Efficient MPI_Bcast across Different Process Arrival Patterns

Pitch Patarasuk Xin Yuan
Department of Computer Science, Florida State University
Tallahassee, FL 32306
{patarasu, xyuan}@cs.fsu.edu

Abstract

A Message Passing Interface (MPI) collective operation such as broadcast involves multiple processes. The process arrival pattern *denotes the timing when each process arrives at a collective operation. It can have a profound impact on the performance since it decides the time when each process can start participating in the operation. In this paper, we investigate the broadcast operation with different process arrival patterns. We analyze commonly used broadcast algorithms and show that they cannot guarantee high performance for different process arrival patterns. We develop two process arrival pattern aware algorithms for broadcasting large messages. The performance of proposed algorithms is theoretically within a constant factor of the optimal for any given process arrival pattern. Our experimental evaluation confirms the analytical results: existing broadcast algorithms cannot achieve high performance for many process arrival patterns while the proposed algorithms are robust and efficient across different process arrival patterns.*

1 Introduction

A Message Passing Interface (MPI) collective operation such as broadcast involves multiple processes. The *process arrival pattern* denotes the timing when each process arrives at a collective operation [**?**]. It can have a profound impact on the performance because it decides the time when each process can start participating in the operation. A process arrival pattern is said to be *balanced* when all processes arrive at the call site at the same time (and thus, start the operation simultaneously). Otherwise, it is said to be *imbalanced*.

It has been shown in [**?**] that (1) process arrival patterns for collective operations in MPI applications are usually sufficiently imbalanced to affect the communication performance significantly; (2) it is virtually impossible for MPI application developers to control the process arrival patterns in their applications; and (3) the performance of collective communication algorithms is sensitive to process arrival patterns. Hence, for an MPI collective operation to be efficient in practice, it must be able to achieve high performance for both balanced and imbalanced process arrival patterns. Process arrival pattern is one of the most important factors that affect the performance of collective communication operations. Unfortunately, this important factor has been largely overlooked by the research and development community. Almost all existing algorithms for MPI collective operations were designed, analyzed, and evaluated under an unrealistic assumption that all processes start the operation at the same time (a balanced process arrival pattern).

Broadcast operation is one of the most common collective operations. In this operation, a message from a process, called *root*, is sent to all other processes. The MPI routine that realizes this operation is *MPI_Bcast* [**?**]. Existing broadcast algorithms that are commonly considered as efficient include the binomial tree algorithm [**?**, **?**], the pipelined algorithms [**?**, **?**, **?**, **?**], and the scatter followed by all-gather algorithm [**?**]. These algorithms have been adopted in widely used MPI libraries such as MPICH [**?**] and OPEN MPI [**?**]. While all of these algorithms perform reasonably well with a balanced process arrival pattern (they were designed under such an assumption), their performance with more practical imbalanced process arrival patterns has never been thoroughly studied.

This paper investigates the broadcast operation with different process arrival patterns. We analyze commonly used broadcast algorithms including the flat-tree algorithm, the binomial-tree algorithm, the pipelined algorithms, and the scatter-allgather algorithm, and show that all of these algorithms cannot guarantee high performance for different process arrival patterns. Our analysis follows a competitive analysis framework where the performance of an algorithm relative to the best possible algorithm is given. The performance of an algorithm under different process arrival patterns is characterized by the *competitive ratio* that bounds the ratio between the performance of the algorithm and the

performance of the optimal algorithm for any given process arrival pattern. We show that the competitive ratios of the commonly used broadcast algorithms are either unbounded or $\Omega(n)$, where n is the number of processes in the operation. This indicates that all of these algorithms may perform significantly worse than the optimal algorithms with some process arrival patterns. We developed two process arrival pattern aware algorithms for broadcasting large messages. Both algorithms have constant competitive ratios: they perform within a constant factor of the optimal *for any process arrival pattern*. We empirically evaluate the commonly used and the proposed broadcast algorithms with different process arrival patterns. The experimental evaluation confirms our analytical results: existing broadcast algorithms cannot achieve high performance for many process arrival patterns while the proposed algorithms are robust and efficient across different process arrival patterns.

The rest of the paper is organized as follows. Section 2 formally describes the process arrival pattern and introduces the performance metrics for measuring the performance of broadcast algorithms with different process arrival patterns. Section 3 presents the competitive analysis of existing broadcast algorithms. Section 4 details the proposed process arrival pattern aware algorithms. Section 5 reports the results of our experiments. Section 6 discusses the related work. Finally, Section 7 concludes the paper.

2 Background

2.1 Process arrival pattern

Let n processes, p_0, p_1, ..., p_{n-1}, participate in a broadcast operation. Without loss generality, we will assume that p_0 is the root. Let a_i be the time when process p_i arrives at the operation. The process arrival pattern can be represented by the tuple $PAP = (a_0, a_1, ..., a_{n-1})$. The average process arrival time is $\bar{a} = \frac{a_0+a_1+...+a_{n-1}}{n}$. Let f_i be the time when process p_i finishes the operation. The *process exit pattern* can be represented by the tuple $PEP = (f_0, f_1, ..., f_{n-1})$. Let δ_i be the time difference between p_i's arrival time a_i and the average arrival time $\bar{a}$, $\delta_i = |a_i - \bar{a}|$. The imbalance in the process arrival pattern can be characterized by the *average case imbalance time*, $\bar{\delta} = \frac{\delta_0+\delta_1+...+\delta_{n-1}}{n}$, and the *worst case imbalance time*, $\omega = max_i\{a_i\} - min_i\{a_i\}$. Figure **??** depicts the described parameters.

The different process arrival times at a broadcast operation can significantly affect the performance. For instance, if a broadcast algorithm requires a process to forward messages to other processes (this occurs in all tree based broadcast algorithms), the forwarding can happen only after the process arrives. The impacts of an imbalanced process arrival pattern can be better characterized by the number of

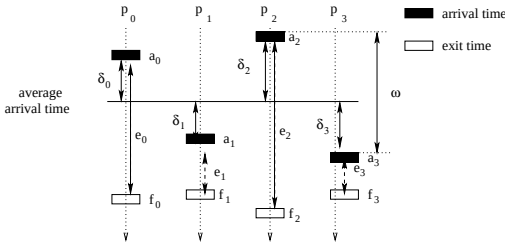


Figure 1. Process arrival pattern

messages that can be sent during the period when some processes arrive while others do not. To capture this notion, the worst case and average case imbalance times are normalized by the time to communicate one message. The normalized results are called the *average/worst case imbalance factor*. Let T be the time to communicate the broadcast message from one process to another process. The *average case imbalance factor* equals to $\frac{\bar{\delta}}{T}$ and the *worst case imbalance factor* equals to $\frac{\omega}{T}$. A worst case imbalance factor of 20 means that by the time the last process arrives at the operation, the first process may have sent 20 messages. It has been shown in [**?**] that the worst case imbalance factors of the process arrival patterns in MPI applications are in many cases much larger than the number of processes involved in the operations.

2.2 Performance metrics

Let the process arrival pattern $PAP = (a_0, ..., a_{n-1})$ and the process exit pattern $PEP = (f_0, ..., f_{n-1})$. The elapsed time of each process p_i, $0 \leq i \leq n-1$, is $e_i = f_i - a_i$. Thus, the total time for the operation is $e_0 + e_1 + ... + e_{n-1}$ and the *average per node time* for this operation is $\bar{e} = \frac{e_0+e_1+...+e_{n-1}}{n}$. The *worst case per node time* is $g = max_i\{e_i\}$.

In an application, the total time or the average per node time accurately reflects the time that the program spends on an operation. Hence, we will use the average per node time ($\bar{e}$) as the performance metric to characterize the performance of a broadcast algorithm with a given process arrival pattern. The worst case per node time (g) is also interesting. However, we consider it secondary since it does not reflect the actual time that an application spends on the operation. Note that the impact of load balancing in a broadcast operation, which is better reflected by the worst case per node time, is not clear with imbalanced process arrival patterns.

In analyzing the performance of broadcast algorithms, we assume an ideal platform such that the process exit pattern (and thus the average per node time) is a function of the broadcast algorithm and the process arrival pattern. We ignore other deterministic and non-deterministic factors that

can affect the performance. Our assumptions will be detailed in the next section. We denotes the performance (average per node time) of a broadcast algorithm r with a process arrival pattern $PAP = (a_0, ..., a_{n-1})$ as $\bar{e}(r, PAP)$.

An optimal broadcast algorithm for a given process arrival pattern PAP is an algorithm that minimizes the average per node time. Formally, the optimal average per node time for a process arrival pattern PAP is given by

$$OPT(PAP) = \min_{\texttt{r is a broadcast algorithm}} \{\bar{e}(r, PAP)\}$$

The *performance ratio* of a given broadcast algorithm r on a given process arrival pattern PAP measures how far is r from being optimal for the process arrival pattern. It is defined as the average per node time of r on PAP divided by the minimum possible average per node time on PAP.

$$PERF(r, PAP) = \frac{\bar{e}(r, PAP)}{OPT(PAP)}$$

The value for $PERF(r, PAP)$ is at least 1. It is exactly 1 if and only if the broadcast algorithm is optimal for the PAP. When a broadcast algorithm is optimized for a specific process arrival pattern, it does not provide any guarantees for other process arrival patterns. The definition of the performance ratio follows the competitive analysis framework where performance guarantees of a certain solution are provided relative to the best possible solution. The definition of performance ratio of a broadcast algorithm can be extended to be with respect to a set of process arrival patterns. Let Γ be a set of process arrival patterns, the performance ratio of a broadcast algorithm r on Γ is defined as

$$PERF(r, \Gamma) = \max_{PAP \in \Gamma} \{PERF(r, PAP)\}$$

When Γ includes all possible process arrival patterns, the performance ratio is referred to as the **competitive ratio**. The competitive ratio of an algorithm r is denoted by $PERF(r)$. The competitive ratio is the worst performance ratio that an algorithm obtains with respect to all process arrival patterns. In our analysis, the performance of a broadcast algorithm with different process arrival patterns is characterized by its competitive ratio.

3 Competitive ratios of existing broadcast algorithms

3.1 System models

The performance of broadcast algorithms is affected heavily by both the system architecture and the process arrival pattern. To focus on analyzing the impact of process arrival patterns, we make the following assumptions.

• When both the sender and the receiver are ready, the time to communicate a message of size $msize$ between any pair of processes is $T(msize)$. Communications between multiple pairs of processes can happen simultaneously and do not interfere with one another. This assumption holds when each process runs on a different compute node and all compute nodes are connected by a cross-bar switch.

• The communications follow the 1-port model. That is, when a process is ready, it can send and receive simultaneously. Most contemporary networking technology such as Ethernet, InfiniBand, and Myrinet supports this model when each compute node is equipped with one network interface card.

• When a process receives a message, it can start forwarding the message as soon as the first bit of the message is received. Furthermore, the time to send a message is proportional to the message size: $T(a \times msize) = a \times T(msize)$. These assumptions simplify the situation when a message is forwarded in a pipelined fashion: the message can be pipelined for each single bit. These assumptions underestimate the communication time by not counting the start-up overheads.

We consider two communication models: the *blocking* model and the *non-blocking* model. These two models differentiate in how the system behaves when the sender tries to send a message and the receiver is not ready for the message (e.g. not arriving at the operation yet). In the blocking model, the sender is blocked until the receiver is ready. The actual communication takes place after the receiver is ready. In the non-blocking model, the sender is not blocked even when the receiver is not ready. The actual communication takes place when the sender is ready to send the message. The message is thus buffered at the receiving end. When the receiver arrives, it can read the message from local memory. We assume that reading a message from the local memory does not take any time. Note that in both models, a process must arrive at the operation before it can forward a message.

Both the blocking and non-blocking models have practical applications. In MPI implementations, small messages are usually buffered at the receiving side and follow the non-blocking model. Large messages are typically communicated using the rendezvous protocol, which follows the blocking model.

Let a process arrival pattern $PAP = (a_0, a_1, ..., a_{n-1})$. Since nothing can happen in the broadcast operation before the root arrives at the operation, we will assume that $a_0 \leq a_i, 1 \leq i \leq n-1$. Note that p_0 is the root. In practice, when $a_i < a_0$, we can treat it as if $a_i = a_0$, and add $a_i - a_0$ in calculating the total communication time. We denote $\Delta = max_i\{a_i\} - a_0$.

3.2 Optimal performance for a given process arrival pattern

To perform the competitive analysis of the algorithms, we will need to establish the optimal broadcast performance for a process arrival pattern. The following lemmas bound the optimal performance.

Lemma 1: Consider broadcasting a message of size $msize$ with a process arrival pattern $PAP = (a_0, ..., a_{n-1})$. Under the non-blocking model, $OPT(PAP) \leq T(msize)$. Under the blocking model, $OPT(PAP) \leq \frac{\Delta}{n} + T(msize)$, where $\Delta = max_i\{a_i\} - a_0$.

Proof: We prove by constructing an algorithm, called *nearopt*, such that $\bar{e}(nearopt, PAP) = T(msize)$ under the non-blocking model and $\bar{e}(nearopt, PAP) = \frac{\Delta}{n} + T(msize)$ under the blocking model.

Let us sort the arrival times in ascending order and let the sorted arrival times be $(a_0, a'_1, a'_2, ..., a'_{n-1})$ and the corresponding processes be p_0, p'_1, ..., p'_{n-1}. Algorithm *nearopt*, depicted in Figure **??**, works as follows: (1) p_0 sends the broadcast message to p'_1 and then exits the operation; and (2) p'_i, $1 \leq i \leq n-2$, forwards the message it receives to p'_{i+1} and then exits the operation.

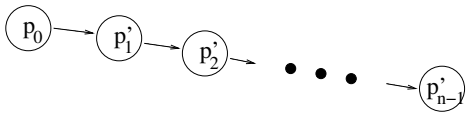


Figure 2. The near optimal broadcast algorithm (*nearopt*)

Under the non-blocking model, since $a'_i \leq a'_{i+1}$, each process can start forwarding data as soon as it reaches the operation. Hence, each process p_i, $0 \leq i \leq n-1$, will exit the operation at time $f_i = a_i + T(msize)$. The total time for this operation is $n \times T(msize)$ and $\bar{e}(nearopt, PAP) = \frac{n \times T(msize)}{n} = T(msize)$. Under the non-blocking model, $OPT(PAP) \leq \bar{e}(nearopt, PAP) = T(msize)$.

Under the blocking model, p_0 will need to wait until p'_1 arrives before it can start sending the data. Thus, $f_0 = a'_1 + T(msize)$. A process p'_i, $1 \leq i \leq n-2$, will need to wait until p'_{i+1} before it can send the data: $f'_i = a'_{i+1} + T(msize)$. The last process p'_{n-1} will exit at $a'_{n-1} + T(msize)$. The total time for this operation is thus, $f_0 - a_0 + f'_1 - a'_1 + ... + f'_{n-1} - a'_{n-1} = \Delta + n \times T(msize)$ and $\bar{e}(nearopt, PAP) = \frac{\Delta}{n} + T(msize)$. Hence, under the blocking model, $OPT(PAP) \leq \bar{e}(nearopt, PAP) = \frac{\Delta}{n} + T(msize)$. □

While the $nearopt$ is not always optimal, it is very close to optimal for any process arrival pattern under both models as shown in the following lemma.

Lemma 2: Under the non-blocking communication model, $OPT(PAP) \geq \frac{n-1}{n} \times T(msize)$. Under the blocking model, $OPT(PAP) \geq \frac{\Delta + (n-1) \times T(msize)}{n}$. □

The proof is omitted due to the space limitation. Basically, for the non-blocking case, the broadcast operation must communication at least $(n-1) \times msize$ messages (each of the receivers must receive a $msize$ sized message) and the total time for this operation is at least $T((n-1) \times msize)$. Hence, $OPT(PAP) \geq \frac{T((n-1) \times msize)}{n} = \frac{n-1}{n} \times T(msize)$. The blocking case follows a similar argument.

3.3 Competitive ratios of common broadcast algorithms

We analyze common broadcast algorithms including the flat tree algorithm (used in OPEN MPI [**?**]), the binomial tree broadcast algorithm (used in OPEN MPI and MPICH [**?**]), the linear-tree pipelined broadcast algorithm (used in SCI-MPICH [**?**]), and the scatter-allgather broadcast algorithm (used in MPICH). In the flat tree algorithm (Figure **??** (a)), the root sends the broadcast message to each of the receivers one-by-one. In the binomial tree algorithm (Figure **??** (b)), broadcast follows a hypercube communication pattern. There are $lg(n)$ steps in this algorithm. In the first step, p_0 sends to p_1; in the second step, p_0 sends to p_2 and p_1 sends to p_3; in the i-th step, processes p_0 to $p_{2^{i-1}-1}$ have the data, each of the process p_i, $0 \leq i \leq 2^{i-1} - 1$, sends the message to process $p_{i \oplus 2^{i-1}}$, where $\oplus$ is the exclusive or operator. In the linear tree pipelined algorithm, the $msize$-byte message is partitioned into X segments of size $\frac{msize}{X}$. The communication is done by arranging the processes into a linear array: p_i, $0 \leq i \leq n-2$, sends to p_{i+1}, as shown in Figure **??** (c). Broadcasting the $msize$-byte message is realized by X pipelined broadcasts of segments of size $\frac{msize}{X}$. In the *scatter followed by all-gather* algorithm, the $msize$-byte message is first distributed to the n processes by a scatter operation (each machine gets $\frac{msize}{n}$-byte data). After that, an all-gather operation is performed to combine messages to all processes. We denote the flat tree algorithm as *flat*, the binomial algorithm as *binomial*, the linear tree pipelined algorithm as *linear_p*, and the scatter-allgather algorithm as *sca-all*.

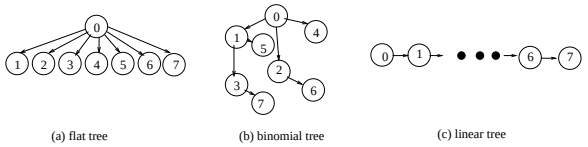


Figure 3. Broadcast algorithms

Lemma 3: Under the non-blocking model, the competitive ratio of any broadcast algorithm that requires a receiver to forward a message is unbounded (can potentially be infinite).

Proof: Let a broadcast algorithm r requires a process p_i to forward a message to p_j in the operation. To show that $PERF(r)$ is unbounded, we must show that there exists a process arrival pattern PAP such that $\frac{\bar{e}(r,PAP)}{OPT(PAP)}$ is unbounded. Let us denote Λ as a very large number. We construct the process arrival pattern where all processes except p_i arrive at the same time as p_0. Process p_i arrives at a much later time $a_0 + \Lambda$. $PAP = (a_0, a_0, .., a_i, .., a_0 = a_0, a_0, .., a_0 + \Lambda, .., a_0)$. Under this arrival pattern, process p_j can only finish the operation after it receives the message forwarded by p_i. Hence, $f_j \geq a_0 + \Lambda$. Thus, $\bar{e}(r, PAP) = \frac{e_0+e_1+...e_{n-1}}{n} \geq \frac{e_j}{n} \geq \frac{\Lambda}{n}$. From Lemma 1, $OPT(PAP) \leq T(msize)$. Hence, $PERF(r) \geq \frac{\bar{e}(r,PAP)}{OPT(PAP)} \geq \frac{\frac{\Lambda}{n}}{T(msize)} = \frac{\Lambda}{n \times T(msize)}$. Since Λ is independent of n or $T(msize)$, $\frac{\Lambda}{n \times T(msize)}$ can potentially be infinite and $PERF(r)$ is unbounded. □

We will use the notion $PERF(r) = \infty$ to denote that $PERF(r)$ is unbounded.

Theorem 1: In both the blocking and non-blocking models, *PERF(flat)* = $\Omega(n)$.

Proof: Consider first the non-blocking model with a balanced process arrival pattern $PAP = (a_0, a_0, ..., a_0)$. Since the root sequentially sends the message to each of the receivers, $f_0 = a_0 + (n-1) \times T(msize)$; $f_1 = a_0 + T(msize)$; $f_2 = a_0 + 2 \times T(msize)$; ...; $f_{n-1} = a_0 + (n-1) \times T(msize)$. Thus, the total time for this operation is $f_0 - a_0 + f_1 - a_0 + ... + f_{n-1} - a_0 = (n-1) \times T(msize) + \sum_{i=1}^{n-1} i \times T(msize) \geq \frac{n(n-1)}{2} \times T(msize)$. Hence, $\bar{e}(flat, PAP) \geq \frac{n-1}{2} \times T(msize)$. From Lemma 1, $OPT(PAP) \leq T(msize)$. We have $PERF(flat) \geq PERF(flat, PAP) = \frac{\bar{e}(flat,PAP)}{OPT(PAP)} \geq \frac{n-1}{2}$. Hence, $PERF(flat) = \Omega(n)$.

In the blocking model, consider the same PAP as the non-blocking case. Since all processes arrive at the same time, $\Delta = 0$. With the flat-tree algorithm, the total time for both the blocking and non-blocking models are the same. From Lemma 1: $OPT(PAP) \leq \frac{\Delta}{n} + T(msize) = T(msize)$. Hence, in the blocking model, $PERF(flat) = \Omega(n)$. □

Theorem 2: In the non-blocking model, *PERF(binomial)* = ∞. In the blocking model, *PERF(binomial)* = $\Omega(n)$.

Proof: In the non-blocking model, since the binomial tree algorithm requires forwarding of messages, from Lemma 3, $PERF(binomial) = \infty$.

In the blocking model, consider the arrival pattern PAP where the $lg(n)$ direct children of the root arrive at $a_0 + \Lambda$ and all other processes arrive at a_0. Let us assume $\frac{\Lambda}{n} \geq T(msize)$. From Lemma 1, $OPT(PAP) \leq \frac{\Lambda}{n} + T(msize) \leq \frac{2\Lambda}{n}$. Clearly, in this operation, all processes will exit the operation after $a_0 + \Lambda$. Hence, the total time for this operation is larger than $(n - lg(n))\Lambda \geq \frac{n}{2}\Lambda$. The average per node time $\bar{e}(binomial, PAP) \geq \frac{\Lambda}{2}$. $PERF(binomial) \geq PERF(binomial, PAP) = \frac{\bar{e}(binomial,PAP)}{OPT(PAP)} \geq \frac{n}{4}$. Hence, $PERF(binomial) = \Omega(n)$.

Theorem 3: In the non-blocking model, *PERF(linear_p)* = ∞. In the blocking model, *PERF(linear_p)* = $\Omega(n)$.

Proof: The linear tree pipelined algorithm requires the intermediate processes to forward messages. From Lemma 3, in the non-blocking model, $PERF(linear\_p) = \infty$.

In the blocking model, consider the process arrival pattern where process p_1 arrives much later than all other nodes: $PAP = (a_0, a_0 + \Lambda, a_0, ..., a_0)$. Let Λ be much larger than $T(msize)$. Following the linear tree, all processes can receive the broadcast message after $a_0 + \Lambda$. Hence, the total time is at least $(n-1)\Lambda \geq \frac{n}{2}\Lambda$. Following the similar argument as that in the binomial tree, $PERF(linear\_p) = \Omega(n)$. □

Theorem 4: In the non-blocking model, *PERF(sca-all)* = ∞. In the blocking model, *PERF(sca-all)* = $\Omega(n)$.

Proof: In the scatter-allgather algorithm, there is an implicit synchronization: in order for any process to finish the all-gather operation, all process must arrive at the operation. This implicit synchronization applies to both the blocking and non-blocking models. Consider arrival pattern when p_1 arrives late: $PAP = (a_0, a_0 + \Lambda, a_0, ..., a_0)$. For both blocking and non-blocking models, due to the implicit synchronization, the total time for this algorithm is at least $(n-1)\Lambda \geq \frac{n}{2} \times \Lambda$ and the per node time is at least $\frac{\Lambda}{2}$.

In the non-blocking model: since $OPT(PAP) \leq T(msize)$, *PERF(sca-all, PAP)* $\geq \frac{\Lambda}{2T(msize)}$, which is unbounded. Hence, *PERF(sca-all)* $\geq$ *PERF(sca-all, PAP)* $= \infty$. In the blocking model, when $\frac{\Lambda}{n} \geq T(msize)$, $OPT(PAP) \leq \frac{\Lambda}{n} + T(msize) \leq \frac{2\Lambda}{n}$. *PERF(sca-all, PAP)* $\geq \frac{n}{4}$. Hence, *PERF(sca-all)* = $\Omega(n)$. □

Theorems 1 to 4 show that the competitive ratios of all of these commonly used algorithms are either unbounded or $\Omega(n)$ for both the blocking or non-blocking models, which indicates that these algorithms cannot guarantee high performance for different process arrival patterns: all of these algorithms can potentially perform much worse than the optimal algorithm for some process arrival patterns.

4 Process arrival pattern aware algorithms

We present two new broadcast algorithms, one for each communication model, with constant competitive ratios:

these algorithms guarantee high performance for any process arrival patterns. The new algorithms are designed for broadcasting large messages. The idea is to add control messages to make the processes aware of and adapt to the process arrival pattern. Note that any efficient algorithm will not wait until all processes arrive before taking any action. Such an algorithm must make on-line decisions as processes arrive: hence, the *nearopt* algorithm in Figure **??** cannot be used.

Since the algorithms are designed for broadcasting large messages, we will assume that sending and receiving a small control message do not take time. Moreover, under the assumption in Section 3.1, broadcasting to a group of processes when all receivers are ready will take $T(msize)$ time (e.g. using the *nearopt* algorithm). In practice, the linear tree pipelined broadcast algorithm can have a communication time very close to $T(msize)$ when the message size is sufficiently large [**?**]. We will call the broadcast to a sub-group of processes *sub-group broadcast*. In a sub-group broadcast, only the root knows the group members. Hence, the root will need to initiate the operation. In practice, this can be done efficiently. Consider for example using the linear tree pipelined broadcast algorithm for a sub-group broadcast. The root can send a header that contains the list of receivers in the sub-group before the broadcast message. When a receiver receives the headers, it will know how to form the linear tree by using the information in the header.

The algorithm for the non-blocking model, called *arrival_nb*, is shown in Figure **??**. When a receiver arrives at the operation, it checks to see whether the broadcast message has been received. If the message has been received, it copies the message from the local memory to the output buffer and the operation is completed. If the message has not been received, the receiver sends a control message ARRIVED to the root and waits for the sub-group broadcast initiated by the root to receive the message. The root repeatedly checks whether some process arrives. If some processes arrive, it initiates a sub-group broadcast among the processes. If no process arrives, the root picks a receiver that have not been sent the broadcast message and sends the message to the receiver.

Theorem 5: Under the assumptions in Section 3.1, ignoring the control message overheads, *PERF(arrival_nb)* ≤ 3.

Proof: In $arrival\_nb$, the algorithm works in rounds, in each round, the root either sends a message to one node, or performs a sub-group broadcast. In both case, $T(msize)$ will be incurred in each round. Since at most $n-1$ rounds are performed by the root, $f_0 \leq a_0 + (n-1)T(msize)$.

Consider a receiving process, p_i, $1 \leq i \leq n-1$. In the worst case, p_i must wait one full round before it starts participating in the sub-group broadcast. It takes

Receiving process:
(1) Check if the message has been buffered locally
(2) If yes, copy the message and exit;
(3) Else, send control message ARRIVED to the root.
(4) Participate in the sub-group broadcast initiated by the root to receive the broadcast message.

Root process:
(1) While (some receivers have not been sent the message)
(2) Check if any ARRIVED messages have arrived
(3) Discard the ARRIVED messages from receivers who have been sent the broadcast message
(4) if (a group of receivers have arrived)
(5) Initiate a sub-group broadcast
(6) else
(7) Select a process p_i that has not been sent the message and send the broadcast message to p_i

Figure 4. Broadcast algorithm for the non-blocking model ($arrival\_nb$)

p_i at most $2T(msize)$ to complete the operation: $f_i \leq a_i + 2T(msize)$. Hence, the total time among all processes in the operation is no more than $(n-1)T(msize) + 2(n-1)T(msize)$ and the average per node time is no more than $\frac{3(n-1)}{n} \times T(msize)$. This applies to any process arrival pattern, PAP. From Lemma 2, $OPT(PAP) \geq \frac{n-1}{n}T(msize)$. $PERF(arrival\_nb) \leq 3$. □

The algorithm for the blocking model, called *arrival_b* is shown in Figure **??**. When a receiver arrives at the operation, it sends a control message ARRIVED to the root and waits for the sub-group broadcast initiated by the root to receive the broadcast message. The root repeatedly checks whether some process arrives. If some processes arrive, it initiates a sub-group broadcast among the processes. Otherwise, it just waits until some processes arrive.

Receiving process:
(1) Send control message ARRIVED to the root.
(2) Participate in the sub-group broadcast initiated by root to receive the broadcast message.

Root process:
(1) While (some receivers have not been sent the message)
(2) Check if any ARRIVED messages have arrived
(3) if (a group of receivers have arrived)
(4) Initiate a sub-group broadcast

Figure 5. Broadcast algorithm for the blocking model ($arrival\_b$)

Theorem 6: Under the assumptions in Section 3.1, ignoring the control message overheads, *PERF(arrival_b)* ≤ 3.

Proof: In $arrival\_b$, the algorithm works in rounds. In each round, the root waits for some processes to arrive and performs the sub-group broadcast. Consider a receiving process, p_i, $1 \leq i \leq n-1$. In the worst case, after p_i arrives at the operation, it must wait $T(msize)$ time before it starts participating in the sub-group broadcast. Hence, it takes p_i at most $2T(msize)$ to complete the operation: $f_i \leq a_i + 2T(msize)$.

The root exits the operation only after the last process receives the broadcast message. After the last process arrives, it will wait at most $2T(msize)$ to receive the message. Hence, $f_0 \leq a_0 + \Delta + 2T(msize)$. Hence, the total time among all processes in the operation is no more than $\Delta + 2T(msize) + 2(n-1)T(msize)$ and the average per node time is no more than $\frac{\Delta+2n\times T(msize)}{n}$. This applies to any process arrival pattern PAP. From Lemma 2, $OPT(PAP) \geq \frac{\Delta+(n-1)\times T(msize)}{n}$. Hence, $PERF(arrival\_b) \leq \frac{\Delta+2n\times T(msize)}{\Delta+(n-1)\times T(msize)} \leq 3$. □

5 Performance study

The study is performed on two clusters: *Draco* and *Cetus*. The Draco cluster has 16 compute nodes with a total of 128 cores. Each node is a Dell PowerEdge 1950 with two 2.33GHz Quad-core Xeon E5345's (8 cores per node) and 8GB memory. The compute nodes are connected by a 20Gbps InfiniBand DDR switch. The MPI library in this cluster is mvapich-0.9.9. The Cetus cluster is a 16-node Ethernet switched cluster. Each node is a Dell Dimension 2400 with a 2.8GHz P4 processor and 640MB memory. All nodes run Linux (Fedora) with 2.6.5-1.358 kernel. The Ethernet card in each machine is Broadcom BCM 5705 1Gbps/100Mbps/10Mbps card with the driver from Broadcom. The Ethernet switch connecting all nodes is a Dell Powerconnect 2724 (24-port, 1Gbps switch). The MPI library in this cluster is MPICH2-1.0.4.

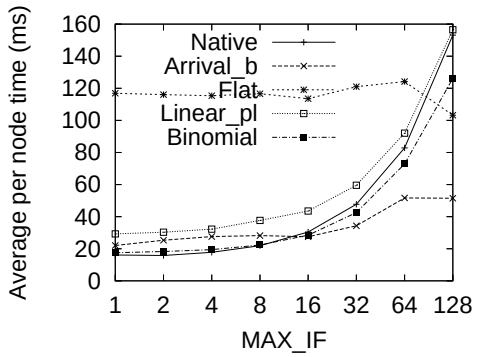
To evaluate the performance of the existing and proposed broadcast algorithms, we implemented these algorithms over MPI point-to-point routines. Since we consider broadcasting large messages and almost all existing implementations for broadcasting large messages follow the blocking model, our evaluation also focuses on the blocking model. We implemented the flat tree algorithm (*flat*), the linear-tree pipelined algorithm (*linear_p*), the binomial tree algorithm (*binomial*), and the proposed process arrival pattern aware algorithm (*arrival_b*). In addition, we also consider the native algorithm from the MPI library (denoted as *native*). In the implementation of *arrival_b*, there can be multiple methods to perform the sub-group broadcast. We choose the most efficient scheme for each platform: the *linear_p* with a segment size of $8KB$ on Cetus and the scatter-allgather algorithm for Draco. Besides studying the blocking model, we also implemented two algorithms that follow the non-blocking model on Draco when each node runs one process: the non-blocking flat tree algorithm (denoted as *flat_nb*) and the process arrival pattern aware non-blocking algorithm (denoted as *arrival_nb*). The non-blocking communication is achieved by chopping a large point-to-point message into a set of small messages to avoid the rendezvous protocol in MPI and other factors that cause communications to block. Note that using the *MPI_Rsend* routine can avoid the rendezvous protocol. However, for some reason, this routine is still blocking in the broadcast operation.

```
(1) r = rand() % MAX_IF;
(2) for (i=0; i<ITER; i++) {
(3)    MPI_Barrier (...);
(4)    for (j=0; j<r; j++) {
(5)      /* computation time equals to one message time */
(6)    }
(7)    t0 = MPI_Wtime();
(8)    MPI_Bcast(...);
(9)    elapse += MPI_Wtime() - t0;
(10)}
```

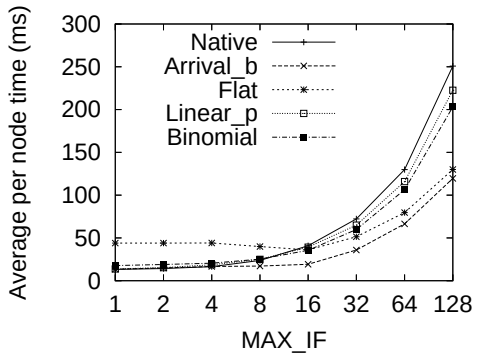
Figure 6. Code segment for studying controlled random process arrival patterns

We perform extensive experiments to study the broadcast algorithms with many different process arrival patterns. We will present two representative experiments. The first study investigates the performance of the algorithms with random process arrival patterns. The second study investigates the performance when a small subset of processes arrive at the operation later than other processes. The code segment in the benchmark to study random process arrival patterns is shown in Figure **??**. In this benchmark, a random value that is bounded by a constant *MAX_IF* is generated and stored in r. For different processes, r is different since different processes have different seeds. The controlled random process arrival pattern is created by first calling an *MPI_Barrier* and then executing the computation in the loop in lines (4) to (6) whose duration is controlled by the value of r. The loop body in line (5) executes roughly the time to send one broadcast message between two nodes. Hence, the maximum imbalance factor (defined in Section 2) is at most *MAX_IF* in this experiment. The code to investigate the effect with a few late arriving processes is similar except that the loop in lines (4)-(6) is always executed *MAX_IF* times and that each process (except the root) has a probability to run this loop.

In reporting the results, we fix either the broadcast mes-



(a) Draco (2MB)

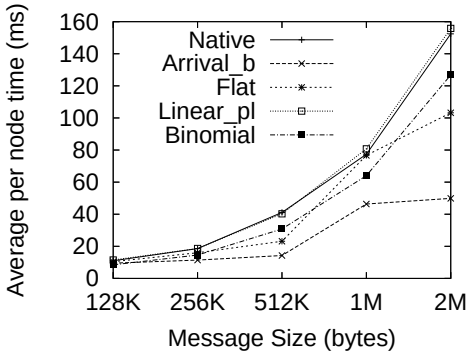


(b) Cetus (256KB)

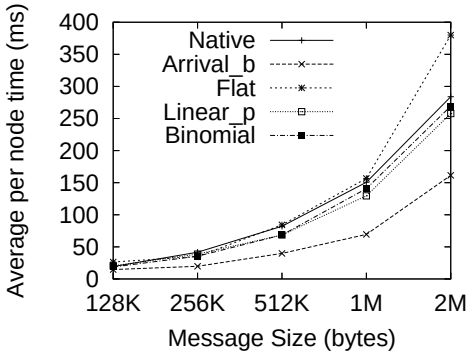
Figure 7. Performance with controlled random process arrival patterns

sage size or the MAX_IF value. Each data point in our results is an average of twenty random samples (twenty random process arrival patterns). In studying the blocking model, $n = 128$ on Draco (8 processes per node) and $n = 16$ on Cetus. In studying the non-blocking model, $n = 16$ on Draco.

Figure **??** shows the performance of different algorithms with controlled random process arrival patterns. The broadcast message size is 2MB on Draco and 256KB on Cetus. On both clusters, when the imbalanced factor is small (*MAX_IF* is small), $arrival\_b$ performs slightly worse than the best performing algorithms. Even in such cases, $arrival\_b$ is competitive: the performance is only slightly worse than the best performing algorithm mainly due to the overheads introduced. When the process arrival pattern becomes more imbalanced (*MAX_IF* is larger), the $arrival\_b$ significantly out-performs all other algorithms. The performance difference is large when *MAX_IF* is large.



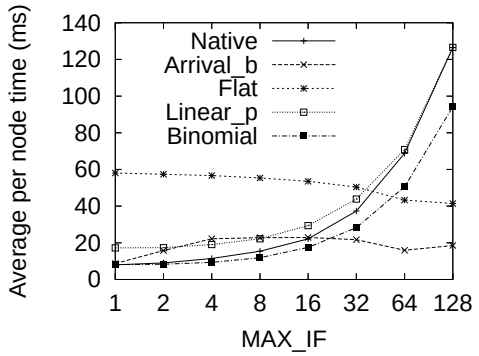
(a) Draco



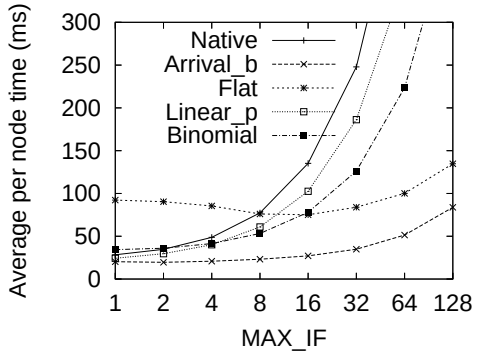
(b) Cetus

Figure 8. Broadcast with different message size with $MAX\_IF = n$

Figure **??** shows the performance with different message sizes. We fixed *MAX_IF* $= n$. On Draco, $n = 128$ and on Cetus, $n = 16$. As can be seen in the figure, on both clusters, $arrival\_b$ maintains its advantage for a wide range of message sizes. When the message size is not sufficiently large, *arrival_b* may not achieve a good performance due to the control message overheads it introduces. This can be seen in the 128KB case on Draco.



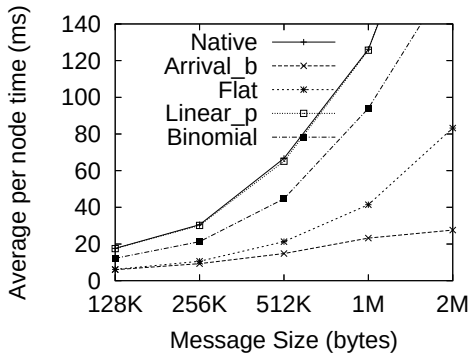
(a) Draco (1MB)



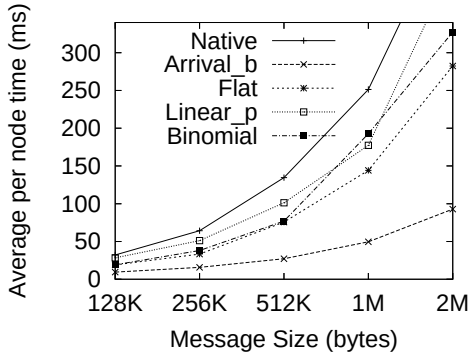
(b) Cetus (512KB)

Figure 9. Performance when 20% of processes arrive late

Figure **??** shows the performance when a small subset of processes arrive late. In this experiment, the sender always arrives early and each of the receiving process has a 20% probability to arrive late. The time for late arrival is roughly equal to the time to send *MAX_IF* messages. The broadcast message size is 1MB on Draco and 512KB on Cetus. As can be seen in the figure, *binomial*, *linear_p*, and *native* behave similarly and perform worse than *arrival_b* when *MAX_IF* is large. The *flat* and *arrival_b* have similar characteristics with *arrival_b* performing much better. Figure **??** shows the performance on different message sizes with the same settings. *MAX_IF* $= n$ (the number of processes) and the broadcast message size varies. *Arrival_b* is efficient for a wide range of message sizes.



(a) Draco

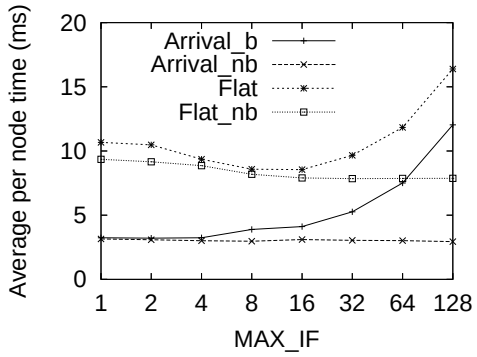


(b) Cetus

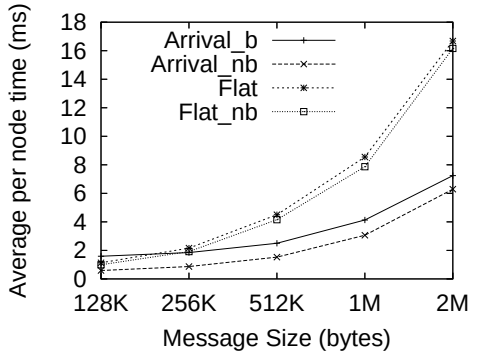
Figure 10. Performance when 20% of processes arrive late (*MAX_IF* $= n$)

Figure **??** shows the performance of the non-blocking algorithms. This figure shows the cases when 20% of processes arrive late. There are two trends in the figure. First, the non-blocking algorithm performs better than the corresponding blocking algorithm. This is because the reduction of the waiting time in a non-blocking algorithm. Second, with a non-blocking algorithm, the average per node time is not very sensitive to the process arrival pattern: see the flat line in Figure **??** (a), which indicates that the non-blocking model is more efficient than the blocking model. Unfortunately, making point-to-point communications with a large amount of data non-blocking is very difficult: many factors such as the limited system buffers in various levels of soft-

ware can make the communication blocking.



(a) 1MB



(b) *MAX_IF* $= 16$

Figure 11. Non-blocking case (Draco with each node running one process ($n = 16$))

Our experimental results confirm the analytical findings. The commonly used algorithms perform poorly when the process arrival pattern has a large imbalance factor. On the other hand, the proposed process arrival pattern aware algorithms react much better to imbalanced process arrival patterns. Moreover, when the imbalance factor is large, using our algorithms improves the performance very significantly.

6 Related Work

The broadcast operation in different environments has been extensively studied. Many algorithms were developed for topologies used in parallel computers such as meshes and hypercubes [**?**, **?**]. Generic broadcast algorithms including the binomial tree algorithms [**?**, **?**], the pipelined broadcast algorithms [**?**, **?**, **?**], the scatter-allgather algorithm [**?**], have been developed and adopted in common MPI libraries. These algorithms achieve high performance with a balanced process arrival pattern. However, their performance with imbalanced process arrival patterns has not been thoroughly studied.

The characteristics and impacts of process arrival patterns in MPI applications were studied in [**?**]. Our work is motivated by the results in [**?**]. While process arrival pattern is an important factor, few collective communication algorithms have been developed to deal with imbalanced process arrival patterns. The only algorithms that can adapt to different process arrival patterns (known to us) were published in [**?**], where algorithms for all-reduce and barrier can automatically change their logical topologies based on the process arrival pattern. We formally analyze the performance of existing broadcast algorithms with different process arrival patterns. This is the first time such analysis is performed on any collective algorithms.

7 Conclusion

We study efficient broadcast schemes with different process arrival patterns. We show that existing algorithms cannot guarantee high performance with different process arrival patterns, and develop new broadcast algorithms that achieve constant competitive ratios for the blocking and non-blocking models. The experiment results show that the proposed algorithms are robust and efficient for different process arrival patterns.

Acknowledgement

This work is supported in part by National Science Foundation (NSF) grants: CCF-0342540, CCF-0541096, and CCF-0551555.

References

[1] A. Faraj, P. Patarasuk, and X. Yuan, "A Study of Process Arrival Patterns for MPI Collective Operations," the *21th ACM ICS*, pages 168-179, June 2007.

[2] S. L. Johnsson and C. T. Ho, "Optimum Broadcasting and Personalized Communication in Hypercube." *IEEE Trans. on Computers*, 38(9):1249-1268, 1989.

[3] R. Kesavan, K. Bondalapati, and D.K. Panda, "Multicast on Irregular Switch-based Networks with Wormhole Routing." *IEEE HPCA*, Feb. 1997.

[4] H. Ko, S. Latifi, and P. Srimani, "Near-Optimal Broadcast in All-port Wormhole-routed Hypercubes using Error Correcting Codes." *IEEE TPDS*, 11(3):247-260, 2000.

[5] A. Mamidala, J. Liu, D. Panda. "Efficient Barrier and Allreduce on InfiniBand Clusters using Hardware Multicast and

Adaptive Algorithms." *IEEE Cluster*, pages 135-144, Sept. 2004.

[6] P.K. McKinley, H. Xu, A. Esfahanian and L.M. Ni, "Unicast-Based Multicast Communication in Wormhole-Routed Networks." *IEEE TPDS*, 5(12):1252-1264, Dec. 1994.

[7] MPICH - A Portable Implementation of MPI. http://www.mcs.anl.gov/mpi/mpich.

[8] The MPI Forum, "The MPI-2: Extensions to the Message Passing Interface," Available at http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html.

[9] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/.

[10] P. Patarasuk, A. Faraj, and X. Yuan, "Pipelined Broadcast on Ethernet Switched Clusters." The *20th IEEE IPDPS*, April 25-29, 2006.

[11] *SCI-MPICH: MPI for SCI-connected Clusters*. Available at: www.lfbs.rwth-aachen.de/ users/joachim/SCI-MPICH/pcast.html.

[12] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimizing of Collective Communication Operations in MPICH," *ANL/MCS-P1140-0304*, Mathematics and Computer Science Division, Argonne National Laboratory, March 2004.

[13] S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications," In *Proceedings of SC'00: High Performance Networking and Computing*, 2000.

[14] J. Watts and R. Van De Gejin, "A Pipelined Broadcast for Multidimensional Meshes." *Parallel Processing Letters*, 5(2)281-292, 1995.