Faster Topology-aware Collective Algorithms Through Non-minimal Communication *

Paul Sack William Gropp
Department of Computer Science
University of Illinois at Urbana-Champaign
paulsack@illinois.edu wgropp@illinois.edu

Abstract

Known algorithms for two important collective communication operations, allgather and reduce-scatter, are minimal-communication algorithms; no process sends or receives more than the minimum amount of data. This, combined with the data-ordering semantics of the operations, limits the flexibility and performance of these algorithms. Our novel non-minimal, topology-aware algorithms deliver far better performance with the addition of a very small amount of redundant communication. We develop novel algorithms for Clos networks and single or multi-ported torus networks. Tests on a 32k-node BlueGene/P result in allgather speedups of up to 6x and reduce-scatter speedups of over 11x compared to the native IBM algorithm. Broadcast, reduce, and allreduce can be composed of allgather or reduce-scatter and other collective operations; our techniques also improve the performance of these algorithms.

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

General Terms Algorithms, Performance

Keywords Collective-communication algorithms

1. Introduction

Known algorithms for collective communication in MPI have been designed to minimize data movement, across the network and in memory. The output of any collective operation must be in a certain order. These requirements together severely curtail the options for collective algorithms. In this work, we show that a small increase in the amount of data communicated can have an enormous effect on performance by reducing congestion and allowing for multi-port algorithms.

* This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357. This work was supported in part by the U.S. Department of Energy under contract DE-FG02-08ER25835 and by the National Science Foundation under grant 0837719. We thank the anonymous reviewers, Brian Greskamp, and Jeff Hammond for their helpful comments.

PPoPP'12, February 25–29, 2012, New Orleans, Louisiana, USA.

Many supercomputers with Clos ("fat-tree") networks have less bisection bandwidth than injection bandwidth. The Roadrunner system, currently at number 7 on the Top 500 list [12], has a two-level tree, where the top level provides for just over 25% of the injection bandwidth from the processor cores into the lower level. All supercomputers with torus networks have less bisection bandwidth than injection bandwidth. In other words, under certain bisect communication patterns, congestion will limit performance.

Unfortunately, many widely-used collective communication algorithms make use of bisect patterns that cause the worst congestion. Also, multi-ported switches on torus networks allow for nodes to exchange messages on multiple links simultaneously, but collective algorithms do not take advantage of this.

The essence of our approach is that we temporarily relax the requirement that the output of an operation must be in a particular order as specified in the MPI standard. Instead, we require only that the output be in the same order on every process. We then add a small extra stage of communication either before or after the operation that restores the correct ordering. On an n-node system, this increases the amount of communication by a factor of only $1/n$ to $2/n$. On the other hand, it eliminates or sharply reduces congestion and allows the use of multi-port algorithms.

Specifically, for all networks, we reorder the communication stages in the recursive-doubling allgather and recursive-halving reduce-scatter algorithms, and, for mesh or torus networks, we execute three or six three-dimensional bucket operations in parallel. This produces the correct result in the wrong order; one extra stage of communication restores the correct order.

These algorithms deliver better performance on any network that can suffer from congestion or allows multi-ported communication. In tests on an IBM Blue Gene/P, our best algorithm delivers up to 6x better performance for allgather, and up to 11x better performance for reduce-scatter over the native algorithm. For larger problems, broadcast, reduce, and allreduce are usually decomposed into a scatter and allgather, reduce-scatter and gather, or reduce-scatter and allgather, respectively. Our methods improve the performance of these operations as well.

In Section 2, we discuss known algorithms and related work and show why the performance of minimal-communication algorithms is limited by network congestion and the use of one port at a time in multiport networks. In Section 3, we present our new algorithms. In Section 4, we evaluate the performance of our algorithms on a large supercomputer and compare predicted and measured performance. We discuss future work and conclude in Section 5.

2. Background

In the allgather operation, each process i of P processes has an input vector X_i of length n. After the operation is complete, each

process has the same copy of output vector Y of length nP. Vector Y consists of each X_i concatenated in rank order. *I.e.*, $Y = X_0X_1X_2 \cdots X_{P-1}$. Essentially, every process broadcasts its input vector X to all the other processes.

In the reduce-scatter operation, each process has an input vector X_i of length nP. Afterwards, each process has an output vector Y_i of length n. In the operation, blocks of n input elements from each X_i are combined using the reduction operator, and the result from block i is stored on the process with rank i. E.g., Y_0 will be formed from the first n elements of each X_i; Y_1 will be formed from the second n elements of each X_i and so on. Formally, $Y_i = \oplus_{p=0}^{P-1} X_p[in : (i+1)n)$, where $\oplus$ is the reduction operator (*e.g.*, sum, maximum, product, etc.).

Allgather can be easily implemented as P broadcast calls; similarly, reduce-scatter can be implemented as P reduce calls, but more sophisticated algorithms deliver better performance.

These two algorithms are particularly important, because several other MPI operations can be efficiently implemented as combinations of allgather or reduce-scatter and other operations. In this work, we focus on allgather; the inverse of the communication pattern in each allgather algorithm is the same as the communication pattern in the corresponding reduce-scatter algorithm.

2.1 Network model

Of the many network performance models that one can use in analyzing collective operations, we base ours on a simple, frequently-used model that incorporates two parameters: α, the startup cost per message, and β, the bandwidth cost per byte. A third parameter, γ, is used in other work to represent the cost of applying the reduction operator. We ignore the cost of applying reduction operators in our model; our algorithms do not affect this cost. We use two models; one, the common α/β model which does not consider congestion or topology, and another that extends the α/β model to consider congestion and topology.

We charge α once for each pair of send and receive messages. We explore *single-port* networks, in which each process can send and receive one message at a time, and *multi-port* networks, where each process can send and receive as many messages as each node has links. We assume there is only one MPI process on each node. Multiple processes on a node are best handled using a hierarchical MPI library, such as [15].

No-congestion model: The first model ignores topology and congestion. β represents the bandwidth with which each processor is connected to the network. Each process can only send and receive one message in each stage in this model. The α term for an algorithm is simply the number of stages, since each process can exchange up to one message per stage. The β term is found by summing the maximum message size per stage over all the stages. This model can also be seen as one in which each processor is connected by one link to a full crossbar interconnect.

Equations derived using this model will be labeled as plain C. This was the model used in developing algorithms for MPICH [16, 17] and other work that ignores congestion or topology.

Congestion-aware model: The second model incorporates topology and congestion. δ replaces β and represents the link speed. We add a penalty factor in the δ term to represent congestion. Topology and congestion-aware equations we will label as C_{Clos} or C_{torus}. In this model, processors on torus networks can send and receive messages on every link simultaneously.

For simplicity, each processor can send and receive up to one message on each link per stage. The α term is derived by summing the maximum number of messages exchanged in each stage by any one process over all the stages. The δ term is derived by summing the amount of data transferred over the busiest link in each stage over all the stages.

On a Clos network, each process has one link, so $\delta = \beta$, and $C_{Clos} = C$ for algorithms that do not suffer from congestion.

On a 3-d torus network, each process has six links, but can only use one at a time in the simple model. Thus, $\delta = \beta$ and $C_{torus} = C$ for single-ported algorithms that do not suffer from congestion.

Our congestion-aware model cannot model multi-port algorithms where the message activity on different ports is unsynchronized. In [14], we propose a more detailed model that can. This is only necessary for modeling our multi-port allgather algorithm on non-cubic 3-d torus networks.

Clos model: For the Clos network, we consider networks parameterized by R, the radix of the switches, and μ, the ratio of the minimum bisection bandwidth to injection bandwidth, which is subject to the constraint: $0 < \mu \leq 1$. In an optimal Clos network, the number of up links and down links for any non-root, non-terminal switch will each be $R/2$. The root switches will each have R down links. Each bottom-level switch will be connected to T nodes and $R - T$ level-two switches, where $\frac{R-T}{T} \geq \mu$. This is the same idealized model of a Clos network as is used in [1] to minimize the number of switches on a system of a given size subject to constraints on μ and R. Switches with a radix of 128 or 256 will likely be used in supercomputers in the near future [4].

Our model for Clos networks underestimates the congestion seen in real-world Clos networks. Hoefler studied the bandwidth delivered by Infiniband Clos networks, which use static routing tables in the switches [9]. The delivered bandwidth was found to be 55-60% of that which could be delivered with ideal routing, due to hot spots. Zahavi inspected the communication patterns of all common collective algorithms on a Clos network and presented an algorithm for generating routing tables that prevent congestion in Clos networks where the bisection bandwidth matches the injection bandwidth [19]. This accelerates collective algorithms by 40% when an application uses the whole system.

In our model, we assume that Clos networks have perfect oracular routing. Our algorithms will improve the performance of collective algorithms on real Clos networks more than our model predicts.

2.2 Non-topology-aware algorithms

One simple allgather algorithm is the ring algorithm, composed of $P - 1$ stages. In every stage, process i sends data to process $i - 1 \mod P$ and receives data from process $i + 1 \mod P$. In the first stage, process i sends its input vector X_i and receives vector $X_{i+1 \mod P}$. In the subsequent $P - 2$ stages, each process sends the vector it received in the previous step. In stage s, where the stages are numbered from 0 to $P - 2$, each process sends $X_{i+s \mod P}$ and receives $X_{i+s+1 \mod P}$[1].

After P steps, each process has the full output vector $Y = X_0X_1X_2 \cdots X_{P-1}$.

The ring algorithm is easy to analyze: there are $P - 1$ stages, so the message-startup cost is $P - 1$. In a 3-d torus or Clos network, the ring algorithm will not cause any congestion, and the bandwidth term will be $n \cdot (P - 1)$. The ring algorithm is optimal with respect to the bandwidth term on single-port networks.

Therefore, the cost is:

$$C = (P - 1)\alpha + n(P - 1)\beta.$$

A similar ring algorithm exists for the reduce-scatter operation. In the first stage, each process i sends process $i - 1$ the elements from its input vector corresponding to process $i + 1$, *i.e.*, $X_i[(i + 1)n : (i + 2)n)$. It receives from process $i + 1$ the data corresponding to process $i+2$ and combines this data (using the re-

[1] Hereafter, arithmetic on process ranks is assumed to be circular on the number of processes P.

duction operator) with the elements in X_i corresponding to process $i+2$. This proceeds similarly for the $s-2$ remaining stages.

The ring algorithms can also be run in the opposite direction.

An important algorithm for allgather is known as the recursive-doubling algorithm. It is composed of $\log_2 P$ stages. In the first stage, process i exchanges its input vector X_i of n elements with the process whose rank only differs in the last bit. In the second stage, process i exchanges $2n$ elements with the process whose rank only differs in the second-last bit. In stage s (from 0 to $\log_2 P - 1$), process i exchanges $2^s n$ elements with process i `XOR` 2^s. It is named the recursive-doubling algorithm because the amount of data exchanged and the portion of the output vector Y that is filled doubles in each stage.

The corresponding algorithm for reduce-scatter is known as the recursive-halving algorithm. In the first stage, processes whose high bits differ exchange half of the input vector, in the second stage, processes whose second-highest bits differ exchange one fourth of the input vector, and so on.

If there is no congestion, the bandwidth term is given by the summation: $\sum_{s=0}^{\log_2 P-1} 2^s n = n(P-1)$, and the cost is:

$$C = \log_2 P\alpha + n(P-1)\beta.$$

This algorithm is optimal with respect to the startup cost; data cannot be broadcast from one process to $P-1$ other processes in under $\log_2 P$ stages. However, on a $\mu < 1$ Clos network or a torus, there will be congestion.

Clos network: Let us consider first a Clos network where each terminal switch is connected to T nodes, where T is a power-of-two. In the first few stages, where 2^s is less than T, all communication will be between nodes connected to the same switch and there will be no congestion. In the remaining stages, all communication will be between nodes connected to different switches, and the performance will suffer by a factor of $1/\mu$. Thus,

$$\begin{aligned} C_{Clos} &= \log_2 P\alpha + n\delta\{(1+2+4+\cdots+T/2) \\ &+ (1/\mu)(T+2T+4T+\cdots+P/2)\} \\ &= \log_2 P\alpha + n\delta\{(T-1)+(1/\mu)(P-T)\} \\ &\approx \log_2 P\alpha + (nP/\mu)\delta, \end{aligned}$$

where the approximation holds for $P \gg T$.

Thus, the effect of congestion reduces the performance of this algorithm by a factor of $1/\mu$ on the bandwidth term.

3-d torus: Let us consider the performance of the algorithm on a system composed of P nodes arranged in a $p \times p \times p$ 3-d torus, where p is a power-of-two. We assume that the nodes are numbered in increasing order in the X dimension, then the Y dimension, then the Z dimension.

In the first stage, nodes at a rank distance of one apart exchange input vectors. These nodes will be neighbors in the X dimension, so there will be no congestion. In the second stage, nodes at a rank distance of two apart exchange twice as much data as in the first step. If $X \geq 4$, these nodes will be two hops away in the X dimension, and there will be a congestion factor of two. In the third step, messages four times as large will travel four hops in the X dimension, with 4-fold congestion, and so on. In the second-last stage in each dimension, stage $\log_2 p - 1$, messages will travel $p/4$ hops and suffer $p/4$-way congestion. Since this is a torus, in the final stage in each dimension, stage $\log_2 p$, messages will travel $p/2$ hops, but will suffer only $p/4$-way congestion because in this stage only, the extra wrap-around link provided by the torus can be used to relieve congestion.

Once we reach the stage where the rank distance between nodes equals X, neighbors in the Y dimension exchange data, and so on.

If we revisit our model, we get a bandwidth term of:

$$\begin{aligned} & n\delta(1+p+p^2)\{1\cdot 1 + 2\cdot 2 + 4\cdot 4 + \cdots + \\ & (p/4)\cdot(p/4) + (p/2)\cdot(p/4)\} \\ = \; & n\delta(1+p+p^2)\{\sum_{i=0}^{\log_2 p-1} 4^i - p^2/8\} \\ = \; & n\delta(1+p+p^2)\{(1/3)(p^2-1) - p^2/8\} \\ = \; & n\delta(1+p+p^2)((5/24)p^2 - 1/3) \\ \approx \; & (5/24)nP^{4/3}\delta \end{aligned}$$

and a total cost:

$$C_{torus} \approx \log_2 P\alpha + (5/24)nP^{4/3}\delta.$$

Each factor 1, p, or p^2 represents the size of the first message exchanged in dimension X, Y, and Z, respectively. The first factor in each term in the summation represents the message size in the $\log_2 p$ stages in that dimension and the second term represents the congestion factor; note that the last two stages in each dimension have the same congestion factor: $p/4$.

The recursive-doubling algorithm is used for shorter messages in MPICH due to its lower startup cost term, and the ring algorithm is used for longer messages [7, 8].

2.3 Topology-aware algorithms

The 3-d bucket allgather algorithm is formed by executing the ring algorithm in each dimension in turn. *E.g.*, on a $P = p \times p \times p$ network, messages of size n circulate in the X dimension, then messages of size np circulate in the Y dimension, then messages of size np^2 circulate in the Z dimension.

This reduces the number of messages substantially and incurs a total cost:

$$C_{torus} = 3(\sqrt[3]{P} - 1)\alpha + n(P-1)\delta$$

Similarly, the 3-d bucket reduce-scatter algorithm is formed by executing three ring reduce-scatter algorithms, starting with np^2 elements per message in the Z dimension, then np elements per message in the Y dimension, then n elements per message in the X dimension.

3. Design

In this section, we present several non-minimal recursive-doubling and bucket algorithms.

We add a stage of communication before the allgather operation and after the reduce-scatter operation that enables us to reorder the stages of the algorithms while preserving correctness. It also enables the use of multiple ports in a multi-port torus network.

3.1 Recursive-doubling algorithm

The recursive-doubling allgather algorithm presented above can be described as a *distance-doubling*, recursive-doubling algorithm, because the numerical distance between communicating processes doubles in every stage $(1, 2, 4, \ldots, P/2)$. Thus, the stages with the largest messages suffer from the worst congestion.

3.1.1 Solution for Clos networks

To resolve this, we propose a recursive-doubling, *distance-halving* allgather algorithm for Clos networks and a reordered recursive-doubling allgather algorithm for mesh or torus networks.

In the distance-halving algorithm, in each stage s, processes exchange $n2^s$ elements, as in all recursive-doubling algorithms. Process i exchanges data with process i `XOR` $P2^{-(s+1)}$. *I.e.*, in the first stage, processes whose ranks differ only in the highest bit

exchange data, in the second stage, processes whose ranks differ only in the next-highest bit exchange data, and so on.

The cost is then:

$$\begin{aligned} C_{Clos} &= \log_2 P\alpha + n\delta\{(1/\mu)(1+2+4+\cdots+P/(2T)) \\ &+ (P/T + 2P/T + 4P/T + \cdots + P/2)\} \\ &= \log_2 P\alpha + n\delta\{(1/\mu)(P/T-1) + (P - P/T)\} \\ &\approx log_2 P\alpha + nP\{1 + 1/(\mu T)\}\delta \end{aligned}$$

From the first stage until the last stage where $P2^{-s+1} \geq T$, all messages must travel through multiple switches and incur congestion determined by the parameter μ. The largest messages in the final stages travel through only one switch with no congestion.

This analysis assumes P and T are powers-of-two. It is a close approximation for other values of T. If 2^a is the largest power of two that evenly divides T, then in the last a stages all the messages will travel through only one switch. For the next handful of stages, some pairs of processes will exchange messages through only one switch and others through several switches. Whether or not congestion occurs in the intermediate stages depends on the value of μ and T, but it is sharply curtailed in any case. There is no concise equation for the exact value when T is not a power-of-two.

This algorithm reduces the bandwidth term in the cost from $(nP/\mu)\delta$ to $nP\{1 + 1/(\mu T)\}\delta$, and is close to the optimal bandwidth term, $nP\delta$, for typical values of μT, however, the data is no longer in the correct order.

In the usual distance-doubling, recursive-doubling algorithm, each process receives a message in each stage containing elements in the output vector adjacent to elements that the process already has. *E.g.*, process 0 starts with element 0, receives element 1 in stage 1, receives elements 2 and 3 in stage 2, 4-7 in stage 3, and so on. In each stage, each process simply concatenates the data it receives with the data it already has.

In the distance-halving, recursive-doubling algorithm, each process still concatenates the data it has with the data it receives in each stage, but the output ends up in the wrong order. The input vector X_i should start at offset ni in the output vector Y, but instead will end up at offset $n\tilde{i}$, where $\tilde{i}$ is the bit-reverse of i.

To rectify this, we simply have processes i and $\tilde{i}$ exchange input vectors before the algorithm begins. Then each input vector ends up at the correct offset in the output vector.

We add this stage to the algorithm's cost:

$$\begin{aligned} C_{Clos} &= \log_2 P\alpha + nP\{1 + 1/(\mu T)\}\delta + \\ &\quad \alpha + (n/\mu)\delta \\ &\approx (\log_2 P + 1)\alpha + nP\{1 + 1/(\mu T)\}\delta. \end{aligned}$$

Each process sends and receives nP bytes instead of $n(P-1)$, a negligible difference.

For typical Clos networks, $1/(\mu T)$ is a small number and:

$$C_{Clos} \approx (\log_2 P)\alpha + nP\delta.$$

For the corresponding recursive-halving, distance-doubling reduce-scatter algorithm, processes i and $\tilde{i}$ exchange output vectors.

3.1.2 Solution for torus networks

For mesh or torus networks, we *could* use the distance-halving allgather algorithm and expect an improvement, since the largest message in the last stage will not be delayed by congestion and the next largest message will have a congestion factor of 2 instead of $\sqrt[3]{P}/4$. The distance-halving algorithm is really a quasi-topology-aware algorithm, since it will perform well for any network in which nodes are numbered rationally.

We can do better if we further reorder the stages. As before, let us assume that the network has $P = p \times p \times p$ nodes, where p is a power-of-two. In stages one through three, nodes $p/2$ hops apart in the X, Y, and Z dimensions, respectively, exchange data, suffering $p/4$-fold congestion. In the next three stages, processes $p/4$ hops apart exchange data, suffering $p/4$-fold congestion again. In the next group of three stages, congestion will be reduced to a factor of $p/8$, and will continue to halve for each subsequent group of three stages. In the last three stages, immediate neighbors exchange data with no congestion.

The bandwidth term is then:

$$\begin{aligned} & (1+2+4)n\{\sum 1 \cdot p/4 + 8 \cdot p/4 + \cdots + P/8 \cdot 1\}\delta \\ \approx\ & 7n(p/2)\{\sum_{i=0}^{\log_2 p - 1} 4^i\}\delta \\ \approx\ & (7/6)np^3\delta \\ =\ & (7/6)nP\delta. \end{aligned}$$

The first message exchanged in each dimensions is of size n times 1, 2, or 4. The first term in the summation reflects the size of the message; the first message exchanged in dimension X will be of size $1n$, the second message exchanged in dimension X will be of size $8n$, etc. The second term in the summation represents the congestion in that stage of the algorithm.

The data exchange to preserve the correct ordering is more complex. Let us define a schedule, S, which dictates the distance between pairs of communicating processes in each stage. More precisely, S is a vector that specifies which bit differs between communicating pairs of processes in each stage.

For the distance-doubling, recursive-doubling algorithm, $S[s] = s$, since in stage s, process i and process i `XOR` 2^s exchange data. For the distance-halving algorithm, $S[s] = \log_2 P - 1 - s$. For the torus-reordered algorithm with XYZ process numbering:

$$\begin{aligned} S[s] = \{&3\log_2 p - 1, 2\log_2 p - 1, \log_2 p - 1, \\ &3\log_2 p - 2, 2\log_2 p - 2, \log_2 p - 2, \\ &\ldots, \\ &2\log_2 p, \log_2 p, 0\}. \end{aligned}$$

For example, for a 512-node system arranged as an 8x8x8 3-d torus, $S[s]$ can range from 0 to 8 (since $\log_2 512 = 8 + 1$). For distance-doubling, $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$. For distance-halving, $S = \{8, 7, 6, 5, 4, 3, 2, 1, 0\}$. For the remapped algorithm, one optimal schedule is $S = \{8, 5, 2, 7, 4, 1, 6, 3, 0\}$. (The entries in S for each group of three consecutive stages can be permuted.)

This table shows the full schedule:

Stage	S[s]	Distance	Hops	Congestion	Size $\times n$
0	8	256	4 (Z)	2	1
1	5	32	4 (Y)	2	2
2	2	4	4 (X)	2	4
3	7	128	2 (Z)	2	8
4	4	16	2 (Y)	2	16
5	1	2	2 (X)	2	32
6	6	64	1 (Z)	1	64
7	3	8	1 (Y)	1	128
8	0	1	1(X)	1	256

Note that the communication in the three stages with the largest messages occurs without congestion.

We define a mapping function R, where $R(S, i)$ is the process which sends its input vector to process i before the allgather operation (or to which process i sends its output vector after the reduce-scatter operation). $R(S, i)$ permutes the bits in i according to S.

The process with rank i can be expressed using bit vector b_i, where $i = \sum_{r=1}^{\lceil \log_2 P \rceil} 2^{r-1} b_i[r]$. *E.g.*, if $P = 16$, $b_{11} = \{1, 1, 0, 1\}$. In the data-swapping stage, rank i then receives its data from rank $R(S, i) = \sum_{r=1}^{\lceil \log_2 P \rceil} 2^{r-1} b_i[S[r]]$.

We have investigated the exact traffic patterns in the data swapping phase for several different network sizes and found that none cause any congestion, but we cannot prove this. However, a 3-d $p \times p \times p$ torus has $2p^2$ bisection bandwidth, therefore the bandwidth cost for this phase cannot exceed $n(P/2)/(2p^2)\delta = (n/4)\sqrt[3]{P}\delta$. The bandwidth cost of the extra stage is marginal compared to the bandwidth cost of the main algorithm.

Thus, the total cost is:

$$\begin{aligned} C_{torus} &\leq (\log_2 P + 1)\alpha + ((7/6)nP + (n/4)\sqrt[3]{P})\delta \\ &\approx (\log_2 P)\alpha + (7/6)nP\delta. \end{aligned}$$

3.2 Bucket algorithm

The bandwidth term in the bucket algorithm is optimal for the single-port model on a 3-d torus. The number of stages, $3\sqrt[3]{P} - 1$, is more than the $\log_2 P$ stages of the recursive-doubling algorithm but much less than the $P - 1$ stages of the 1-d ring algorithm.

However, on a multi-port 3-d torus network, each process can exchange messages with all six of its neighbors simultaneously. The bucket algorithm does not take advantage of this. Our approach extends the bucket algorithm to use all six ports.

In our algorithm, we run six bucket operations simultaneously in a way that avoids congestion. We label the six buckets XYZ^+, XYZ^-, YZX^+, YZX^-, ZXY^+, and ZXY^-. The + buckets run clockwise in each dimension and the - buckets run counter-clockwise in each dimension. XYZ^+ and XYZ^- both circulate in the X dimension, then the Y dimension, then the Z dimension. Similarly, both YZX buckets circulate in the Y dimension, then the Z dimension, then the X dimension. We divide the input and output vectors into six sections, one for each bucket. Buckets 0 and 1 refer to the XYZ buckets, 2 and 3 to the YZX buckets, and 4 and 5 to the ZXY buckets.

The cost is then:

$$\begin{aligned} C_{torus} &= 6 \cdot (3\sqrt[3]{P} - 1)\alpha + (n/6)(P-1)\delta \\ &\approx 18\sqrt[3]{P}\alpha + (n/6)(P-1)\delta \end{aligned}$$

For large problems, the performance loss from the 6-fold increase in the number of messages is more than offset by the 6-fold decrease in the bandwidth term.

However, we face a familiar problem: out-of-order data.

Let us show how the the 6-way bucket algorithm reorders the data. Suppose that each process i has 6 elements in its input vector $X_i = \{A_i, B_i, C_i, D_i, E_i, F_i\}$. The correct output would be $Y = \{A_0 - F_0, A_1 - F_1, \ldots A_{P-1} - F_{P-1}\}$. However, instead, in the first part of Y, we will get $Y[0 : n/6) = \{A_0, A_1, \ldots, A_{P-1}\}$. In the second part, we will get $Y[n/6 : n/3) = \{B_0, B_1, \ldots, B_{P-1}\}$. In the third part (from YZX^+), we will get $Y[n/3 : n/2) = \{C_0, C_p, C_{2p}, \ldots, C_{P-1}\}$, since the output will be in YZX order. The data in the remaining three sections will be similarly perturbed.

To solve this, we number each of the six sections in each input vector X_i. Section j in input X_i is assigned *section identifier* $6i+j$. In the correct output vector, all the input sections are arranged in section id order. In total, we have $6P$ sections. Sections whose id is between 0 and $P - 1$ must be placed in the first bucket, XYZ^+; sections whose id is between P and $2P - 1$ must be placed in the second bucket, XYZ^-, and so on.

Next is the question of *where* to place each section in the correct bucket. Section s must be at offset $s \mod P$ in the output of the corresponding bucket. For the two XYZ buckets, this is achieved by placing section s in the corresponding input bucket on process $s \mod P$.

For the YZX buckets, section s must be placed in one of the YZX buckets on the process whose rank *would be* $s \mod P$ if the processes were numbered in YZX order. Since they are not, we rotate the bits in $s \mod P$ to find the rank of the process which have section s in its input buffer.

Consider an $8 \times 8 \times 8$ torus again. The segment id of the third segment from process 200 is $s = 200 \cdot 6 + 3 = 1203$. $\lfloor 1203/P \rfloor = 2$, therefore this segment will be in bucket 2: YZX^+. Its offset within YZX^+ is $1203 \mod P = 179$. This segment must start from the input vector of the process whose YZX order is 179. We then take the 9-bit quantity 179 and rotate it clockwise 3 bits to put it in XYZ order. The rotated offset is 410. Therefore the segment whose id is 1203 should be placed in $X_{410}[YZX^+]$.

For segments in the ZXY buckets, we rotate $s \mod P$ 6-bits clockwise (or 3-bits counter-clockwise).

To summarize, for each of the 6 sections in each input vector X_i, we assign a section id: $s = 6i + j$, where j is a value from 0 to 5. This section should be placed in bucket $\lfloor s/P \rfloor$. We then find the quantity $s \mod P$. If the section maps to either XYZ bucket, then this section should be sent to process $s \mod P$. If the section is in either YZX bucket, we rotate $s \mod P$ by $\log_2 p$ bits clockwise and send the data to process $(s \mod P) >> \log_2 p$.[2] If the section is in either ZXY bucket, we rotate $s \mod P$ by $\log_2 p$ bits counter-clockwise and send the data to $(s \mod P) << \log_2 p$.

The correct data remapping for this algorithm or the remapped recursive-doubling algorithm can also be determined empirically quite easily by observing the order of the output.

Including the data-shuffling stage, the cost is:

$$\begin{aligned} C_{torus} &= (18\sqrt[3]{P} + 6)\alpha + ((n/6)(P-1) + n)\delta \\ &\approx 18\sqrt[3]{P}\alpha + (n/6)(P-1)\delta. \end{aligned}$$

The additional data movement adds negligible overhead.

Non-cube torus networks are easily handled. Each of the six bucket operations waits until all the others have finished circulating in each dimension before moving on to circulate in the next dimension. We found that performance suffered greatly if we allowed multiple bucket operations to compete for the same links.

Performance in the first two stages will be limited by the pair of buckets circulating on the largest dimension. In the last stage, in which the messages are much larger than in the first two stages, the buckets will have more equal performance. The buckets circulating along the longest dimension will have the smallest messages, and the buckets circulating along the shortest dimension will have the largest messages.

Consider a network where $X < Y < Z$. In the last stage, the slowest bucket will be the XYZ buckets and the quickest the YZX buckets. XYZ buckets will circulate $Z-1$ messages of XYn bytes, for a total of $XY(Z-1)n$ bytes. The YZX buckets will circulate $X - 1$ messages of YZ bytes, for a total of $(X-1)YZ$ bytes. On large systems, the difference is not important.

Messages whose size is not a multiple of six are also simple to handle. We use a similar technique to that used in [18] for the irregular `MPI_Allgatherv` problem. We round-up the input size to the next multiple of six bytes and shift the data from higher processes to lower processes. Processes whose rank assignment is near P may have empty input buffers. This will involve each process sending at most two messages or receiving up to six messages, but typically no

[2] $>>$ and $<<$ are the cyclical clockwise and counter-clockwise bit rotation operators.

process will receive more than three messages. The amount of data exchanged in this stage will be no more than n bytes per process.

For 3-d meshes, or partitions of 3-d tori, the 6-way bucket algorithm will cause a congestion load of 2 on each link (since there is no wrap-around link). The 3-way bucket algorithm would then be preferred for its lower startup cost.

A variation of this algorithm was presented in [11]. Instead of adding an extra stage of communication to restore the correct data ordering, they reorder the data after the allgather operation or before the reduce-scatter operation. We also considered three alternatives to the extra stage: using MPI types to send and receive messages with non-contiguous memory addresses, packing the data into temporary buffers before sending messages and unpacking the data from temporary buffers after receiving messages, and reordering the data in-memory at the end. We found all three to have poor performance. The presentation in [11] is difficult to follow, since they extend the bucket algorithm but present the ring algorithm instead in their description of the base bucket algorithm. They explored multiple methods of multithreading the operation of the 6-way bucket algorithm; we do not. We will compare our performance results with theirs in the following section.

3.3 Other algorithms

Long broadcast operations in MPICH use the Van de Geijn algorithm, in which the operation is composed of a scatter operation followed by an allgather operation [3]. Similarly, the long reduce operation can be composed of an reduce-scatter, followed by an gather, and the long allreduce can be broken into an reduce-scatter call followed by allgather. Our algorithms can accelerate all of these operations. Further, the data reordering stages can often be made to cancel out; this happens naturally for allreduce. With simple changes, the data reordering can be folded into the gather or scatter operation. In fact, MPICH already uses a recursive-halving, distance-doubling reduce-scatter as the basis for the long reduce and allreduce operations, with no explanation.

3.4 Other related work

The algorithms used in OpenMPI [6] and MPICH, broadly-speaking, are chosen as those that are best under our two-parameter model, but they ignore network congestion, since these are generic libraries. The specific algorithms used in MPICH are described in [17]. Notably, the algorithm for the big allgather and big reduce-scatter problems uses an embedded one-dimensional ring. In [16], the same authors provide more detail but cover fewer algorithms. In particular, the often-encountered recursive-doubling and recursive-halving patterns are well-explained.

In [2], a clever broadcast algorithm is developed for two-dimensional mesh topologies, which achieves $\log_2 P$ scaling. Their algorithm is based on the common binary-tree algorithm. For an $p \times p$ network, in step s, each node that has a copy of the message sends it to a node that does not at a distance of $p2^{-s}$ hops away. This avoids the congestion inherent in running generic binary-tree algorithms on meshes.

In [13], several reduction algorithms are analyzed and implemented on a two-dimensional mesh, including binomial tree-based algorithms, recursive halving algorithms, a two-phase bucket-based algorithm, and several hybrids. The optimal choice, analytically and empirically, depended upon the size of the reduction.

4. Evaluation

All experiments were performed on a 40k-node Blue Gene/P system. Partitions of 512 nodes or more form 3-d tori. Smaller partitions form 3-d meshes. The Blue Gene/P has a torus network for point-to-point messages and some collective-communication operations and a tree network for other collective-communication operations. Both networks have special features to accelerate collective communication. The torus network has a "deposit bit" feature in which a message can be put on the network once and every node along a line in the torus can receive it. This is particularly helpful for broadcast operations. The switches in the tree network support in-place integer reduction operations.

The 512-node allocation forms an 8x8x8-node cube. The 32k-node allocations forms a 32x32x32-node cube. The allocations in between are non-cubic. The Z dimension doubles until it reaches 32 nodes, then the Y dimension doubles until 32, then the X dimension doubles. We run one MPI process per node. The system provides a simple interface for finding the size of each dimension in an allocation and the coordinates of each process in the allocation. By default, processes are numbered in XYZ order.

Each node is comprised of four PowerPC 450 microprocessors and a switch. Each switch has 3 bidirectional links on the tree network of 850 MB/s per direction and 6 bidirectional links on the torus network of 425 MB/s per switch per direction. The minimum packet size is 32 bytes, with 16 bytes of payload data and 16 bytes of header data. The maximum size is 256 bytes with 240 bytes of payload data. After accounting for the overhead in the packet headers and the acknowledgement packets, the maximum bandwidth per link is 88% of the raw bandwidth. According to IBM, with 6-way bidirectional communication, 93% of that is the peak that can be delivered, *i.e.*, $425\text{ MB/s} \times 6 \times 2 \times .88 \times .93 = 4.18$ GB/s [10]. In our data, we report the one-way bandwidth into or out of a node. The bandwidth calculations count the time but not the data exchanged in the extra stage for the reordered recursive-doubling algorithms. *I.e.*, we report $n(P-1)/t$, where n is the input message size on each process, P is the number of processes, and t is the amount of time each operation takes.

The output of each combination of algorithm, size, and number of processes was verified for correctness.

In each figure, the message size refers to the size of the input vector for the allgather operation or the output vector for the reduce-scatter operation, *i.e.*, the value of n from Section 3. The size of the output allgather vector or input reduce-scatter vector is this value times the number of processes, *i.e.*, nP.

4.1 Allgather performance

The left plot in Figure 1 shows the performance of the one-port allgather algorithms on an 8x8x8 512-node 3-d torus. The recursive-doubling, distance-doubling algorithm (*rd doubling*) is the better of the two non-topology-aware algorithms for smaller messages, due to the lower message count of the recursive-doubling algorithms, and the ring algorithm is better for larger messages, due to the absence of congestion in the ring algorithm. The quasi-topology-aware recursive-doubling, distance-halving algorithm (*rd halving*), with one more stage of communication, beats the recursive-doubling, distance-halving algorithm for all message sizes.

For the topology-aware algorithms, the optimally-scheduled recursive-doubling algorithm (*rd optimal*) is better for smaller messages and the bucket algorithm is better for larger messages. This is expected, since the recursive-doubling algorithm has fewer stages but a larger bandwidth term. The bucket algorithm does reach the upper bound of 375 MB/s. The native algorithm is generally better than all the single-port algorithms for small messages on small partitions. We plot data from the native algorithm on the default `MPI_COMM_WORLD` communicator.

Figure 2 shows the performance of the multi-port algorithms. We include the native BlueGene/P allgather algorithm in this figure. For small messages, the native algorithm running on a copy of `MPI_COMM_WORLD` performs far worse than the native algorithm on `MPI_COMM_WORLD`. The 3-bucket algorithm delivers up to 1120

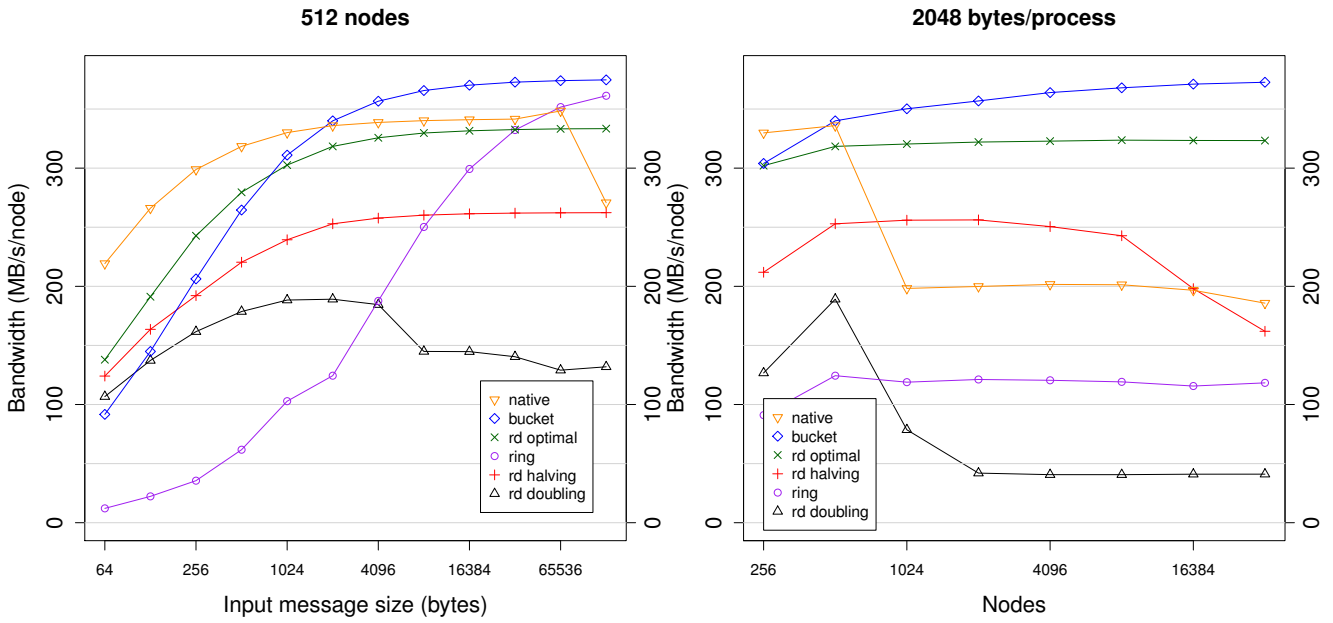

Figure 1. Performance of single-port allgather algorithms.

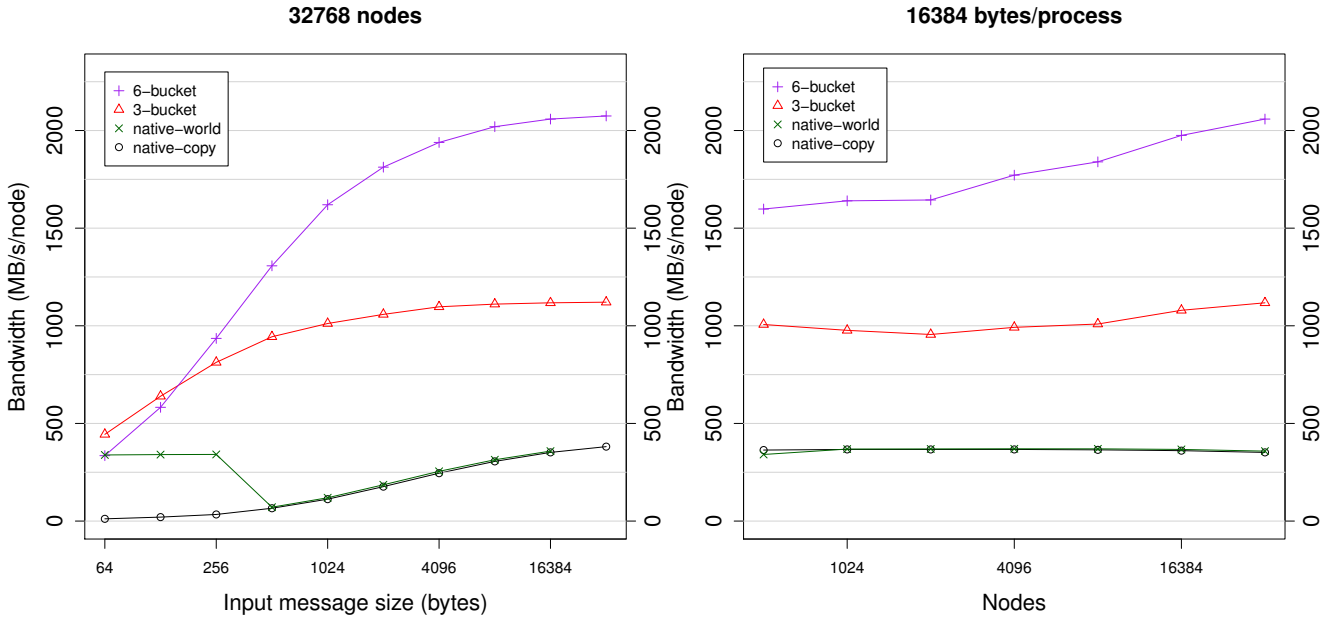

Figure 2. Performance of multi-port allgather algorithms.

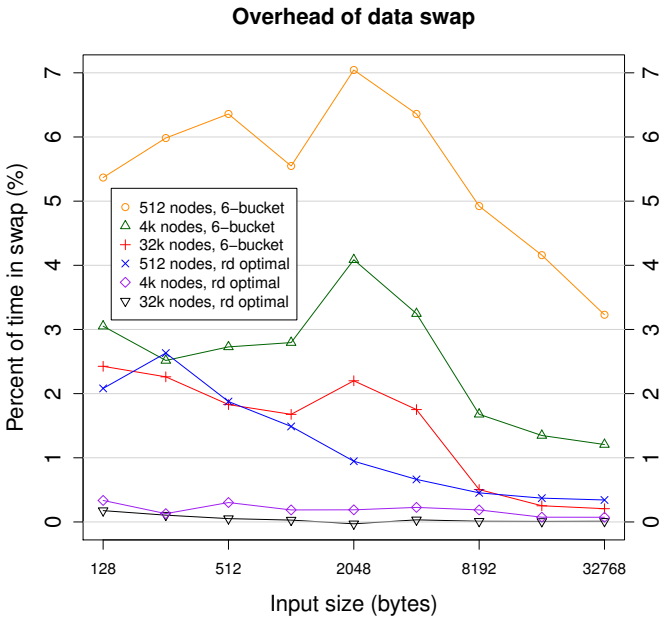

Figure 3. Overhead of input data shuffle stage

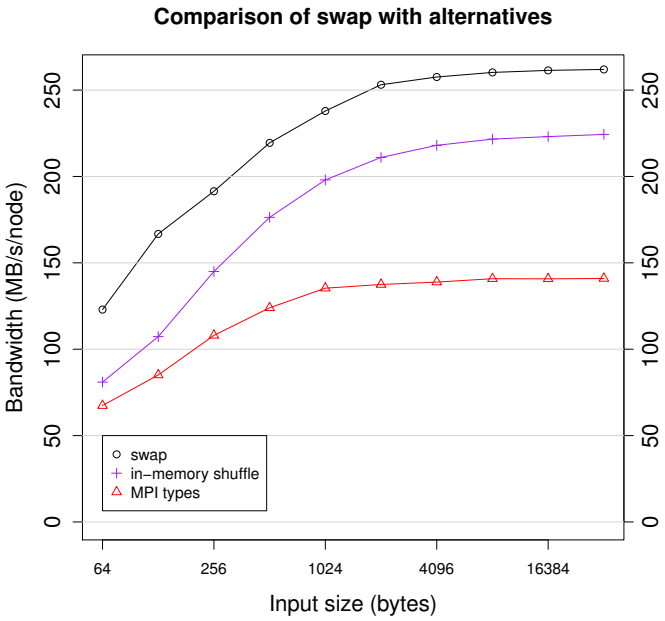

Figure 4. Alternatives to input data shuffle

MB/s, or 99.5% of the achievable link bandwidth of three ports. The 6-bucket algorithm delivers up to 2075 MB/s, which is 99% of the achievable link bandwidth of 6 ports. This is 5.5x better than the native algorithm.

Figure 3 shows the amount of time spent in the additional data-exchange steps for the optimally-remapped recursive-doubling algorithm (*rd optimal*) and the 6-way bucket algorithm. Restoring the correct order and handling arbitrary message sizes involves many more messages for the multi-port bucket algorithm. Moreover, the bulk of the bucket algorithm is about six times quicker than the recursive-doubling algorithm. This is why we observe more overhead for the bucket algorithms. In any case, the overhead is quite small and decreases as we scale the number of processes.

In Figure 4, we show the performance of two alternatives to the additional stage of communication for the recursive-doubling, distance-halving allgather operation running on 512 nodes (*swap*). We could reorder the data in memory after the recursive-doubling operation is complete (*in-memory shuffle*). We load data from bit-reversed offsets in a temporary output buffer and store the data in sequential order in the proper output buffer; this was quicker than the reverse. As we explained in the previous section, the extra stage in our non-minimal algorithms involves exchanging n to $2n$ bytes per process across the network, whereas the in-memory shuffle requires shuffling nP bytes of data in memory. We also tried using MPI types to read and write strided, non-contiguous data. Both proved slower than our method with an extra stage of communication. We chose to use the distance-halving algorithm for this experiment, since the system is likely to deliver better performance exchanging strided, non-contiguous messages than non-contiguous messages with a more complex memory layout.

4.2 Reduce-scatter performance

Figure 5 shows the performance of the reduce-scatter variations. As mentioned above, the communication pattern of reduce-scatter is the reverse of that of the allgather operation. The input vector on each process is arranged as 32-bit integer values and we sum them to form the output vector.

The native algorithm is slowest for small messages. During development, we observed that our algorithms were spending much time in memory-allocation routines. We rearranged the code to use preallocated temporary buffers; the native algorithm must allocate temporary storage. Further, our implementation only supports the case where each process receives an equal portion of the output vector; the native algorithm also supports unequal distributions of the output vector. Supporting irregular distribution of the output vector could be easily folded into the extra order-restoring stage of communication. The native algorithm could also check whether the distribution is regular with a short reduce operation.

We again see that the native algorithm is much slower on a copy of `MPI_COMM_WORLD`. The performance is equally poor for a subpartition of `MPI_COMM_WORLD`. It seems likely that the hardware-supported reduction on the tree network is only available on `MPI_COMM_WORLD`.

The non-topology-aware recursive-halving, distance-halving algorithm (*rh halving*), performs the worst. The semi-topology-aware distance-doubling algorithm (*rh doubling*) performs better, followed by the optimally-remapped recursive-halving algorithm (*rh optimal*). The bucket and 6-bucket algorithms perform best, except for smaller messages with smaller allocations. The 6-bucket algorithm performs much better than the others, except for small messages with the smaller 512-node allocation.

4.3 Measured versus predicted performance

We now compare the measured performance for several allgather algorithms with the performance predicted by our model from Section 2.1. The δ term is simply the reciprocal of the link speed (375 MB/s) for one or three-port algorithms, and 7% less for six-port algorithms. The ring algorithm on the 32k-node partition consists of 32767 stages comprised of one message per stage per process and has a latency of 206 ms with a zero-byte input message. Thus, $\alpha = 206\text{ ms}/32767 = 6.29\mu\text{s}$.

In Section 3, we developed approximate equations for the performance of different allgather algorithms; we ignored minor terms, such as the extra messages our algorithms introduce. On

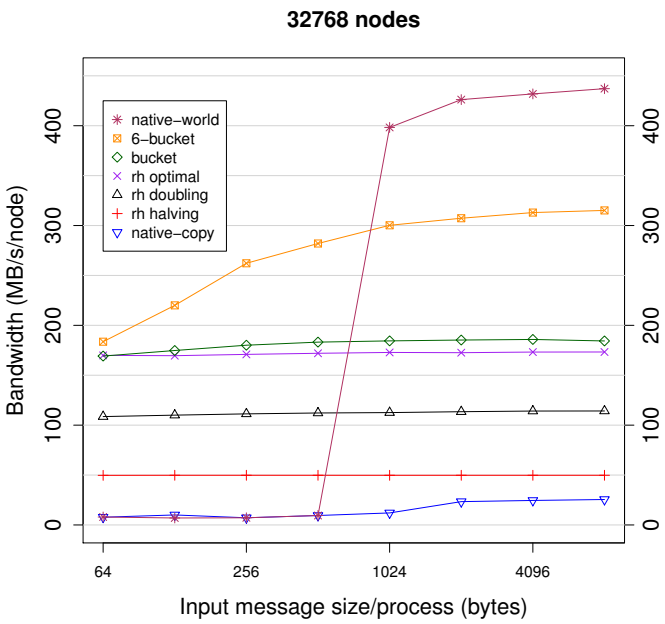

Figure 5. Performance of reduce-scatter algorithms.

large systems, these terms are negligible. In this analysis, we do not throw out any terms. Further, our equations assumed cubic torus networks. Here, we directly apply our congestion-aware performance model by counting messages and bytes; we do not use any equations. Our model here assumes that the messages sent on different ports in multi-port algorithms are synchronized; they all begin and end at the same time and have the same size. This model cannot model multi-port bucket algorithms on non-cubic torus networks. In [14], we present a more detailed model that handles this case. Essentially, the α term is still charged for every message on every port, and the β term is determined by the slowest of the 3 or 6 buckets in each phase of the algorithm. On cubic torus networks, the two models are equivalent.

In Figure 6, we compare the predicted versus measured performance of the optimally-remapped and distance-doubling recursive-doubling, and the single-port, three-port, and six-port bucket allgather algorithms with 64-byte or 32-kilobyte input messages per process. For clarity, we exclude the 64-byte 6-bucket algorithm from the plot. We introduce a factor of 0.93 into the predicted performance for the 6-port bucket algorithm, since the system penalizes the per-port bandwidth under 6-port communication by 7%. We see the least agreement between the model and the data for small messages on small partitions. This indicates that message start-up costs are optimistically modeled by our model; for larger messages or large partitions, the more accurate bandwidth term in our model dominates. This is not surprising: our model implicitly assumes zero-latency global synchronization between stages of communication and the absence of system noise or operating system effects. For small messages, this inaccuracy is magnified.

4.4 Comparison with related work

As mentioned above in Section 3.2, Jain and Sabharwal also implemented 6-way bucket algorithms [11]. Their evaluation was also on a Blue Gene/P system. Their multi-threaded allgather algorithm delivers about the same performance as our single-threaded implementation with an extra stage. Their single-threaded algorithm performs about 30% worse. On the other hand, their multi-threaded reduce-scatter algorithm performs much better than ours, since they can spread the application of the reduction operator over multiple threads. Their single-threaded reduce-scatter algorithm has similar performance to ours. Instead of an extra stage of communication, they reorder the data in memory after the allgather algorithm and before the reduce-scatter algorithm, as we do in the experiment whose results are shown in 4. It is unclear to us how they can do this reordering so efficiently, since we tried this and performance suffered greatly, as Figure 4 shows.

5. Conclusion and future work

5.1 Future work

The 6-way bucket algorithms perform very well, but have the drawback of a larger number of messages compared to the recursive-doubling or recursive-halving algorithms. We plan to investigate the performance of a hybrid algorithm. In the hybrid algorithm, we would replace the first two phases of the 6-bucket algorithm with a 3-way reordered recursive-doubling operation. The recursive-doubling portion of the algorithm would only operate in two dimension. Overall, we would have three simultaneous recursive-doubling operations followed by six simultaneous bucket operations. This is in the same vein as the work on hybrid reduction algorithms for 2-d meshes explored in [13].

The recursive-doubling algorithms only work when the number of processes is a power-of-two. We would like to use our technique to improve the efficiency of Bruck's algorithm [5], where, in each stage, process i sends data to process $i + 2^s$ and receives data from process $i - 2^s$.

We would also like to validate our algorithms on a Clos network; we chose to use a Blue Gene/P system for our first experiments since topology information is so accessible. Further, the Blue Gene/P job scheduler schedules contiguous blocks of nodes. Many job schedulers on Clos networks do not, since Clos networks generally have a larger bisection bandwidth to injection bandwidth ratio, and fewer problems naturally map to a Clos network, so topology information is less useful.

Finally, we plan to develop multi-port versions of other MPI operations, such as scatter. We do not think a multi-port recursive-doubling algorithm would be useful, since it has only a small advantage over the bucket algorithm for small messages due to its low message count. We evaluated the Van de Geijn broadcast algorithm (scatter followed by allgather) using our allgather algorithms. We found the native scatter to limit our performance; with an efficient scatter algorithm, our Van de Geijn broadcast algorithm should deliver good performance.

5.2 Conclusion

Many widely-used allgather and reduce-scatter algorithms do not reach their full potential for several reasons. They are single-ported or not topology-aware. In all of them, processes send and receive the minimal amount of data. In contrast, our algorithms are non-minimal because we add an extra stage of communication to restore the correct data order. This flexibility enables us to reorder the stages of communication or run multiple operations in parallel on a multi-port network. The overhead of the extra stage is small and this leads to substantially better performance compared to minimal-communication algorithms.

To summarize: we show that non-minimal collective algorithms deliver far better performance than minimal collective algorithms. We show that collective algorithms need not preserve the correct ordering as long as the misordering is universal, as a simple extra stage can restore the correct ordering. We present a novel semi-topology-aware algorithm that works well on a variety of networks and is optimal on Clos networks, a novel topology-aware recursive-

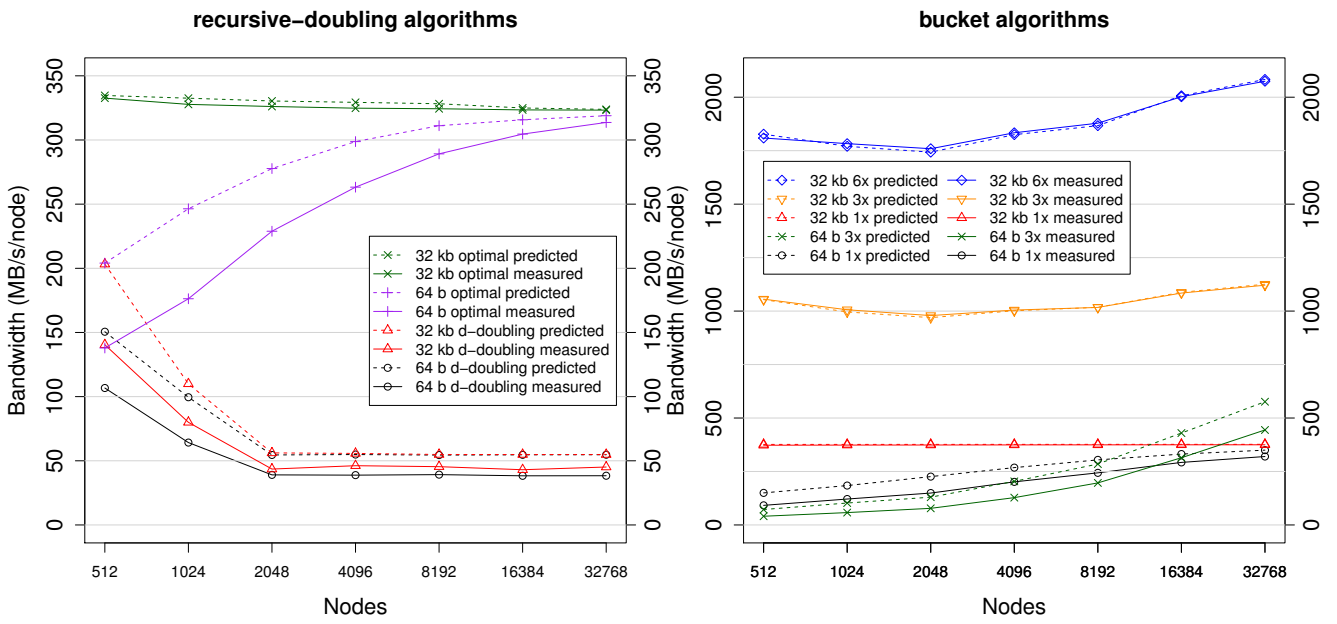

Figure 6. Measured versus predicted performance.

doubling allgather and recursive-halving reduce-scatter algorithm for mesh or torus networks, and a novel multi-port bucket algorithm for multi-port mesh or torus networks. Our allgather algorithm delivers within 1% of the maximum deliverable multi-port bandwidth of a Blue Gene/P system, which is 5.5x better than the native algorithm, and our analytical model matches measured results.

References

[1] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, routing, and packaging of efficient large-scale networks. In *Proc of the Conf. on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 41:1–41:11, New York, NY, USA, 2009. ACM.

[2] M. Barnett, D. G. Payne, and R. A. van de Geijn. Optimal broadcasting in mesh-connected architectures. Technical report, Austin, TX, USA, 1991.

[3] M. Barnett, S. Gupta, D. G. Payne, L. Shuler, R. van de Geijn, and J. Watts. Interprocessor collective communication library (intercom). In *Proc. of the Scalable High Performance Computing Conf.*, pages 357–364. IEEE Computer Society Press, 1994.

[4] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems, 2008.

[5] J. Bruck, , C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multi-port message-passing systems. In *IEEE Transactions on Parallel and Distributed Systems*, pages 298–309, 1997.

[6] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proc. 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[7] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[8] W. D. Gropp and E. Lusk. *User's Guide for* `mpich`*, a Portable Implementation of MPI*. Mathematics and Computer Science Division, Argonne National Laboratory, 1996. ANL-96/6.

[9] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In *Proc. of the 2008 IEEE Int. Conf. on Cluster Computing*. IEEE Computer Society, Oct. 2008. ISBN 978-1-4244-2640.

[10] IBM-Blue-Gene-Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2):199–220, 2008.

[11] N. Jain and Y. Sabharwal. Optimal bucket algorithms for large MPI collectives on torus interconnects. In *Proc. of the 24th ACM Int. Conf. on Supercomputing*, ICS '10, pages 27–36, 2010.

[12] H. Meuer, E. Strohmaier, J. Dongarra, and H. Simon. *TOP500 Supercomputing Sites*, 2011 (accessed May 13, 2011). `http://top500.org`.

[13] B. L. Payne, M. Barnett, R. Littlefield, D. G. Payne, and R. A. van De Geijn. Global combine on mesh architectures with wormhole routing. In *Proc. of 7 th Int. Parallel Proc. Symp*, 1993.

[14] P. Sack. *Scalable Collective Message-passing Algorithms*. PhD thesis, University of Illinois at Urbana-Champaign, 2011.

[15] H. Tang and T. Yang. Optimizing threaded MPI execution on SMP clusters. In *Proc. of 15th ACM int. conf. on supercomputing*, pages 381–392. ACM Press, 2001.

[16] R. Thakur and W. Gropp. Improving the performance of collective operations in MPICH. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 2840 in LNCS, Springer Verlag (2003) 257-267 10th European PVM/MPI User's Group Meeting*, pages 257–267. Springer Verlag, 2003.

[17] R. Thakur and R. Rabenseifner. Optimization of collective communication operations in MPICH. *Int. Journal of High Performance Computing Applications*, 19:49–66, 2005.

[18] J. L. Träff, A. Ripke, C. Siebert, P. Balaji, R. Thakur, and W. Gropp. A pipelined algorithm for large, irregular all-gather problems. *Int. Journal High Performance Computing Applications*, 24:58–68, February 2010.

[19] E. Zahavi. Fat-trees routing and node ordering providing contention free traffic for MPI global collectives. *Parallel and Distributed Processing Workshops and PhD Forum*, 0:761–770, 2011.