynamic leaders performs the best. This demonstrates how the dynamic leader algorithms can be used with various inter-node algorithms, which is useful in adapting the algorithms to different situations. It also highlights the importance of maintaining the binomial tree communication pattern between the nodes, even though it creates some unnecessary dependencies.

3.9.3 Alltoall

Table 3.8 gives the speedups for alltoall over the MPICH Isend algorithm. Unlike the broadcast and reduce, the hierarchical versions does not produce a slowdown. Due to the alltoall data dependency of the collective, the intra-node barriers do not add to the synchronization costs of the hierarchical algorithms. The speed-up for the Cray versions of the collective is also given, though the exact implementation is unknown, a store and forward algorithm is used for small messages, and all versions

Table 3.8.: Alltoall mean latency speedups over MPICH, averaged over all buffer sizes and levels of imbalance

Number Cores	Cray (default)	MPICH Isend Hier.	Hier. aware
512	1.5	1.5	2.8
1024	1.9	1.8	3.3
2048	3.0	2.1	4.7
4096	2.9	2.4	6.3
8192	4.0	3.6	8.4
16384	4.1	3.7	7.0
32768	10.4	6.7	15.4

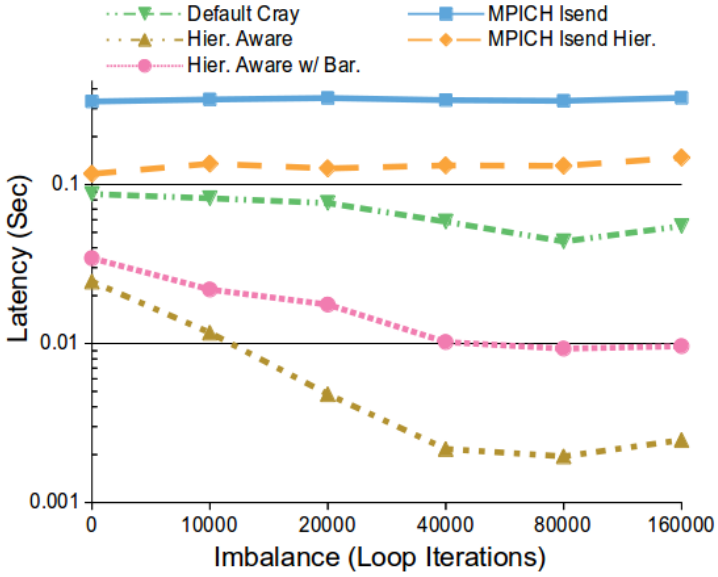


Fig. 3.24.: Mean latency after last arriving process for alltoall vs process imbalance, 8192 cores, 131k Byte buffer

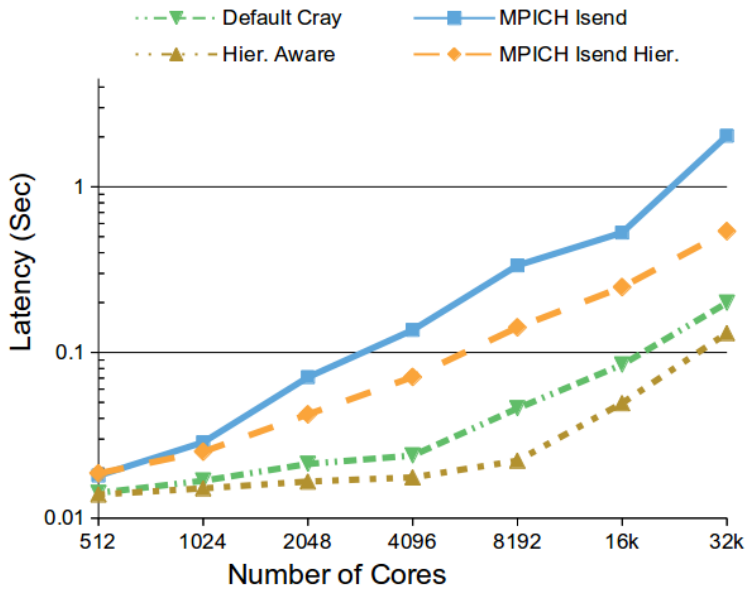


Fig. 3.25.: Mean latency for alltoall vs world size, 10k iteration imbalance, 131k Byte buffer

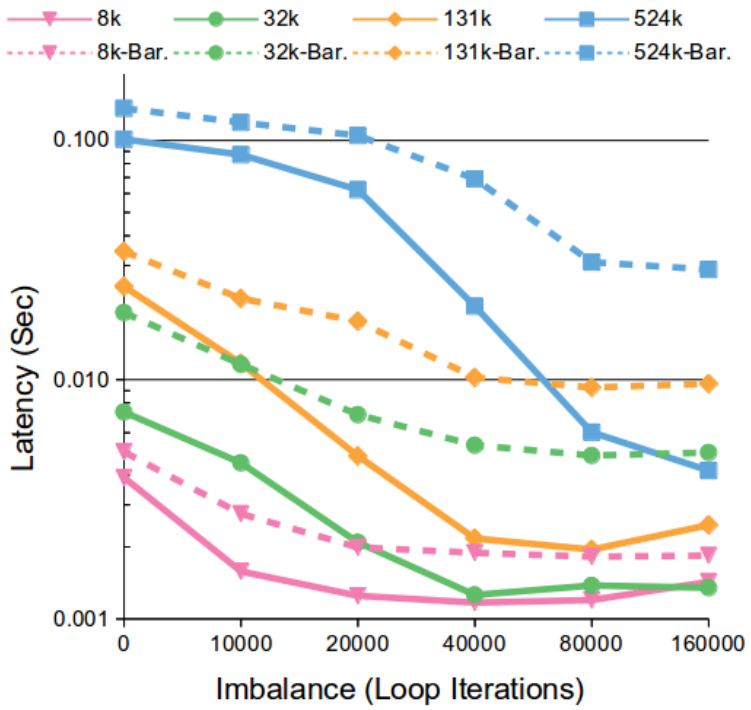


Fig. 3.26.: Mean latency after last arriving process vs process imbalance for alltoall aware and aware with barrier, 8192 cores, multiple buffer sizes

use an *automatically balanced injection* feature. There is no reason to think that this feature could not also be used on our hierarchy-aware version.

The hierarchy-aware version outperforms the other versions, because of its ability to pre-send data from early arriving processes using *opportunistic message fragmentation*. This is highlighted in Figure 3.24 which shows the latency of the collective *after* the arrival of the last process, which essentially removes the synchronization delay from the latency. The decreasing latency of the hierarchy-aware version as the imbalance increases is caused by the ability to pre-send data from the early arriving processes. With extremely high imbalance, this latency would be equal to the latency of one process (the slowest) sending one message to all others, and receiving one message from all others. It should be noted that the speedups given in Table 3.8 and Figure 3.25 use the entire collective latency, not the latency after the last arriving process.

The hierarchy-aware algorithm also produces a speedup in the zero imbalance case. As mentioned in Section 3.7.3, using multiple senders on the same node provides a speedup due to better network utilization. To isolate this effect the hierarchy-aware with barrier version was tested. This versions adds a intra-node barrier to prevent the pre-sending of any data, but retain the multiple senders of the hierarchy-aware version. Having multiple receivers still provides some means for early arriving nodes to avoid late arriving leaders, but this effect cannot be completely eliminated. Figure 3.26 show the imbalance aware version as well as the imbalance aware with barrier version for several buffer sizes over various level of imbalance. In all cases the barrier version prevents the algorithm from pre-sending data, leading the the gap in latency as the imbalance increases.

3.9.4 Macro Benchmarks

One macro benchmark was run for each collective that isolated that collective in a simple application. These tests show that the extra control messages being sent in

Table 3.9.: Floyd-Warshall benchmark run-time and speedup

Processes	Binomial Tree	Hier. MPICH Isend	Dyn. Leader
128	18.9	19.6 (.96)	17.6 (1.08)
256	9.4	9.1 (1.04)	7.6 (1.23)
512	8.1	7.7 (1.05)	6.3 (1.28)
1024	8.2	7 (1.17)	6.9 (1.19)

Table 3.10.: Matrix vector multiply macro benchmark runtime and speedup

Processes	Binomial Tree	Hier. Bi. Tree	Dyn. Leader
256	18.75	19.15 (0.98)	18.325 (1.02)
512	10.775	10.95 (0.98)	10.775 (1)
1024	6.95	7.25 (0.96)	6.375 (1.09)
2048	5.325	5.15 (1.03)	4.775 (1.12)
4096	5.3	4.8 (1.1)	4.4 (1.2)
8192	4.575	4.7 (0.97)	3.675 (1.24)

the dynamic leader algorithm, and the fragmented messages in the alltoall algorithm do not negatively effect the performance of an application when collectives are being called repeatedly. These benchmarks were chosen to isolate each of the collectives, and do not scale as far as the micro-benchmark tests. For broadcast, an implementation of the Floyd-Warshall all-pairs shortest path algorithm with 8k vertices was used, with results given in Table 3.9. The reduction was tested with a dense matrix vector multiply with a rotating root for each iteration and an 8k x 8k matrix of floats. Alltoall was tested with the the FT benchmark from the NAS Parallel Benchmark suite [47]. Unfortunately this benchmark switches the processor decomposition as the number of processors increases, leading to a true alltoall only occurring with smaller numbers of processes. The results in Table 3.11 are reported for the largest numbers of processes before this switch for several test sizes. The other benchmarks are run on increasing processes until the scaling stops.

Table 3.11.: FT benchmark speedup, for given test size and number of processes

Test	Hier. Bi. Tree	Dyn. Leader
A-128	1.86	2.11
B-256	1.88	1.98
C-512	1.58	1.62

3.10 Future Systems

These algorithms are also well positioned to be effective in future systems. As the number or cores per node increases, the *dynamic leader selection* will have more opportunity to utilize early arriving processes, and need fewer leaders as well as control messages. The *opportunistic message fragmentation* employed by alltoall will also have more data potentially available for early communication. New high performance systems that utilize intelligent network interfaces to offload communication can also take advantage of these ideas. Fan-in and Fan-out algorithms could utilize the network interface as a persistent leader, so long as careful consideration was taken to ensure minimal dependence on the process arrival pattern. This could allow for the communication of data to and from a node before *any* processes arrive. The alltoall algorithm could also be employed with the network interface tasked with managing message fragments and pre-sending data. This would remove the need for multiple senders, but also guarantee all other nodes are available to receive message fragments.

3.11 Conclusion

In this work, we offer three major contributions that help exploit process imbalance to improve MPI collective operations in hierarchical systems. Our first contribution is a detailed examination of the the nature of process imbalance in a hierarchical system. Using a custom modified profiler and hardware performance counters we describe this imbalance in greater detail than prior works. Our next contribution is developing the method of *dynamic leader selection* to create imbalance tolerant algorithms for fan-

in and fan-out collectives. Finally we introduce the idea of *opportunistic message fragmentation* as a method for pre-sending data in all-to-all collectives. Using these contributions we implement reduce, broadcast, and alltoall algorithms, which achieve speedups over MPICH of 18.9x, 5.3x, and 6.9x respectively.

4. CONCLUSION

This work focuses on accelerating collective communications for modern high performance systems. Collective communications are an important aspect of scientific applications, and are important for the scalability of these applications.

The first part of this work presents the universal hierarchical algorithm for MPI collectives that allows the utilization of hierarchy-unaware collectives for inter-node communication. To do this we sacrifice inter- and intra-node communication overlap that past works achieved, but the potential gains realized from having the ability to use any inter-node algorithm justify the sacrifice, especially as the numbers of processes grow. We show that our algorithm works for a wide variety of collectives, including vector versions of collectives as well as alltoall, which has never been done hierarchically with shared memory. We test our algorithms with as many as 65536 cores and see speedups over the baseline averaging 14.2x for alltoallv, 26x for allgather, and 32.7x for reduce-scatter. We demonstrate the flexibility of our algorithm by proposing novel improvements for the inter- and intra-node portions of communication, and show these changes effortlessly plug into the universal hierarchical algorithm. We believe that our algorithm is an ideal solution for future MPI implementations. There exists a large number of existing collectives that are best used in different scenarios, and this algorithm allows them to be used in a hierarchical way without major modification. As new algorithms are designed for complex networks, or intra-node copies are optimized for specific architectures, the MPI implementation will be able to seamlessly integrate these features into the universal algorithm.

The second part of this work offers three major contributions that help exploit process imbalance to improve MPI collective operations in hierarchical systems. Our first contribution is a detailed examination of the the nature of process imbalance in a hierarchical system. Using a custom modified profiler and hardware performance

counters we describe this imbalance in greater detail than prior works. Our next contribution is developing the method of *dynamic leader selection* to create imbalance tolerant algorithms for fan-in and fan-out collectives. Finally we introduce the idea of *opportunistic message fragmentation* as a method for pre-sending data in all-to-all collectives. Using these contributions we implement reduce, broadcast, and alltoall algorithms, which achieve speedups over MPICH of 18.9x, 5.3x, and 6.9x respectively.

These sets of optimizations thoroughly investigate collective communications on modern clusters and offer myriad of methods for improving the latency of the communications. This work also thoroughly investigates the nature of process imbalance in order to better understand its impact on collectives. These improvements obtain impressive performance results and are well positioned to work with future systems.

LIST OF REFERENCES

LIST OF REFERENCES

[1] J. S. Vetter and F. Mueller, "Communication characteristics of large-scale scientific applications for contemporary cluster architectures," *J. Parallel Distrib. Comput.*, vol. 63, no. 9, pp. 853–865, Sept. 2003.

[2] Q. Ali, S. P. Midkiff, and V. S. Pai, "Efficient high performance collective communication for the Cell blade," in *Proc. of the 23rd Int. Conf. on Supercomputing*. ACM, 2009.

[3] P. Sanders and J. L. Träff, "The hierarchical factor algorithm for all-to-all communication," in *Euro-Par 2002 Parallel Process.*, 2002, vol. 2400, pp. 799–803.

[4] N. T. Karonis, B. R. de Supinski, I. T. Foster, W. D. Gropp, E. L. Lusk, and J. Bresnahan, "Exploiting hierarchy in parallel comput. networks to optimize collective operation performance," in *14th Int. Parallel and Distributed Process. Symp., IEEE Electron. Library*, Apr. 2000.

[5] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda, "MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics," in *Proc. 2008 8th IEEE Int. Symp. Cluster Computing and Grid*, 2008.

[6] A. Mamidala, A. Vishnu, and D. Panda, "Efficient shared memory and RDMA based design for MPI allgather over Infiniband," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2006, vol. 4192, pp. 66–75.

[7] K. Kandalla, H. Subramoni, G. Santhanaraman, M. Koop, and D. K. Panda, "Designing multi-leader-based allgather algorithms for multi-core clusters," in *Proc. of the 2009 IEEE Int. Symp. on Parallel&Distributed Process.*, 2009, pp. 1–8.

[8] V. Tipparaju, J. Nieplocha, and D. Panda, "Fast collective operations using shared and remote memory access protocols on clusters," in *Parallel and Distributed Process. Symp., 2003. Proc.. Int.* IEEE, 2003, pp. 10–pp.

[9] S. Sistare, R. vandeVaart, and E. Loh, "Optimization of MPI collectives on clusters of large-scale SMP's," in *Proc. of the 1999 ACM/IEEE Conf. on Supercomputing*. ACM, 1999.

[10] R. Graham, M. Venkata, J. Ladd, P. Shamis, I. Rabinovitz, V. Filipov, and G. Shainer, "Cheetah: A framework for scalable hierarchical collective operations," in *Cluster, Cloud and Grid Computing, 2011 11th IEEE/ACM Int. Symp. on*, 2011.

[11] H. Zhu, D. Goodell, W. Gropp, and R. Thakur, "Hierarchical collectives in MPICH2," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2009.

[12] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in MPICH," *Int. J. of High Performance Computing Applicat.*, vol. 19, no. 1, pp. 49–66, 2005.

[13] J. Matienzo and N. Enright Jerger, "Performance analysis of broadcasting algorithms on the Intel single chip cloud comput." in *Int. Symp. on Performance Anal. of Syst. and Software*, 2013.

[14] S. Li, T. Hoefler, and M. Snir, "NUMA-Aware shared memory collective communication for MPI," in *Proc. of the 22nd Int. Symp. on High-performance parallel and distributed computing*. ACM, Jun. 2013, pp. 85–96.

[15] P. Sack and W. Gropp, "Faster topology-aware collective algorithms through non-minimal communication," in *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*. ACM, 2012, pp. 45–54.

[16] A. Faraj, P. Patarasuk, and X. Yuan, "A study of process arrival patterns for MPI collective operations," *International Journal of Parallel Programming*, vol. 36, no. 6, pp. 543–570, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10766-008-0070-9

[17] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. Träff, "MPI on a million processors," ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2009. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_9

[18] J. Pješivac-Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of MPI collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, June 2007.

[19] "MPICH, a high-performance and portable implementation of the MPI," http://www.mpich.org/, accessed 10/07/2013.

[20] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MagPIe: MPIs collective communication operations for clustered wide area systems," in *Proc. PPoPP 1999*, 1999, pp. 131–140.

[21] T. Kielmann, H. E. Bal, and S. Gorlatch, "Bandwidth-efficient collective communication for clustered wide area systems," in *In Proc. Int. Parallel and Distributed Process. Symp.* IEEE, 1999, pp. 492–499.

[22] R. L. Graham and G. Shipman, "MPI support for multi-core architectures: Optimized shared memory collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, 2008, vol. 5205, pp. 130–140.

[23] B. Goglin and S. Moreaud, "Dodging non-uniform I/O access in hierarchical collective operations for multicore clusters," in *Parallel and Distributed Process. Workshops and Phd Forum, 2011 IEEE Int. Symp. on*, 2011, pp. 788 –794.

[24] R. Brightwell, K. Pedretti, and T. Hudson, "SMARTMAP: Operating system support for efficient data sharing among processes on a multi-core processor," in *Proc. of the 2008 ACM/IEEE conference on Supercomputing*, 2008, pp. 25:1–25:12.

[25] R. Brightwell and K. Pedretti, "Optimizing multi-core MPI collectives with SMARTMAP," in *Parallel Process. Workshops, 2009 ICPPW, Int. Conf. on*, 2009, pp. 370 –377.

[26] T. Ma, G. Bosilca, A. Bouteiller, B. Goglin, J. Squyres, and J. Dongarra, "Kernel assisted collective intra-node MPI communication among multi-core and many-core CPUs," in *Parallel Process. (ICPP), 2011 Int. Conf. on*, 2011, pp. 532 –541.

[27] S. Moreaud, B. Goglin, R. Namyst, and D. Goodell, "Optimizing MPI communication within large multicore nodes with kernel assistance," in *Parallel Distributed Process., Workshops and Phd Forum, 2010 IEEE Int. Symp. on*, 2010.

[28] T. Ma, G. Bosilca, A. Bouteiller, and J. Dongarra, "HierKNEM: An adaptive framework for kernel-assisted and topology-aware collective communications on many-core clusters," in *Parallel Distributed Process. Symp. (IPDPS), 2012 IEEE 26th Int.*, 2012, pp. 970–982.

[29] H. Tang and T. Yang, "Optimizing threaded MPI execution on SMP clusters," in *Proc. of the 15th Int. Conf. on Supercomputing*. ACM, 2001, pp. 381–392.

[30] R. W. Hockney, "The communication challenge for MPP: Intel Paragon and Meiko CS-2," *Parallel Comput.*, vol. 20, no. 3, pp. 389–398, Mar. 1994.

[31] J. Bruck, C. tien Ho, S. Kipnis, E. Upfal, and D. Weathersby, "Efficient algorithms for all-to-all communications in multi-port message-passing systems," in *IEEE Trans. on Parallel and Distributed Syst.*, 1997, pp. 298–309.

[32] T. Kielmann, R. F. H. Hofman, H. E. Bal, A. Plaat, and R. A. F. Bhoedjang, "MPI's reduction operations in clustered wide area systems," in *In Proc. MPIDC'99, Message Passing Interface Developer's and User's Conf.*, 1999, pp. 43–52.

[33] "Message passing toolkit (MPT) 6.0 man pages," http://docs.cray.com/cgi-bin/craydoc.cgi?mode=Show;f=man/xe_mptm/60/cat3/intro_mpi.3.html, accessed 10/14/2013.

[34] M. P. I. Forum, "MPI: A message-passing interface standard version 3.0," Sept. 2012.

[35] B. S. Parsons and V. S. Pai, "Accelerating MPI collective communications through hierarchical algorithms without sacrificing inter-node communication flexibility," in *Parallel and Distributed Processing Symposium, IEEE 28th International*, May 2014.

[36] R. Kumar, A. Mamidala, and D. Panda, "Scaling alltoall collective on multi-core systems," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–8.

[37] A. E. Eichenberger and S. G. Abraham, "Impact of load imbalance on the design of software barriers," in *in Proceedings of the 1995 International Conference on Parallel Processing*, 1995, pp. 63–72.

[38] A. R. Mamidala, J. Liu, and D. K. Panda, "Efficient barrier and allreduce infiniband clusters using hardware multicast and adaptive algorithms," in *In Proceedings of Cluster Computing*, 2004.

[39] A. Faraj, X. Yuan, and D. Lowenthal, "STAR-MPI: Self tuned adaptive routines for MPI collective operations," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 199–208. [Online]. Available: http://doi.acm.org/10.1145/1183401.1183431

[40] P. Patarasuk and X. Yuan, "Efficient MPI bcast across different process arrival patterns," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, April 2008, pp. 1–11.

[41] Y. Qian and A. Afsahi, "Process arrival pattern and shared memory aware alltoall on Infiniband," in *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 250–260. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03770-2_31

[42] ——, "Process arrival pattern aware alltoall and allgather on Infiniband clusters," *International Journal of Parallel Programming*, vol. 39, no. 4, pp. 473–493, 2011.

[43] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, no. 3, pp. 146–158, 1989. [Online]. Available: http://dx.doi.org/10.1007/BF01784024

[44] R. Gusella and S. Zatti, "The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3BSD," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-87-337, Jan 1987. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/1987/5405.html

[45] T. Jones and G. Koenig, "A clock synchronization strategy for minimizing clock variance at runtime in high-end computing environments," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2010 22nd International Symposium on*, Oct 2010, pp. 207–214.

[46] D. Becker, R. Rabenseifner, and F. Wolf, "Implications of non-constant clock drifts for the timestamps of concurrent events," in *Cluster Computing, 2008 IEEE International Conference on*, Sept 2008, pp. 59–68.

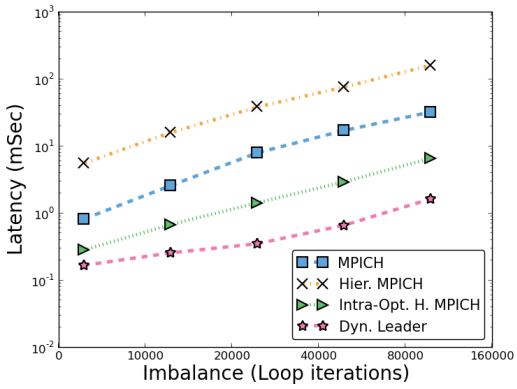
[47] "NAS parallel benchmarks," https://www.nas.nasa.gov/publications/npb.html, accessed 4/21/2014.

[48] F. Petrini, D. J. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, ser. SC '03. New York, NY, USA: ACM, 2003, pp. 55–. [Online]. Available: http://doi.acm.org/10.1145/1048935.1050204

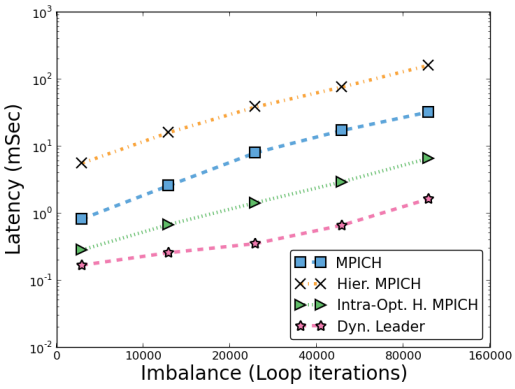
[49] "mpiP: Lightweight, scalable MPI profiling," http://mpip.sourceforge.net/, accessed 4/21/2014.

APPENDICES

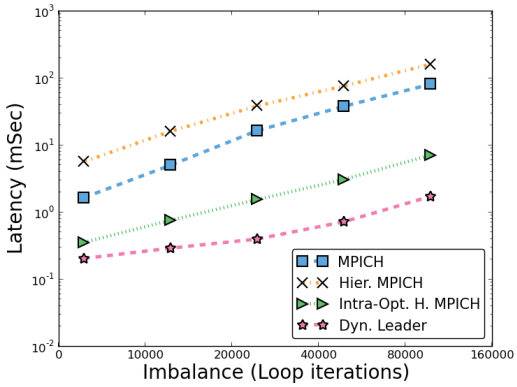
A. COMPLETE REDUCTION DATA



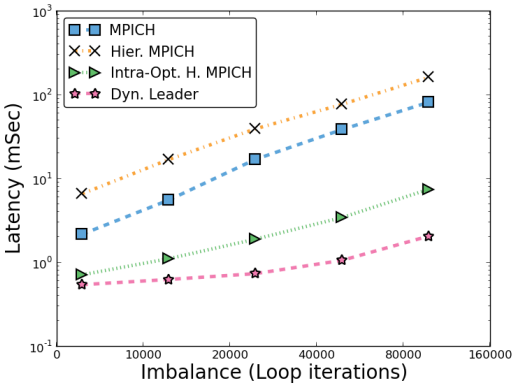
(a) 32



(b) 512

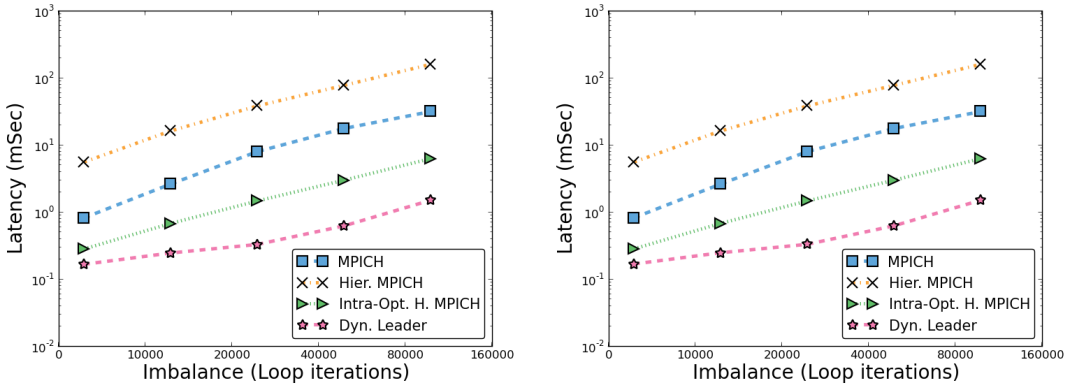


(c) 8192



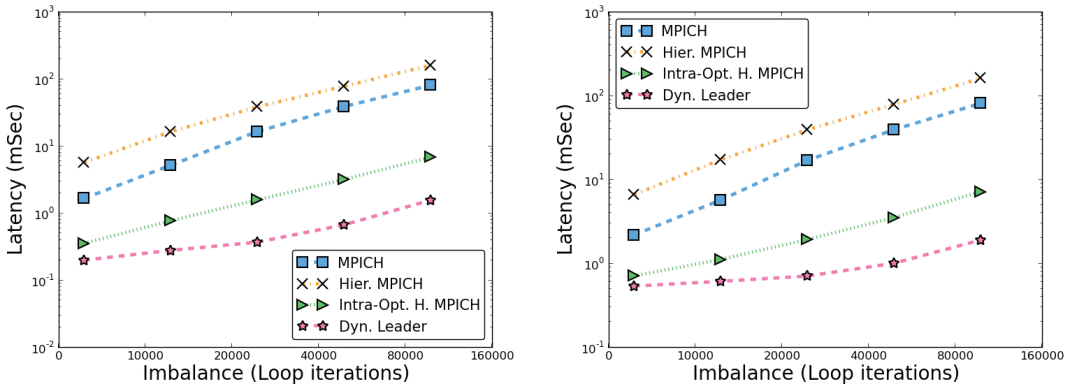
(d) 131072

Fig. A.1.: Reduction latency 512 Cores, varying buffer sizes



(a) 32

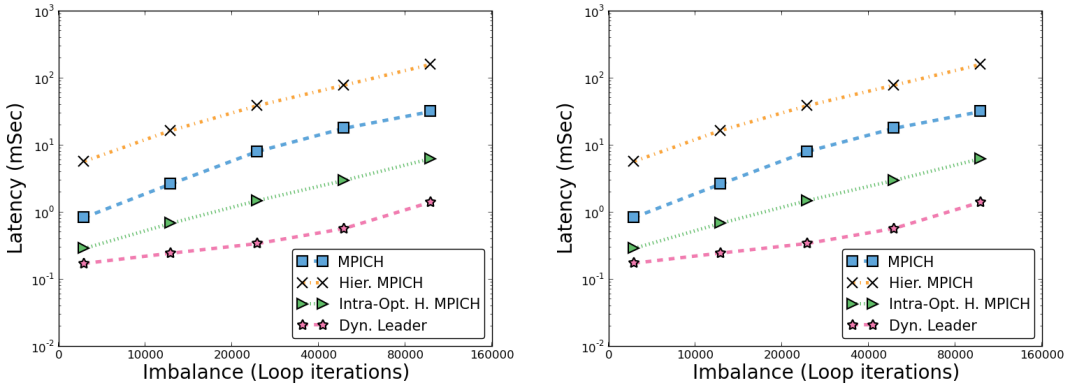
(b) 512



(c) 8192

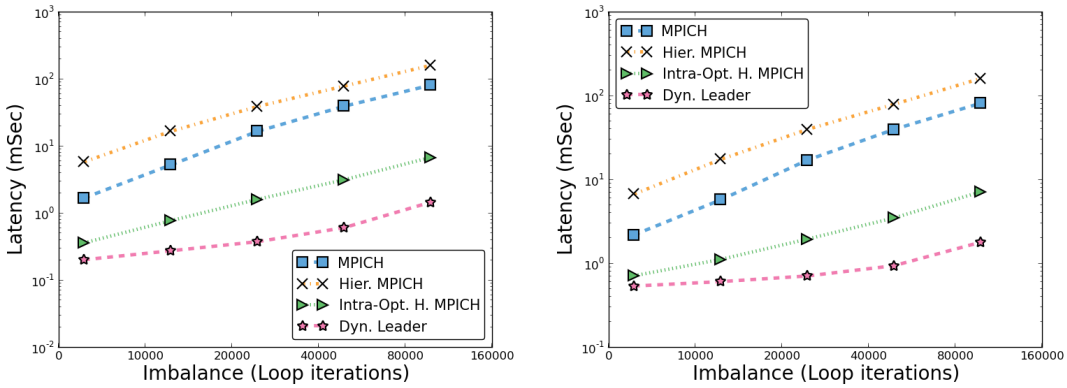
(d) 131072

Fig. A.2.: Reduction latency 1024 Cores, varying buffer sizes



(a) 32

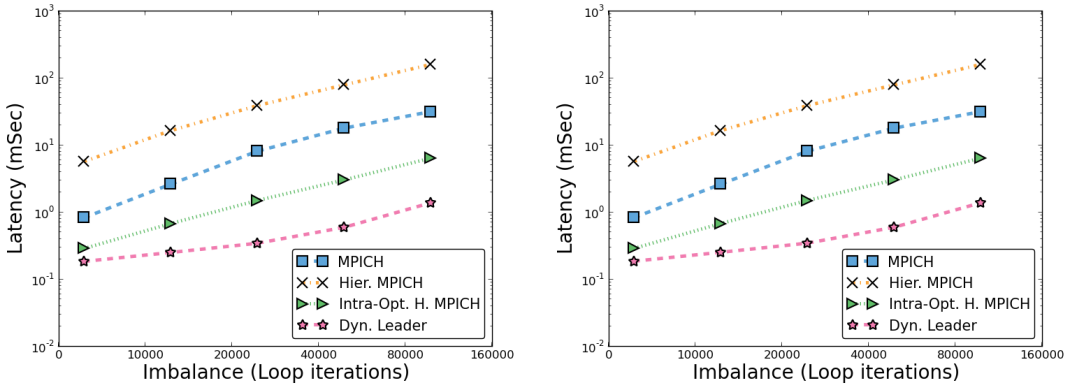
(b) 512



(c) 8192

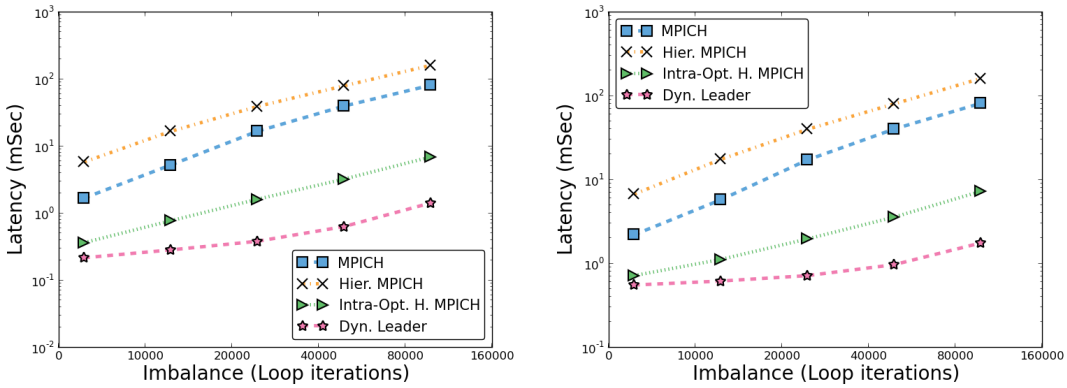
(d) 131072

Fig. A.3.: Reduction latency 2048 Cores, varying buffer sizes



(a) 32

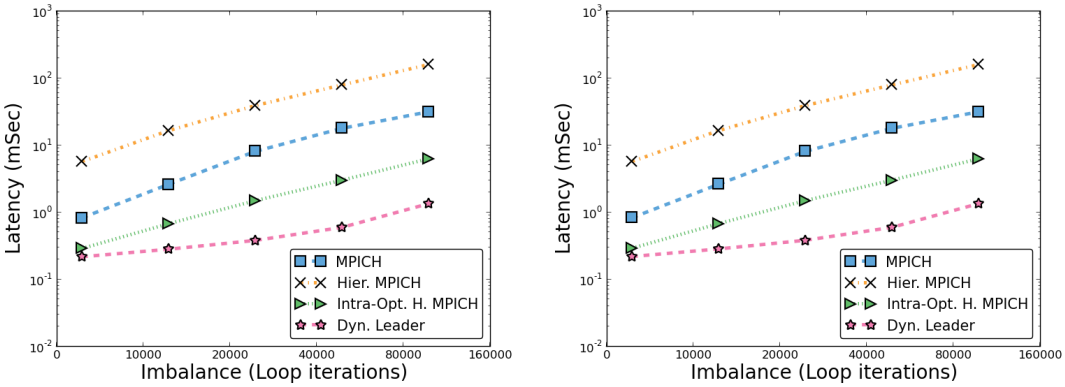
(b) 512



(c) 8192

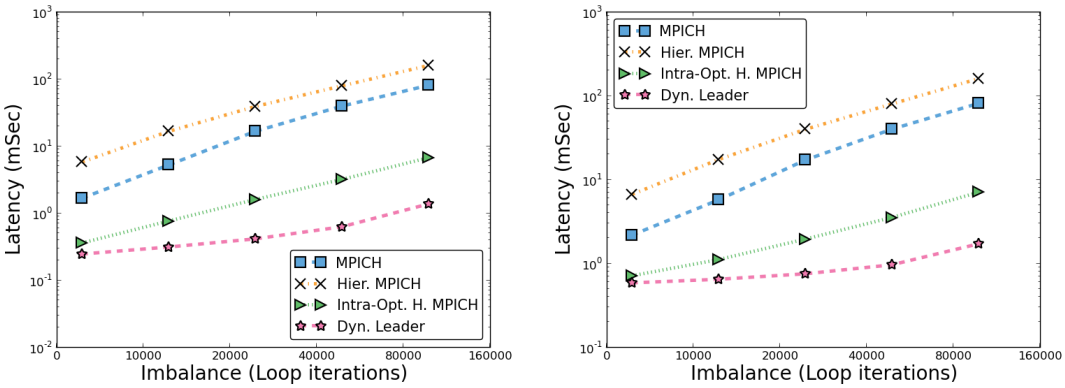
(d) 131072

Fig. A.4.: Reduction latency 4096 Cores, varying buffer sizes



(a) 32

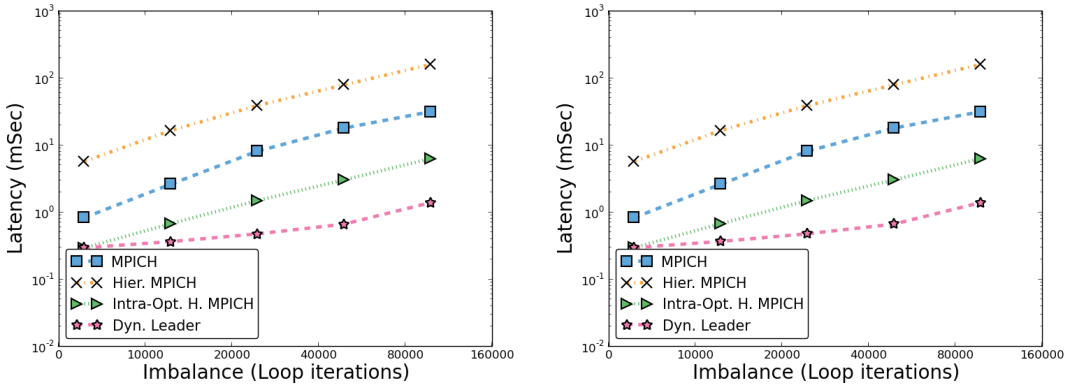
(b) 512



(c) 8192

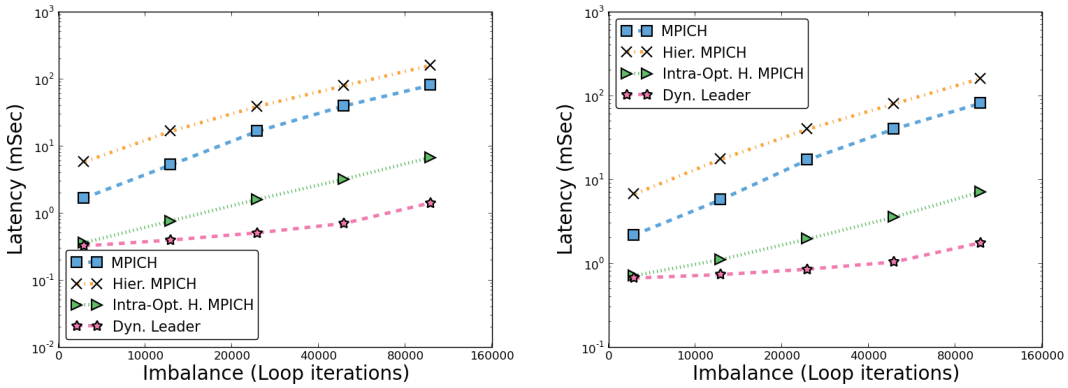
(d) 131072

Fig. A.5.: Reduction latency 8192 Cores, varying buffer sizes



(a) 32

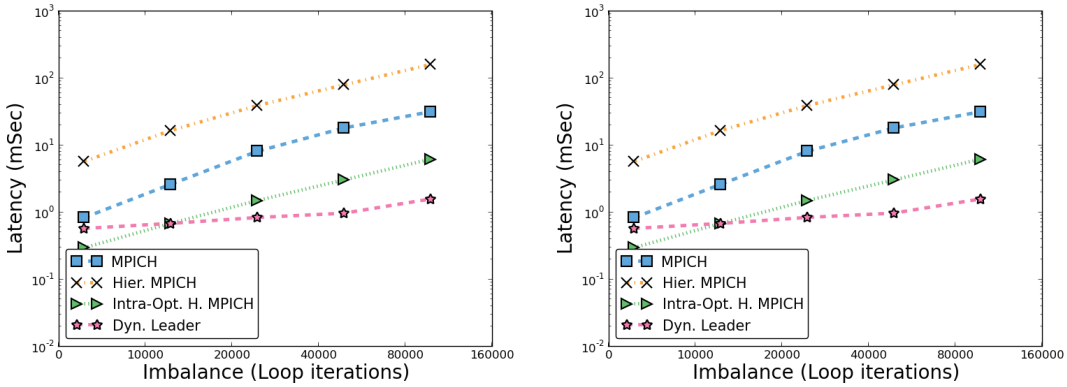
(b) 512



(c) 8192

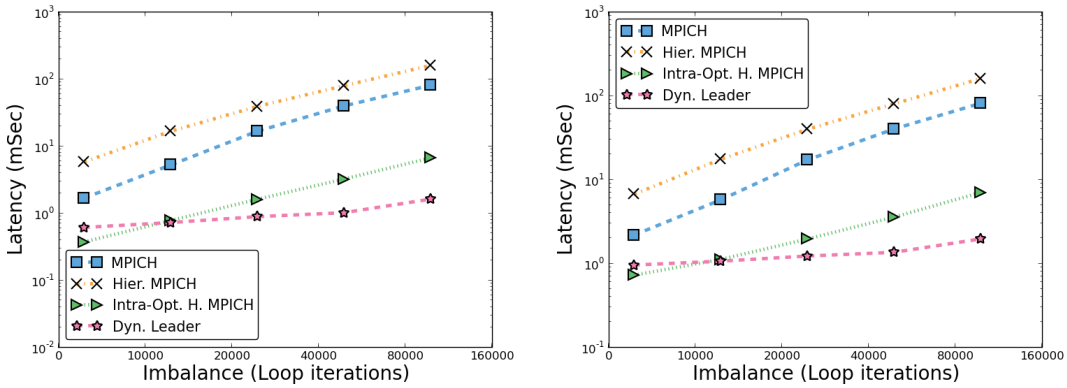
(d) 131072

Fig. A.6.: Reduction latency 16384 Cores, varying buffer sizes



(a) 32

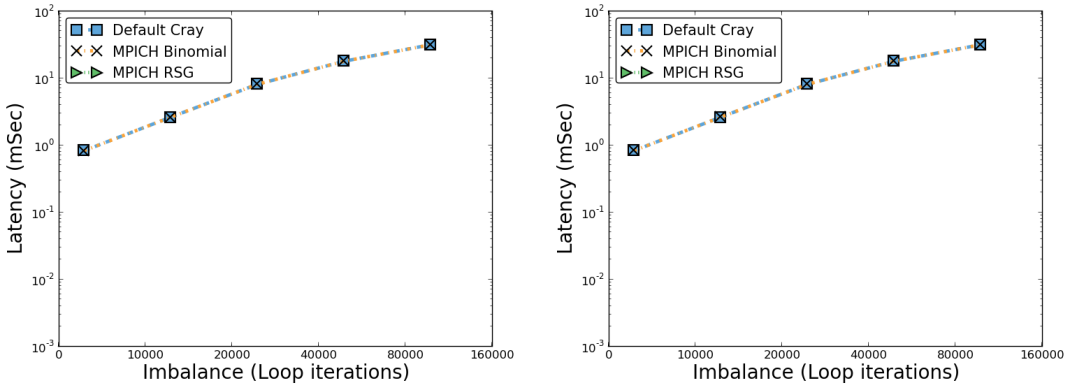
(b) 512



(c) 8192

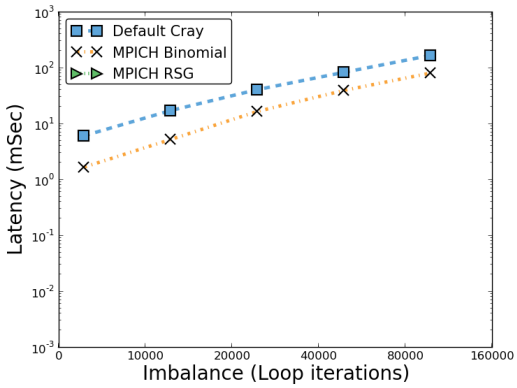
(d) 131072

Fig. A.7.: Reduction latency 32768 Cores, varying buffer sizes

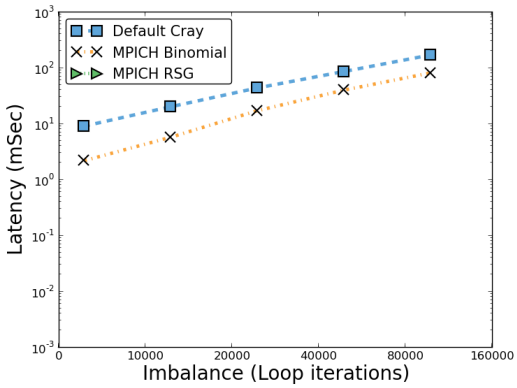


(a) 32

(b) 512

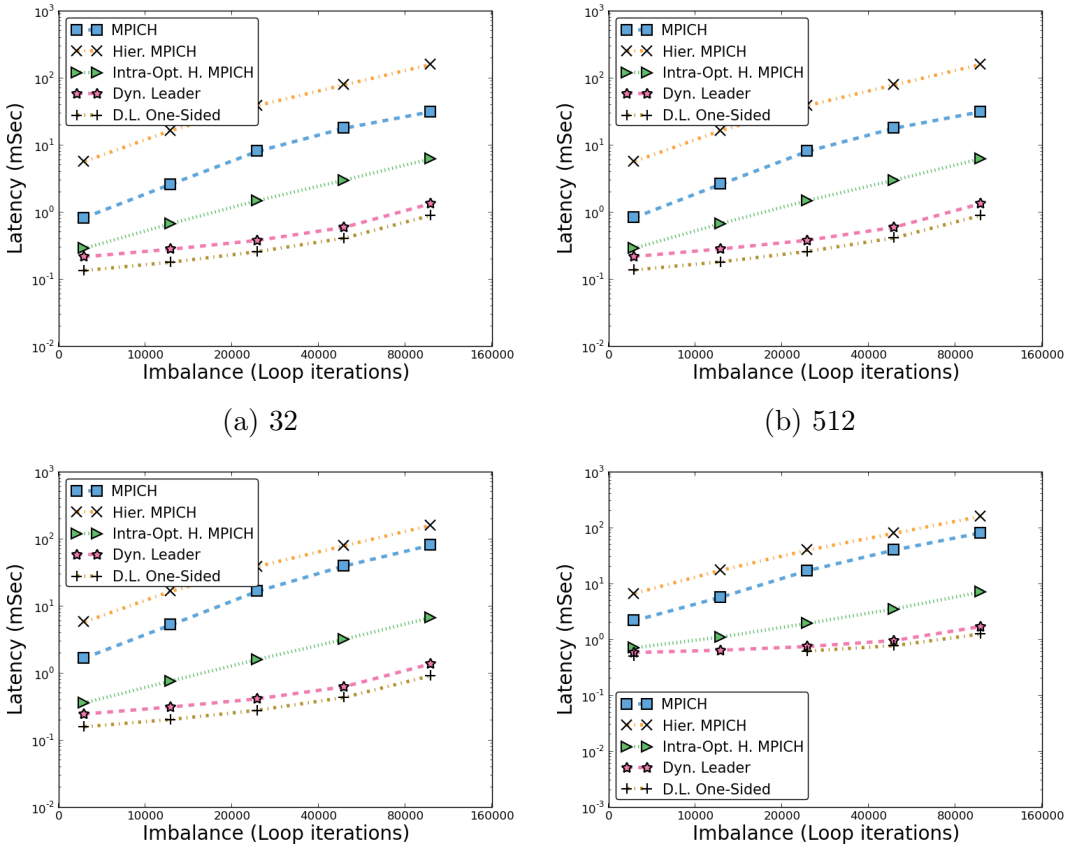


(c) 8192



(d) 131072

Fig. A.8.: Reduction latency for Cray and MPICH 8192 cores, varying buffer sizes



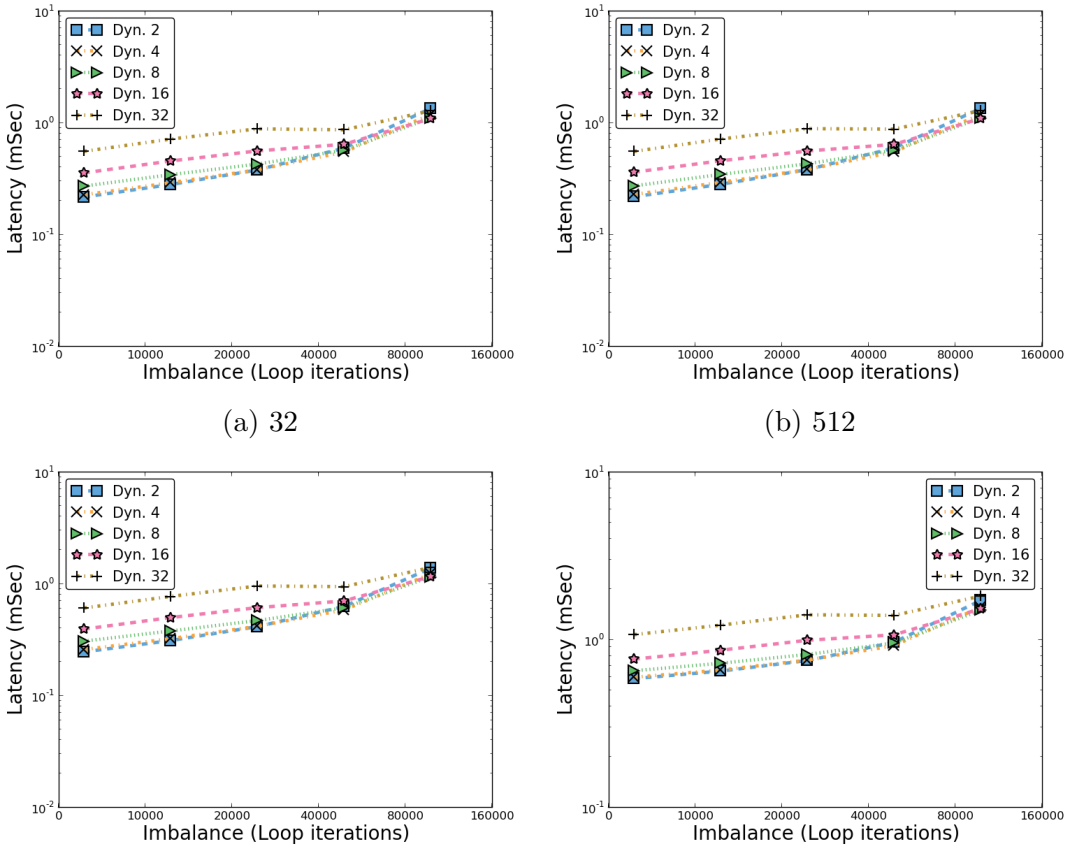
(a) 32

(b) 512

(c) 8192

(d) 131072

Fig. A.9.: Reduction latency including one-sided version 8192 cores, varying buffer sizes



(a) 32

(b) 512

(c) 8192

(d) 131072

Fig. A.10.: Reduction latency for K-degree trees 8192 cores, varying buffer sizes

Table A.1.: Reduction latency 512 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.27E-005	8.21E-004	2.58E-003	7.96E-003	1.71E-002	3.23E-002
	MPICH Bi.	2.15E-005	8.20E-004	2.58E-003	7.96E-003	1.71E-002	3.23E-002
	MPICH Bi. Hier.	4.07E-005	5.57E-003	1.59E-002	3.83E-002	7.60E-002	1.61E-001
	MPICH Bi. Hier. Opt.	2.08E-005	2.83E-004	6.76E-004	1.43E-003	2.93E-003	6.58E-003
	Dynamic Leader	2.45E-005	1.67E-004	2.56E-004	3.51E-004	6.61E-004	1.63E-003
512	Default Cray	2.50E-005	8.27E-004	2.58E-003	7.96E-003	1.71E-002	3.23E-002
	MPICH Bi.	2.42E-005	8.22E-004	2.58E-003	7.96E-003	1.71E-002	3.23E-002
	MPICH Bi. Hier.	4.46E-005	5.58E-003	1.59E-002	3.83E-002	7.60E-002	1.61E-001
	MPICH Bi. Hier. Opt.	2.20E-005	2.84E-004	6.77E-004	1.43E-003	2.93E-003	6.59E-003
	Dynamic Leader	2.55E-005	1.68E-004	2.58E-004	3.52E-004	6.62E-004	1.63E-003
8192	Default Cray	1.13E-004	6.34E-003	1.70E-002	3.96E-002	7.86E-002	1.70E-001
	MPICH Bi.	6.14E-005	1.65E-003	5.03E-003	1.65E-002	3.80E-002	8.09E-002
	MPICH Bi. Hier.	1.24E-004	5.68E-003	1.61E-002	3.85E-002	7.62E-002	1.61E-001
	MPICH Bi. Hier. Opt.	4.52E-005	3.55E-004	7.57E-004	1.54E-003	3.09E-003	7.10E-003
	Dynamic Leader	5.10E-005	2.04E-004	2.91E-004	3.98E-004	7.21E-004	1.72E-003
131072	Default Cray	7.64E-003	9.45E-003	2.01E-002	4.26E-002	8.16E-002	1.73E-001
	MPICH Bi.	7.63E-004	2.17E-003	5.55E-003	1.70E-002	3.85E-002	8.15E-002
	MPICH Bi. Hier.	1.47E-003	6.60E-003	1.69E-002	3.93E-002	7.70E-002	1.62E-001
	MPICH Bi. Hier. Opt.	4.44E-004	7.13E-004	1.11E-003	1.90E-003	3.44E-003	7.46E-003
	Dynamic Leader	4.09E-004	5.44E-004	6.27E-004	7.37E-004	1.06E-003	2.06E-003

Table A.2.: Reduction latency 1024 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.28E-005	8.25E-004	2.63E-003	8.01E-003	1.77E-002	3.17E-002
	MPICH Bi.	2.17E-005	8.23E-004	2.63E-003	8.01E-003	1.77E-002	3.17E-002
	MPICH Bi. Hier.	4.10E-005	5.60E-003	1.63E-002	3.84E-002	7.79E-002	1.60E-001
	MPICH Bi. Hier. Opt.	2.08E-005	2.83E-004	6.80E-004	1.47E-003	3.00E-003	6.29E-003
	Dynamic Leader	2.59E-005	1.67E-004	2.47E-004	3.31E-004	6.25E-004	1.51E-003
512	Default Cray	2.58E-005	8.26E-004	2.63E-003	8.01E-003	1.77E-002	3.17E-002
	MPICH Bi.	2.49E-005	8.25E-004	2.63E-003	8.01E-003	1.77E-002	3.17E-002
	MPICH Bi. Hier.	4.69E-005	5.60E-003	1.63E-002	3.84E-002	7.80E-002	1.60E-001
	MPICH Bi. Hier. Opt.	2.26E-005	2.84E-004	6.81E-004	1.47E-003	3.00E-003	6.30E-003
	Dynamic Leader	2.67E-005	1.69E-004	2.49E-004	3.32E-004	6.27E-004	1.51E-003
8192	Default Cray	1.09E-004	6.16E-003	1.72E-002	3.96E-002	8.12E-002	1.69E-001
	MPICH Bi.	6.18E-005	1.66E-003	5.19E-003	1.65E-002	3.89E-002	8.09E-002
	MPICH Bi. Hier.	1.24E-004	5.70E-003	1.64E-002	3.86E-002	7.81E-002	1.61E-001
	MPICH Bi. Hier. Opt.	4.57E-005	3.56E-004	7.70E-004	1.58E-003	3.19E-003	6.83E-003
	Dynamic Leader	4.97E-005	2.00E-004	2.79E-004	3.71E-004	6.70E-004	1.57E-003
131072	Default Cray	7.12E-003	9.15E-003	2.01E-002	4.25E-002	8.41E-002	1.72E-001
	MPICH Bi.	7.67E-004	2.19E-003	5.71E-003	1.70E-002	3.95E-002	8.14E-002
	MPICH Bi. Hier.	1.51E-003	6.63E-003	1.73E-002	3.94E-002	7.89E-002	1.62E-001
	MPICH Bi. Hier. Opt.	4.45E-004	7.12E-004	1.12E-003	1.93E-003	3.54E-003	7.19E-003
	Dynamic Leader	4.14E-004	5.38E-004	6.13E-004	7.08E-004	1.01E-003	1.91E-003

Table A.3.: Reduction latency 2048 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.31E-005	8.33E-004	2.63E-003	8.04E-003	1.78E-002	3.17E-002
	MPICH Bi.	2.39E-005	8.32E-004	2.63E-003	8.04E-003	1.78E-002	3.17E-002
	MPICH Bi. Hier.	4.03E-005	5.71E-003	1.65E-002	3.88E-002	7.83E-002	1.60E-001
	MPICH Bi. Hier. Opt.	2.11E-005	2.87E-004	6.81E-004	1.49E-003	2.98E-003	6.29E-003
	Dynamic Leader	2.83E-005	1.72E-004	2.45E-004	3.40E-004	5.71E-004	1.41E-003
512	Default Cray	2.55E-005	8.34E-004	2.64E-003	8.05E-003	1.78E-002	3.17E-002
	MPICH Bi.	2.48E-005	8.34E-004	2.64E-003	8.05E-003	1.78E-002	3.17E-002
	MPICH Bi. Hier.	4.49E-005	5.71E-003	1.65E-002	3.88E-002	7.83E-002	1.60E-001
	MPICH Bi. Hier. Opt.	2.28E-005	2.89E-004	6.83E-004	1.49E-003	2.98E-003	6.29E-003
	Dynamic Leader	2.87E-005	1.74E-004	2.46E-004	3.42E-004	5.73E-004	1.42E-003
8192	Default Cray	1.05E-004	6.18E-003	1.73E-002	4.02E-002	8.17E-002	1.67E-001
	MPICH Bi.	6.11E-005	1.68E-003	5.24E-003	1.66E-002	3.93E-002	8.12E-002
	MPICH Bi. Hier.	1.24E-004	5.82E-003	1.66E-002	3.89E-002	7.85E-002	1.60E-001
	MPICH Bi. Hier. Opt.	4.61E-005	3.61E-004	7.73E-004	1.61E-003	3.15E-003	6.80E-003
	Dynamic Leader	5.13E-005	2.02E-004	2.75E-004	3.76E-004	6.08E-004	1.46E-003
131072	Default Cray	7.08E-003	9.14E-003	2.02E-002	4.31E-002	8.45E-002	1.70E-001
	MPICH Bi.	7.56E-004	2.19E-003	5.74E-003	1.71E-002	3.98E-002	8.17E-002
	MPICH Bi. Hier.	1.50E-003	6.71E-003	1.75E-002	3.97E-002	7.93E-002	1.61E-001
	MPICH Bi. Hier. Opt.	4.42E-004	7.14E-004	1.12E-003	1.96E-003	3.50E-003	7.15E-003
	Dynamic Leader	4.11E-004	5.38E-004	6.08E-004	7.10E-004	9.42E-004	1.79E-003

Table A.4.: Reduction latency 4096 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.25E-005	8.28E-004	2.62E-003	8.11E-003	1.79E-002	3.16E-002
	MPICH Bi.	2.17E-005	8.27E-004	2.61E-003	8.11E-003	1.79E-002	3.16E-002
	MPICH Bi. Hier.	4.11E-005	5.68E-003	1.65E-002	3.88E-002	7.90E-002	1.60E-001
	MPICH Bi. Hier. Opt.	2.25E-005	2.87E-004	6.76E-004	1.49E-003	3.03E-003	6.39E-003
	Dynamic Leader	2.17E-005	8.27E-004	2.61E-003	8.11E-003	1.79E-002	3.16E-002
512	Default Cray	2.57E-005	8.30E-004	2.62E-003	8.12E-003	1.79E-002	3.16E-002
	MPICH Bi.	2.49E-005	8.30E-004	2.62E-003	8.12E-003	1.79E-002	3.16E-002
	MPICH Bi. Hier.	4.52E-005	5.71E-003	1.65E-002	3.88E-002	7.90E-002	1.60E-001
	MPICH Bi. Hier. Opt.	2.40E-005	2.89E-004	6.77E-004	1.50E-003	3.03E-003	6.40E-003
	Dynamic Leader	2.49E-005	8.30E-004	2.62E-003	8.12E-003	1.79E-002	3.16E-002
8192	Default Cray	1.09E-004	6.11E-003	1.72E-002	4.05E-002	8.32E-002	1.68E-001
	MPICH Bi.	6.14E-005	1.68E-003	5.23E-003	1.66E-002	3.96E-002	8.09E-002
	MPICH Bi. Hier.	1.47E-004	5.79E-003	1.66E-002	3.90E-002	7.93E-002	1.60E-001
	MPICH Bi. Hier. Opt.	5.34E-005	3.63E-004	7.71E-004	1.61E-003	3.22E-003	6.90E-003
	Dynamic Leader	6.64E-005	2.16E-004	2.84E-004	3.81E-004	6.32E-004	1.42E-003
131072	Default Cray	7.24E-003	9.09E-003	2.02E-002	4.34E-002	8.61E-002	1.71E-001
	MPICH Bi.	7.53E-004	2.20E-003	5.75E-003	1.72E-002	4.01E-002	8.14E-002
	MPICH Bi. Hier.	1.54E-003	6.70E-003	1.74E-002	3.98E-002	8.01E-002	1.61E-001
	MPICH Bi. Hier. Opt.	4.53E-004	7.17E-004	1.12E-003	1.96E-003	3.57E-003	7.25E-003
	Dynamic Leader	4.20E-004	5.54E-004	6.17E-004	7.17E-004	9.68E-004	1.76E-003

Table A.5.: Reduction latency 8192 cores, cont.

		0	10000	20000	40000	80000	160000
32	Default Cray	2.44E-005	8.26E-004	2.61E-003	8.07E-003	1.78E-002	3.14E-002
	MPICH Bi.	2.32E-005	8.25E-004	2.61E-003	8.07E-003	1.78E-002	3.14E-002
	MPICH Bi. Hier.	4.28E-005	5.68E-003	1.65E-002	3.89E-002	7.90E-002	1.59E-001
	MPICH Bi. Hier. Opt.	2.54E-005	2.87E-004	6.72E-004	1.49E-003	3.00E-003	6.26E-003
	Dynamic Leader	4.12E-005	2.17E-004	2.80E-004	3.79E-004	5.93E-004	1.34E-003
	MPICH RSG						
	Dynamic-4	4.43E-005	2.27E-004	2.91E-004	3.80E-004	5.47E-004	1.18E-003
	Dynamic-8	4.88E-005	2.72E-004	3.43E-004	4.27E-004	5.68E-004	1.10E-003
	Dynamic-16	6.24E-005	3.56E-004	4.55E-004	5.60E-004	6.42E-004	1.09E-003
	Dynamic-32	9.98E-005	5.55E-004	7.15E-004	8.83E-004	8.66E-004	1.28E-003
	Dynamic-OS	3.05E-005	1.35E-004	1.79E-004	2.56E-004	4.10E-004	8.91E-004

Table A.6.: Reduction latency 8192 cores, cont.

512	Default Cray	2.74E-005	8.28E-004	2.61E-003	8.07E-003	1.78E-002	3.14E-002
	MPICH Bi.	2.64E-005	8.27E-004	2.61E-003	8.08E-003	1.78E-002	3.14E-002
	MPICH Bi. Hier.	4.67E-005	5.69E-003	1.65E-002	3.89E-002	7.90E-002	1.59E-001
	MPICH Bi. Hier. Opt.	2.68E-005	2.89E-004	6.74E-004	1.49E-003	3.00E-003	6.26E-003
	Dynamic Leader	3.96E-005	2.18E-004	2.82E-004	3.81E-004	5.95E-004	1.34E-003
	MPICH RSG						
	Dynamic-4	4.04E-005	2.28E-004	2.93E-004	3.82E-004	5.49E-004	1.19E-003
	Dynamic-8	4.71E-005	2.73E-004	3.45E-004	4.30E-004	5.68E-004	1.11E-003
	Dynamic-16	5.53E-005	3.61E-004	4.57E-004	5.59E-004	6.41E-004	1.09E-003
	Dynamic-32	8.23E-005	5.55E-004	7.15E-004	8.86E-004	8.78E-004	1.29E-003
	Dynamic-OS	3.20E-005	1.37E-004	1.81E-004	2.58E-004	4.13E-004	8.93E-004

Table A.7.: Reduction latency 8192 cores, cont.

		0	10000	20000	40000	80000	160000
8192	Default Cray	1.33E-004	6.09E-003	1.73E-002	4.09E-002	8.34E-002	1.68E-001
	MPICH Bi.	6.53E-005	1.67E-003	5.23E-003	1.66E-002	3.96E-002	8.09E-002
	MPICH Bi. Hier.	1.32E-004	5.79E-003	1.66E-002	3.90E-002	7.93E-002	1.60E-001
	MPICH Bi. Hier. Opt.	4.95E-005	3.61E-004	7.64E-004	1.60E-003	3.19E-003	6.74E-003
	Dynamic Leader	6.20E-005	2.46E-004	3.11E-004	4.14E-004	6.27E-004	1.38E-003
	MPICH RSG	1.96E-003	1.81E-002	4.04E-002	9.87E-002	2.10E-001	4.13E-001
	Dynamic-4	6.49E-005	2.58E-004	3.23E-004	4.17E-004	5.84E-004	1.22E-003
	Dynamic-8	7.32E-005	3.07E-004	3.79E-004	4.69E-004	6.10E-004	1.15E-003
	Dynamic-16	9.08E-005	3.93E-004	4.99E-004	6.12E-004	7.02E-004	1.16E-003
	Dynamic-32	1.29E-004	6.07E-004	7.71E-004	9.56E-004	9.44E-004	1.40E-003
	Dynamic-OS	5.38E-005	1.59E-004	2.02E-004	2.82E-004	4.34E-004	9.14E-004

Table A.8.: Reduction latency 8192 cores, cont.

		0	10000	20000	40000	80000	160000
131072	Default Cray	7.11E-003	9.01E-003	2.01E-002	4.37E-002	8.62E-002	1.70E-001
	MPICH Bi.	7.39E-004	2.18E-003	5.74E-003	1.71E-002	4.01E-002	8.14E-002
	MPICH Bi. Hier.	1.49E-003	6.68E-003	1.74E-002	3.99E-002	8.01E-002	1.61E-001
	MPICH Bi. Hier. Opt.	4.46E-004	7.12E-004	1.11E-003	1.95E-003	3.54E-003	7.09E-003
	Dynamic Leader	4.04E-004	5.86E-004	6.50E-004	7.51E-004	9.65E-004	1.71E-003
	MPICH RSG	7.71E-003	2.15E-002	4.35E-002	1.02E-001	2.13E-001	4.17E-001
	Dynamic-4	4.03E-004	5.97E-004	6.59E-004	7.56E-004	9.22E-004	1.56E-003
	Dynamic-8	4.36E-004	6.53E-004	7.24E-004	8.16E-004	9.56E-004	1.50E-003
	Dynamic-16	4.94E-004	7.65E-004	8.65E-004	9.92E-004	1.07E-003	1.53E-003
	Dynamic-32	5.98E-004	1.07E-003	1.22E-003	1.41E-003	1.40E-003	1.82E-003
	Dynamic-OS	4.24E-004	5.08E-004	5.47E-004	6.25E-004	7.80E-004	1.26E-003

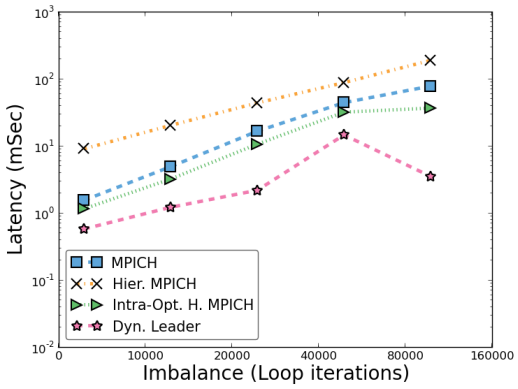
Table A.9.: Reduction latency 16384 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.34E-005	8.27E-004	2.61E-003	8.11E-003	1.80E-002	3.16E-002
	MPICH Bi.	2.21E-005	8.26E-004	2.61E-003	8.11E-003	1.80E-002	3.16E-002
	MPICH Bi. Hier.	4.12E-005	5.70E-003	1.65E-002	3.89E-002	7.92E-002	1.59E-001
	MPICH Bi. Hier. Opt.	2.94E-005	2.93E-004	6.74E-004	1.49E-003	3.03E-003	6.30E-003
	Dynamic Leader	5.44E-005	2.95E-004	3.64E-004	4.75E-004	6.64E-004	1.39E-003
512	Default Cray	2.62E-005	8.29E-004	2.61E-003	8.11E-003	1.80E-002	3.16E-002
	MPICH Bi.	2.55E-005	8.28E-004	2.61E-003	8.11E-003	1.80E-002	3.16E-002
	MPICH Bi. Hier.	4.75E-005	5.70E-003	1.65E-002	3.89E-002	7.92E-002	1.59E-001
	MPICH Bi. Hier. Opt.	3.15E-005	2.95E-004	6.75E-004	1.49E-003	3.03E-003	6.31E-003
	Dynamic Leader	4.92E-005	2.96E-004	3.66E-004	4.77E-004	6.66E-004	1.39E-003
8192	Default Cray	1.51E-004	6.10E-003	1.74E-002	4.12E-002	8.40E-002	1.68E-001
	MPICH Bi.	6.28E-005	1.68E-003	5.25E-003	1.67E-002	3.97E-002	8.10E-002
	MPICH Bi. Hier.	1.29E-004	5.80E-003	1.66E-002	3.90E-002	7.94E-002	1.60E-001
	MPICH Bi. Hier. Opt.	5.40E-005	3.63E-004	7.65E-004	1.61E-003	3.22E-003	6.79E-003
	Dynamic Leader	7.04E-005	3.26E-004	3.98E-004	5.10E-004	7.05E-004	1.43E-003
131072	Default Cray	9.73E-004	6.76E-003	1.81E-002	4.18E-002	8.46E-002	1.69E-001
	MPICH Bi.	7.31E-004	2.19E-003	5.75E-003	1.72E-002	4.02E-002	8.15E-002
	MPICH Bi. Hier.	1.50E-003	6.71E-003	1.75E-002	3.99E-002	8.03E-002	1.61E-001
	MPICH Bi. Hier. Opt.	4.40E-004	7.16E-004	1.11E-003	1.96E-003	3.57E-003	7.14E-003
	Dynamic Leader	4.10E-004	6.71E-004	7.37E-004	8.55E-004	1.04E-003	1.77E-003

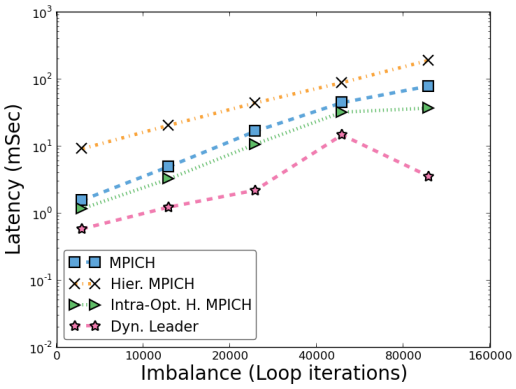
Table A.10.: Reduction latency 32768 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.39E-005	8.26E-004	2.61E-003	8.09E-003	1.80E-002	3.15E-002
	MPICH Bi.	2.24E-005	8.28E-004	2.60E-003	8.09E-003	1.80E-002	3.15E-002
	MPICH Bi. Hier.	4.16E-005	5.71E-003	1.65E-002	3.89E-002	7.93E-002	1.59E-001
	MPICH Bi. Hier. Opt.	3.85E-005	2.94E-004	6.81E-004	1.49E-003	3.03E-003	6.19E-003
	Dynamic Leader	1.08E-004	5.72E-004	6.81E-004	8.34E-004	9.67E-004	1.58E-003
512	Default Cray	2.66E-005	8.28E-004	2.61E-003	8.10E-003	1.80E-002	3.15E-002
	MPICH Bi.	2.57E-005	8.28E-004	2.61E-003	8.10E-003	1.80E-002	3.15E-002
	MPICH Bi. Hier.	4.67E-005	5.70E-003	1.65E-002	3.89E-002	7.93E-002	1.59E-001
	MPICH Bi. Hier. Opt.	4.01E-005	2.96E-004	6.80E-004	1.49E-003	3.03E-003	6.19E-003
	Dynamic Leader	9.19E-005	5.73E-004	6.80E-004	8.36E-004	9.70E-004	1.58E-003
8192	Default Cray	1.42E-004	6.11E-003	1.76E-002	4.17E-002	8.46E-002	1.68E-001
	MPICH Bi.	6.35E-005	1.68E-003	5.24E-003	1.67E-002	3.98E-002	8.10E-002
	MPICH Bi. Hier.	1.37E-004	5.81E-003	1.66E-002	3.91E-002	7.95E-002	1.60E-001
	MPICH Bi. Hier. Opt.	6.88E-005	3.73E-004	7.78E-004	1.61E-003	3.23E-003	6.67E-003
	Dynamic Leader	1.13E-004	6.13E-004	7.25E-004	8.89E-004	1.02E-003	1.62E-003
131072	Default Cray	1.03E-003	6.81E-003	1.82E-002	4.24E-002	8.52E-002	1.69E-001
	MPICH Bi.	7.17E-004	2.18E-003	5.74E-003	1.72E-002	4.03E-002	8.15E-002
	MPICH Bi. Hier.	1.44E-003	6.70E-003	1.75E-002	3.99E-002	8.04E-002	1.61E-001
	MPICH Bi. Hier. Opt.	4.41E-004	7.26E-004	1.12E-003	1.96E-003	3.58E-003	7.02E-003
	Dynamic Leader	4.07E-004	9.59E-004	1.07E-003	1.23E-003	1.36E-003	1.96E-003

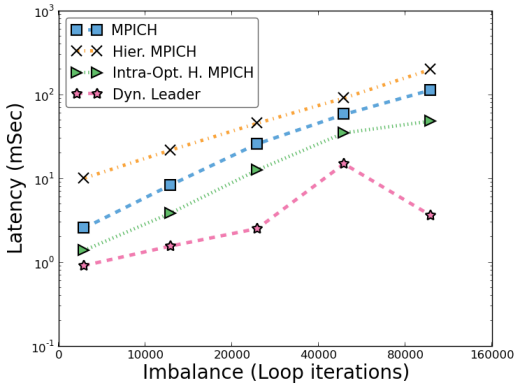
B. COMPLETE BROADCAST DATA



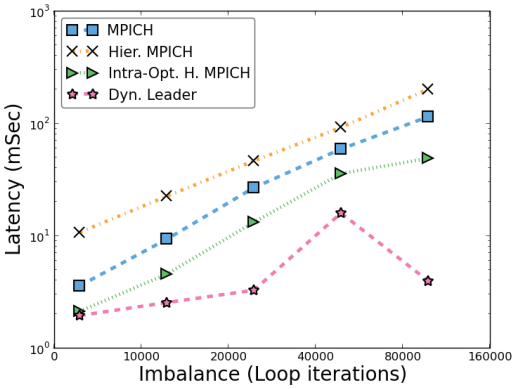
(a) 32



(b) 512

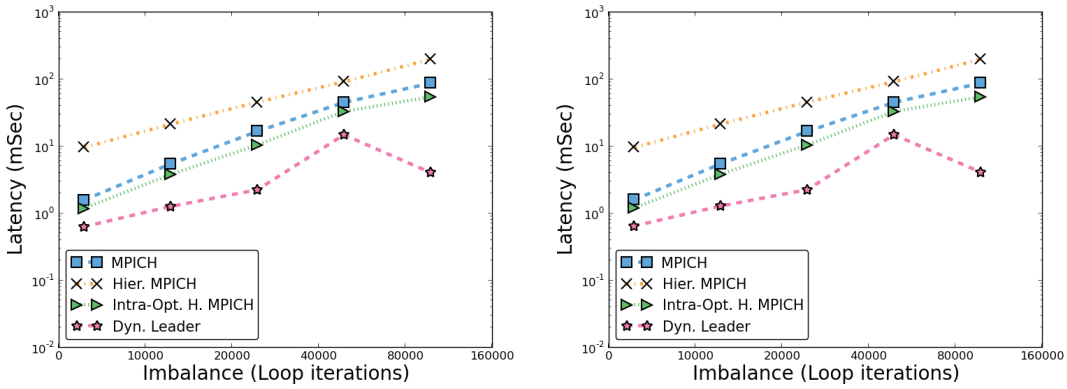


(c) 8192



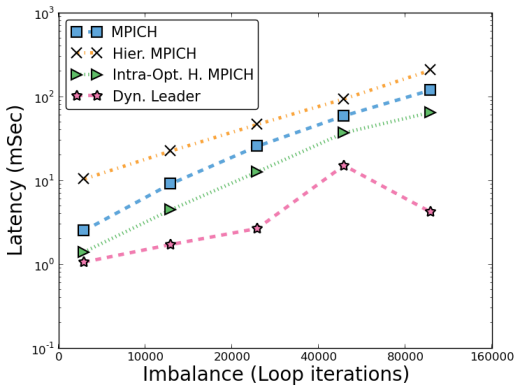
(d) 131072

Fig. B.1.: Broadcast latency 512 cores, varying buffer sizes

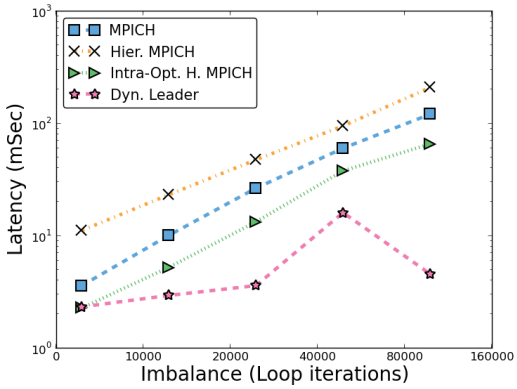


(a) 32

(b) 512

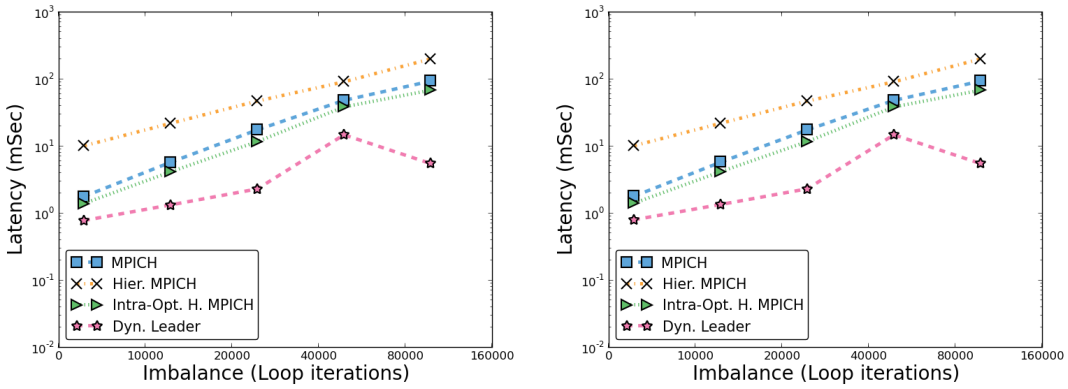


(c) 8192



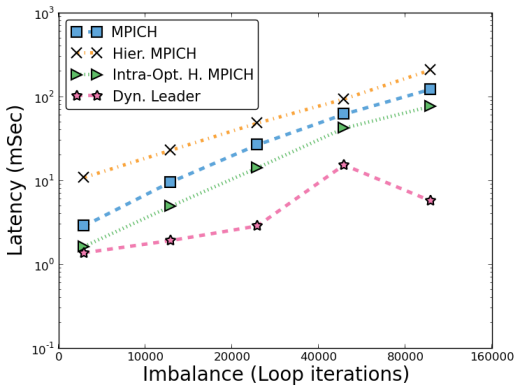
(d) 131072

Fig. B.2.: Broadcast latency 1024 cores, varying buffer sizes

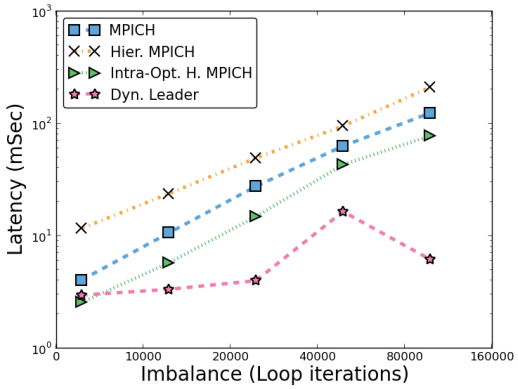


(a) 32

(b) 512

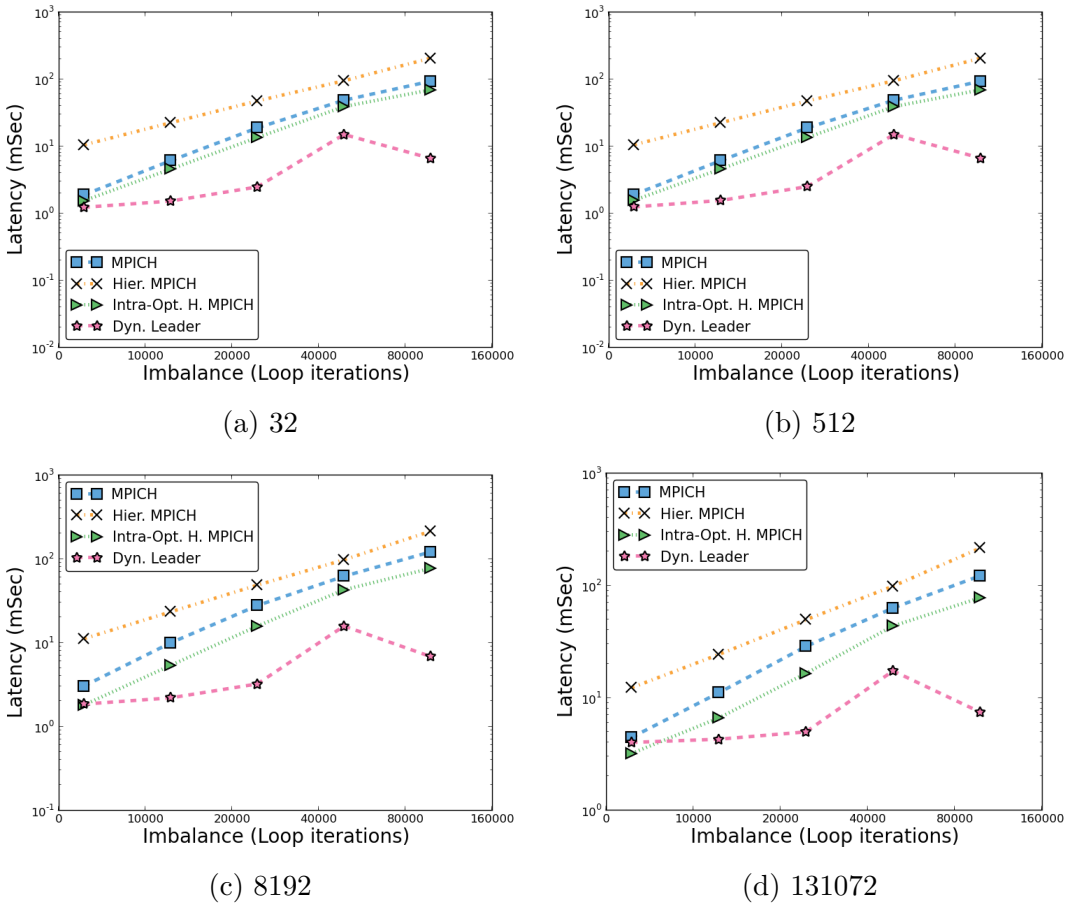


(c) 8192



(d) 131072

Fig. B.3.: Broadcast latency 2048 cores, varying buffer sizes



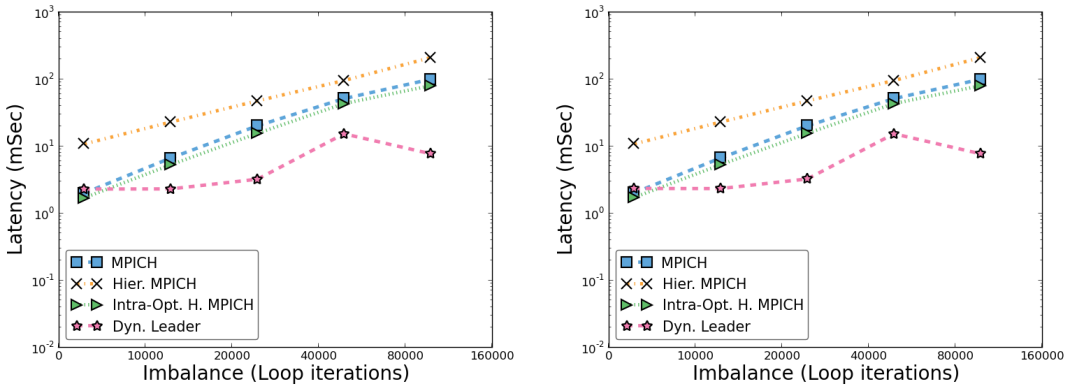
(a) 32

(b) 512

(c) 8192

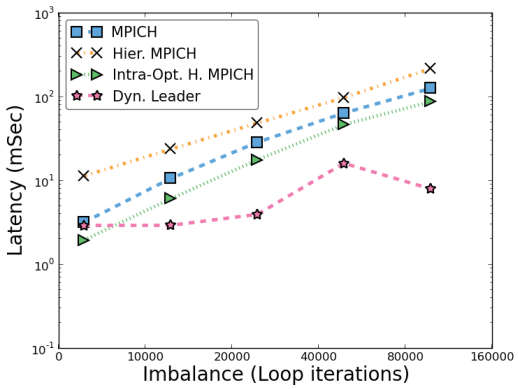
(d) 131072

Fig. B.4.: Broadcast latency 4096 cores, varying buffer sizes

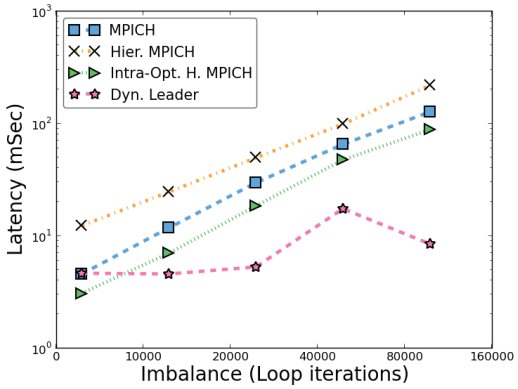


(a) 32

(b) 512

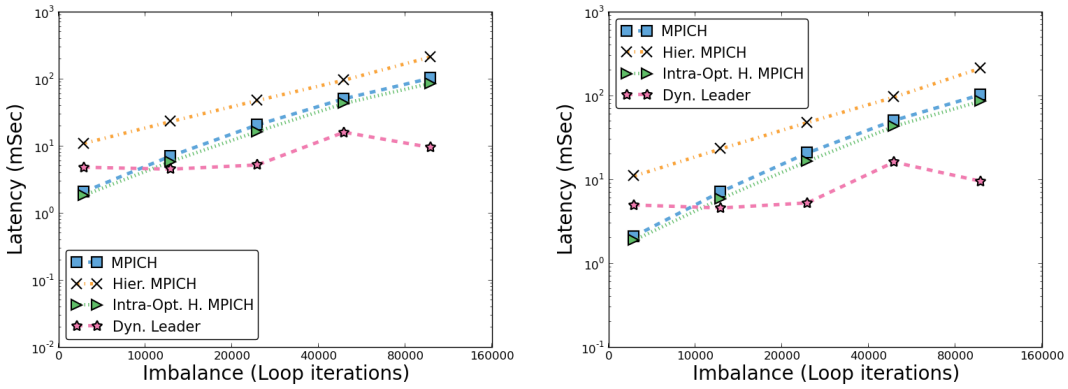


(c) 8192



(d) 131072

Fig. B.5.: Broadcast latency 8192 cores, varying buffer sizes



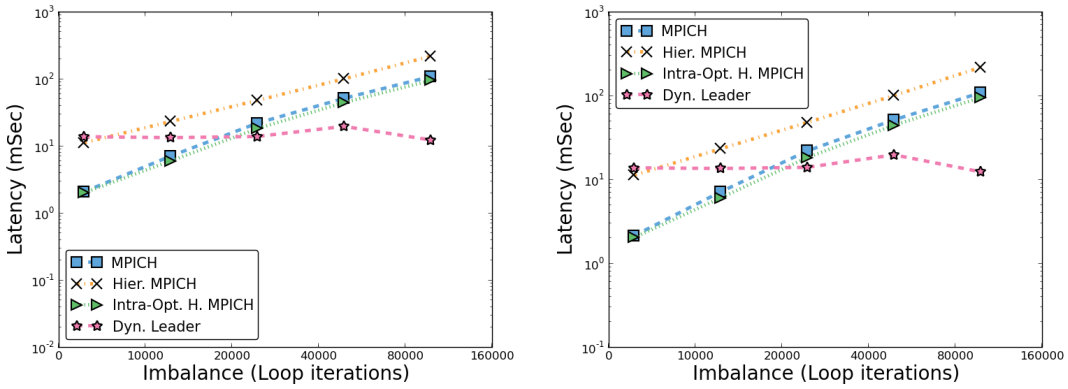
(a) 32

(b) 512

(c) 8192

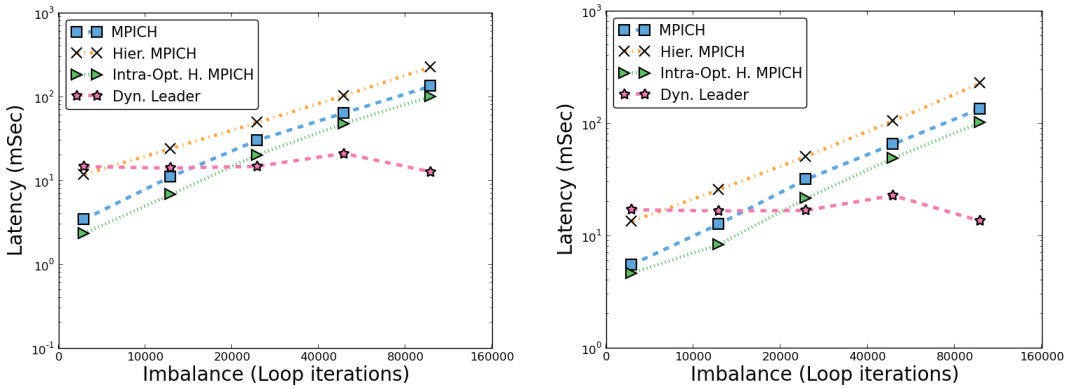
(d) 131072

Fig. B.6.: Broadcast latency 16384 cores, varying buffer sizes



(a) 32

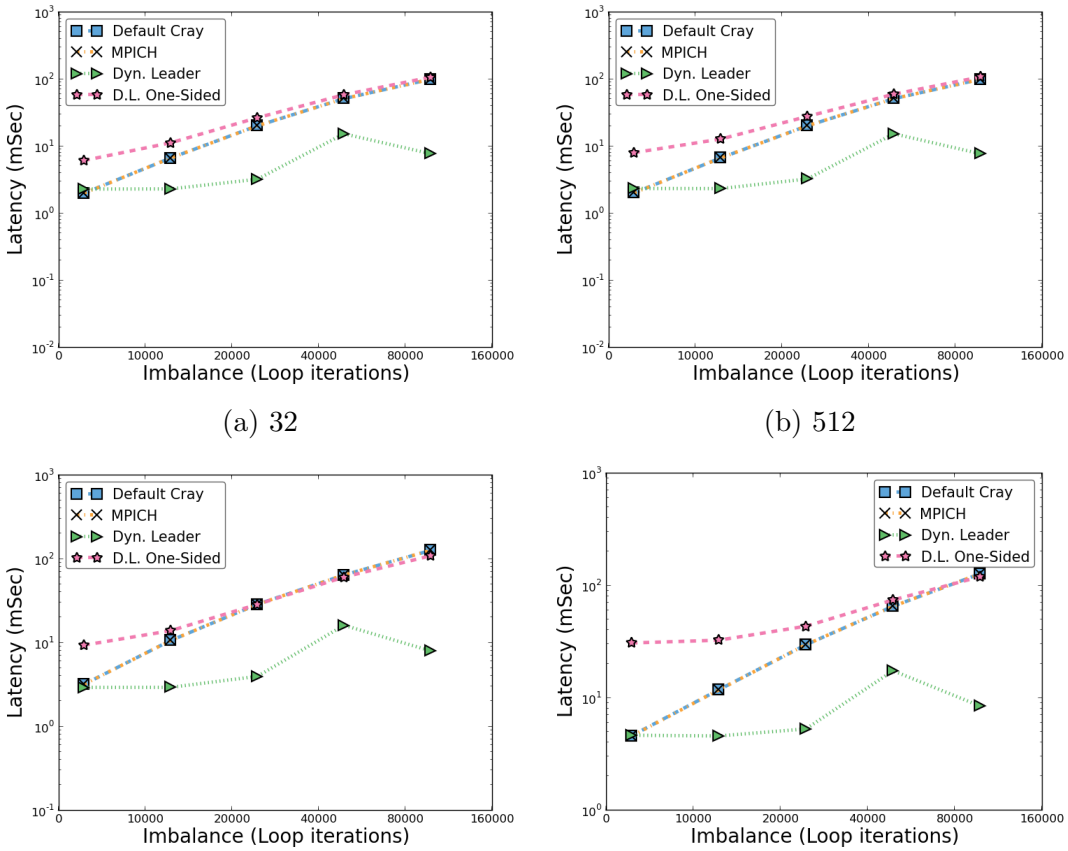
(b) 512



(c) 8192

(d) 131072

Fig. B.7.: Broadcast latency 32768 cores, varying buffer sizes



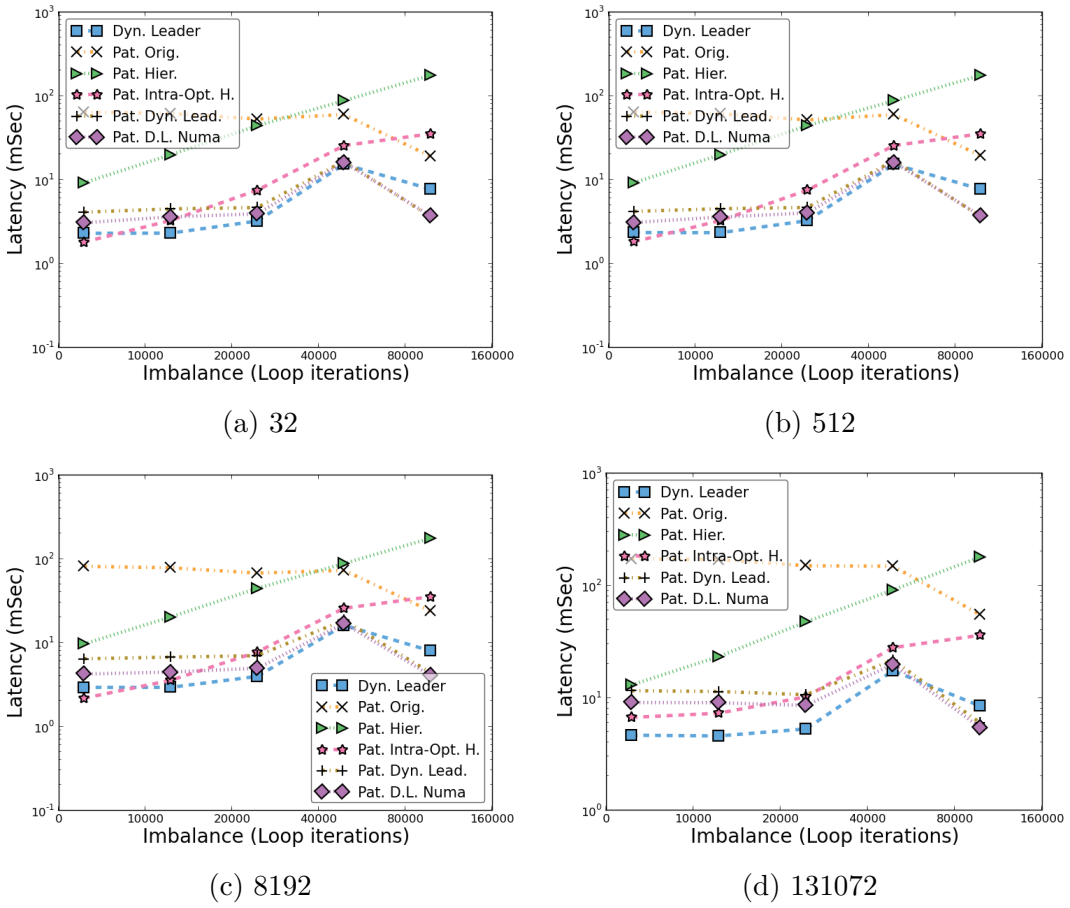
(a) 32

(b) 512

(c) 8192

(d) 131072

Fig. B.8.: Broadcast latency with cray and one sided dynamic 8192 cores, varying buffer sizes



(a) 32

(b) 512

(c) 8192

(d) 131072

Fig. B.9.: Broadcast latency for Pat. algorithm 8192 cores, varying buffer sizes

Table B.1.: Broadcast latency 512 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.87E-005	1.56E-003	4.94E-003	1.67E-002	4.45E-002	7.86E-002
	MPICH Bi.	2.74E-005	1.56E-003	4.94E-003	1.67E-002	4.45E-002	7.86E-002
	MPICH Bi. Hier.	5.00E-005	9.11E-003	2.04E-002	4.41E-002	8.85E-002	1.90E-001
	MPICH Bi. Hier. Opt.	2.63E-005	1.16E-003	3.24E-003	1.07E-002	3.21E-002	3.69E-002
	Dynamic Leader	1.09E-004	5.81E-004	1.23E-003	2.20E-003	1.47E-002	3.54E-003
512	Default Cray	3.29E-005	1.56E-003	4.94E-003	1.67E-002	4.45E-002	7.86E-002
	MPICH Bi.	3.46E-005	1.56E-003	4.94E-003	1.67E-002	4.45E-002	7.86E-002
	MPICH Bi. Hier.	5.50E-005	9.12E-003	2.04E-002	4.41E-002	8.86E-002	1.90E-001
	MPICH Bi. Hier. Opt.	3.62E-005	1.17E-003	3.24E-003	1.07E-002	3.21E-002	3.69E-002
	Dynamic Leader	1.17E-004	5.85E-004	1.23E-003	2.20E-003	1.47E-002	3.56E-003
8192	Default Cray	1.86E-004	2.57E-003	8.41E-003	2.60E-002	5.85E-002	1.14E-001
	MPICH Bi.	1.74E-004	2.57E-003	8.41E-003	2.61E-002	5.85E-002	1.14E-001
	MPICH Bi. Hier.	1.72E-004	1.01E-002	2.19E-002	4.58E-002	9.15E-002	2.01E-001
	MPICH Bi. Hier. Opt.	1.60E-004	1.39E-003	3.89E-003	1.27E-002	3.51E-002	4.86E-002
	Dynamic Leader	3.02E-004	9.19E-004	1.57E-003	2.54E-003	1.50E-002	3.65E-003
131072	Default Cray	1.52E-003	3.61E-003	9.29E-003	2.68E-002	5.92E-002	1.15E-001
	MPICH Bi.	1.51E-003	3.58E-003	9.32E-003	2.68E-002	5.92E-002	1.15E-001
	MPICH Bi. Hier.	1.36E-003	1.07E-002	2.25E-002	4.64E-002	9.21E-002	2.02E-001
	MPICH Bi. Hier. Opt.	1.43E-003	2.13E-003	4.55E-003	1.32E-002	3.57E-002	4.89E-002
	Dynamic Leader	1.67E-003	1.95E-003	2.55E-003	3.28E-003	1.59E-002	3.96E-003

Table B.2.: Broadcast latency 1024 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	2.93E-005	1.57E-003	5.44E-003	1.66E-002	4.49E-002	8.67E-002
	MPICH Bi.	2.89E-005	1.57E-003	5.44E-003	1.66E-002	4.49E-002	8.67E-002
	MPICH Bi. Hier.	5.18E-005	9.55E-003	2.12E-002	4.53E-002	9.11E-002	1.97E-001
	MPICH Bi. Hier. Opt.	3.86E-005	1.18E-003	3.78E-003	1.04E-002	3.29E-002	5.43E-002
	Dynamic Leader	1.56E-004	6.31E-004	1.26E-003	2.23E-003	1.48E-002	4.03E-003
512	Default Cray	5.47E-005	1.58E-003	5.45E-003	1.66E-002	4.49E-002	8.67E-002
	MPICH Bi.	5.17E-005	1.58E-003	5.45E-003	1.66E-002	4.49E-002	8.67E-002
	MPICH Bi. Hier.	7.57E-005	9.58E-003	2.13E-002	4.53E-002	9.11E-002	1.98E-001
	MPICH Bi. Hier. Opt.	5.78E-005	1.19E-003	3.80E-003	1.04E-002	3.29E-002	5.43E-002
	Dynamic Leader	1.77E-004	6.44E-004	1.28E-003	2.25E-003	1.48E-002	4.09E-003
8192	Default Cray	1.72E-004	2.53E-003	9.16E-003	2.57E-002	5.94E-002	1.20E-001
	MPICH Bi.	1.69E-004	2.53E-003	9.16E-003	2.57E-002	5.94E-002	1.20E-001
	MPICH Bi. Hier.	1.51E-004	1.04E-002	2.25E-002	4.68E-002	9.44E-002	2.08E-001
	MPICH Bi. Hier. Opt.	1.42E-004	1.41E-003	4.49E-003	1.27E-002	3.72E-002	6.52E-002
	Dynamic Leader	3.44E-004	1.07E-003	1.73E-003	2.69E-003	1.52E-002	4.23E-003
131072	Default Cray	1.61E-003	3.62E-003	1.01E-002	2.65E-002	6.02E-002	1.21E-001
	MPICH Bi.	1.59E-003	3.60E-003	1.01E-002	2.65E-002	6.02E-002	1.21E-001
	MPICH Bi. Hier.	1.51E-003	1.11E-002	2.32E-002	4.75E-002	9.50E-002	2.08E-001
	MPICH Bi. Hier. Opt.	1.59E-003	2.27E-003	5.23E-003	1.33E-002	3.78E-002	6.56E-002
	Dynamic Leader	2.00E-003	2.33E-003	2.94E-003	3.60E-003	1.61E-002	4.57E-003

Table B.3.: Broadcast latency 2048 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	3.51E-005	1.78E-003	5.75E-003	1.75E-002	4.81E-002	9.32E-002
	MPICH Bi.	3.73E-005	1.78E-003	5.75E-003	1.75E-002	4.81E-002	9.32E-002
	MPICH Bi. Hier.	1.32E-004	1.01E-002	2.19E-002	4.70E-002	9.10E-002	2.00E-001
	MPICH Bi. Hier. Opt.	6.33E-005	1.40E-003	4.15E-003	1.17E-002	3.85E-002	6.83E-002
	Dynamic Leader	2.67E-004	7.79E-004	1.33E-003	2.30E-003	1.48E-002	5.50E-003
512	Default Cray	7.02E-005	1.79E-003	5.76E-003	1.75E-002	4.82E-002	9.32E-002
	MPICH Bi.	6.87E-005	1.79E-003	5.76E-003	1.75E-002	4.82E-002	9.32E-002
	MPICH Bi. Hier.	8.85E-005	1.01E-002	2.19E-002	4.70E-002	9.10E-002	2.00E-001
	MPICH Bi. Hier. Opt.	1.21E-004	1.41E-003	4.16E-003	1.17E-002	3.85E-002	6.83E-002
	Dynamic Leader	2.50E-004	7.98E-004	1.35E-003	2.31E-003	1.48E-002	5.50E-003
8192	Default Cray	2.15E-004	2.91E-003	9.63E-003	2.67E-002	6.20E-002	1.23E-001
	MPICH Bi.	2.67E-004	2.88E-003	9.62E-003	2.67E-002	6.20E-002	1.23E-001
	MPICH Bi. Hier.	2.10E-004	1.09E-002	2.30E-002	4.85E-002	9.40E-002	2.09E-001
	MPICH Bi. Hier. Opt.	1.71E-004	1.64E-003	4.95E-003	1.42E-002	4.23E-002	7.70E-002
	Dynamic Leader	4.50E-004	1.38E-003	1.93E-003	2.88E-003	1.54E-002	5.74E-003
131072	Default Cray	1.85E-003	4.07E-003	1.06E-002	2.76E-002	6.29E-002	1.24E-001
	MPICH Bi.	1.80E-003	4.04E-003	1.06E-002	2.75E-002	6.29E-002	1.24E-001
	MPICH Bi. Hier.	1.66E-003	1.16E-002	2.37E-002	4.92E-002	9.47E-002	2.10E-001
	MPICH Bi. Hier. Opt.	1.74E-003	2.55E-003	5.73E-003	1.48E-002	4.29E-002	7.75E-002
	Dynamic Leader	2.31E-003	2.96E-003	3.34E-003	3.98E-003	1.64E-002	6.16E-003

Table B.4.: Broadcast latency 4096 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	4.86E-005	1.88E-003	6.06E-003	1.87E-002	4.84E-002	9.29E-002
	MPICH Bi.	4.56E-005	1.88E-003	6.07E-003	1.87E-002	4.84E-002	9.29E-002
	MPICH Bi. Hier.	7.34E-005	1.05E-002	2.23E-002	4.72E-002	9.44E-002	2.05E-001
	MPICH Bi. Hier. Opt.	8.36E-005	1.54E-003	4.54E-003	1.34E-002	3.89E-002	6.91E-002
	Dynamic Leader	3.30E-004	1.23E-003	1.52E-003	2.47E-003	1.49E-002	6.57E-003
512	Default Cray	7.88E-005	1.90E-003	6.08E-003	1.87E-002	4.84E-002	9.29E-002
	MPICH Bi.	7.70E-005	1.89E-003	6.08E-003	1.87E-002	4.84E-002	9.29E-002
	MPICH Bi. Hier.	1.17E-004	1.05E-002	2.23E-002	4.72E-002	9.44E-002	2.05E-001
	MPICH Bi. Hier. Opt.	1.14E-004	1.56E-003	4.56E-003	1.34E-002	3.89E-002	6.91E-002
	Dynamic Leader	3.60E-004	1.24E-003	1.55E-003	2.50E-003	1.50E-002	6.62E-003
8192	Default Cray	2.48E-004	3.04E-003	9.95E-003	2.77E-002	6.20E-002	1.21E-001
	MPICH Bi.	2.48E-004	3.04E-003	9.95E-003	2.77E-002	6.20E-002	1.21E-001
	MPICH Bi. Hier.	2.43E-004	1.12E-002	2.33E-002	4.86E-002	9.74E-002	2.14E-001
	MPICH Bi. Hier. Opt.	2.42E-004	1.81E-003	5.36E-003	1.58E-002	4.27E-002	7.71E-002
	Dynamic Leader	6.12E-004	1.85E-003	2.20E-003	3.22E-003	1.58E-002	6.83E-003
131072	Default Cray	2.42E-003	4.52E-003	1.11E-002	2.86E-002	6.29E-002	1.22E-001
	MPICH Bi.	2.36E-003	4.44E-003	1.11E-002	2.86E-002	6.30E-002	1.22E-001
	MPICH Bi. Hier.	2.53E-003	1.23E-002	2.43E-002	4.95E-002	9.84E-002	2.15E-001
	MPICH Bi. Hier. Opt.	2.41E-003	3.20E-003	6.61E-003	1.65E-002	4.35E-002	7.77E-002
	Dynamic Leader	3.24E-003	4.00E-003	4.28E-003	4.97E-003	1.73E-002	7.45E-003

Table B.5.: Broadcast latency 8192 cores

		0	10000	20000	40000	80000	160000
32	Default Cray	4.55E-005	1.99E-003	6.65E-003	2.01E-002	5.11E-002	9.95E-002
	MPICH Bi.	4.46E-005	1.99E-003	6.65E-003	2.01E-002	5.11E-002	9.95E-002
	Dynamic Leader	5.45E-004	2.29E-003	2.30E-003	3.21E-003	1.53E-002	7.71E-003
	Dynamic Leader 4	4.81E-004	2.03E-003	2.12E-003	3.07E-003	1.51E-002	7.60E-003
	Dynamic Leader 8	4.27E-004	1.78E-003	1.98E-003	2.93E-003	1.51E-002	7.40E-003
	Dynamic Leader 16	3.44E-004	1.43E-003	1.76E-003	2.73E-003	1.49E-002	5.73E-003
	Dynamic Leader OS	1.47E-003	6.11E-003	1.12E-002	2.65E-002	5.84E-002	1.07E-001
	MPICH Bi. Hier.	6.57E-005	1.07E-002	2.28E-002	4.73E-002	9.51E-002	2.09E-001
	MPICH Bi. Hier. Opt.	1.46E-004	1.69E-003	5.27E-003	1.55E-002	4.31E-002	7.97E-002
	Pat.	5.42E-002	6.41E-002	6.15E-002	5.30E-002	5.98E-002	1.90E-002
	Pat. Hier.	5.73E-004	9.18E-003	1.98E-002	4.41E-002	8.76E-002	1.76E-001
	Pat. DL	3.06E-003	4.10E-003	4.48E-003	4.65E-003	1.67E-002	3.60E-003
	Pat. OL	5.64E-004	1.80E-003	3.26E-003	7.49E-003	2.55E-002	3.53E-002
	Pat. DLN	2.09E-003	3.07E-003	3.60E-003	3.92E-003	1.62E-002	3.69E-003

Table B.6.: Broadcast latency 8192 cores

		0	10000	20000	40000	80000	160000
512	Default Cray	8.76E-005	2.01E-003	6.67E-003	2.01E-002	5.12E-002	9.95E-002
	MPICH Bi.	8.37E-005	2.01E-003	6.67E-003	2.01E-002	5.12E-002	9.95E-002
	Dynamic Leader	5.73E-004	2.34E-003	2.33E-003	3.24E-003	1.53E-002	7.69E-003
	Dynamic Leader 4	4.92E-004	2.07E-003	2.15E-003	3.10E-003	1.52E-002	7.63E-003
	Dynamic Leader 8	4.43E-004	1.81E-003	2.00E-003	2.96E-003	1.51E-002	7.44E-003
	Dynamic Leader 16	3.80E-004	1.46E-003	1.79E-003	2.75E-003	1.49E-002	5.72E-003
	Dynamic Leader OS	3.71E-003	7.99E-003	1.27E-002	2.77E-002	5.96E-002	1.08E-001
	MPICH Bi. Hier.	1.11E-004	1.08E-002	2.29E-002	4.73E-002	9.52E-002	2.09E-001
	MPICH Bi. Hier. Opt.	1.51E-004	1.72E-003	5.29E-003	1.55E-002	4.31E-002	7.97E-002
	Pat.	5.44E-002	6.42E-002	6.10E-002	5.17E-002	5.97E-002	1.93E-002
	Pat. Hier.	6.20E-004	9.21E-003	1.99E-002	4.41E-002	8.76E-002	1.76E-001
	Pat. DL	3.12E-003	4.18E-003	4.51E-003	4.66E-003	1.68E-002	3.62E-003
	Pat. OL	5.96E-004	1.82E-003	3.26E-003	7.50E-003	2.55E-002	3.53E-002
	Pat. DLN	2.11E-003	3.09E-003	3.58E-003	4.01E-003	1.61E-002	3.70E-003

Table B.7.: Broadcast latency 8192 cores

		0	10000	20000	40000	80000	160000
8192	Default Cray	2.50E-004	3.18E-003	1.06E-002	2.87E-002	6.42E-002	1.26E-001
	MPICH Bi.	2.45E-004	3.18E-003	1.06E-002	2.87E-002	6.42E-002	1.26E-001
	Dynamic Leader	8.56E-004	2.93E-003	2.93E-003	3.96E-003	1.61E-002	7.98E-003
	Dynamic Leader 4	8.33E-004	2.76E-003	2.95E-003	4.03E-003	1.62E-002	7.91E-003
	Dynamic Leader 8	9.21E-004	2.81E-003	2.98E-003	4.03E-003	1.63E-002	7.74E-003
	Dynamic Leader 16	1.01E-003	2.96E-003	3.31E-003	4.43E-003	1.64E-002	6.20E-003
	Dynamic Leader OS	5.13E-003	9.35E-003	1.39E-002	2.87E-002	6.06E-002	1.09E-001
	MPICH Bi. Hier.	2.27E-004	1.14E-002	2.37E-002	4.86E-002	9.79E-002	2.17E-001
	MPICH Bi. Hier. Opt.	2.76E-004	1.95E-003	6.07E-003	1.77E-002	4.66E-002	8.75E-002
	Pat.	7.11E-002	8.14E-002	7.83E-002	6.76E-002	7.29E-002	2.38E-002
	Pat. Hier.	1.14E-003	9.66E-003	2.03E-002	4.45E-002	8.80E-002	1.76E-001
	Pat. DL	3.91E-003	6.43E-003	6.72E-003	7.00E-003	1.85E-002	4.38E-003
	Pat. OL	1.08E-003	2.18E-003	3.56E-003	7.74E-003	2.57E-002	3.53E-002
	Pat. DLN	2.72E-003	4.22E-003	4.46E-003	4.95E-003	1.70E-002	4.06E-003

Table B.8.: Broadcast latency 8192 cores

		0	10000	20000	40000	80000	160000
131072	Default Cray	2.21E-003	4.60E-003	1.18E-002	2.97E-002	6.52E-002	1.27E-001
	MPICH Bi.	2.18E-003	4.56E-003	1.17E-002	2.96E-002	6.52E-002	1.27E-001
	Dynamic Leader	3.26E-003	4.64E-003	4.57E-003	5.28E-003	1.74E-002	8.44E-003
	Dynamic Leader 4	3.44E-003	4.85E-003	4.90E-003	5.60E-003	1.77E-002	8.50E-003
	Dynamic Leader 8	4.06E-003	5.43E-003	5.39E-003	6.05E-003	1.80E-002	8.47E-003
	Dynamic Leader 16	5.07E-003	6.51E-003	6.67E-003	7.18E-003	1.88E-002	7.28E-003
	Dynamic Leader OS	2.72E-002	3.07E-002	3.26E-002	4.34E-002	7.42E-002	1.20E-001
	MPICH Bi. Hier.	2.04E-003	1.23E-002	2.45E-002	4.94E-002	9.88E-002	2.18E-001
	MPICH Bi. Hier. Opt.	2.08E-003	3.07E-003	6.98E-003	1.84E-002	4.74E-002	8.80E-002
	Pat.	1.61E-001	1.73E-001	1.70E-001	1.50E-001	1.48E-001	5.49E-002
	Pat. Hier.	5.85E-003	1.30E-002	2.33E-002	4.73E-002	9.12E-002	1.79E-001
	Pat. DL	9.00E-003	1.16E-002	1.14E-002	1.06E-002	2.17E-002	5.98E-003
	Pat. OL	5.89E-003	6.69E-003	7.29E-003	1.02E-002	2.80E-002	3.60E-002
	Pat. DLN	7.63E-003	9.07E-003	9.05E-003	8.54E-003	1.99E-002	5.45E-003

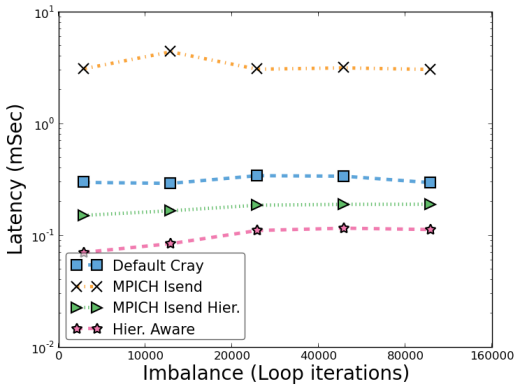
Table B.9.: Broadcast latency 16384 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	8.96E-005	2.08E-003	7.11E-003	2.08E-002	5.10E-002	1.03E-001
	MPICH Bi.	9.01E-005	2.09E-003	7.12E-003	2.08E-002	5.09E-002	1.03E-001
	MPICH Bi. Hier.	7.18E-005	1.10E-002	2.32E-002	4.79E-002	9.63E-002	2.14E-001
	MPICH Bi. Hier. Opt.	2.91E-004	1.85E-003	5.88E-003	1.65E-002	4.32E-002	8.64E-002
	Dynamic Leader	1.08E-003	4.85E-003	4.54E-003	5.25E-003	1.61E-002	9.56E-003
512	Default Cray	1.33E-004	2.15E-003	7.17E-003	2.08E-002	5.09E-002	1.03E-001
	MPICH Bi.	1.33E-004	2.11E-003	7.16E-003	2.08E-002	5.09E-002	1.03E-001
	MPICH Bi. Hier.	1.64E-004	1.10E-002	2.33E-002	4.79E-002	9.63E-002	2.14E-001
	MPICH Bi. Hier. Opt.	3.25E-004	1.89E-003	5.92E-003	1.65E-002	4.33E-002	8.64E-002
	Dynamic Leader	1.11E-003	4.99E-003	4.60E-003	5.29E-003	1.62E-002	9.59E-003
8192	Default Cray	5.25E-004	3.41E-003	1.12E-002	2.93E-002	6.35E-002	1.29E-001
	MPICH Bi.	4.95E-004	3.41E-003	1.12E-002	2.94E-002	6.35E-002	1.29E-001
	MPICH Bi. Hier.	3.62E-004	1.17E-002	2.41E-002	4.91E-002	9.91E-002	2.21E-001
	MPICH Bi. Hier. Opt.	3.75E-004	2.18E-003	6.71E-003	1.87E-002	4.67E-002	9.27E-002
	Dynamic Leader	1.49E-003	5.57E-003	5.31E-003	6.05E-003	1.72E-002	9.87E-003
131072	Default Cray	2.78E-003	5.02E-003	1.25E-002	3.04E-002	6.46E-002	1.30E-001
	MPICH Bi.	2.73E-003	5.13E-003	1.24E-002	3.04E-002	6.46E-002	1.30E-001
	MPICH Bi. Hier.	2.83E-003	1.28E-002	2.51E-002	5.03E-002	1.00E-001	2.23E-001
	MPICH Bi. Hier. Opt.	2.93E-003	3.74E-003	8.08E-003	1.95E-002	4.76E-002	9.34E-002
	Dynamic Leader	4.45E-003	7.62E-003	7.16E-003	7.48E-003	1.87E-002	1.06E-002

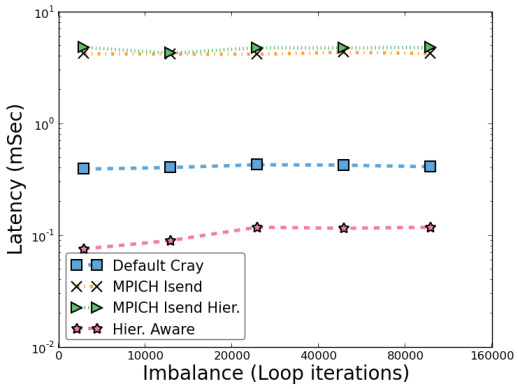
Table B.10.: Broadcast latency 32768 cores, varying buffer sizes

		0	10000	20000	40000	80000	160000
32	Default Cray	1.53E-004	2.12E-003	7.10E-003	2.20E-002	5.17E-002	1.09E-001
	MPICH Bi.	8.58E-005	2.09E-003	7.10E-003	2.20E-002	5.16E-002	1.09E-001
	MPICH Bi. Hier.	1.65E-004	1.12E-002	2.33E-002	4.81E-002	1.01E-001	2.19E-001
	MPICH Bi. Hier. Opt.	5.87E-004	2.02E-003	6.01E-003	1.83E-002	4.46E-002	9.53E-002
	Dynamic Leader	2.76E-003	1.38E-002	1.34E-002	1.39E-002	1.97E-002	1.24E-002
512	Default Cray	3.26E-004	2.15E-003	7.13E-003	2.21E-002	5.17E-002	1.09E-001
	MPICH Bi.	2.42E-004	2.14E-003	7.15E-003	2.21E-002	5.17E-002	1.09E-001
	MPICH Bi. Hier.	2.41E-004	1.13E-002	2.34E-002	4.81E-002	1.01E-001	2.19E-001
	MPICH Bi. Hier. Opt.	6.50E-004	2.03E-003	6.06E-003	1.84E-002	4.47E-002	9.54E-002
	Dynamic Leader	2.74E-003	1.39E-002	1.36E-002	1.41E-002	1.98E-002	1.24E-002
8192	Default Cray	5.89E-004	3.64E-003	1.11E-002	3.03E-002	6.41E-002	1.34E-001
	MPICH Bi.	5.37E-004	3.46E-003	1.11E-002	3.04E-002	6.41E-002	1.34E-001
	MPICH Bi. Hier.	4.39E-004	1.19E-002	2.41E-002	4.92E-002	1.03E-001	2.26E-001
	MPICH Bi. Hier. Opt.	7.11E-004	2.35E-003	6.83E-003	2.03E-002	4.79E-002	1.01E-001
	Dynamic Leader	3.16E-003	1.47E-002	1.41E-002	1.48E-002	2.11E-002	1.28E-002
131072	Default Cray	3.74E-003	5.40E-003	1.28E-002	3.17E-002	6.54E-002	1.35E-001
	MPICH Bi.	3.64E-003	5.53E-003	1.27E-002	3.17E-002	6.54E-002	1.35E-001
	MPICH Bi. Hier.	3.61E-003	1.34E-002	2.57E-002	5.07E-002	1.05E-001	2.28E-001
	MPICH Bi. Hier. Opt.	3.87E-003	4.65E-003	8.34E-003	2.14E-002	4.90E-002	1.02E-001
	Dynamic Leader	7.75E-003	1.71E-002	1.66E-002	1.68E-002	2.28E-002	1.36E-002

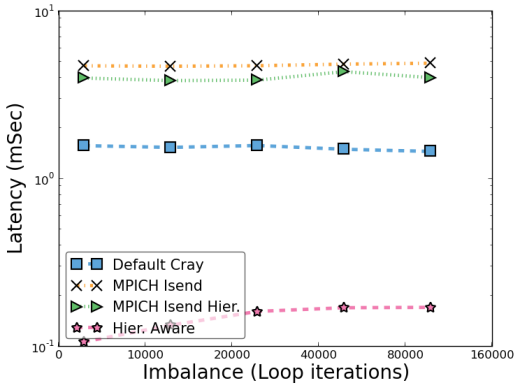
C. COMPLETE ALLTOALL DATA



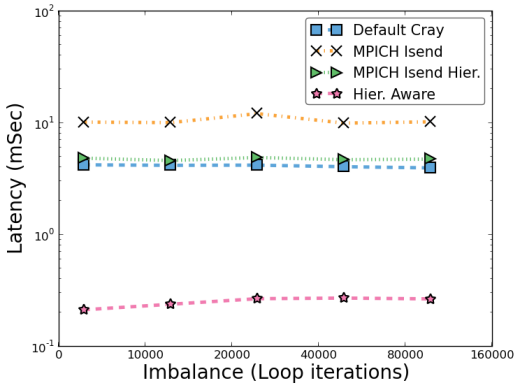
(a) 512



(b) 2048

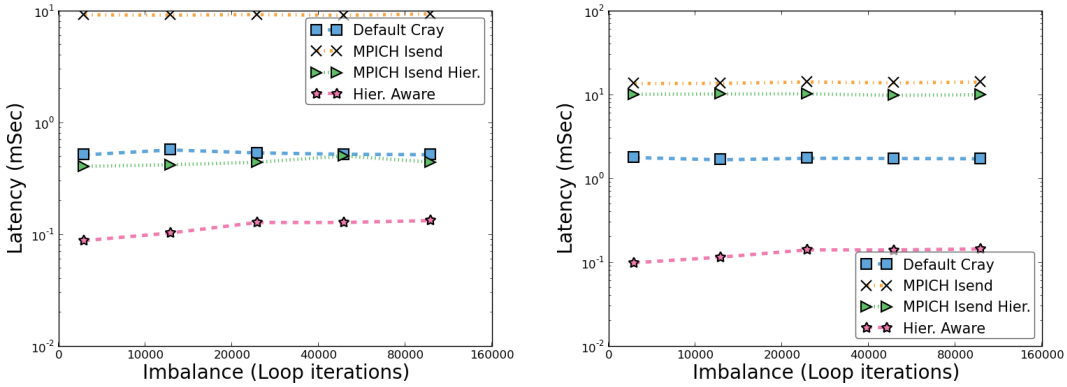


(c) 8192



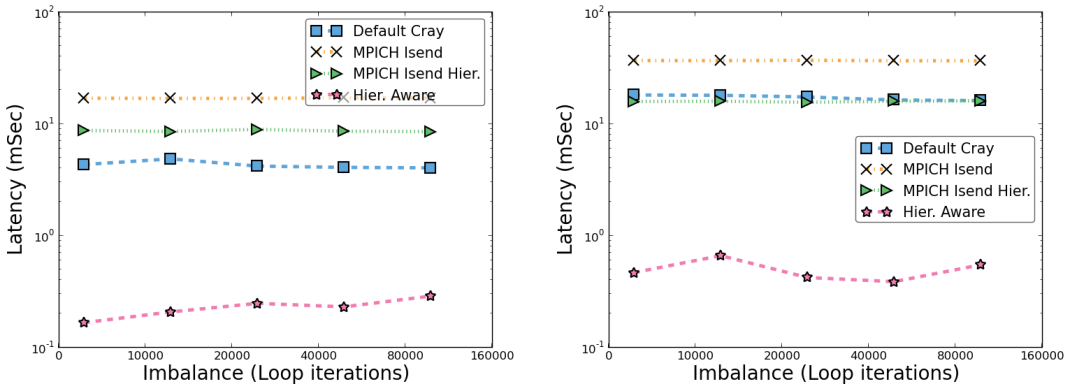
(d) 32768

Fig. C.1.: Alltoall latency 512 cores, varying buffer sizes



(a) 1024

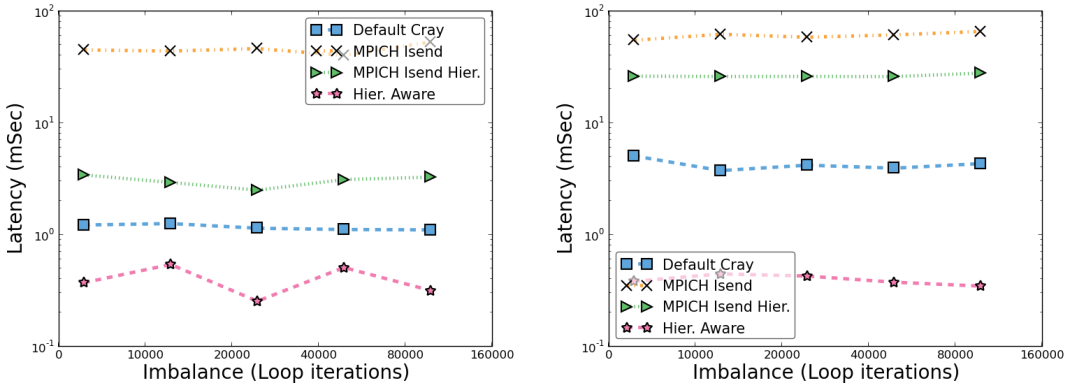
(b) 4096



(c) 16384

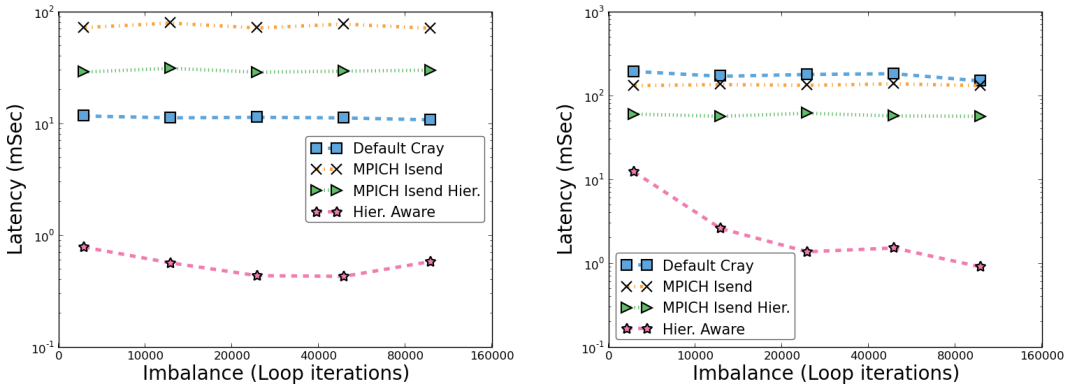
(d) 65536

Fig. C.2.: Alltoall latency 1024 cores, varying buffer sizes



(a) 1024

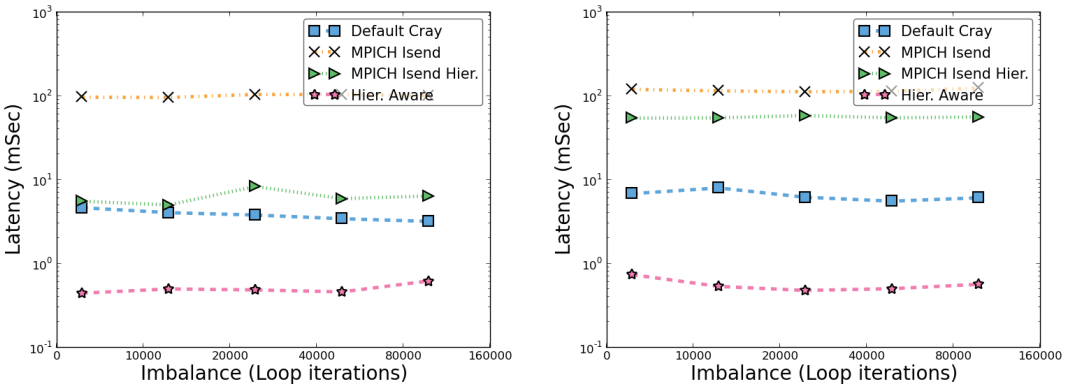
(b) 8192



(c) 32768

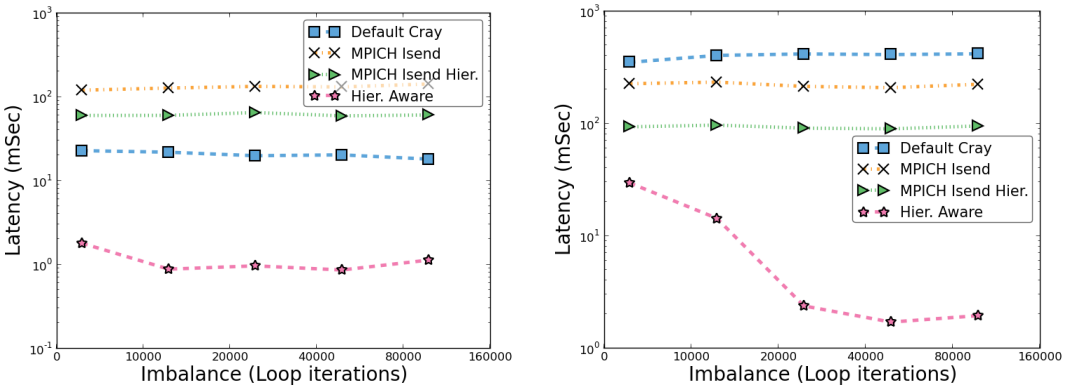
(d) 131072

Fig. C.3.: Alltoall latency 2048 cores, varying buffer sizes



(a) 4096

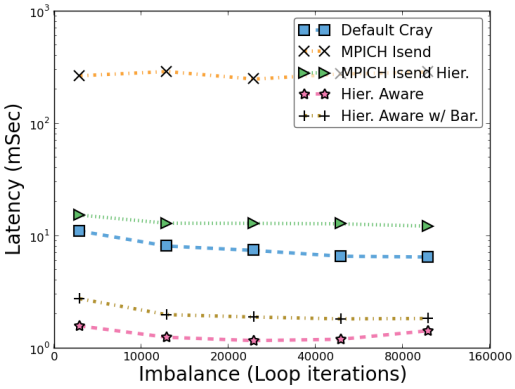
(b) 16384



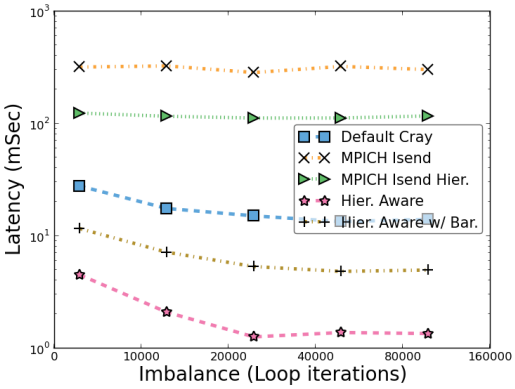
(c) 65536

(d) 262144

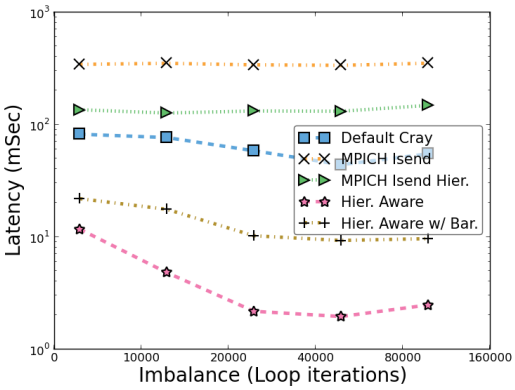
Fig. C.4.: Alltoall latency 4096 cores, varying buffer sizes



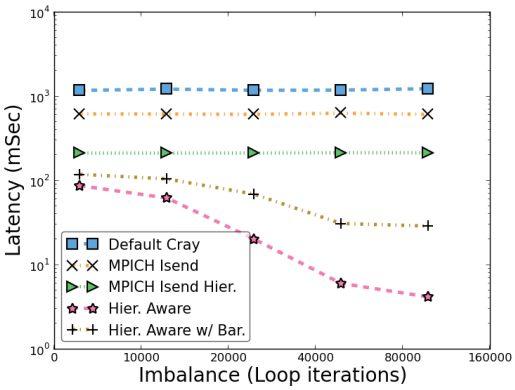
(a) 8192



(b) 32768

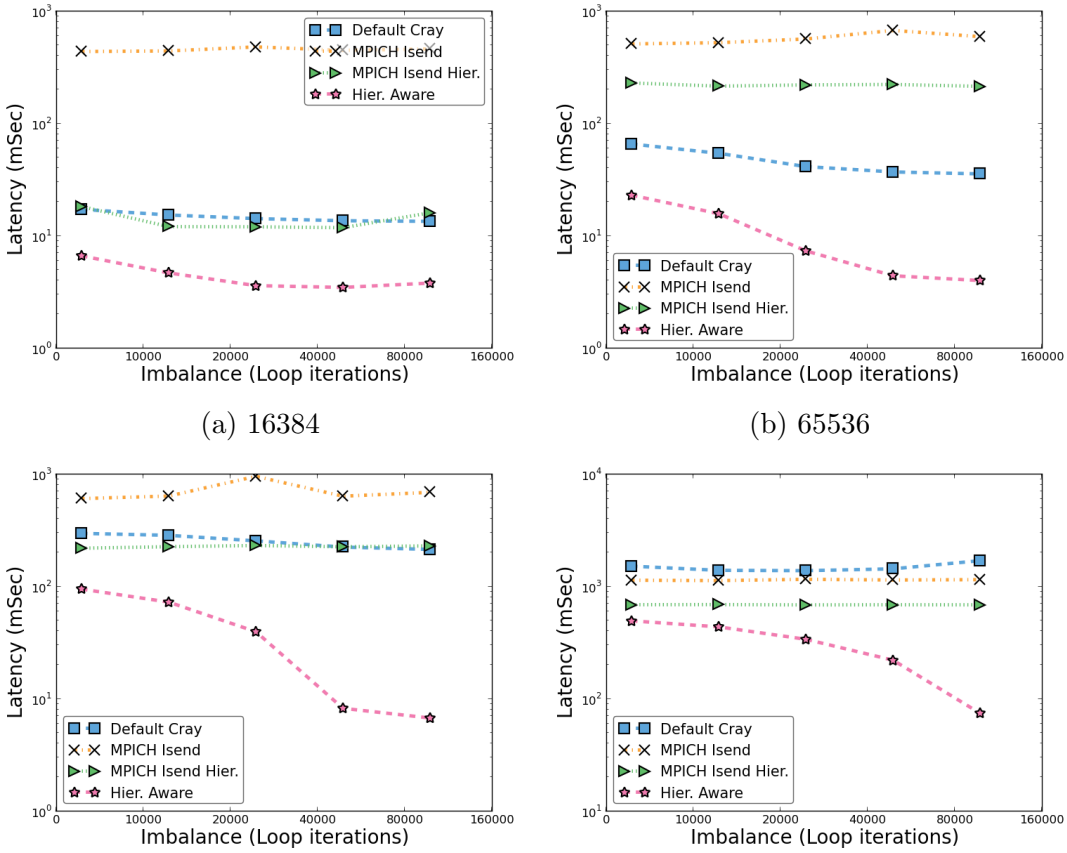


(c) 131072



(d) 524288

Fig. C.5.: Alltoall latency 8192 cores, varying buffer sizes



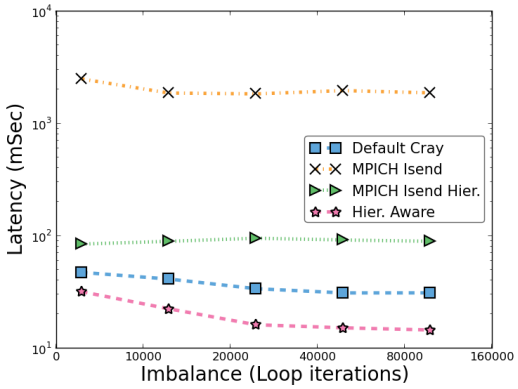
(a) 16384

(b) 65536

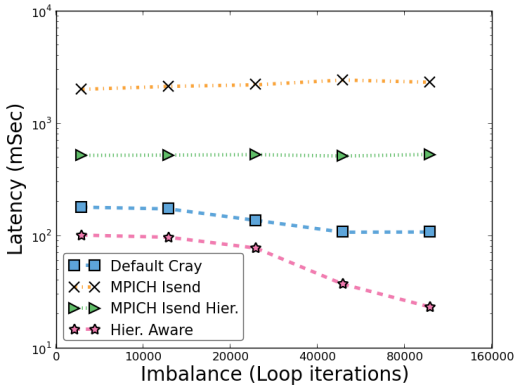
(c) 262144

(d) 1048576

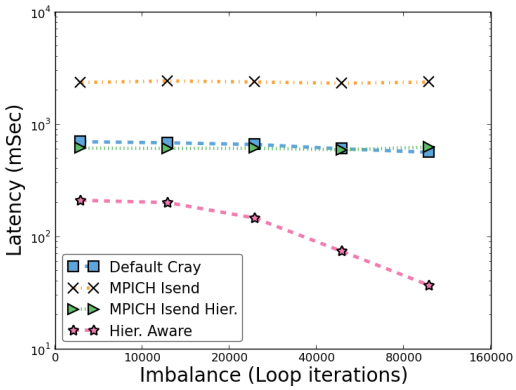
Fig. C.6.: Alltoall latency 16384 cores, varying buffer sizes



(a) 32768

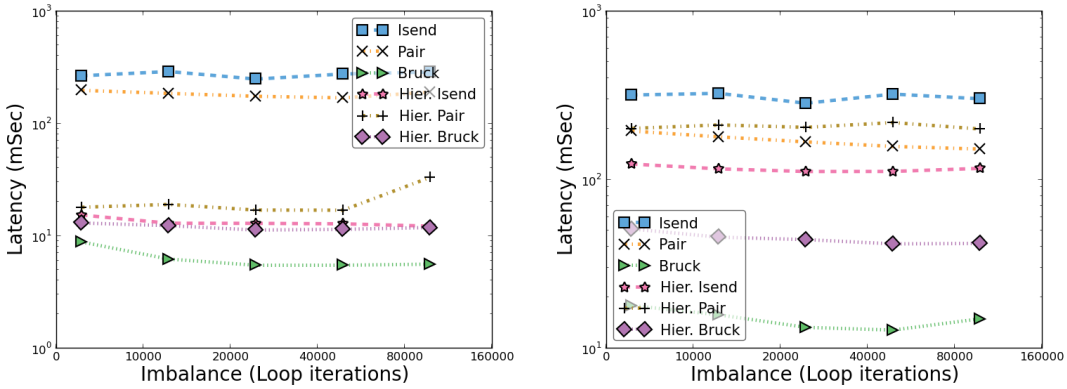


(b) 131072



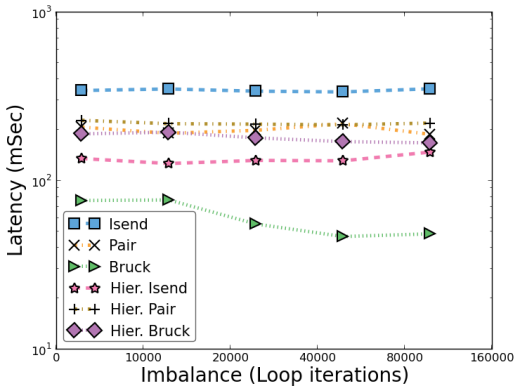
(c) 524288

Fig. C.7.: Alltoall latency 32768 cores, varying buffer sizes

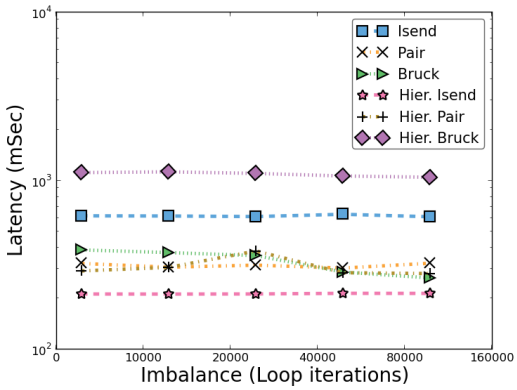


(a) 8192

(b) 32768

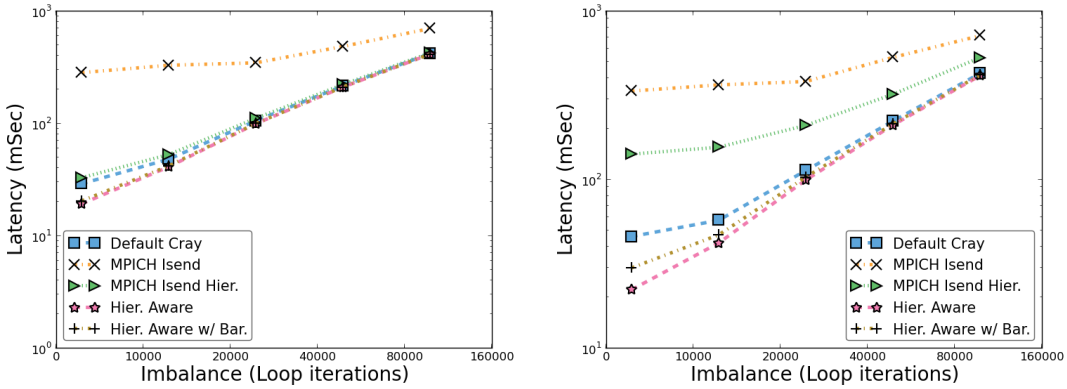


(c) 131072



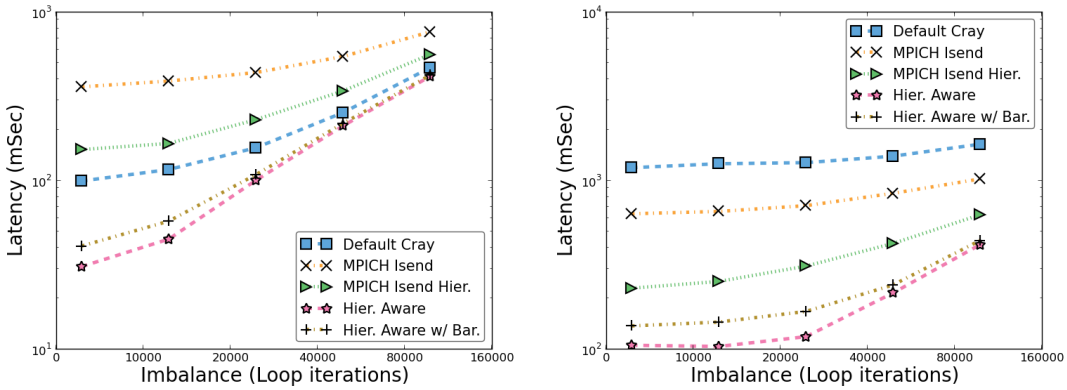
(d) 524288

Fig. C.8.: Alltoall latency for MPICH 8192 cores, varying buffer sizes



(a) 8192

(b) 32768



(c) 131072

(d) 524288

Fig. C.9.: Alltoall latency include sync. time 8192 cores, varying buffer sizes

Table C.1.: Alltoall latency 512, varying buffer sizes

		0	10000	20000	40000	80000	160000
512	Default Cray	3.34E-004	2.99E-004	2.93E-004	3.44E-004	3.40E-004	2.97E-004
	MPICH Isend	2.99E-003	3.11E-003	4.40E-003	3.08E-003	3.16E-003	3.06E-003
	MPICH Isend Hier.	2.17E-004	1.52E-004	1.67E-004	1.87E-004	1.90E-004	1.91E-004
	Dynamic Leader	8.12E-005	7.07E-005	8.47E-005	1.11E-004	1.16E-004	1.13E-004
2048	Default Cray	1.02E-003	3.94E-004	4.06E-004	4.30E-004	4.27E-004	4.13E-004
	MPICH Bi.	4.16E-003	4.23E-003	4.20E-003	4.19E-003	4.36E-003	4.25E-003
	MPICH Bi. Hier.	4.31E-003	4.82E-003	4.29E-003	4.79E-003	4.76E-003	4.82E-003
	Dynamic Leader	3.59E-004	7.60E-005	9.04E-005	1.19E-004	1.16E-004	1.19E-004
8192	Default Cray	3.02E-003	1.57E-003	1.54E-003	1.58E-003	1.50E-003	1.45E-003
	MPICH Bi.	4.61E-003	4.71E-003	4.68E-003	4.72E-003	4.82E-003	4.88E-003
	MPICH Bi. Hier.	3.91E-003	3.98E-003	3.85E-003	3.87E-003	4.35E-003	4.01E-003
	Dynamic Leader	8.37E-004	1.06E-004	1.34E-004	1.62E-004	1.70E-004	1.71E-004
32768	Default Cray	6.87E-003	4.21E-003	4.16E-003	4.18E-003	4.05E-003	3.95E-003
	MPICH Bi.	1.02E-002	1.01E-002	1.00E-002	1.21E-002	9.93E-003	1.02E-002
	MPICH Bi. Hier.	4.76E-003	4.83E-003	4.60E-003	4.89E-003	4.68E-003	4.73E-003
	Dynamic Leader	2.76E-003	2.12E-004	2.38E-004	2.67E-004	2.70E-004	2.66E-004

Table C.2.: Alltoall latency 1024, varying buffer sizes

		0	10000	20000	40000	80000	160000
1024	Default Cray	5.97E-004	5.17E-004	5.72E-004	5.38E-004	5.23E-004	5.18E-004
	MPICH Isend	9.01E-003	9.25E-003	9.19E-003	9.30E-003	9.16E-003	9.41E-003
	MPICH Isend Hier.	5.94E-004	4.07E-004	4.21E-004	4.43E-004	5.05E-004	4.46E-004
	Dynamic Leader	2.34E-004	8.83E-005	1.03E-004	1.28E-004	1.28E-004	1.33E-004
4096	Default Cray	3.36E-003	1.79E-003	1.68E-003	1.75E-003	1.74E-003	1.73E-003
	MPICH Bi.	1.35E-002	1.37E-002	1.37E-002	1.42E-002	1.39E-002	1.42E-002
	MPICH Bi. Hier.	8.84E-003	1.02E-002	1.02E-002	1.03E-002	9.86E-003	1.00E-002
	Dynamic Leader	9.08E-004	9.93E-005	1.16E-004	1.42E-004	1.41E-004	1.45E-004
16384	Default Cray	6.40E-003	4.32E-003	4.85E-003	4.18E-003	4.08E-003	4.03E-003
	MPICH Bi.	1.73E-002	1.69E-002	1.69E-002	1.69E-002	1.70E-002	1.69E-002
	MPICH Bi. Hier.	7.67E-003	8.72E-003	8.53E-003	8.91E-003	8.58E-003	8.51E-003
	Dynamic Leader	2.79E-003	1.66E-004	2.07E-004	2.47E-004	2.30E-004	2.87E-004
65536	Default Cray	2.12E-002	1.81E-002	1.80E-002	1.74E-002	1.63E-002	1.61E-002
	MPICH Bi.	3.64E-002	3.68E-002	3.67E-002	3.70E-002	3.66E-002	3.66E-002
	MPICH Bi. Hier.	1.69E-002	1.58E-002	1.59E-002	1.56E-002	1.59E-002	1.61E-002
	Dynamic Leader	1.07E-002	4.66E-004	6.61E-004	4.22E-004	3.87E-004	5.47E-004

Table C.3.: Alltoall latency 2048, varying buffer sizes

		0	10000	20000	40000	80000	160000
2048	Default Cray	1.66E-003	1.18E-002	1.86E-002	3.37E-002	7.95E-002	1.83E-001
	MPICH Isend	4.48E-002	5.55E-002	6.13E-002	7.87E-002	1.19E-001	2.35E-001
	MPICH Isend Hier.	3.91E-003	1.40E-002	2.03E-002	3.51E-002	8.15E-002	1.85E-001
	Dynamic Leader	1.31E-003	1.10E-002	1.79E-002	3.31E-002	7.89E-002	1.83E-001
8192	Default Cray	9.20E-003	1.56E-002	2.12E-002	3.68E-002	8.23E-002	1.87E-001
	MPICH Bi.	5.92E-002	6.53E-002	7.93E-002	9.09E-002	1.40E-001	2.48E-001
	MPICH Bi. Hier.	2.76E-002	3.67E-002	4.34E-002	5.86E-002	1.04E-001	2.10E-001
	Dynamic Leader	3.66E-003	1.10E-002	1.79E-002	3.31E-002	7.98E-002	1.83E-001
32768	Default Cray	1.68E-002	2.23E-002	2.87E-002	4.40E-002	8.97E-002	1.93E-001
	MPICH Bi.	7.46E-002	8.32E-002	9.70E-002	1.05E-001	1.56E-001	2.54E-001
	MPICH Bi. Hier.	3.15E-002	3.96E-002	4.89E-002	6.16E-002	1.08E-001	2.12E-001
	Dynamic Leader	7.98E-003	1.09E-002	1.80E-002	3.30E-002	7.88E-002	1.83E-001
131072	Default Cray	1.67E-001	2.06E-001	1.89E-001	2.13E-001	2.62E-001	3.33E-001
	MPICH Bi.	1.35E-001	1.43E-001	1.54E-001	1.66E-001	2.17E-001	3.15E-001
	MPICH Bi. Hier.	4.88E-002	7.17E-002	7.45E-002	9.46E-002	1.36E-001	2.39E-001
	Dynamic Leader	2.90E-002	2.29E-002	2.01E-002	3.40E-002	7.99E-002	1.83E-001

Table C.4.: Alltoall latency 4096, varying buffer sizes

		0	10000	20000	40000	80000	160000
4096	Default Cray	7.21E-003	4.63E-003	4.04E-003	3.79E-003	3.42E-003	3.18E-003
	MPICH Isend	9.69E-002	9.64E-002	9.57E-002	1.04E-001	1.04E-001	1.01E-001
	MPICH Isend Hier.	6.61E-003	5.53E-003	5.00E-003	8.30E-003	5.95E-003	6.38E-003
	Dynamic Leader	1.31E-003	4.43E-004	4.97E-004	4.84E-004	4.59E-004	6.19E-004
16384	Default Cray	1.04E-002	6.81E-003	7.97E-003	6.17E-003	5.54E-003	6.08E-003
	MPICH Bi.	1.10E-001	1.19E-001	1.15E-001	1.12E-001	1.13E-001	1.24E-001
	MPICH Bi. Hier.	5.70E-002	5.42E-002	5.47E-002	5.82E-002	5.47E-002	5.59E-002
	Dynamic Leader	2.90E-003	7.36E-004	5.34E-004	4.77E-004	5.01E-004	5.64E-004
65536	Default Cray	2.78E-002	2.28E-002	2.19E-002	1.99E-002	2.03E-002	1.81E-002
	MPICH Bi.	1.26E-001	1.20E-001	1.27E-001	1.33E-001	1.31E-001	1.42E-001
	MPICH Bi. Hier.	5.79E-002	6.00E-002	6.01E-002	6.50E-002	5.93E-002	6.08E-002
	Dynamic Leader	1.02E-002	1.79E-003	8.81E-004	9.64E-004	8.60E-004	1.13E-003
262144	Default Cray	3.41E-001	3.50E-001	4.03E-001	4.16E-001	4.10E-001	4.17E-001
	MPICH Bi.	2.32E-001	2.26E-001	2.33E-001	2.14E-001	2.08E-001	2.23E-001
	MPICH Bi. Hier.	9.47E-002	9.32E-002	9.67E-002	9.08E-002	8.95E-002	9.49E-002
	Dynamic Leader	4.41E-002	2.96E-002	1.43E-002	2.37E-003	1.70E-003	1.95E-003

Table C.5.: Alltoall latency 8192, varying buffer sizes

		0	10000	20000	40000	80000	160000
8192	Default Cray	1.38E-002	1.10E-002	8.10E-003	7.40E-003	6.58E-003	6.47E-003
	MPICH Bi.	2.90E-001	2.66E-001	2.90E-001	2.48E-001	2.76E-001	2.87E-001
	MPICH Pair	1.91E-001	1.97E-001	1.85E-001	1.75E-001	1.69E-001	1.90E-001
	MPICH Bruck	1.22E-002	8.80E-003	6.20E-003	5.46E-003	5.47E-003	5.59E-003
	MPICH Bi. Hier.	1.07E-002	1.53E-002	1.29E-002	1.29E-002	1.28E-002	1.22E-002
	MPICH Pair Hier.	1.99E-002	1.79E-002	1.90E-002	1.69E-002	1.69E-002	3.31E-002
	MPICH Bruck Hier.	1.38E-002	1.30E-002	1.23E-002	1.12E-002	1.14E-002	1.18E-002
	Dynamic Leader	3.90E-003	1.58E-003	1.25E-003	1.17E-003	1.20E-003	1.43E-003
32768	Default Cray	2.76E-002	2.77E-002	1.75E-002	1.50E-002	1.34E-002	1.40E-002
	MPICH Bi.	3.06E-001	3.18E-001	3.25E-001	2.84E-001	3.22E-001	3.02E-001
	MPICH Pair	1.91E-001	1.94E-001	1.79E-001	1.67E-001	1.57E-001	1.52E-001
	MPICH Bruck	2.11E-002	1.79E-002	1.58E-002	1.33E-002	1.28E-002	1.49E-002
	MPICH Bi. Hier.	1.14E-001	1.24E-001	1.16E-001	1.12E-001	1.12E-001	1.17E-001
	MPICH Pair Hier.	1.85E-001	2.01E-001	2.11E-001	2.04E-001	2.18E-001	2.00E-001
	MPICH Bruck Hier.	5.14E-002	5.10E-002	4.56E-002	4.41E-002	4.15E-002	4.18E-002
	Dynamic Leader	7.32E-003	4.49E-003	2.09E-003	1.26E-003	1.38E-003	1.35E-003

Table C.6.: Alltoall latency 8192, varying buffer sizes cont.

		0	10000	20000	40000	80000	160000
131072	Default Cray	8.71E-002	8.19E-002	7.64E-002	5.83E-002	4.38E-002	5.47E-002
	MPICH Bi.	3.33E-001	3.43E-001	3.50E-001	3.39E-001	3.36E-001	3.51E-001
	MPICH Pair	1.88E-001	2.08E-001	1.91E-001	1.99E-001	2.18E-001	1.88E-001
	MPICH Bruck	7.85E-002	7.62E-002	7.68E-002	5.53E-002	4.66E-002	4.83E-002
	MPICH Bi. Hier.	1.17E-001	1.35E-001	1.26E-001	1.32E-001	1.31E-001	1.48E-001
	MPICH Pair Hier.	1.88E-001	2.28E-001	2.18E-001	2.16E-001	2.14E-001	2.20E-001
	MPICH Bruck Hier.	1.92E-001	1.89E-001	1.94E-001	1.79E-001	1.70E-001	1.67E-001
	Dynamic Leader	2.45E-002	1.17E-002	4.80E-003	2.17E-003	1.95E-003	2.47E-003
524288	Default Cray	1.19E+000	1.18E+000	1.22E+000	1.18E+000	1.19E+000	1.23E+000
	MPICH Bi.	6.12E-001	6.18E-001	6.16E-001	6.12E-001	6.31E-001	6.10E-001
	MPICH Pair	3.20E-001	3.22E-001	3.06E-001	3.15E-001	3.03E-001	3.23E-001
	MPICH Bruck	4.06E-001	3.88E-001	3.74E-001	3.59E-001	2.87E-001	2.65E-001
	MPICH Bi. Hier.	2.19E-001	2.12E-001	2.12E-001	2.13E-001	2.14E-001	2.14E-001
	MPICH Pair Hier.	2.77E-001	2.91E-001	3.05E-001	3.82E-001	2.84E-001	2.81E-001
	MPICH Bruck Hier.	1.12E+000	1.12E+000	1.13E+000	1.10E+000	1.07E+000	1.05E+000
	Dynamic Leader	1.01E-001	8.71E-002	6.22E-002	2.03E-002	6.03E-003	4.17E-003

Table C.7.: Alltoall latency 16384, varying buffer sizes

		0	10000	20000	40000	80000	160000
16384	Default Cray	1.95E-002	1.72E-002	1.53E-002	1.42E-002	1.36E-002	1.34E-002
	MPICH Isend	4.32E-001	4.37E-001	4.42E-001	4.80E-001	4.48E-001	4.61E-001
	MPICH Isend Hier.	1.88E-002	1.82E-002	1.21E-002	1.20E-002	1.18E-002	1.60E-002
	Dynamic Leader	1.00E-002	6.62E-003	4.67E-003	3.60E-003	3.47E-003	3.79E-003
65536	Default Cray	6.02E-002	6.56E-002	5.42E-002	4.13E-002	3.70E-002	3.55E-002
	MPICH Bi.	5.34E-001	5.10E-001	5.23E-001	5.64E-001	6.73E-001	5.93E-001
	MPICH Bi. Hier.	1.82E-001	2.29E-001	2.15E-001	2.20E-001	2.22E-001	2.14E-001
	Dynamic Leader	3.51E-002	2.30E-002	1.57E-002	7.34E-003	4.41E-003	3.99E-003
262144	Default Cray	3.03E-001	2.97E-001	2.85E-001	2.54E-001	2.24E-001	2.14E-001
	MPICH Bi.	5.98E-001	6.05E-001	6.37E-001	9.54E-001	6.36E-001	6.88E-001
	MPICH Bi. Hier.	1.96E-001	2.19E-001	2.26E-001	2.31E-001	2.25E-001	2.30E-001
	Dynamic Leader	1.21E-001	9.46E-002	7.25E-002	3.95E-002	8.22E-003	6.74E-003
1048576	Default Cray	1.35E+000	1.52E+000	1.39E+000	1.38E+000	1.43E+000	1.70E+000
	MPICH Bi.	1.44E+000	1.14E+000	1.13E+000	1.16E+000	1.14E+000	1.15E+000
	MPICH Bi. Hier.	6.96E-001	6.88E-001	6.90E-001	6.85E-001	6.87E-001	6.86E-001
	Dynamic Leader	5.54E-001	4.94E-001	4.37E-001	3.38E-001	2.21E-001	7.42E-002

Table C.8.: Alltoall latency 32768, varying buffer sizes

		0	10000	20000	40000	80000	160000
32768	Default Cray	5.60E-002	4.72E-002	4.12E-002	3.38E-002	3.10E-002	3.10E-002
	MPICH Isend	1.87E+000	2.49E+000	1.87E+000	1.83E+000	1.96E+000	1.87E+000
	MPICH Isend Hier.	9.37E-002	8.44E-002	8.93E-002	9.51E-002	9.18E-002	8.93E-002
	Dynamic Leader	3.62E-002	3.18E-002	2.24E-002	1.61E-002	1.52E-002	1.45E-002
131072	Default Cray	1.88E-001	1.80E-001	1.73E-001	1.37E-001	1.08E-001	1.08E-001
	MPICH Bi.	2.11E+000	2.01E+000	2.14E+000	2.21E+000	2.43E+000	2.33E+000
	MPICH Bi. Hier.	4.72E-001	5.20E-001	5.22E-001	5.27E-001	5.13E-001	5.30E-001
	Dynamic Leader	1.15E-001	1.01E-001	9.70E-002	7.77E-002	3.71E-002	2.32E-002
524288	Default Cray	7.06E-001	7.02E-001	6.86E-001	6.63E-001	6.07E-001	5.65E-001
	MPICH Bi.	2.53E+000	2.36E+000	2.44E+000	2.39E+000	2.33E+000	2.38E+000
	MPICH Bi. Hier.	5.80E-001	6.15E-001	6.12E-001	6.14E-001	5.95E-001	6.28E-001
	Dynamic Leader	2.38E-001	2.11E-001	2.02E-001	1.47E-001	7.41E-002	3.70E-002

VITA

VITA

Benjamin Parsons graduated from Purdue University in 2009 with a bachelor's degree in computer engineering. He then enrolled in the graduate program at Purdue, continuing to work with Vijay Pai as his research adviser. In 2011 he received a master's degree in computer engineering with his research focusing on probabilistic hard disk timing models. In 2011 he received funding from the SMART scholarship and starting working with the Army Corps of Engineers Research and Development Center. This gave him access to several large scale systems, and as a result the focus of his research turned to collective communications on HPC systems.