Algorithmica (2009) 53: 425–453
DOI 10.1007/s00453-007-9113-7

An Approximation Algorithm and Dynamic Programming for Reduction in Heterogeneous Environments

Pangfeng Liu · May-Chen Kuo · Da-Wei Wang

Received: 20 November 2006 / Accepted: 25 September 2007 / Published online: 30 October 2007

Abstract Network of workstation (NOW) is a cost-effective alternative to massively parallel supercomputers. As commercially available off-the-shelf processors become cheaper and faster, it is now possible to build a cluster that provides high computing power within a limited budget. However, a cluster may consist of different types of processors and this heterogeneity complicates the design of efficient collective communication protocols. For example, it is a very hard combinatorial problem to find an optimal reduction schedule for such heterogeneous clusters. Nevertheless, we show that a simple technique called *slowest-node-first* (SNF) is very effective in designing efficient reduction protocols for heterogeneous clusters. First, we show that SNF is actually a 2-approximation algorithm, which means that an SNF schedule length is always within twice of the optimal schedule length, no matter what kind of cluster is given. In addition, we show that SNF does give the optimal reduction time when the cluster consists of two types of processors, when the ratio of communication speed between them is at least two. When the communication speed ratio is less than two, we develop a dynamic programming technique to find the optimal schedule. Our dynamic programming utilizes the monotone property of the objective function, and can significantly reduce the amount of computation time. Finally, combined with an approximation algorithm for broadcast 2004, we propose an all-reduction algorithm which sends the reduction answer to *all* processors, with approximation ratio 3.5.

P. Liu · M.-C. Kuo
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan

P. Liu (✉)
Graduated Institute of Networking and Multimedia, National Taiwan University, Taipei, Taiwan
e-mail: pangfeng@csie.ntu.edu.tw

D.-W. Wang
Institute of Information Science, Academia Sinica Nankang, Taipei, Taiwan

We conduct three groups of experiments. First, we show that SNF performs better than the built-in `MPI_Reduce` in a test cluster. Second, we observe a factor of 93 times saving in computation time to find the optimal schedule, when compared with a naive dynamic programming implementation. Thirdly, we apply the theoretical results to a branch-and-bound search and show that they can reduce the search time of the optimal reduction schedule by a factor of 500, when the cluster has three kinds of processors.

Keywords Heterogeneous workstation cluster · Scheduling optimization · Reduction protocol · Slowest-node-first heuristic · Dynamic programming · Branch-and-bound search

1 Introduction

Network of workstation (NOW) is a cost-effective alternative to massively parallel supercomputers [1]. As commercially available off-the-shelf processors become cheaper and faster, it is now possible to build a PC or workstation cluster that provides high computing power within a limited budget. High performance parallelism is achieved by dividing the computation into manageable subtasks, and distributing these subtasks to the processors within the cluster. These off-the-shelf high-performance processors provide a much higher performance to cost ratio so that a high performance cluster can be built inexpensively. In addition, the processors can be conveniently connected by industry standard network components. For example, Gigabit technology provides up to 1 gigabit per second (Gbps) of bandwidth with inexpensive network adaptors and switches.

In parallel to the development of inexpensive and standardized hardware components for network of workstations, system software for programming on NOW are also advancing rapidly. For example, the *Message Passing Interface* (MPI) library has evolved into a standard for writing message-passing parallel code [13, 17, 27]. MPI programmers use a standardized high-level programming interface to exchange information among processes, instead of a native machine-specific communication library. MPI programmers can write highly portable parallel codes and run them on any parallel machine (including network of workstation) that has MPI implementation [9, 12, 36].

Most of the literature on cluster computing emphasizes on *homogeneous* clusters—clusters consisting of the same type of processors. However, we argue that heterogeneity is one of the key issues that must be addressed in improving performance of NOW. First it is always the case that one wishes to connect as many processors as possible into a cluster to increase parallelism and reduce execution time. Despite the increased computing power, the scheduling management of such a *heterogeneous network of workstation* (HNOW) becomes complicated since these processors will have different communication performance from one another. Second, since most of the processors that are used to build a cluster are commercially off-the-shelf products, they will very likely be outdated by faster successors before they become unusable. Very often a cluster will consist of "leftovers" from the previous installation, and

"new comers" that are recently purchased. The issue of heterogeneity is both scientific and economic.

Any workstation cluster, be it homogeneous or heterogeneous, requires efficient collective communications [2, 28, 29]. For example, a barrier synchronization is often placed between two successive phases of parallel computation to ensure that all processors finish the first phase before any processor goes to the second. In addition, a scatter operation distributes input data from the source to all the other processors for parallel processing, then a global reduction operation combines the partial solutions obtained from individual processors into the final answer. The efficiency of these collective communications will affect the overall parallel performance, sometimes dramatically.

The focus of this paper is to optimize the reduction operation. A 5-year-profiling in production mode at the University of Stuttgart has shown that more than 40% of the execution time of Message Passing Interface (MPI) routines is spent in the collective communication routines MPI Allreduce and MPI Reduce [29]. As a result it is vital to develop efficient algorithm to collect and combine the data in a cluster environment, especially in a heterogeneous environment.

It is very intuitive that one might want to use a heterogeneous broadcast algorithm [6, 7, 10, 22, 25, 26] and work in the reverse direction for reduction, since if all the data flow in the opposite direction, a broadcast becomes a reduction. However, this is not the case because in a reduction *every* processor must send and contribute its data, so *every* processor must send at least one message. On the other hand, a broadcast is for a single processor to send the *same* messages to all the other processors. Although other processor could help relay this message, but this relay will not necessarily make the broadcast faster. For example, if there is an extremely fast processor that wants to broadcast a message to a lot of other extremely slow processors, the best schedule is for the fast processor to send the messages *one by one* to all other slow processors, without any replaying. In contrast the reduction operation requires *every* processor to send exactly one message.

There has been some work in the literature that focus on the optimization of collection reduction [19, 23, 29, 30]. However, these works consider mostly homogeneous environments, and does not take the diversity of processor speed into consideration—mostly because these approaches rely on recursive processor halving techniques and low level communication protocol improvements, and they may not perform well in heterogeneous clusters. Vadhiyar et al. [33] gave an experimental approach to tune the collective communication, including reduction, via exhaustive search on heterogeneous clusters.

Heterogeneity of a cluster complicates the design of efficient collective communication protocols. When the processors send and receive messages at different speeds, it is difficult to synchronize them so that the message can arrive at the right processor at the right time for maximum communication throughput. On the other hand, in a homogeneous NOW every processor requires the same amount of time to transmit a message. For example it is straightforward to implement a reduction operation on P processors as $\log P$ phases, and in each phase we reduce the number of processors that have received the combined results from other processors by half. In a heterogeneous environment it is no longer clear how to complete the same task within the

minimum amount of time. As a result, the focus of this paper is to propose algorithms that perform reduction efficiently in irregular and loosely coupled clusters, instead of regular network topology, because homogeneous and regular network topology like butterfly, can perform vector reduction much more efficiently.

This paper treats the heterogeneity of a cluster mathematically, and derives provably efficient algorithms for reduction operations. The goal is to find an optimal reduction schedule that uses the least amount of time to complete the reduction. This paper shows that a simple heuristic called *slowest-node-first* (SNF) is very effective in designing reduction protocols for heterogeneous cluster systems. Despite the fact that SNF heuristic does not guarantee the optimal reduction time we show that SNF is actually a 2-approximation algorithm. That means for *any cluster*, the total SNF schedule length is guaranteed to be within twice of the optimal schedule length. This is very important since we prove that the performance of SNF is within twice of the optimum, no matter what kind of cluster we are given. In addition, we show that SNF does give an optimal reduction time when the cluster consists of two types of processors and the communication speed ratio between them is at least two. When the ratio is less than 2, we develop a dynamic programming to find the optimal schedule. Our dynamic programming utilizes the monotone property of the objective function, and can significantly reduce the amount of computation time.

Here is a summary of the theoretical results from this paper.

- SNF does give an optimal reduction time when the communication time of any processor is a multiple of the communication time of any other faster processor in the cluster.
- SNF is not optimal, but is a 2-approximation algorithm.
- SNF does give an optimal reduction time when the cluster consists of two types of processors and the communication speed ratio between them is at least two.
- A dynamic programming to find an optimal reduction for cluster with two types of processors, in time $O(f^2 s \log s)$ where s and f are the number of slow and fast processors.
- A all-reduction algorithm with an approximation ratio 3.5.

It is a very difficult to find an optimal reduction schedule for heterogeneous clusters. In practice we often resort to branch-and-bound search [2]. This paper shows that we can reduce the number of search tree nodes we have to examine by the theoretical results developed in the proof of SNF results. We want to emphasize that the exhaustive search techniques in this paper are guided by the theoretical foundation, and these techniques can be incorporated into the framework suggested by Vadhiyar et. al [33] that actually measures the key communication parameters, to further optimize the search efficiency.

We conduct experiments to show that these techniques can improve the search efficiency by a factor of 500 when there are three types of processors. For the case of two types of processors, we experiment our dynamic programming and show that our technique can reduce the amount of computation time by a factor of 93.

The rest of the paper is organized as follows. Section 2 describes the communication model in our treatment of reduction problem in HNOW. Section 3 describes the concept of earliest possible schedule and Sect. 4 describes the slowest-node-first

heuristic for reduction in heterogeneous cluster. Section 5 states the theoretical results, including a special dynamic programming for efficiently solving a cluster that has two kinds of processors. Section 6 describes how the theoretical results help improve the efficiency in finding optimal schedule, and Sect. 7 concludes.

2 Communication Model

There have been two classes of models for collective communication in homogeneous cluster environments. The first group of models assume that all the processors are fully connected. As a result it takes the same amount of time for a processor to send a message to any other processor. For example, both Postal model [5] and LogP model [20] use a set of parameters to capture the communication costs. In addition Postal model and LogP model assume that the sender can engage in other activities after a fixed startup cost, during which the sender injects the message into the network and is ready for the next message. Optimal broadcast scheduling algorithm for these homogeneous models can be found in [5, 20]. The second group of models assume that the processors are connected by an arbitrary network. It has been shown that even when every edge has a unit communication cost (denoted as the Telephone model), finding an optimal broadcast schedule remains NP-hard [14]. Efficient algorithms and network topologies for other similar problems related to broadcast, including multiple broadcast, gossiping and reduction, can be found in [11, 15, 16, 18, 24, 31, 34, 35].

Communication models for heterogeneous environments have also been proposed in the literature. Bar-Noy et al. introduced a heterogeneous postal model [4] in which the communication costs among links are not uniform. In addition, the sender may engage in another communication before the current one is finished, just like homogeneous postal and LogP model. An approximation algorithm for multicast is given, with an approximation ratio $logk$ where k is the number of destinations of the multicast [4]. Banikazemi et al. [2] proposed a simple model in which the heterogeneity among processors is characterized by the speed of sending processors. Based on this model, a 2-approximation algorithm for broadcast is given in [25]. The approximation ratio for the same algorithm is later improved to 1.5, and the problem is shown to be NP-complete in [22]. We adopt the simple model from [2] for its simplicity and the high level abstraction of network topology. Other models for heterogeneous clusters can be found in [8, 21].

The communication model of a heterogeneous environment is defined as follows. The system consists of n processors $\{p_1, \ldots, p_n\}$, each is capable of direct point-to-point communication to one another. A processor p_i is characterized by its transmission time $t(p_i)$, i.e. the time it takes for p_i to send a message to any other processor.

In order to make the communication model realistic, we further assume that two communications cannot overlap, i.e. neither the sender nor the receiver of an ongoing communication can engage in any other communication at the same time. This is referred to as 1-port model in the literature [2, 3, 22, 25, 32]. Although communication hardware often allows some degree of overlapping, e.g., a receiver may receive several messages at nearly the same time by using MPI non-blocking routines, this degree of communication parallelism may be limited because processors may only

have only limited resource for communication, and multiple communications using the same resource eventually have to be serialized. As a result, we focus on those clusters in which all processors have very limited resources, and different communications have to be serialized.

Despite that we model every processor by a "communication time", the computation time can also be incorporated into the model as follows. For every processor p we define its $t(p)$ as the sum of the computation time to *generate the data*, plus the communication time to *delivery the data*. For example, let a processor p receive the data from another processor q, combine its data with the data of q, then send the combined data to another processor r. The computation time to generate the data corresponds to combining the data of p with the data of q, and the communication time to delivery the data corresponds to sending combined data to processor r.

We define the reduction problem for the communication model we just described. Suppose each processor in the system has a unit of information, and we would like to combine all these informations into the final answer, how do we schedule the processors so that the reduction takes the least amount of time? For example, each processor may have a number and we want to compute the total sum of these numbers by message passing. For a homogeneous system a simple tree algorithm can be used so that during each iteration half of the processors send their information to the other half where the informations are combined. The algorithm takes $\log n$ rounds, where n is the number of processors, to combine all the informations since the number of active processors reduces by half per iteration. However, due to the differences in communication speeds within a heterogeneous environment, this algorithm cannot guarantee minimum reduction time.

We formally define the reduction as a scheduling problem. We assume that the reduction operator is both *associative* and *commutative*, such that all possible combinations are valid. This assumption is based on the fact that all 12 MPI reduction operators, including maximum, minimum, sum, product, logical and, logical or, etc., are associative and commutative. We observe that during the reduction process exactly $n-1$ messages are sent by $n-1$ different processors—each processor except the final reduction destination, sends out exactly one message. Based on this observation, we define a reduction schedule as a mapping function from a processor to the time that it starts sending its message. Formally we define a function s so that for any processor p (except the reduction destination), p starts sending its message at time $s(p)$, and completes sending its message at time $s(p)+t(p)$. Let $c(s, p) = s(p)+t(p)$ be the *completion time* of p under the scheduling function s. The interval $[s(p), c(s, p)]$ is defined as the *transmission window* of p under s.

To show that it is not beneficial for the reduction destination to send out any message, we define that a processor p can *reach* the reduction destination d if and only if there exists a series of transmission windows $w_1, w_2, \ldots, w_k$, and the following properties are true.

1. The source of w_1 is p.
2. The destination of w_k is d.
3. The destination of w_i is the source of w_{i+1}, for $1 \leq i < k$.
4. The starting time of w_{i+1} could not be earlier than the ending time of w_i, for $1 \leq i < k$.

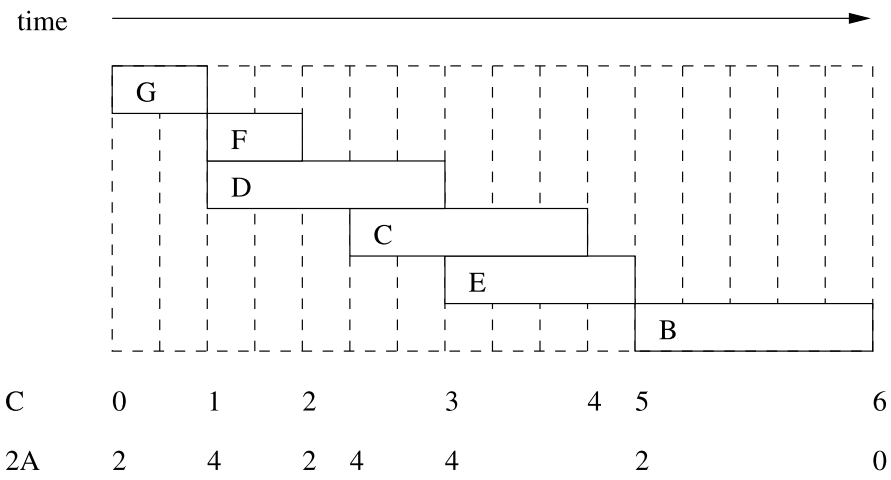

Fig. 1 A valid reduction schedule showing A and C functions defined in (1). The system consists of seven processors—indicated by the uppercase letters from A to G. Each processor has a transmission window to indicate the time interval during which it is sending its message. The transmission times, which correspond to the duration of the windows, for these seven processors, are 10, 5, 5, 5, 4, 2, and 2, respectively

A schedule is "correct" if and only if all the processors (except the destination d) can reach d. It is easy to see that if a schedule is correct, then we can remove all the transmission windows with d as the source, and all the other processors can still reach d. Therefore, without lose of generality, we assume that the destination *will not* send any messages in our scheduling.

Due to the constraint that a processor can only participate a single communication at any given time, we must distinguish valid schedule from invalid ones. To formalize this constraint, we define two sets of processors for any schedule at any given time. Let $A(s,t)$ be the number of processors that are still actively sending their messages, and $C(s,t)$ be the number of processors that have completed sending their messages at time t under schedule s.

$$\begin{aligned} A(s,t) &= |\{p_i : s(p_i) \le t < c(s,p_i)\}|, \\ C(s,t) &= |\{p_i : t \ge c(s,p_i)\}|, \\ 2A(s,t) + C(s,t) &\le n. \end{aligned} \tag{1}$$

A scheduling function s is *valid* if and only if the inequality above (see (1)) is true. The inequality says that at any time t, the total number of senders and receivers, and those processors that have sent out their messages, should not be more than the number of processors in the system. Every valid schedule can be scheduled in a cluster without violating our assumption that a processor can engage only a single communication. Although we only know the senders of all messages, later in Lemma 1 we will show how to find a receiver for all these messages. Figure 1 show a valid schedule and the corresponding A and C functions.

3 Earliest-Possible Schedule

This section describe a technique called *earliest possible* scheduling that can "normalize" all the possible valid schedules. We use this canonical form to simplify the discussion of finding an optimal schedule.

An earliest possible (EP) schedule is one in which all the communications are initiated as earlier as possible. A new communication can be initiated as soon as the number of *free processors* reaches 2—one for the sender and one for the receiver. Let $F(s,t)$ denote the number of processors that we are free to use to initiate a new communication at time t under schedule s. That is, $F(s,t) = n - 2A'(s,t) - C(s,t)$, where $A'(s,t) = |\{p_i | s(p_i) < t < s(p_i) + t(p_i)\}|$ is the number of processor that are in the middle of sending and receiving messages. Note that $A'(s,t)$ is different from $A(s,t)$. $A'(s,t)$ does not include processors that will start sending and receiving message at time t but $A(s,t)$ does. As a result all the first $\lfloor n/2 \rfloor$ processors will start sending messages at time 0, and the rest of the processors will start sending messages as soon as the number of free processors reaches two, as illustrated by Fig. 2. Note that after a message is received, only the *receiver* is considered "free" since it has the information from the sender, and should be kept active. The sender will no longer be "free" and will not participate in any further communication. The algorithm EP will assign non-decreasing start time to the processors in the order they appear in the input processor sequence P. Since we do not need to schedule the reduction destination for sending, there are only $n-1$ processors in the input processor sequence P.

By definition an earliest possible schedule s has the following properties.

1. $n - 1 \leq 2A(s,t) + C(s,t) \leq n$.
2. For any processor p that $s(p) > 0$, there exist another processor q such that $s(p) = c(s,q)$.

Note that it is trivial to convert any valid schedule into an EP schedule without increasing the total time—we just move the transmission window of each processor

Fig. 2 The pseudo code for the earliest possible scheduling algorithm

```
Algorithm EP(P)
{
  i = 1; time = 0; free = n;
  Active = empty set; Complete = empty set;
  while (i ≤ n-1) {
    do while (free ≥ 2) {
      set the start time of the i-th processor in P to time
      add the i-th processor in P into Active;
      i = i + 1;
      free = free − 2;
    }
    find the set of processors p that has the smallest
    completion time in Active;
    time = the completion time of p;
    move p from Active to Complete;
    free = free + the number of elements in p ;
  }
}
```

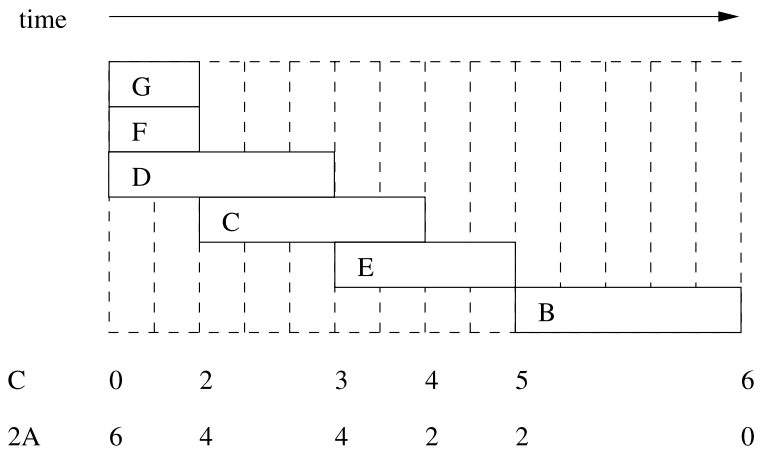

Fig. 3 The EP schedule converted from the schedule in Fig. 1. Note that when time is 2, both processors G and F have completed sending their messages, and will no longer be "free" ($C = 2$). Meanwhile processor D is sending its message to another processor ($2A' = 2$). The number of "free" processors is $7 - 2 - 2 = 3$, which is more 2, the number of initiating another communication. Therefore we can initiate processor C to start sending another message. Now we have $2A = 4$ and the number of free processors is below 2

to the earliest possible time. As a result EP schedule servers as a canonical form in which we will limit our search of an optimal schedule. Since the EP algorithm can completely determine a schedule once the order of processors in the sequence P is fixed, we only have to consider up to $(n-1)!$ different schedules. Figure 3 shows the EP schedule that is converted from the schedule in Fig. 1.

3.1 Sender Dependency and Destination Assignment

Given an EP schedule, we need to specify the destinations for all messages in order to program a cluster to perform a reduction. We will show that under any EP schedule, it is always possible to do so without violating the constraint that a processor cannot participate more than one communication simultaneously.

Lemma 1 *For any earliest possible schedule we can assign the destinations for all processors so that no two overlapping transmission windows have any sender or receiver in common.*

Proof First we would like to establish the dependency among the processors. Note that in algorithm EP it requires two free processors to start a new transmission. Recall that only the *receiver* of a communication remain "free" after the message passing, we need to wait for *two* communications to finish in order to have two new free processors. As a result we define the *predecessors* of a sender processor to be the *two* senders that must complete before the new transmission can start.[1] This dependency forms a binary tree among all sender processors.

[1] Note that when n is an odd number, one of the processor will have only one predecessor.

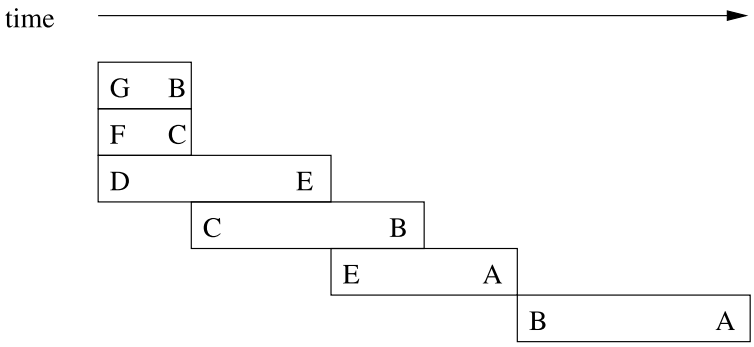

Fig. 4 Find the destinations for all senders in Fig. 3. Note that A is the destination processor

Now we can start filling in the destinations for all senders. First we assign the reduction destination d as the destination of the sender that has the latest completion time. Then for a sender/receiver pair s and r, we assign s and r to be the destinations of the two predecessors of s. We assign destinations for transmission windows in a top-down manner within the binary dependency tree. Since both predecessors complete sending their messages before s starts, no overlapping transmission windows will have processors in common. □

Figure 4 shows an example of how to find the destination for the schedule in Fig. 3. Note that the assignment is not unique since we can assign s and r to either of the two predecessors.

4 Slowest-Node-First Scheduling

We now introduce a simple scheduling heuristic called *slowest-node-first* (SNF) for the reduction problem. SNF simply sorts the processors in $P = (p_1, p_2, \ldots, p_{n-1})$ in non-increasing transmission time order, i.e., $t(p_i) \geq t(p_j)$ if $i < j$. SNF then gives the sorted sequence P to algorithm EP for scheduling. In the next section we show that this simple technique is very effective in obtaining a good reduction schedule. Figure 5 shows the slowest-node-first schedule with the same cluster as in Fig. 3.

The rationale of having the slowest processors to send messages first is as follows. At the beginning of the reduction process, we would like to overlap as many communication windows as possible. Intuitively we let all the slow processors send first so that they will overlap with each other, instead of having to wait for each other at the end of the reduction.

5 Theoretical Results

Before we describe the main results, we describe an exchange lemma that clarifies our intuition that slow processors should send first, as we did in SNF scheduling.

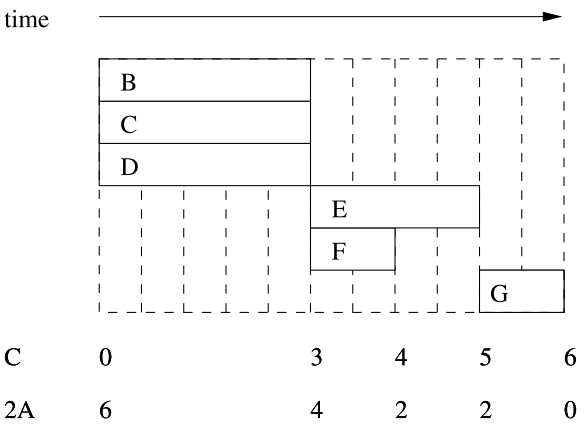

Fig. 5 Slowest-node-first scheduling result from the same cluster as in Figs. 1 and 3

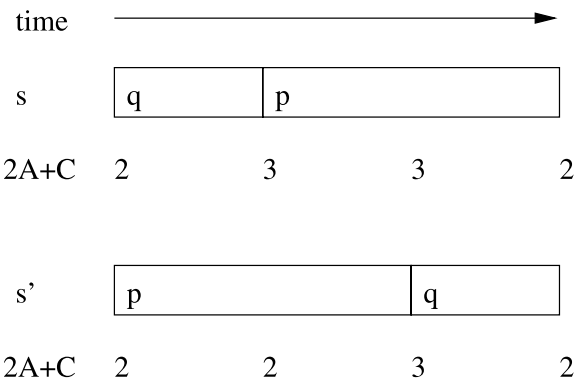

Fig. 6 An illustration on the contribution to the sum of C and $2A$ before and after we switch the transmission windows of p and q

Lemma 2 *Let s be a valid schedule and processor p starts right after processor q ends, i.e. $s(p) = c(s, q)$. If $t(p) > t(q)$ then we can exchange p and q so that the modified schedule s' in which $s'(p) = s(q)$ and $s'(q) = c(s', p)$ is also valid.*

Proof From Fig. 6 we observe that the contribution of p and q to the sum of C and A functions is always higher in s than in s', therefore if s can satisfy Inequality 1, so can s'. □

From Lemma 2 it follows that in the search of an optimal reduction schedule we can neglect those schedules that have a slower sender p with a faster predecessor q. There are two cases to consider. First if $s(p) = c(s, q)$ then by Lemma 2 we can switch p and q. On the other hand, if there is a gap between the transmission windows of p and q, then we can delay the window of q until it touches p's window. We can do so because there is no new transmission initiated between $s(p)$ and $c(s, q)$, since q is a predecessor of p. As a result we will consider only those schedules that all the senders have predecessors of smaller or equal transmission time.

Corollary 1 *There exists an optimal schedule such that the predecessors of every senders have an equal or higher transmission time than the sender.*

5.1 Approximation Ratio

We now consider a special class of clusters in which the transmission time of every processor is a power of 2. Without lose of generality we assume that the fastest processor has a transmission time of 1. We call this kind of cluster a *power 2* cluster and show that SNF generates optimal schedules for all power 2 clusters.

Theorem 1 *The slowest-node-first method gives optimal schedules for all power* 2 *clusters.*

Proof We show that every schedule for a power 2 cluster can be converted into the SNF schedule without increasing the total reduction time. We reschedule all the slowest processors as early as possible, until there is no faster processor ahead of them. Then we reschedule the second slowest processors the same way, and repeat this rescheduling until the final schedule is the same as SNF.

When the transmission time of every processors is a power of 2, we claim that there is an optimal schedule s so that for every processor p its starting time $s(p)$ is a multiple of its transmission time $t(p)$. Since every processor has a predecessor that ends when it starts, we locate a "chain" of predecessors all the way back to time 0. In addition, every one of these predecessors has a transmission time of equal or larger power of 2. Therefore the start time $s(p)$, which is the sum of the transmission time of these predecessors, is also a multiple of its transmission time.

Consider a processor p and the set F of faster processors that starts before p. Let q be the processor that completes last in F. If q finishes right when p starts, then by Lemma 2 we can exchange p and q. If there is a gap between $s(p)$ and $s(q) + t(q)$, we claim that there will be no transmission windows located within this gap. If a processors starts or ends within this gap it must fall into this gap completely, since all processors must start and end at the multiple of its transmission time. If that is the case q will not be the processor with the largest completion time in F. Since there is no transmission window in the gap, we can safely delay the transmission window of q so that q ends when p starts. By Lemma 2 we can then exchange p and q and the theorem follows. □

Theorem 2 *The slowest-node-first schedule has a total reduction time no greater than twice of the time of an optimal schedule.*

Proof Let s be an optimal schedule for a cluster H. Without lose of generality we assume that the fastest processor in H has a transmission time 1. We first convert H into a power 2 cluster by increasing the transmission time of every processor p to $2^{\lceil \log t(p) \rceil}$. We call this new cluster H'.

We argue that there exists a schedule s' for H' in which every processor p starts at time no later than $2s(p)$. The claim follows from a simple induction that if every processor p in H' waits for the two predecessors q and r defined by s, then it can

start at time no later than $2s(p)$, since both p and q starts no later than $2s(q)$ and $2s(q)$, and their transmission time at most doubles in H'.

Now we have constructed a new schedule s' for H' that has a reduction time at most twice of s. Since s' is a schedule for a power 2 cluster, its reduction time is at least the time of the optimal SNF schedule s^* on H'. Finally if we apply SNF on H and obtain schedule s'', it will be as fast as the SNF schedule s^* on H', since for every processor both of its predecessors can start earlier and send messages faster. As a result the reduction time of s'' is no more than s^*, which in turn is no more than twice of s. □

5.2 Approximation Ratio for All-reduction

There is a similar problem called *all-reduction*, in which the final reduction answer should be sent to all processors. It is easy to see that the all-reduction is at least as expensive as a fastest broadcast from any source, or a fastest reduction into any destination.

In a previous paper [25] we showed that starting from any processor as the source, we can perform a broadcast in no more than twice of the fastest broadcast time from that same source, with an technique called *fastest-node-first* [2]. The fastest-node-first technique is later shown to have an approximation ratio 1.5 when the broadcast source is the fastest processor [22]. We have already shown that with *slowest-node-first* scheduling, we can perform reduction to any destination with time no more than twice the optimal time to the same destination. Combining these two algorithms, we can perform an all-reduction within 3.5 times of the optimal time. We first perform a reduction into the fastest processor with SNF, which will complete in twice of the optimal all-reduction time, then we do a broadcast from that fastest processor, which takes at most 1.5 times of the optimal all-reduction time.

Theorem 3 *There exists a* 3.5*-approximation all-reduction algorithm. That is, this algorithm gives an all-reduction schedule and the length is within* 3.5 *times of the optimal schedule, for* all *possible clusters.*

5.3 Two Types of Processors

In this section we prove that SNF does give the optimal reduction time when the cluster consists of two types of processors, and the communication speed ratio between them is at least two. We also find a counterexample which shows that if the communication speed ratio between them is less than two then SNF is not optimal.

Lemma 3 *Let s be a valid schedule, p be a slow processor and q_1, q_2, q_3 be fast processors. If $s(p)$ is within the transmission window of q_1, q_1 stars right after q_2, and q_2 starts right after q_3, then there exists a schedule s' with following properties.*

1. *$s'(p) = s(q_3)$, i.e. process p starts sending message in s' the time process q_3 starts sending message in s.*

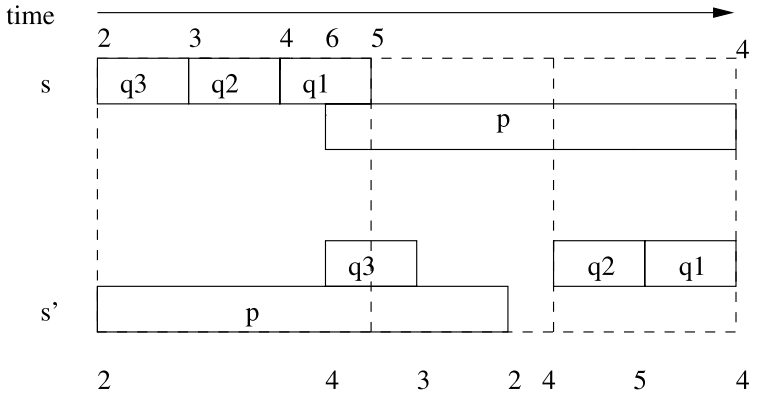

Fig. 7 Rearrange the communication windows of p and q_1, q_2, q_3

2. $s'(q_3) = s(p)$, *i.e. process* q_3 *starts sending messages in* s' *the time process* p *starts sending message in* s.
3. *Process* q_1 *starts right after* q_2 *and* q_1 *ends in* s' *at the same time as* p *ends in* s.

Proof From Fig. 7 we observe the followings:

1. $2A(s, s(p)) + C(s, s(p)) = 6$ because p is at least twice as slow as q_1.
2. $s'(q_2) < c(s', p)$, i.e. p ends before q_2 starts in s'.

From the observations above we conclude that at any time t the value of $2A + C$ is always equal or higher in s than in s', and the lemma follows. □

According to Lemma 2 we know that there exists an optimal schedule in which no slow processor will start *right after* a fast processor. Therefore, we can assume for every slow processor, its predecessors are slow processors as well. As a result the starting time of every slow processor is a multiple of its transmission time, and we can layer the slow processors according to their starting time as follows.

Definition 1 Layer one processors are the set of slow processors starting at time 0. Layer $i + 1$ processors are the set of slow processors starting right after layer i processors.

Theorem 4 *SNF is optimal when there are only two types of processors and the communication speed ratio between them is at least two.*

Proof Assume that there exists an optimal solution in which there exits a slow processor that starts after a fast processor does. Let p be the earliest such slow processor, and q be the last fast processor starts before p starts. We show that it is always possible to reschedule p so that it starts earlier than q without increasing the total time.

There are three cases to consider—processor p can start *before*, *at*, or *after* the processor q ends. For the second case we apply Lemma 2 and switch p and q. For

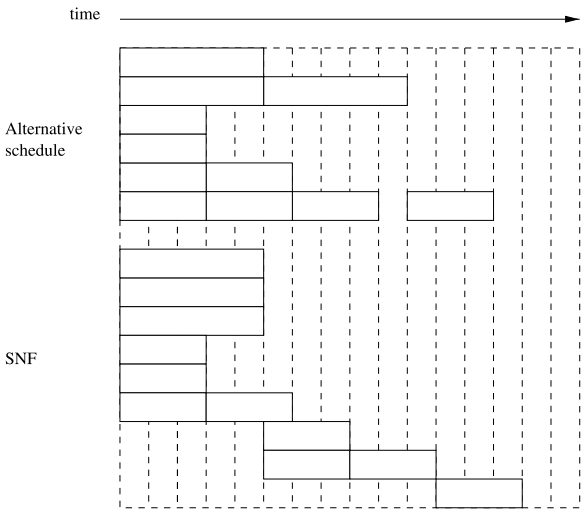

Fig. 8 A counterexample that SNF always gives optimal reduction time in a cluster consisting of two types of processors

the third case, since q finishes last among all the fast processors starting before p, there will be no processor starting in the gap between q ends and p starts. As a result we can delay the transmission window of q so that q ends right when p starts, then we apply Lemma 2 to move p earlier.

We now consider the first case. Since every processor has a predecessor that ends when it starts, we locate a "chain" of predecessors for q all the way back to time 0. For processor q we need at least two levels of predecessors and both of them must be fast ones, since the ratio of communication speed is at least two and the slow processors must start at a multiple of its transmission time. By Lemma 3 we find a new schedule in which p starts before q. We repeatedly reschedule the slow processors p earlier until no fast processor starts before it, and the theorem follows. □

Figure 8 shows a counterexample that SNF always gives optimal reduction time. This cluster has 4 slow processors with transmission time x, and 8 fast processors with transmission time 1. The alternative schedule requires $2x + 1$ time when $1.5 \leq x < 2$, or 4 when $1 < x < 1.5$. In contrast SNF requires $x + 3$ time for both cases, and has a longer reduction time for all x between 1 and 2. As a result the bound of two in Theorem 4 is tight.

5.3.1 Small Speed Ratio

From the counter-example in Fig. 8, we know that SNF cannot find optimal solution for a cluster of only two types of processors with speed ratio no more than 2. However, in the following we show that we can still use dynamic programming to find the optimal solution in this situation. In addition, we show that this dynamic programming can be significantly improved by a theoretical study.

To ease the explanation we suggest a binary tree representation for the reduction problems. Figure 9 illustrates a reduction problem. Now if we traverse backward in time, we obtain Fig. 10, which has the same total time as in Fig. 9. For each

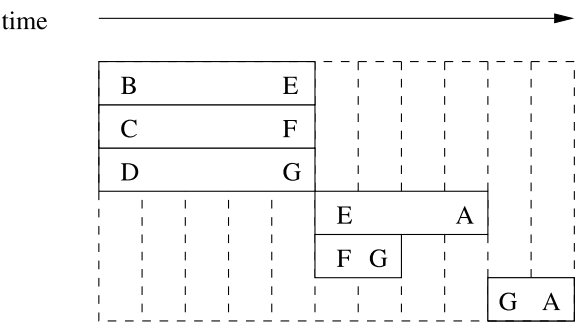

Fig. 9 An example of reduction scheduling where processor *A* is the final destination processor

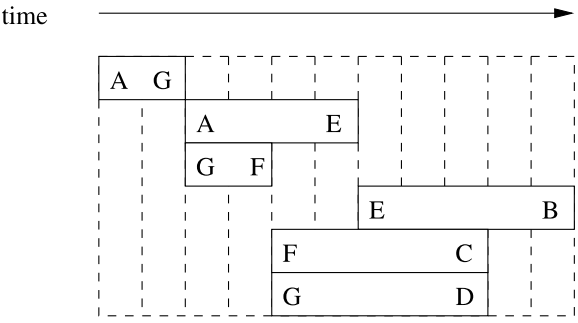

Fig. 10 Figure 9 traversed backward in time

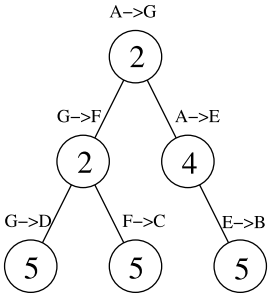

Fig. 11 A binary tree representation of the schedule in Fig. 10. The labels next to the nodes indicate the sender and the receiver of the transmission window, and the numbers within the nodes indicate the transmission time

transmission window in the time-reversed diagram (Fig. 9), we construct a node, with a weight equal to the length of the transmission window. We place an edge from a node a to another node b if the transmission window represented by b can only occur *after* the transmission window represented by node a is completed. The tree will be a binary tree since a node only needs wait for up to two nodes. The corresponding binary tree of the example from Fig. 9 is illustrated in Fig. 11.

We define the length of a tree path to be the sum of the weights of the tree nodes along the path, and the critical path length to be the maximum tree path from the root to a leaf. To find an optimal reduction schedule is equivalent to constructing a binary tree that has the minimum critical path length, when given the weights of the nodes. We use *MCP* to denote the minimum critical path.

5.3.2 Dynamic Programming

Before we describe the dynamic programming algorithm for finding the minimum critical path, we make the following observation. From Observation 1 we immediately conclude that there exists an optimal MCP tree in which all the fast nodes are on top of slow processors in the tree, that is, no slow processor has fast child.

Observation 1 *An MCP tree exists in which every node has a cost no greater than its children.*

This observation follows from the fact that if an MCP tree does have a parent with a cost larger than one of its children, we switch these two nodes and the overall MCP length will not increase. In fact this is a special case of Lemma 2. We can also conclude that an optimal MCP tree exists in which every path from the root to a leaf has non-decreasing weights on the nodes.

We describe a dynamic programming method for finding the optimal MCP tree for clusters with two classes of processors. From Theorem 4 we only need to focus on the cases where the speed ratio is less than 2. Let $T(f,s)$ be the minimum critical path length for a cluster consisting of f fast processors, s slow processors, and a destination whose transmission time is irrelevant. The transmission time of fast and slower processors are t_f and t_s respectively. We define $T(f,s)$ recursively as the following (2). This recursion follows from Observation 1 that an optimal MCP tree exists with the root being the minimum cost in the tree - in our case a fast processor.

$$T(f,s) = t_f + \min_{\substack{f_l+f_r=f-1 \\ s_l+s_r=s}} (\max(T(f_l,s_l), T(f_r,s_r))), \quad f > 0 \tag{2}$$

$$= \lceil \log_2(s+1) \rceil t_s, \quad \text{when } f = 0. \tag{3}$$

The recursion above suggests a dynamic programming solution to the MCP problem. The optimal solutions of MCP are kept in a two dimensional array, with the number of fast and slow processors as the indices. The algorithm makes these table entries from lower to higher indices. The execution time, given f and s as the number of processors, is bounded by $O(f^2 s^2)$, since there are fs entries to compute, and the time for computing each entry is bounded by $O(fs)$ time.

5.3.3 Improvement from the Monotone of s Coordinate

The dynamic programming described in the previous section can be improved if we understand the structure of the $T(f,s)$ function. We describe this structures and show how it reduces the computation costs of $T(f,s)$. It is not difficult to see that the $T(f,s)$ function is monotonic with respect to s, that is, when we increase the number of slow processors, the optimal MCP length will not decrease.

Lemma 4 $T(f,s+1) \geq T(f,s)$ *for all* $s \geq 0$.

Proof Consider an optimal MCP tree $\mathcal{T}$ with f fast and $s+1$ slow processors. Suppose this tree has a shorter MCP than the tree with one less slow processor, that is,

$T(f, s+1) < T(f, s)$. From Observation 1 we can always locate slow processor n as a leaf in $\mathcal{T}$. It is easy to see that by removing n we will not increase the MCP length. As a result we have constructed a tree that has a shorter critical path than $T(f, s)$, a contradiction to the definition of $T(f, s)$. □

With Lemma 4 in place we examine (2). For a given f_l and f_r pair, we must examine all the s_l and s_r pairs such that $s_l + s_r = s$. This is equivalent to computing the following function:

$$M(f_l, f_r, s) = \min_{0 \le s' \le s} (\max(T(f_l, s'), T(f_r, s - s'))). \tag{4}$$

Equation (4) leads us to consider the *crossover* point of $T(f_l, s')$ and $T(f_r, s - s')$. Since $T(f_l, s')$ is non-decreasing and $T(f_r, s - s')$ is non-increasing with respect to s' (from Lemma 4), the minimum of the maximum of the two functions is located around $T(f_l, s') = T(f_r, s - s')$. This crossover point of can be easily found by a binary search with $O(\log s)$ time. On the other hand, if one of the functions is strictly larger than the other, the minimum of the maximum of two functions can be found at either $s' = 0$ or $s' = s$, and these special cases can be easily checked. We summarize our discussion with the following theorem:

Theorem 5 *Given the number of fast and slow processors in an MCP length problem, the optimal time $T(f, s)$ can be found in time $O(f^2 s \log s)$.*

Proof The theorem follows from the improvement of reducing the computation time for each cell to $O(f \log s)$, and there are fs cells to compute. □

5.3.4 Improvement from the Monotone of f Coordinate

We now establish the monotone of $T(f, s)$ with respect to f. One would think that, like adding slow processors, adding fast processors will delay the schedule. Surprisingly, this intuition is only partially true—when the number of fast processors is small, adding fast processors actually helps reduce the broadcast time since the workload of telling all the slow processors is *shared* by the added fast processors.

The main purpose of having fast processors is to share the workload of sending messages to slow processors. As a result, if a fast processor only has one child in an optimal MCP tree, we conclude that the number of slow processors is not large enough compared with the number of fast processors to be beneficial, so this redundant fast processor can be safely removed without increasing the MCP length.

Observation 2 *Any tree node with only one child can be removed* (*by connecting its parent to its only child*) *without increasing the MCP length.*

With this observation, we proceed to prove the monotonic result.

Lemma 5 *$T(f + 1, s) \ge T(f, s)$ for all $f \ge s - 1$.*

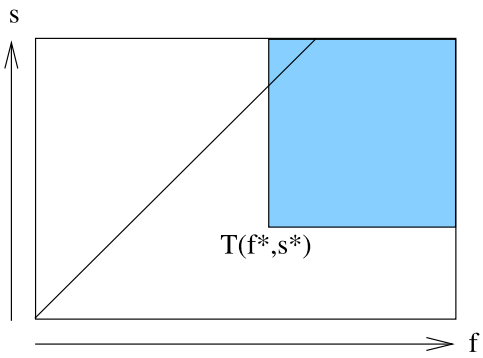

Fig. 12 An illustration that any entry with indices higher than f^* and s^* will have a larger $T(f,s)$ value

Proof We prove by contradiction and assume that $T(f+1,s) < T(f,s)$ for an $f \geq s-1$. Consider this optimal MCP tree having $f+1$ fast and s slow processors. From Observation 1 all $f+1$ fast processors (including the root) are connected into a tree $\mathcal{T}^*$. The $f+1$ fast processors in $\mathcal{T}^*$ can connect up to $f+1+1 \geq s+1$ other tree nodes as children. Since we do not have enough slow processors to cover every fast processor in $\mathcal{T}^*$ with two children, there exists a fast processor with at most one child. From Observation 2 we remove this one-child fast processor without increasing the MCP length, and by doing so we construct a tree with f fast processors, s slow processors, and an MCP length smaller than $T(f,s)$—a contradiction to the definition of $T(f,s)$. □

To see how Lemma 5 helps improve dynamic programming efficiency, let's consider a table entry $T(f^*,s^*)$ where $f^* > s^*$. If $T(f^*,s^*)$ is already worse than the current best solution, we do not need to consider any $T(f,s)$ where $f \geq f^*$ and $s \geq s^*$. The reason is that $T(f,s)$ is monotonic with respect to f, we have $T(f,s^*) \geq T(f^*,s^*)$ for $f \geq f^*$. In addition, $T(f,s)$ is also monotonic with respect to s, so $T(f,s) \geq T(f^*,s^*)$ for $s \geq s^*$. As a result it is possible to eliminate a rectangular area in the $T(f,s)$ table (Fig. 12) if we know the solution $T(f,s)$ is worse than the current best solution.

We apply the results from Lemmas 4 and 5 to our dynamic programming. We first define a relation between two (f,s) pairs while computing the T function in (2). We call two (f,s) pairs *buddies* if they appear within the maximum function in (2). For example (f_l,s_l) and (f_r,s_r) are buddies if $f_l+f_r = f-1$ and $s_l+s_r = s$ for given f and s (Fig. 13). For example, point A in Fig. 13 has a buddy B on the right side of the f-s plane. We also define $F(A)$ and $S(A)$ to be the f and s coordinates of point A. Let C be the point $(\lceil \frac{s+1}{2} \rceil - 1, \lceil \frac{s+1}{2} \rceil)$ on the $s = f+1$ line (Fig. 13).

We improve efficiency in dynamic programming by eliminating buddy pairs that could not possibly be the answer. There are three cases to consider. In all cases we assume that A is a point on the line $s = f+1$, and B is the buddy of A.

$S(A) \leq S(B)+1$ Consider a point A' under A and its buddy B' (Fig. 13). From Lemma 4 we know that $T(B') \geq T(B)$, and from Lemmas 4 and 5 we conclude that $T(B') \geq T(A)$. As a result the pair A' and B' could not possibly produce the answer

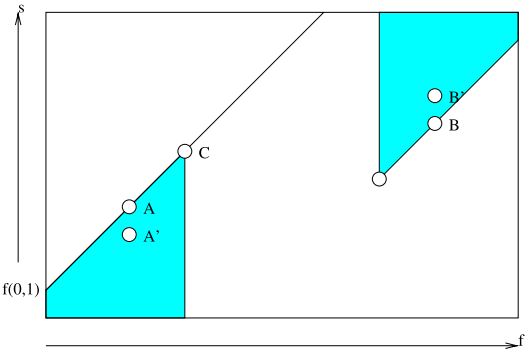

Fig. 13 An illustration of two buddies A and B. Point C is on the line $s = f + 1$ and has coordinate $(\lceil \frac{s+1}{2} \rceil - 1, \lceil \frac{s+1}{2} \rceil)$

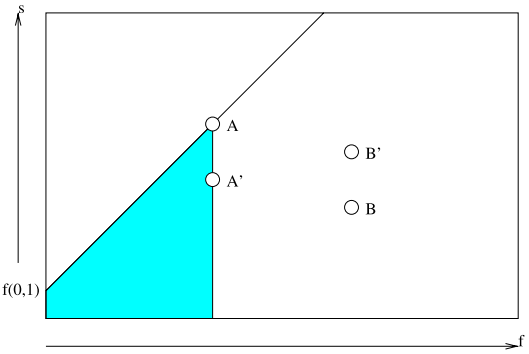

Fig. 14 An illustration when $S(A)$ is larger than $S(B) + 1$ and $T(A) \leq T(B)$

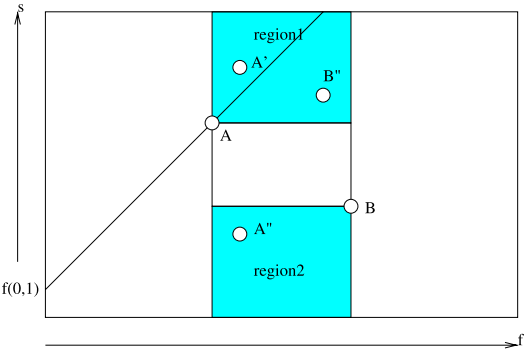

Fig. 15 An illustration for the third case that $S(A)$ is larger than $S(B) + 1$ and $T(A) > T(B)$

for (2) since A and B would be a better choice. We do not need to consider the shaded area in Fig. 13.

$S(A) > S(B) + 1$ and $T(A) \leq T(B)$ In Fig. 14 any point A' below A could not be the solution, since the buddy of A' (call it B') has cost $T(B') \geq T(B) \geq T(A)$. As a result the pair A' and B' could not possibly produce the answer, and we do not need to consider the shaded area in Fig. 14.

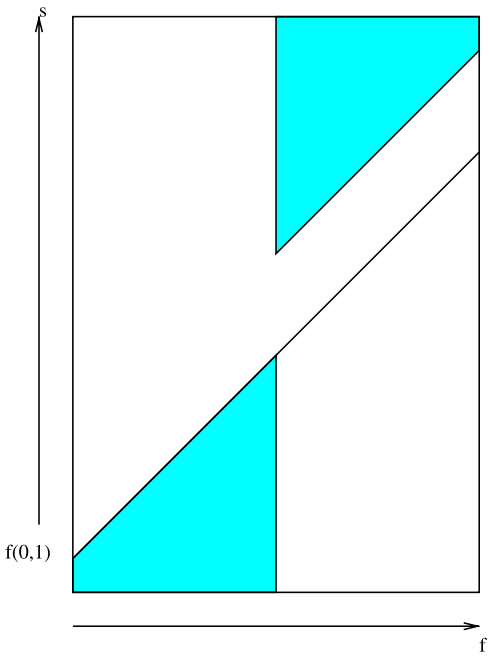

Fig. 16 An illustration for the case that s is larger than f

$S(A) > S(B) + 1$ *and* $T(A) > T(B)$ Any point A' in the upper region (region 1 in Fig. 15) could not be the solution since $T(A') \geq T(A) > T(B)$. On the other hand, any point $A*$ in the lower area (region 2 in Fig. 15) could not be the solution either since the buddy of $A*$ (call it $B*$) has a T value no less than either A or B. We only need to consider the rectangular region in the middle for this case (Fig. 15).

We now describe the overall dynamic programming algorithm in details. The algorithm will try to ignore as many cells as possible, given the three cases we just described. The algorithm will ignore the region in Fig. 13 when the s coordinate is less than or equal to $\lceil \frac{s+1}{2} \rceil$ (point C in Fig. 13), as discussed in the first case. Once we pass point C, we repeatedly compare $T(A)$ and $T(B)$ (B is the buddy of A) and try to find the first A on the line $s = f + 1$ such that $T(A) > T(B)$. The second case applies to those s coordinates before the loop stops, and their trapezoid areas can be safely ignored. When the loop stops, the third case applies and we need to consider a rectangular area only (Fig. 15). For those areas that cannot be ignored, we use a binary search to find the crossover point, as in the discussion of (4).

In Figs. 13, 14, and 15 we all assume that f is larger than s. When s is larger than f, the second and the third case will not occur, and we can actually simplify the dynamic programming by ignoring the area illustrated in Fig. 16.

5.3.5 Improvement for Higher Dimensions

It is easy to generalize Theorem 5 to a higher number of classes of processors, since Lemma 4 holds for the slowest processors.

Theorem 6 *If a cluster has n processors from k classes, the optimal MCP length can be computed in* $O(n^{2k-1}\log n)$ *time.*

Proof It is easy to see that Lemma 4 holds for the slowest processors. That is, the optimal time function given the number of processors of each kind, is monotonic with respect to the number of the slowest processors. As a result we can derive an equation similar to (4), that is, we fix the number of other processors, and try to find the minimum by a binary search for the crossover point along the coordinate of the slowest processor. The dynamic programming requires a table of n^k entries since the number of each type of processors is bounded by n. For each entry we need $O(\log n)$ time for a binary search, and there are n^{k-1} entries. The total time is bounded by $O(n^k \times n^{k-1}\log n)$ and the theorem follows. □

6 Experimental Results

Our experiment has three parts. The first part is to compare the SNF technique with MPI reduction implementation. The second part is to use our theoretical results to guide a exhaustive search and improve its performance. The third part is to use the theoretical results to improve the dynamic programming while finding the optimal schedule in a cluster with two classes of processors.

6.1 SNF and MPI Reduction

We compare the performance of SNF and the MPI built-in reduction. The first implementation is from our SNF technique and the second implementation is a direct invocation of `MPI_Reduce` function,

The cluster has 8 processors, four fast Intel XEON processor running at 3.2 GHz, with one gigabytes of DDRII 400 memory, and four slow AthlonMP processors running at 2 GHz, with one gigabytes of DDR 266 memory. The cluster is connected by fast Ethernet.

We conduct two sets of reductions—the MPI built-in maximum operator `MPI_MAX`, and a user defined *greatest-common-divisor* (gcd) operation. Both operators are associative and commutative. The reason that we include the gcd operator is to see the effects of a slightly more computational intensive reduction operator. The MPI library we use is MPICH 1.2.6. The number of data each processor will provide for reduction varies from 4 to 4096. We run each experiment for 1000 times and take the average

Figure 17 shows the total reduction time of SNF and `MPI_Reduce` under different number of data. SNF scheduling outperforms MPI implementation in both MPI built-in maximum operator `MPI_MAX` and the user defined *greatest-common-divisor* (gcd) operation, and for all data numbers from 4 to 4096.

6.2 Exhaustive Search in General Clusters

The previous section describes the theoretical results that guarantees the optimality of SNF method for special cases, and provides performance guarantee for general

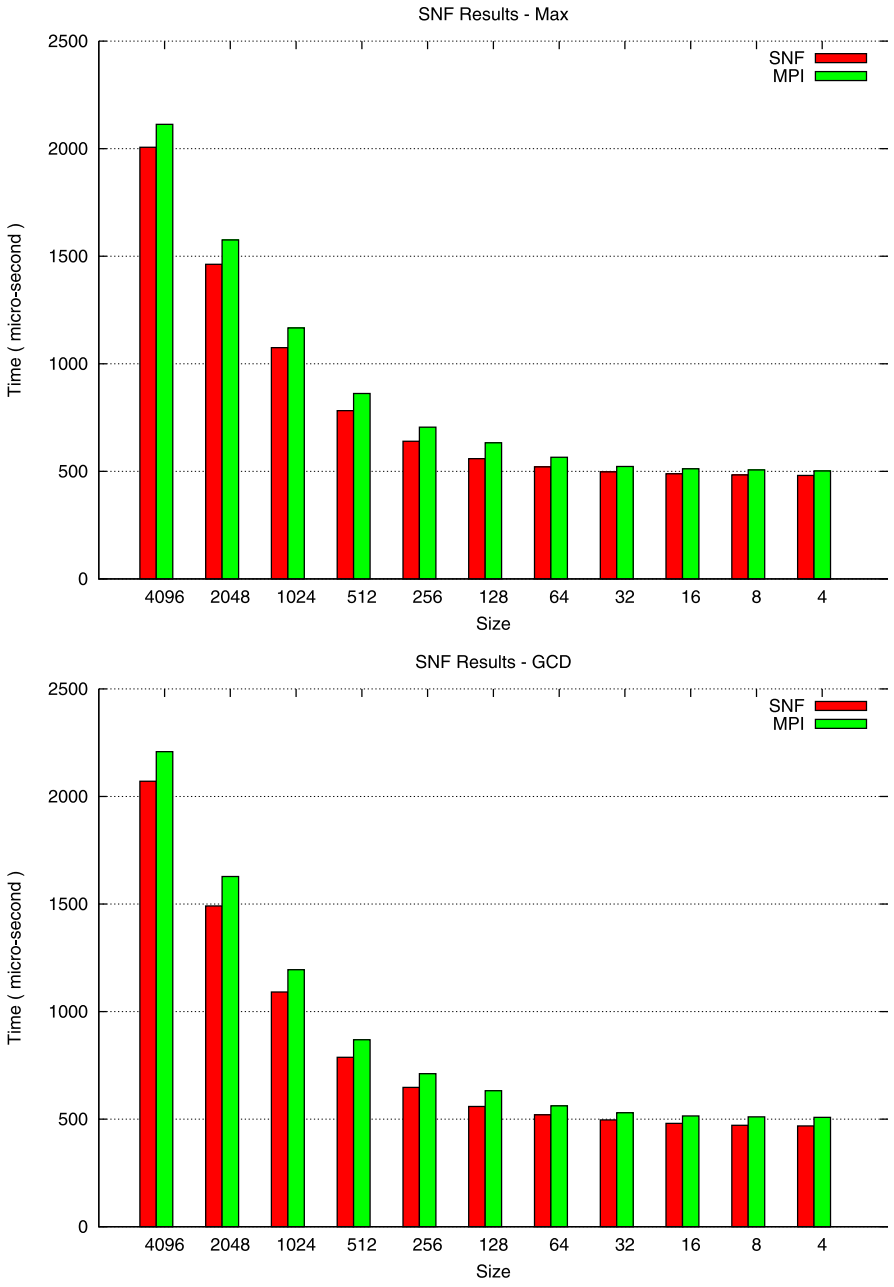

Fig. 17 A comparison on the performance of SNF and `MPI_Reduce`

cases. However, in practice one may wish to find the optimal reduction schedule for a particular cluster that contains more than two kinds of processors. In such cases

we have to search for the optimal schedule since SNF does not guarantee optimality. This section describes the techniques that we used to speed up the search process.

As described in Sect. 3, every reduction schedule can be converted into a earliest-possible schedule, which can be represented by a sequence of processors. As a result we can find an optimal reduction schedule among these $(n-1)!$ possible sequences, where n is the number of processors in the cluster. However, for a typical cluster $(n-1)!$ is such a large number that we apparently cannot try all these permutations, even by a branch-and-bound procedure. To overcome this problem, we conduct experiments to show that by Corollary 1 we can dramatically reduce the search space.

We use three techniques to reduce the number of sequences we have to consider. First of all, we examine the sequences in such an order that those sequences with slower processors appearing first will be examined first. Formally we define the *priority* of a sequence to be the number processors that have longer or equal transmission time than the next processor in the sequence. In other words, the SNF schedule has the highest priority and will be considered first.

The second technique is to apply Corollary 1 so that when a fast processor is scheduled to send the message to a slower processor, we can prune that subtree immediately. In addition, it is possible for several senders to send messages simultaneously so that more than one processor can receive at the same time. In that case if any sender is faster than any of those possible receivers then we can drop this partial solution completely.

Finally, we use the standard branch-and-bound technique to explore the search tree. If the cost of a partially examined sequence is already larger than the current optimal, the entire subtree is pruned. This technique is most effective when the difference among processor speed is large.

We conduct the experiments on a Pentinum 3-450 PC running FreeBSD 3.2 UNIX. The PC has 128 Mbytes memory and we use gcc 2.7.2-1 to compile the code. The input cluster configurations for our experiments are generated as follow. We assume that the number of classes in a cluster is 3. This assumption is practical since processors are usually purchased in batches and the number of batches is usually small. We vary the cluster size from 10 to 21. For each processor we randomly assign a communication speed from the three possible values. For each cluster size we repeat the experiments for 50 times and compute the average for the quantities we measured.

We quantify the *search ratio* of an algorithm as the percentage of the entire search tree the algorithm has to examine in order to find the optimal solution. As a result, an algorithm that scans n tree nodes before finding the optimal one the search ratio is $\frac{n}{N}$, where N is the total number of nodes in the entire search tree.

Table 1 compares the efficiency of our algorithm with a simple branch-and-bound search. Guided by various heuristics described earlier, our algorithm searches much fewer tree nodes than the generic branch-and-bound method, and consequently runs much faster. For large clusters consisting up to 21 nodes our algorithm runs about five hundred times faster than the generic algorithm, and can find the optimal solution within 15 second, while the generic algorithm runs more than two hours.

The first two columns in Table 1 indicate the average number of nodes and leaves of the search trees generated. The next three columns are the number of tree nodes examined, the search time (in second), and the search ratio from our algorithm. The

Table 1 The comparison of two search programs

P	Search tree		SNF search			Generic branch-and-bound			Search time
	N	L	n	Time	n/N	n	Time	n/N	ratio
10	7852	2406	475	0.002	6.049%	5642	0.05	71.854%	25
11	21544	6586	824	0.005	3.825%	15330	0.17	71.157%	34
12	55243	16785	2151	0.01	3.894%	39367	0.48	71.262%	48
13	163186	49695	3715	0.03	2.277%	115419	1.64	70.728%	54
14	440848	134154	9908	0.07	2.247%	310731	4.76	70.485%	68
15	1141417	344573	14845	0.12	1.301%	804970	13.61	70.524%	113
16	3824647	1160027	46688	0.41	1.221%	2683898	48.48	70.174%	118
17	10379374	3147823	70259	0.66	0.677%	7248825	144.32	69.839%	218
18	29444739	8902547	187034	1.85	0.635%	20576306	444.51	69.881%	240
19	78392528	23692980	293681	3.28	0.375%	54742994	1115.72	69.832%	340
20	189185942	56704807	848633	10.43	0.449%	132562430	2476.89	70.069%	237
21	600236924	180412877	1187418	15.28	0.198%	420191444	8173.73	70.004%	544

next three columns are from a generic branch-and-bound algorithm. The last column shows the performance ratio between these two algorithms.

6.3 Dynamic Programming

To demonstrate how theoretical results help improve the efficiency of our dynamic programming, we design experiments to compare the number of pairs of table entries that we need to examine in order to compute the optimal schedule. In other words, we use the number of table entry lookups as the cost measurement of dynamic programming.

We conduct experiments on a Pentinum 3-450 PC running Windows 2000 5.00.2195. The PC has 128 Mbytes memory. We use Microsoft Visual C++R 6.0 to compile the code. The experiment has four parameters—c_s, c_f, s and f. These parameters are the transmission costs of slow and fast processors, and the number of slow and fast processors in the cluster. Since we are only interested in counting the number of referenced pairs while computing the optimal schedule, only 3 parameters are needed: $\frac{c_s}{c_f}$, s and f. In other words, we can safely assume that the transmission cost of a fast processor is 1, therefore, for ease of notation, we only consider parameter c_s.

We compare three algorithms: The first is a naive algorithm that checks all the pairs in the table, that is, it has to check $\lceil \frac{s(f-1)}{2} \rceil$ pairs. In this algorithm, parameter c_s does not affect the result.

The second algorithm utilizes Lemma 4 and uses a binary search on the s-axis to find the optimal solution. This algorithm reduces the number of table references since, while computing (4), a binary search requires only $O(\log s)$ references.

We now consider both Lemma 4 and Lemma 5, and derive the third algorithm. This algorithm deletes certain pairs from the computation and uses a binary search on the rest of the points along the s-coordinate. This algorithm does not reduce the

Table 2 The number of references from a straightforward dynamic programming

150	0	750	1500	2250	3000	3750	4500	5250	6000	6750	7500	8250	9000	9750	10500	11250
140	0	700	1400	2100	2800	3500	4200	4900	5600	6300	7000	7700	8400	9100	9800	10500
130	0	650	1300	1950	2600	3250	3900	4550	5200	5850	6500	7150	7800	8450	9100	9750
120	0	600	1200	1800	2400	3000	3600	4200	4800	5400	6000	6600	7200	7800	8400	9000
110	0	550	1100	1650	2200	2750	3300	3850	4400	4950	5500	6050	6600	7150	7700	8250
100	0	500	1000	1500	2000	2500	3000	3500	4000	4500	5000	5500	6000	6500	7000	7500
90	0	450	900	1350	1800	2250	2700	3150	3600	4050	4500	4950	5400	5850	6300	6750
80	0	400	800	1200	1600	2000	2400	2800	3200	3600	4000	4400	4800	5200	5600	6000
70	0	350	700	1050	1400	1750	2100	2450	2800	3150	3500	3850	4200	4550	4900	5250
60	0	300	600	900	1200	1500	1800	2100	2400	2700	3000	3300	3600	3900	4200	4500
50	0	250	500	750	1000	1250	1500	1750	2000	2250	2500	2750	3000	3250	3500	3750
40	0	200	400	600	800	1000	1200	1400	1600	1800	2000	2200	2400	2600	2800	3000
30	0	150	300	450	600	750	900	1050	1200	1350	1500	1650	1800	1950	2100	2250
20	0	100	200	300	400	500	600	700	800	900	1000	1100	1200	1300	1400	1500
10	0	50	100	150	200	250	300	350	400	450	500	550	600	650	700	750
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s/f	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150

Table 3 The optimized dynamic programming method that considers only the monotony along the s coordinate

150	0	17	35	45	93	79	83	90	89	97	97	99	98	103	108	117
140	0	19	28	62	74	70	76	70	75	75	80	85	95	100	105	105
130	0	28	30	68	59	57	58	61	66	75	80	85	90	95	100	105
120	0	29	53	40	42	48	54	79	114	151	185	168	130	101	154	210
110	0	16	38	31	39	57	99	171	174	164	174	158	125	96	149	180
100	0	16	34	35	81	105	105	106	103	113	116	126	123	95	148	179
90	0	18	26	62	68	83	75	71	81	85	97	103	105	103	107	157
80	0	15	39	54	50	57	54	59	68	75	81	85	99	101	111	102
70	0	16	37	37	41	45	49	54	60	67	71	77	83	92	97	107
60	0	26	23	28	58	99	65	75	96	89	87	91	86	91	98	102
50	0	15	49	58	57	63	62	72	104	131	156	179	142	129	124	129
40	0	18	28	28	32	44	51	60	67	99	116	137	146	195	232	176
30	0	14	31	34	50	46	45	49	58	68	83	89	120	140	173	219
20	0	15	19	25	32	62	74	116	85	87	90	102	110	132	181	236
10	0	10	16	36	46	52	59	89	134	179	224	221	226	233	238	243
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s/f	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150

time complexity order, but will reduce the constant factor since many pairs will not be examined. Please refer to Figs. 13, 14 and 15 for savings.

Table 2 lists the costs of computing the optimal schedule using the three algorithms. We chose $c_s = 2$ and compute $T(f, s)$ all the way till $f \leq 150$ and $s \leq 150$.

From Tables 2, 3 and 4 it is obvious that the second and third algorithms are much better than the first naive algorithm. For example, when $f = s = 150$ the first

Table 4 The optimized dynamic programming method that considers both the monotony along the s and f coordinates

150	0	19	33	50	96	94	91	89	85	105	112	119	122	125	122	120
140	0	18	28	60	76	69	74	86	97	101	111	122	134	139	132	120
130	0	22	34	66	56	60	77	87	97	111	121	145	151	153	142	134
120	0	20	45	56	47	73	88	100	119	141	192	190	176	170	174	184
110	0	15	47	35	59	74	102	169	159	153	169	157	170	178	166	166
100	0	17	33	48	75	103	123	110	116	118	110	110	136	145	146	141
90	0	21	29	59	80	88	90	83	84	92	93	97	94	108	117	126
80	0	14	36	59	54	62	68	72	74	71	75	83	89	84	94	95
70	0	17	39	37	50	59	67	66	62	59	55	55	58	59	61	70
60	0	19	26	44	59	96	87	80	79	70	64	60	57	58	62	61
50	0	13	39	62	58	55	65	68	71	93	123	134	126	122	121	125
40	0	21	27	31	30	37	43	48	47	53	85	119	145	182	201	197
30	0	15	31	42	44	36	33	30	34	33	39	64	109	142	193	257
20	0	14	18	19	20	41	59	83	81	81	77	85	89	104	154	199
10	0	9	8	25	38	39	41	59	115	160	205	202	207	213	218	223
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s/f	0	10	20	30	40	50	60	70	80	90	100	110	120	130	140	150

algorithm has to refer to 11250 pairs, while the second and third only has to look up 117 and 120 pairs respectively.

7 Conclusion

This paper shows that the slowest-node-first scheduling is a very efficient reduction protocols for heterogeneous cluster systems. We show that SNF is a 2-approximation algorithm. In addition, we show that SNF does give the optimal reduction time when the cluster consists of two types of processors, and communication speed ratio between them is at least two. When the communication speed ratio is less than two, we develop a dynamic programming technique to find the optimal schedule. Our dynamic programming utilizes the monotone property of the objective function, and can significantly reduce the amount of computation time. Finally when we combine SNF with previous approximation algorithms for heterogeneous broadcast [22, 25], we have an all-reduction algorithm which sends the reduction answer to *all* processors, with approximation ratio 3.5.

We also conduct experiments to demonstrate that SNF performs better than the built-in `MPI_Reduce` in a test cluster. We also observe a factor of 93 times saving in computation time to find the optimal schedule in a cluster with two classes of processor than a naive dynamic programming implementation. Finally we apply these theoretical results to branch-and-bound search and show that they can reduce the search time by a factor of 500.

It will be interesting to extend this technique to other communication protocols and models. For example, in our model the communication time is determined solely

by the sender. In a more practical and complex model the communication time may be a function of both the send and the receiver [8]. In addition, it will be worthwhile to investigate the possibility to extend the analysis to similar protocols like parallel prefix, or all-to-all broadcast. These questions are very fundamental in designing collective communication protocols in heterogeneous clusters, and will certainly be the focus of further investigations in this area.

Acknowledgements The authors thank Mr. Tzu-Hao Sheng for implementing the heuristic search programs, Mr. Jun-Chen Xu for implementing the MPI reduction programs. This work is supported in part by National Science Council of Taiwan, under grant number NSC95-2221-E-002-071, and by the Excellence Research Program/Frontier and Innovative Research Program of National Taiwan University, under grant number NTU-ERP-95R0062-AE00-07.

References

1. Anderson, T., Culler, D., Patterson, D.: A case for networks of workstations (now). In: IEEE Micro, February 1995, pp. 54–64 (1995)
2. Banikazemi, M., Moorthy, V., Panda, D.K.: Efficient collective communication on heterogeneous networks of workstations. In: Proceedings of International Parallel Processing Conference, pp. 460–467 (1998)
3. Banino, C., Beaumont, O., Carter, L., Ferrante, J., Legrand, A., Robert, Y.: Scheduling strategies for master–slave tasking on heterogeneous processor platforms. IEEE Trans. Parallel Distrib. Syst. **15**(4), 319–330 (2004)
4. Bar-Noy, A., Guha, S., Naor, J., Schieber, B.: Multicast in heterogeneous networks. In: Proceedings of the 13th Annual ACM Symposium on Theory of Computing (1998)
5. Bar-Noy, A., Kipnis, S.: Designing broadcast algorithms in the postal model for message-passing systems. Math. Syst. Theory **27**(5), 431–452 (1994)
6. Beaumont, O., Legrand, A., Marchal, L., Robert, Y.: Pipelining broadcasts on heterogeneous platforms. IEEE Trans. Parallel Distrib. Syst. **16**(4), 300–313 (2005)
7. Beaumont, O., Marchal, L., Robert, Y.: Broadcast trees for heterogeneous platforms. In: 19th IEEE International Parallel and Distributed Processing Symposium, vol. 1, p. 80b. IEEE Computer Society, Los Alamitos (2005)
8. Bhat, P.B., Raghavendra, C.S., Prasanna, V.K.: Efficient collective communication in distributed heterogeneous systems. In: Proceedings of the International Conference on Distributed Computing Systems (1999)
9. Cui, A.Q., Street, R.L.: Large-eddy simulation of coastal upwelling flow. Environ. Fluid Mech. **4**(2), 197–223 (2004)
10. den Burger, M., Kielmann, T., Bal, H.E.: Balanced multicasting: high-throughput communication for grid applications. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, p. 46. IEEE Computer Society, Washington (2005)
11. Dinneen, M., Fellows, M., Faber, V.: Algebraic construction of efficient networks. In: Applied Algebra, Algebraic Algorithms, and Error Correcting Codes. Lecture Notes in Computer Science, vol. 539, p. 9. Springer, Berlin (1991)
12. Dubinski, J., Kim, J., Park, C., Humble, R.: GOTPM: a parallel hybrid particle-mesh treecode. New Astronomy **9**(2), 111–126 (2004)
13. Bruck, J. et al.: Efficient message passing interface (MPI) for parallel computing on clusters of workstations. J. Parallel Distrib. Comput. **40**(1), 19–34 (1997)
14. Garey, M.R., Johnson, D.S.: Computer and Intractability: A Guide to the Theory of NP-Completeness. Freeman, New York (1979)
15. Gargang, L., Vaccaro, U.: On the construction of minimal broadcast networks. Network **19**(6), 673–689 (1989)
16. Grigni, M., Peleg, D.: Tight bounds on minimum broadcast networks. SIAM J. Discrete Math. **4**, 207–222 (1991)
17. Gropp, W., Lusk, E., Doss, N., Skjellum, A.: A high-performance, portable implementation of the mpi: a message passing interface standard. Parallel Comput. **22**(6), 789–828 (1996)

18. Hedetniemi, S.M., Hedetniem, S.T., Liestman, A.L.: A survey of gossiping and broadcasting in communication networks. Networks **18**(4), 319–349 (1991)
19. Karonis, N., de Supinski, B., Foster, I., Gropp, W., Lusk, E., Bresnahan, J.: Exploiting hierarchy in parallel computer networks to optimize collective operation performance. In: Proceedings of the 14th International Parallel and Distributed Processing Symposium (2000)
20. Karp, R., Sahay, A., Santos, E., Schauser, K.E.: Optimal broadcast and summation in the LogP model. In: Proceedings of 5th Annual Symposium on Parallel Algorithms and Architectures (1993)
21. Kesavan, R., Bondalapati, K., Panda, D.: Multicast on irregular switch-based networks with wormhole routing. In: Proceedings of International Symposium on High Performance Computer Architecture (1997)
22. Khuller, S., Kim, Y.: On broadcasting in heterogeneous networks. In: Proceedings of the 16th Annual ACM Symposium on Parallel Architectures and Algorithms (2004)
23. Kielmann, T., Hofman, R.F.H., Bal, H.E., Plaat, A., Raoul, A., Bhoedjang, F.: Mpi'sa reduction operations in clustered wide area systems. In: Proceedings of the Message Passing Interface Developer's and User's Conference (1999)
24. Liestman, A.L., Peters, J.G.: Broadcast networks of bounded degree. SIAM J. Discrete Math. **1**, 531–540 (1988)
25. Liu, P.: Broadcast scheduling optimization for heterogeneous cluster systems. J. Algorithms **42**, 135–152 (2002)
26. Liu, P., Wang, D., Guo, Y.: An approximation algorithm for broadcast scheduling in heterogeneous cluster. In: The 9th International Conference on Realtime Computing Systems and Applications, Taiwan (2003)
27. Luecke, G.R., Kraeva, M., Yuan, J., Spanoyannis, S.: Performance and scalability of MPI on PC clusters. Concurr. Comput. Pract. Exp. **16**(1), 79–107 (2004)
28. Mpich, O.: Improving the performance of collective operations in mpich. Improving the performance of collective. In: Proceedings of the 11th EuroPVM/MPI Conference (2003)
29. Rabenseifner, R.: Optimization of collective reduction operations. In: Proceedings of International Conference on Computational Science (2004)
30. Rabenseifner, R., Träff, J.L.: More efficient reduction algorithms for non-power-of-two number of processors in message-passing parallel systems. In: PVM/MPI, pp. 36–46 (2004)
31. Richards, D., Liestman, A.L.: Generalization of broadcast and gossiping. Networks **18**(2), 125–138 (1988)
32. Steffenel, L.: A framework for adaptive collective communications on heterogeneous hierarchical networks. Research Report 6036, INRIA (2006)
33. Vadhiyar, S.S., Fagg, G.E., Dongarra, J.: Automatically tuned collective communications. In: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing, p. 46. IEEE Computer Society, Los Alamitos (2000)
34. Ventura, J.A., Weng, X.: A new method for constructing minimal broadcast networks. Networks **23**(5), 481–497 (1993)
35. West, D.B.: A class of solutions to the gossip problem. Discrete Math. **39**(33), 307–326 (1992)
36. Yin, Z., Clercx, H.J.H., Montgomery, D.C.: An easily implemented task-based parallel scheme for the Fourier pseudospectral solver applied to 2D Navier–Stokes turbulence. Comput. Fluids **33**(4), 509–520 (2004)