Deadlock-free Asynchronous Communication Strategies for Unstructured Computations on iPSC/860

Sesh Venugopal
Dept. of Computer Science
Rutgers University
New Brunswick, NJ 08903.

Vijay K. Naik
IBM Research Division
T. J. Watson Research Center
Yorktown Heights, NY 10598.

Abstract: In this paper, we describe efficient, scalable, and deadlock-free asynchronous communication algorithms suitable for the parallel implementation of unstructured computations on iPSC/860. Using these deadlock-free strategies, which incur small overhead, we have optimized the communication in parallel sparse Cholesky factorization. We present experimental results to show that such optimizations can lead to more than 50% saving in communication costs.

1 Introduction

Typically, parallel unstructured computations are characterized by irregular data dependencies and communication patterns that are not known at compile time. Treating these as *asynchronous data-driven* computations turns out to be advantageous. The non-blocking (at the application level) send and receive protocol supported by the MPI standard [4] and by many other communication libraries provides an efficient means for realizing asynchronous point-to-point communication on most scalable parallel processing systems. Further, this protocol allows for straightforward implementation of the asynchronous data-driven computations. On current scalable architectures, the non-blocking send/receive protocol performs efficiently if a message is posted at the destination processor prior to its arrival. One way to accomplish this is by guaranteeing that a message is sent (at the application level), only after the corresponding receive is posted at the destination. We refer to such a non-blocking send/receive protocol as a *direct* protocol. Under the direct protocol, asynchronous message passing is non-blocking from end-to-end; i.e., there is direct communication at application level with minimum buffering or blocking by the communication subsystem.

In this paper, we describe an efficient, scalable communication strategy that conforms to the direct protocol and is suitable for unstructured, data-driven computations. We show that our communication strategy is deadlock-free and, based on this strategy, we develop an optimization algorithm for message passing under the direct protocol. Using this message-passing algorithm, non-blocking send and receive messages are posted *incrementally*, during the application program execution. The incremental nature of message posting recycles both memory space and message handles, while guaranteeing deadlock-free communication. We show the performance gains resulting from our strategy using the sparse Cholesky factorization as an example of unstructured, data-driven application. On the iPSC/860, incorporation of our strategy in a block sparse Cholesky factorization resulted in a reduction of more than 50% in communication cost.

The principles developed in this paper are applicable to a more general class of problems than the example we consider and these principles can be applied at the system level or can be embedded in a user program.

2 Parallel computation and communication model

For describing our communication strategy, we model the unstructured computations and the parallel architecture for asynchronous communication as follows. We represent the unstructured computations by a directed (acyclic) task graph where nodes represent tasks and arcs represent the data flow. We consider the class of unstructured computations where such a task graph can be constructed at run-time (after some input values are made available to the program), but prior to the actual numerical computations. We assume that the tasks are distributed among processors by some allocation scheme. More than one task may be assigned to each processor and the tasks are scheduled with the aim of minimizing the total parallel execution time. We assume a distributed-memory model of parallel computation, where data is transferred among processors by explicit message passing. A task can be computed when data on all its input arcs become available; some of this input data may be computed locally by other tasks and some may be computed on one or more remote processors. When a task is computed, the data generated is sent to remote processors if tasks residing on those processors use that data as input. Thus, there are three types of activities on each processor: executing computational tasks, sending outgoing messages and receiving incoming messages.

For realizing good performance, it is desirable to use static computation and communication schedules that conform to the actual data flow. For the class of unstructured computations considered here, the data flow is determined at run-time but is available prior to the actual execution of the computational tasks. Although the computations we consider are data-driven, for performance reasons we treat them as schedulable at run-time, but prior to the actual task execution. Therefore, in our model, we assume that, prior to beginning the task executions, each processor has a schedule for the order in which the tasks assigned to that processor are to be computed. In the following, we refer to these as *static schedules*. (In this paper, we will not discuss the mechanisms for generating these schedules.) Thus, each processor executes, in a sequence, a completely ordered list of tasks. As long as the necessary data is available for execution of a task, the processor executes the task without any interruption. If some remote input data required in computing a task has not arrived when that task is scheduled for execution, then that processor *blocks* on the receive, until the data arrives. After computing the task, the generated data is sent out immediately to other processors that may need the data in their task computations. Thus, the static task schedule implicitly determines the order in which messages are sent out. Execution of a task may generate data that is needed at more than one processors and, in that case, the messages containing this data could be sent in any order. In such cases, we arbitrarily impose an order among these messages. This scheme of sending messages as soon as the corresponding tasks are computed and assigning an order to multiple messages generated by the execution of a single task, gives rise to a total order among the outgoing messages. The incoming messages may arrive at any time, but must be received prior to the execution of the task that requires the data from that message. (If multiple tasks on a processor require that data, then the message must arrive prior to the execution of the earliest scheduled task.) Thus, the latest a message may be received on a processor is just before its consumption in a task execution. Therefore, we schedule the incoming messages so that they are received just before they are needed – a strategy that allows for maximum possible flexibility. As in the case of outgoing messages, more than one message may have to received to execute a task. In such cases, we assign an arbitrary order to the receipt of these messages. Thus, from the static task schedule, a static communication schedule is implicitly defined where messages are sent and received in a predetermined order. We reiterate that under our communication schedule, messages are sent out as soon as (and no later) the data to be sent out becomes available and the messages are received just before (and no earlier) the data in those messages is needed in the execution of a scheduled task.

The complete ordering of messages in a static schedule gives rise to chains or *runs* of sends and receives. A *send run* on processor P_i with respect to processor P_j is an ordered set of consecutive sends from P_i to P_j without any intervening receive on P_i from P_j (but with intervening computations). A *receive run* is defined analogously. The message passing on a processor, with respect to a given "pairing" processor (to which it sends messages and from which it receives messages), is an alternating sequence of send and receive runs. We denote the message passing on P_i with respect to P_j by the sequence,

$$\mathcal{S}_1^{(P_i)}, \mathcal{R}_1^{(P_i)}, \mathcal{S}_2^{(P_i)}, \mathcal{R}_2^{(P_i)}, \ldots, \mathcal{S}_u^{(P_i)}, \mathcal{R}_u^{(P_i)},$$

where $\mathcal{S}_k^{(P_i)}$ and $\mathcal{R}_k^{(P_i)}$ denote the kth send and kth receive runs on P_i, respectively. Similarly, the message passing on P_j with respect to P_i is denoted by the sequence,

$$\mathcal{S}_1^{(P_j)}, \mathcal{R}_1^{(P_j)}, \mathcal{S}_2^{(P_j)}, \mathcal{R}_2^{(P_j)}, \ldots, \mathcal{S}_v^{(P_j)}, \mathcal{R}_v^{(P_j)},$$

where $\mathcal{S}_k^{(P_j)}$ and $\mathcal{R}_k^{(P_j)}$ denote the kth send and kth receive runs on P_j, respectively. Any of $\mathcal{S}_1^{(P_i)}$, $\mathcal{S}_1^{(P_j)}$, $\mathcal{R}_u^{(P_i)}$, or $\mathcal{R}_v^{(P_j)}$ could be an empty run, but both $\mathcal{S}_1^{(P_i)}$ and $\mathcal{S}_1^{(P_j)}$ cannot be empty and, similarly, both $\mathcal{R}_u^{(P_i)}$ and $\mathcal{R}_v^{(P_j)}$ cannot be empty.

We note here that the sends and receives in the communication schedule are interspersed with task computations. Data must be received before the corresponding tasks are computed and the sends can take place only after the completion of the corresponding task computations. Thus, before any of the messages in a send run $\mathcal{S}_k^{(P_i)}$, $k > 1$, can be sent, all messages in the receive run $\mathcal{R}_{k-1}^{(P_i)}$ must be received by P_i; otherwise, those messages would have been scheduled to be sent in $\mathcal{S}_{k-1}^{(P_i)}$.

We further note that because of the manner in which the communication schedules are set up, not any arbitrary sequence of send and receive runs is feasible. A communication schedule between a pair of processors P_i and P_j is *feasible* if and only if, for any m, the number of messages received by P_i in its first m messages (sent to and received from P_j) is at most equal to the number of messages sent by P_j in its first m messages (sent to and received from P_i). It is easy to see that a contradiction of this condition leads to a physically impossible situation in which a message that is not sent from the sender is received on the receiving processor. Note that there is no notion of time involved in this definition and these conditions should hold regardless of whether the message passing between the two processors is synchronous, loosely synchronous, or totally asynchronous. We assume that all communication schedules considered in this paper are feasible schedules.

The *length* of a send or a receive run $\mathcal{Q}$ is the number of messages in that run and is denoted by $|\mathcal{Q}|$. We use the notation

$$\mathcal{M}^{(P_i)} = \left(|\mathcal{S}_1^{(P_i)}|, |\mathcal{R}_1^{(P_i)}|, |\mathcal{S}_2^{(P_i)}|, |\mathcal{R}_2^{(P_i)}|, \ldots, |\mathcal{S}_u^{(P_i)}|, |\mathcal{R}_u^{(P_i)}|\right)$$

to denote the sequence of lengths of the send and receive runs on P_i with respect to some other processor P_j.

Note that neither $|\mathcal{S}_k^{(P_i)}|$ and $|\mathcal{R}_k^{(P_i)}|$ nor $|\mathcal{S}_k^{(P_i)}|$ and $|\mathcal{R}_k^{(P_j)}|$ nor $|\mathcal{S}_k^{(P_i)}|$ and $|\mathcal{S}_k^{(P_j)}|$ are necessarily equal. However,

$$\sum_{k=1}^{u} |\mathcal{S}_k^{(P_i)}| = \sum_{k=1}^{v} |\mathcal{R}_k^{(P_j)}|$$

and

$$\sum_{k=1}^{u} |\mathcal{R}_k^{(P_i)}| = \sum_{k=1}^{v} |\mathcal{S}_k^{(P_j)}|.$$

As stated in the introductory section, we wish to implement message passing among pairs of processors using the static communication schedules under the direct protocol. Non-blocking send and receives are to be used and for each message in a receive run, a non-blocking receive is *posted* ahead of time. In our architectural model, messages arrive in the same order as they are sent. Further, posting of a receive message involves specifying storage location where the data is to be received and assigning an identifier to the posted message. Specification of storage location may require allocation of a storage buffer and assigning a message identifier may involve obtaining a *message handle* from the system. In our model, the direct protocol is realized by informing the sending processor of the posted non-blocking receive, by means of a token message. A single token may be used to indicate the posting of multiple receives. A data message is sent by the sender only after receiving a token for the corresponding receive. If a processor is ready to send a message but has not received the corresponding token, then it *blocks on that send*. When the token arrives, the send operation is completed and the processor is released. A posted non-blocking receive is *cleared* once the message arrives and is retrieved into the user space. This releases the message handle for reuse. Until the posted message is cleared, the receive remains *pending*. If a task is scheduled for execution on a processor and the message containing some of the input data has not arrived from a remote processor, then the processor *blocks on that receive* until the corresponding message arrives. Clearly, under this model of execution, a pair of processors may *deadlock* if each processor blocks on a send or a receive, waiting to be released by the other.

The problem we consider in paper is that of finding an efficient and deadlock-free implementation strategy for a given static communication schedule using the direct protocol. Note that blocking of a processor on a send or receive leads to inefficiency, which must be minimized by any communication strategy.

A simple solution to this problem is to post, before commencing any of the computations on a processor, all messages to be received from all other processors during the entire program execution. This approach was used in [5] for sparse Cholesky factorization. However, this approach tends to be impractical for typical data-driven applications primarily because of two limitations inherent to this approach. First, on any architecture there is a limit on the number of receive messages that can be pending at any time. Second, posting receives beforehand requires allocation of buffer space for all incoming messages. Typical data-driven applications are highly communication-intensive. This results in a significant demand on memory leading to a limitation on the number of pending receives. Although both of these limitations can be made less restrictive by increasing the resources, there will always be problems which will exceed these limitations. Thus, the naive approach is nonscalable and, in some cases, impractical.

In the following, we describe run-time communication strategies that are not bound by any inherent (architecture dependent or problem dependent) limitation and are suitable for parallel unstructured computations. These strategies are shown to be deadlock-free and adhere to the direct protocol. Under these strategies, messages are posted *incrementally* and, thus, difficulties from limited resources are avoided and scalability is maintained.

3 Deadlock-free optimizations

In the following, we describe three strategies for implementing static communication schedule under the direct protocol. The first strategy is the most restrictive and the last is the most flexible and hence most practical. We develop communication algorithms based on these strategies and show that they are deadlock-free provided the task schedule across the processors is deadlock-free. We consider message passing between a pair of processors only. Extension of these results to all communicating processor-pairs is straightforward. In the following, $\mathbf{s}_i^{(P_1)}$ and $\mathbf{r}_j^{(P_1)}$ denote the ith message sent by processor P_1 and jth message received by processor P_1, respectively.

3.1 Receives cleared in order

First we consider a restricted case, where on any processor messages are posted and cleared in the same order in which they are received. Since messages arrive in the same order as they are sent, this assumption implies that messages are posted and cleared in the same order as they are sent. Considering P_1 and P_2, the constraints are: $\mathbf{r}_i^{(P_1)} = \mathbf{s}_i^{(P_2)}$ and $\mathbf{r}_i^{(P_2)} = \mathbf{s}_i^{(P_1)}$.

We make a further assumption that only one message is posted at a time. At the very beginning of execution, each processor posts one receive and sends a token to the sender. During execution, as soon as a processor receives a message, it posts a receive for the next message to be received (if any), and sends a token to the sender. Thus, until all messages are received, there is at least one posted receive at any time, on each processor.

Communication strategies based on these two assumptions maximize the reuse of message handles and memory buffer space. Only when a receive is cleared and the corresponding handle released, a new receive is posted. However, there are two disadvantages to this protocol. This protocol results in a pair-wise synchronization for every message sent. Moreover, there is a possibility of deadlock, as we describe next.

We first show that a pair of processors may deadlock only if both end up blocking on a send. The deadlock situations where both processors block on receives or one is blocked on a receive and the other blocked on a send would not arise.

Claim 1 *In a feasible communication schedule, the only possible deadlock situation between a pair of processors is one in which each processor blocks on a send.*

Proof: Assume that there is a deadlock situation in which P_1 blocks while sending $\mathbf{s}_i^{(P_1)}$ and P_2 blocks while receiving $\mathbf{r}_j^{(P_2)}$.

Messages are posted and cleared in the same order as they are sent, and hence P_2 cannot remain blocked on a message that P_1 has already sent. Since P_1 has successfully sent $i-1$ messages, j must be greater than $i-1$; i.e., $j \not< i$. Also, the receive cannot be for a message that would be sent by P_1 after sending $\mathbf{s}_i^{(P_1)}$ on which it is assumed to block. If this were so, then it would mean that P_2 has successfully received $j-1$ messages including the ith message; i.e., $j \not> i$. Therefore, i and j must be the same; i.e., the receive on which P_2 is assumed to block is for the message that P_1 is trying to send.

Since at any time there is at least one pending receive on P_2, and P_2 is ready to clear the next receive, P_1 cannot block. Also, since the receives for messages in P_2 are cleared in the order in which the messages are sent from P_1, P_1 can send the message to P_2 and release both itself and P_2, which contradicts the deadlock assumption.

Next, assume that there is a deadlock situation in which both P_1 and P_2 block on receive. Under the feasible communication schedule, it is not possible for both processors to block on receives that will be satisfied by future sends. To see this, let P_1 block on its mth message (counting both sends and receives) and let P_2 block on its nth message (sends plus receives). Without loss of generality, let m be at most equal to n. Let r be the total number of receives in the m messages on P_1 (including the mth message – a receive). To satisfy the condition of feasible communication schedule, there must be at least r sends in the first m messages (hence in the first n messages) handled by P_2. Thus, the mth message on P_1 is guaranteed to be cleared and P_1 to be released by a message sent from P_2 prior to its blocking on the n message. Thus, P_1 and P_2 cannot deadlock while both are blocked on receives. ∎

We now consider the conditions under which two processors will deadlock while sending messages. The example shown in Figure 1 illustrates a deadlock situation in a feasible communication schedule between P_1 and P_2. In that figure, the circled numbers are the lengths of the receive runs and the **bold** numbers are the lengths of the send runs; i.e., $\mathcal{M}^{(P_1)} = (2,5,3,5,2,2,1,0)$ and $\mathcal{M}^{(P_2)} = (0,2,6,2,2,1,4,3)$. This results in a deadlock situation with P_1 blocked in $\mathcal{S}_3^{(P_1)}$ and P_2 blocked in $\mathcal{S}_4^{(P_2)}$. At the very beginning, each processor posts a receive for the first message to be received. After that, every time a receive is cleared in a receive run, a new receive is posted. At the end of $\mathcal{R}_2^{(P_1)}$, P_1 has posted a total of 11 receives, and at the end of $\mathcal{R}_3^{(P_2)}$, P_2 has posted a total of 6 receives. Before beginning run $\mathcal{S}_3^{(P_1)}$, P_1 has sent a total of 5 messages. It sends the first message in $\mathcal{S}_3^{(P_1)}$, and then blocks on the second send, waiting for P_2 to post more receives. P_2, on the other hand, has sent a total of 8 messages before beginning run $\mathcal{S}_4^{(P_2)}$. It sends the first 3 messages in $\mathcal{S}_4^{(P_2)}$, and then blocks on the fourth send, waiting for P_1 to post more receives. Thus, both processors are in a deadlock.

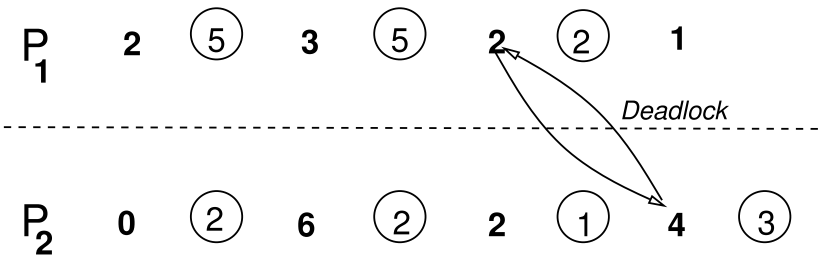

FIGURE 1
A deadlock situation.

We now state the general condition for deadlock as illustrated in the above example. Let $\mathcal{T}_i^{(P_1)}$ be the total number of receives posted on processor P_1 at the end of $\mathcal{R}_i^{(P_1)}$. Similarly we define $\mathcal{T}_i^{(P_2)}$ for processor P_2.

Claim 2 *A deadlock, with P_1 blocking in $\mathcal{S}_\alpha^{(P_1)}$ and P_2 blocking in $\mathcal{S}_\beta^{(P_2)}$ occurs if and only if*

$$\mathcal{T}_{\alpha-1}^{(P_1)} < \sum_{j=1}^{\beta} |\mathcal{S}_j^{(P_2)}| \ \ \textit{and} \ \ \mathcal{T}_{\beta-1}^{(P_2)} < \sum_{j=1}^{\alpha} |\mathcal{S}_j^{(P_1)}|.$$

Proof: The sufficiency of the condition is easy to see. When P_1 begins the send run $\mathcal{S}_\alpha^{(P_1)}$ and P_2 begins the send run $\mathcal{S}_\beta^{(P_2)}$, neither processor has sufficient number of posted receives to satisfy all the sends in the other processor's send run. On the other hand, no fresh receives can be posted unless a send run is terminated. Therefore, both P_1 and P_2 block, causing a deadlock. Now we show the necessity of this condition by contradiction. Let $\mathcal{T}_{\alpha-1}^{(P_1)}$ be at least equal to $\sum_{j=1}^{\beta} |\mathcal{S}_j^{(P_2)}|$. (A symmetric argument applies for $\mathcal{T}_{\beta-1}^{(P_2)} \geq \sum_{j=1}^{\alpha} |\mathcal{S}_j^{(P_1)}|$.) Then, P_2 will complete the run $\mathcal{S}_\beta^{(P_2)}$ without blocking. By Claim 1, there cannot be a deadlock with P_2 in a receive run, and hence, P_2 will complete $\mathcal{R}_\beta^{(P_2)}$, and enter $\mathcal{S}_{\beta+1}^{(P_2)}$. Set $\beta \leftarrow \beta + 1$ and we are back to the invariant in the claim. Continuing this process, eventually both P_1 and P_2 will complete all send and receive runs. ∎

The condition for deadlock as stated in the above claim, does not depend on the number of messages posted at a time. When a fixed (a constant) number of messages are posted at a time, the condition for deadlock can be made more precise. Under our current set of assumptions, after clearing each receive, one more receive is posted. In general, one could post any fixed number of messages at a time and repost the same number of messages whenever all posted messages are cleared. If $\lambda^{(P_1)}$ and $\lambda^{(P_2)}$ are the number of receives posted at a time by P_1 and P_2, respectively, then the two processors deadlock whenever the following conditions holds:

$$\lambda^{(P_1)} + \sum_{j=1}^{\alpha-1} |\mathcal{R}_j^{(P_1)}| \;<\; \sum_{j=1}^{\beta} |\mathcal{S}_j^{(P_2)}|$$

and

$$\lambda^{(P_2)} + \sum_{j=1}^{\beta-1} |\mathcal{R}_j^{(P_2)}| \;<\; \sum_{j=1}^{\alpha} |\mathcal{S}_j^{(P_1)}|.$$

If each processor has exactly one pending receive at all times, as in the example of Figure 1, then $\lambda^{(P_1)}$ and $\lambda^{(P_2)}$ are both equal to 1. For the example of Figure 1, the above conditions indicate that there is a deadlock when α is 3 and β is 4.

To get around the deadlock situation, we propose a new communication strategy which allows for posting of a variable number of messages. An outline of the new communication strategy, for message passing between P_i and P_j, is as follows:

i. The receive messages are posted only at the beginning of a send run (but not necessarily at the beginning of every send run). The actual receives take place in the receive runs, as before.

ii. At a time, multiple messages may be posted in *batches*.

iii. The number of messages, $m_k^{(P_i)}$, to be posted before beginning with the send run $\mathcal{S}_k^{(P_i)}$ on P_i is determined as follows. After completing a receive run $\mathcal{R}_{k-1}^{(P_i)}$ on P_i, if there is a send run $\mathcal{S}_k^{(P_i)}$, then we first identify $\mathbf{s}_n^{(P_i)}$ such that $n = \sum_{l=1}^{k} |\mathcal{S}_l^{(P_i)}|$; i.e., $\mathbf{s}_n^{(P_i)}$ is the last message sent by P_i in send run $\mathcal{S}_k^{(P_i)}$ and n is its sequence number. We then identify, on P_j, the receive run $\mathcal{R}_{k'}^{(P_j)}$ that contains the receive message $\mathbf{r}_n^{(P_j)}$, which corresponds to $\mathbf{s}_n^{(P_i)}$. From this, $m_k^{(P_i)}$, the number of messages to be posted, is determined as the following:

$$\sum_{l=1}^{k} m_l^{(P_i)} \;=\; \max\left(\sum_{l=1}^{k} |\mathcal{R}_l^{(P_i)}|,\;\; \sum_{l=1}^{k'-1} |\mathcal{S}_l^{(P_j)}|\right); \tag{1}$$

i.e.,

$$m_k^{(P_i)} = \max\left(\sum_{l=1}^{k} |\mathcal{R}_l^{(P_i)}|,\;\; \sum_{l=1}^{k'-1} |\mathcal{S}_l^{(P_j)}|\right) - \sum_{l=1}^{k-1} m_l^{(P_i)}. \tag{2}$$

Length of the send run $\mathcal{S}_0^{(P_j)}$ is assumed to be 0. Also, if $m_k^{(P_i)}$ is negative then we set $m_k^{(P_i)}$ to 0 and do not post any messages.

iv. A similar strategy is followed on P_j to determine the number of messages to be posted before entering each of its send runs.

Note that in the above algorithm, when P_i enters the kth send run, $\mathcal{S}_k^{(P_i)}$, at least all the messages in the kth receive run, $\mathcal{R}_k^{(P_i)}$, are guaranteed to be posted. Note that for the execution of the above stated algorithm on P_i, it is necessary to know the range for all send and receive runs of P_j (and vice versa, for P_j). We refer to this algorithm as the *In-order* communication algorithm. In the following, we show that the strategy followed by this algorithm guarantees deadlock-free message passing.

Claim 3 *The Algorithm* In-order *guarantees deadlock-free message passing.*

Proof: Consider processor P_i. Suppose that all messages in the receive run $\mathcal{R}_{k-1}^{(P_i)}$ are received and the processor is about to begin sending messages in the next send run $\mathcal{S}_k^{(P_i)}$. Processor P_i will be able to send all messages in $\mathcal{S}_k^{(P_i)}$ without blocking if P_j is able to post receive messages for all messages in $\mathcal{S}_k^{(P_i)}$ including $\mathbf{s}_n^{(P_i)}$ which is the last message in that send run.

Suppose that $\mathbf{s}_n^{(P_i)}$ is received by $\mathbf{r}_n^{(P_i)}$ on P_j in the receive run $\mathcal{R}_{k'}^{(P_j)}$. From Eq (1), we note that, in algorithm *In-order* , messages for all of $\mathcal{R}_{k'}^{(P_j)}$ are guaranteed to be posted if P_j arrives at the send run $\mathcal{S}_{k'}^{(P_j)}$ (where the $m_{k'}^{(P_j)}$ messages are posted), without being blocked earlier.

Processor P_j is guaranteed to arrive at $\mathcal{S}_{k'}^{(P_j)}$ if it is able to send all messages in all of its send runs up to and including $\mathcal{S}_{k'-1}^{(P_j)}$. (By Claim 1 the processor will not be blocked in a receive run.) Thus, if P_i has posted receive messages corresponding to all messages in $\{\mathcal{S}_1^{(P_j)}, \mathcal{S}_2^{(P_j)}, \ldots, \mathcal{S}_{k'-1}^{(P_j)}\}$ prior to entering the send run $\mathcal{S}_k^{(P_i)}$ then P_j is guaranteed to arrive at $\mathcal{S}_{k'}^{(P_j)}$ and subsequently post all receive messages in $\mathcal{R}_{k'}^{(P_j)}$.

By posting $m_k^{(P_i)}$ messages (as determined by Eq (1)) on P_i before it enters *srunki*, we guarantee that altogether P_i has posted $\sum_{l=1}^{k'-1} |\mathcal{S}_l^{(P_j)}|$ messages, which ensures that P_j will arrive at send run $\mathcal{S}_{k'}^{(P_j)}$ without being blocked. (Note that messages are received in the order they are sent.)

Thus, P_i will not be blocked in any send run, $\mathcal{S}_k^{(P_i)}$.

By a similar proof, we can show that P_j will not be blocked in any of its send runs. ∎

We now revisit the example of Figure 1 and apply the new communication strategy for message passing. Using Eq (2) to determine the number of messages to be posted before each send run, we obtain the batch sizes for this example problem as follows:

$$m_1^{(P_1)} = 5,\ m_2^{(P_1)} = 5,\ m_3^{(P_1)} = 2, \text{ and } m_4^{(P_1)} = 0.$$

$$m_1^{(P_2)} = 2,\ m_2^{(P_2)} = 2,\ m_3^{(P_2)} = 1, \text{ and } m_4^{(P_2)} = 2.$$

It may be verified that by posting messages in batches as specified above, deadlock-free execution is guaranteed.

We note here again that for the communication strategy outline above, the messages must be posted and cleared in the same order as they are sent. Posting messages in batches of multiple messages alleviates this problem to some extent, but the message order must be preserved across the batches. This implies that the computation and communication must be scheduled in a consistent fashion across the processors. As a result, depending on the nature of the unstructured computations, the dependency structure of the entire task graph may have to be taken into account in order to determine feasible schedules. Another point to note is that there is an implicit synchronization among processors so as to adhere to the direct protocol outlined in Section 1. The frequency of synchronization and hence the synchronization overhead is determined by the average size of the send and receive runs. Small size send and receive runs imply small batch sizes and hence larger synchronization overhead. The batch sizes can be made large by bundling together messages from multiple receive runs without sacrificing the deadlock-free property. The upper limit on the number of messages to be posted in a batch is determined by the total number of message handles available at a time and the amount of memory available for allocating the buffer space. Even with such modifications, there may be considerable synchronization overhead if there a significant time lag between the sends in one processor and the corresponding receives on the other processor. This can happen when the computations and message generations proceed at a higher rate as compared to the message transmission rate (e.g., high cpu rate and low latency as compared to the bandwidth). In such cases, the computations would tend to be more and more synchronous as the message passing would be determined by mostly the availability of tokens rather than the availability of data or of messages to send.

3.2 Out-of-order receives

In the above discussion, we assumed that the receives are cleared in the order in which they are sent from the sending processor. The strategies based on this assumption work well when the data is consumed on each processor in the same order in which it is produced on remote processors. Scheduling computations, in this manner, for general unstructured applications is difficult, if not impossible. We now consider the problem of developing deadlock-free message passing strategies that are based on less restrictive assumptions. Specifically, we consider cases where arriving messages may be used, in the receiving processor, in an order different from their order of arrival. We assume, however, that all messages arrive in the same order as the order in which they are sent. Thus, for these cases, the order in which messages are cleared need not match with the order in which they are sent. We refer to these as cases with *out-of-order receives.* Thus, the strategy we develop here is useful where data is not necessarily consumed in the same order in which it is produced on the remote processors.

An immediate consequence of allowing out-of-order receives is that, under the direct protocol, deadlocks cannot be avoided by posting the next message whenever a previously posted message is cleared. This is because, the sender might be blocked on sending a message other than the one the receiver has posted and for which it has sent a token to the sender. Thus, multiple messages must be posted. But simply posting multiple messages in batches of size given by Eq (2) does not ensure deadlock-free message-passing. The number of messages posted must be selected so that every arriving message finds a matching posted message; i.e., the number of messages posted at any time must be large enough to cover both messages that will be arriving and those that will be needed in the next scheduled computations (i.e., those that are part of the next receive runs). More precisely, the number of messages posted, k, is such that every message in the next k arrivals finds a matching message in the set of posted messages. The order of arrivals of the k messages need not match the order in which the k messages are posted or cleared. We define a set of sends to *match* a set of receives if and only if, for every message in the send set, there is a matching message in the receive set. Note that if the receiver tries to receive messages in the exact reverse order as of the sender's, then the receiver must post all messages at the very beginning in order to avoid a deadlock. However, in practice, this type of worst case scenario is rarely encountered.

As an example, consider the case where messages numbered 47 through 54 are sent in that order from P_1 to P_2. Suppose they are needed in P_2 in the order 47, 49, 50, 48, 52, 51, 53, and 54. Then, if P_2 posts receives for the first five messages that it needs (47, 49, 50, 48, and 52) in one batch and blocks until it receives all of them, it will get the messages 47 through 50, but it will block on the pending receive for message 52. On the other hand, P_1 blocks on the send of message 51 since it does not have a token from P_2 for this sending this message. Clearly, there is a deadlock. To avoid deadlock, P_2 must post at least six receives to match the six sends from P_1.

Thus, to guarantee deadlock-free communication, the messages must be grouped accordingly. We refer to each group of messages that form matching sets among the arriving messages and the messages to be received as a as a *window* of messages for communication. Each receive message belongs to one window. A window may include messages from one or more receive runs, but the concept of window does not obey the receive run boundaries. Overall, however, the messages follow the same original order. Receives are posted in batches as before, where a batch consists of one or more windows of messages. Note also that all the messages in a window must be posted simultaneously.

First we formalize the definition of a window of consecutive receive messages. The following inductive definition specifies how a sequence of receives on processor P_1 is grouped into non-overlapping windows.

Definition 1: Let $W_1^{(P_1)}, W_2^{(P_1)}, \ldots, W_{i-1}^{(P_1)}$ be the first $i-1$ windows on processor P_1 and suppose that altogether l messages are covered by these $i-1$ windows. The ith window, $W_i^{(P_1)}$, is defined to consist of the next k, $k \geq 1$, messages to be received on P_1 so that k is the smallest number of receives that would match the same number of sends at P_2, starting from message number $l+1$.

Note that the concept of windows forms an order among the messages sent and received. Although the messages are received in a different order than the order in which they are sent, all the messages in the window $W_{i-1}^{(P_1)}$ are sent and received before any of the messages in the next window $W_i^{(P_1)}$. Further, if there are n messages in the first i windows (i.e., $n = \sum_{w=1}^{i} |W_w^{(P_1)}|$), then those n messages are the receive messages corresponding to the first n messages sent on P_2.

The following example illustrates the division of messages into windows. Suppose that P_1 sends to P_2, messages numbered 1 through 12, in that order and that P_2 clears the messages in the order $< 1, 3, 2, 5, 4, 6, 7, 9, 10, 8, 12, 11 >$. These messages may belong to one or more send runs on P_1 and to one or more receive runs on P_2. Then, the windows of receives in P_2 are $< 1 >$, $< 3, 2 >$, $< 5, 4 >$, $< 6 >$, $< 7 >$, $< 9, 10, 8 >$, and $< 12, 11 >$, where each tuple is a window of one or more messages identified by their message identification numbers. By posting and clearing messages on P_2 in batches formed by the windows, message passing can take place without any deadlock.

Thus, a communication strategy for accommodating out-of-order receives is to first form a sequence of windows that cover all the messages to be received (according to the inductive Definition 1) and then post and clear messages in each window. However, if messages are posted from one window at a time, then a deadlock can take place. This is because, by definition, windows do not necessarily obey receive run boundaries and hence, two processors could be in the midst of their respective send runs when the windows on both processors are exhausted. Since neither processor is able to clear messages until it has completed its current send run, the next window of messages would not be posted and this would lead to a deadlock. A simple example is the case where, on both processors, messages happen to be received in the same order as they are sent, but both processors begin with a send run consisting of more than one message. In this example, there is one message per window, but more than one message in the first send run. Thus, if each processor posts the first window of messages (actually, one message) and starts with the send run, then each processor will block on the second send and there will be deadlock.

To avoid deadlocks such as the one described above, we propose an algorithm that is similar to algorithm *In-order* , but here messages are posted in batches of multiple windows instead of batches of messages. As before, messages are posted only before entering a send run. The number of windows of messages to be posted on P_1 when it is about to enter $\mathcal{S}_k^{(P_1)}$ is determined as follows. Let $\mathbf{s}_n^{(P_1)}$ be the last message in $\mathcal{S}_k^{(P_1)}$. We first determine the corresponding receive message $\mathbf{r}_{n'}^{(P_2)}$ on P_2 and the window to which this message belongs to. Let $W_m^{(P_2)}$ be the window which covers $\mathbf{r}_{n'}^{(P_2)}$. and let $\mathcal{R}_{m'}^{(P_2)}$ be the earliest receive run covered by $W_m^{(P_2)}$. If $\mathbf{s}_l^{(P_2)}$ is the last message in the send run $\mathcal{S}_{m'-1}^{(P_2)}$, then let $\mathbf{r}_{l'}^{(P_1)}$ be the corresponding receive on P_1 and let $W_{m'}^{(P_1)}$ which covers $\mathbf{r}_{l'}^{(P_1)}$. Then to avoid deadlock, before entering $\mathcal{S}_k^{(P_1)}$, P_1 must have posted altogether s windows so that

$$\sum_{w=1}^{s} |W_w^{(P_1)}| \geq \max\left(\sum_{r=1}^{k} |\mathcal{R}_r^{(P_1)}|, \quad \sum_{r=1}^{m'-1} |\mathcal{S}_r^{(P_2)}|\right). \tag{3}$$

Thus, the number of new receives to be posted on P_1 before it enters $\mathcal{S}_k^{(P_1)}$ is given by,

$$m_k^{(P_1)} = \sum_{w=w'}^{s} |W_w^{(P_1)}|, \tag{4}$$

where w' is the number of windows already posted.

We refer to the above algorithm as the *Out-of-order* communication algorithm. As mentioned earlier, this algorithm is similar to algorithm *In-order* except that now the messages are to be posted and received in terms of multiple windows instead of multiple messages. The proof that this algorithm performs communication in a deadlock-free manner is similar to that for *In-order* .

3.3 A general scalable algorithm

We now present a strategy that is more flexible and just as efficient as the one presented above. First, notice that posting more messages than the required minimum given by Eq (4) does not introduce deadlocks. Therefore, to simplify the algorithm, first we use the length of the largest send run and of the largest receive run instead of the lengths of individual send and receive runs. Second, instead of posting a new batch of messages at the beginning of every send run, we allow posting at any point in some receive run. For this approximate scheme, the following describes how to choose the batch size, and when to post a new batch of receives.

In the generalized *Out-of-order* algorithm, first $m_1^{(P_i)}$ receive messages are posted on P_i at the very beginning of execution, and subsequently, a fresh batch of $m_k^{(P_i)}$ receives are posted whenever the number of pending receives dwindles to Δ_i. The size of $m_k^{(P_i)}$ and Δ_i are given by the following claim. The messages could be posted at any point in any receive run. Let $U_s^{(P_i)}$ and $U_r^{(P_i)}$ be the lengths of the longest send run and the longest receive run in P_i.

Claim 4 *A choice of* $m_k^{(P_i)} = \Delta_k^{(P_i)} = \max(U_s^{P_j}, U_r^{P_i})$, *and* $m_k^{(P_j)} = \Delta_k^{(P_j)} = \max(U_s^{P_i}, U_r^{P_j})$ *guarantees deadlock-free communication between processors* P_i *and* P_j.

4 Communication optimization for sparse Cholesky factorization on iPSC860

In the following, we first briefly discuss the message passing protocols on the iPSC/860 and then show how our communication optimization algorithm is applicable to reduce latency cost by a significant margin.

4.1 Message passing on iPSC/860

Message passing calls on the iPSC/860 are classified into asynchronous and synchronous types. These are not to be confused with the asynchronous communication discussed above.

All communication in our application is asynchronous, but the communication may be carried out by using the iPSC/860 synchronous or asynchronous message-passing calls. Messages of both types are characterized by a length, a type and an id. A message is uniquely identified by its message type, and is supplied by the user. The id is an identifier used to check for the completion of asynchronous messages, and is generated by the system.

On the iPSC/860, a normal message with more than 100 bytes of data follows a three-trip communication protocol, involving handshaking between the sender and the receiver. The sending processor first sends the length of the message to the receiver. The receiving processor checks if the corresponding receive has already been posted. If not, it allocates space for the message in the system buffer. It then sends an acknowledgement to the sending processor. Upon receiving the acknowledgement, the sending processor transmits the message. A *force type* message, on this system, however, bypasses normal control flow mechanisms. The message is sent without any handshaking. It assumes that the receiving processor has allocated space and sends the message off in a single trip. If the destination processor has not issued a receive, the message is lost. In [1], the time in microseconds to send a force type message of length m bytes over distance d on the iPSC/860 is shown to be $t = 95 + 0.394m + 10.3d$. The unforced messages take the same time if the message length is at most 100 bytes; for larger lengths, the time is $t = 164 + 0.398m + 29.9d$. Thus, for low communication costs, it desirable to use the force type messages.

We implement the direct protocol using the *force type* messages on the iPSC/860. In this section, we present the communication performance results of a parallel sparse block Cholesky factorization code. For further information on the sparse Cholesky factorization refer to [3] and on the parallel sparse block Cholesky factorization refer to [6, 7] and the references therein.

Matrix	Description	Order	Nonzeros
BCSSTK13	Fluid flow problem	2003	265575
BCSSTK15	Module of an offshore problem	3948	637680
BCSSTK16	Corps of Engineers dam	4884	749027
BCSSTK17	Elevated Pressure Vessel	10974	1020970
L7PT16	7-point Laplacian discretization	4096	257288
L7PT20	7-point Laplacian discretization	8000	646839
L9PT127	9-point Laplacian discretization	16129	563345

TABLE 1
Test matrices.

4.2 Test Matrices

For our experiments, we used both structured and unstructured sparse matrices. A brief description of each test case appears in Table 1. The first four cases are from the Harwell-

Boeing sparse matrix collection [2], and are highly unstructured. For brevity, these matrices will also be referred to as B13, B15, B16, and B17, respectively. The last three cases are 7-point Laplacian discretizations of structured grid problems. These cases will be referred to as L16, L20, and L127, respectively. The entry "nonzeros in factor" is the number of nonzeros in the lower triangular factor including the elements on the diagonal.

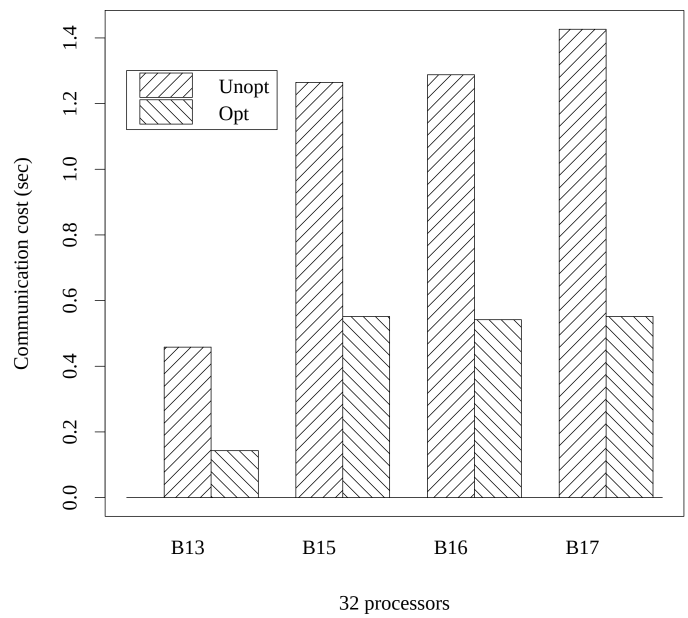

FIGURE 2
Comparison of communication costs for unstructured matrices on 32 processors.

4.3 Communication performance

We compare the communication costs on 32 processors for the unstructured matrices in Figure 2 and for the structured cases in Figure 3. The unoptimized communication costs are those using the regular blocking send/receive mechanism. The optimized communication costs are those using the non-blocking sends/receives implemented using the modified *Out-of-order* communication algorithm described in Section 3.3. In Figures 4(a) and 4(b), we show the improvements in communication cost performance for test matrices B15 and L127, respectively.

Notice that in all the test cases shown, the reduction in communication cost is at least 50%, with up to 70% reduction in cost for the best case. The reduction in communication is an important gain. Our experiments show that without any optimization of communication, the communication contributes twice as much or more to the total execution time as computation [6]. After communication optimization, the communication time is never more than the computation time, and we thus arrive at a balance between computation and communication

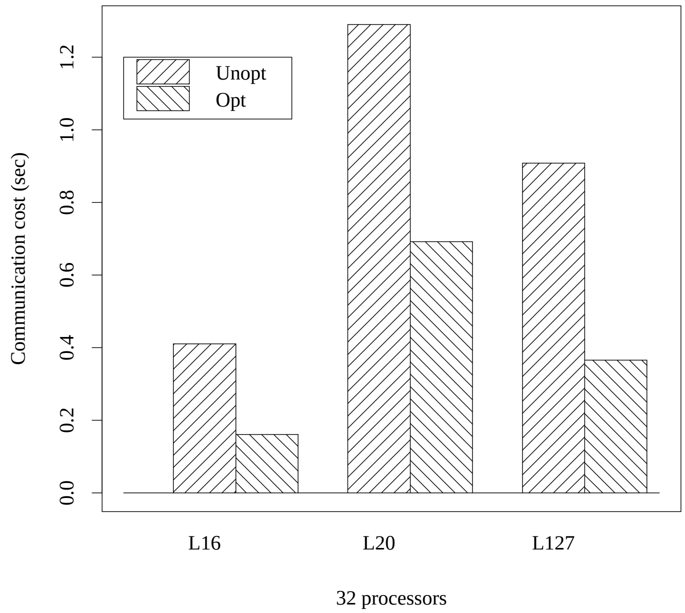

FIGURE 3
Comparison of communication costs for structured matrices on 32 processors.

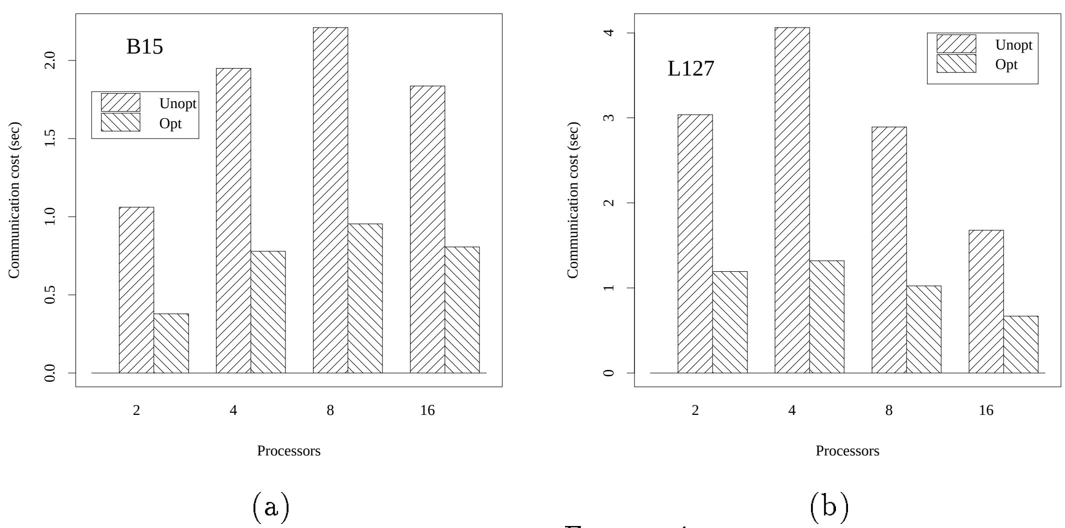

FIGURE 4
Comparison of communication costs for (a) BCSSTK15 and (b) L9PT127 sparse matrices.

time on a processor.

5 Conclusions

In this paper, we have described efficient communication algorithms that are suitable for unstructured computations. The algorithms we have developed are based on non-blocking send and receive primitives and are designed to adhere to the direct protocol, where a message is sent only after a corresponding receive is posted at the destination. Our communication strategies minimize the blocking time spent by the processors and also guarantee that there is no deadlock. Using these deadlock-free strategies, we have optimized the communication in parallel sparse Cholesky factorization. We present experimental results to show that such optimizations lead to more than 50% saving in communication costs. Currently we are testing these strategies with other type of sparse matrix computations. As a final comment, we note that although our the asynchronous communication model resembles closely with that on iPSC/860, the model is quite general and can be used as a model for examining many existing and proposed MIMD message passing systems.

Acknowledgement: The authors would like to thank Yousuff Hussaini of ICASE, NASA Langley Research Center for generously granting the use of their 32 node iPSC/860.

References

[1] S. H. Bokhari. Complete exchange on the iPSC-860. Technical Report ICASE Report No. 91-4, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA, 1991.

[2] I. S. Duff, R. G. Grimes, and J. G. Lewis. Sparse matrix test problems. *ACM Transactions on Mathematical Software*, 15-1:1–14, 1989.

[3] J. A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Prentice-Hall, Inc., Englewood Cliff, NJ, 1981.

[4] D. W. Walker. The design of a standard message passing interface for distributed memory concurrent computers. *Parallel Computing*, 20:657–673, 1994.

[5] S. Venugopal, V. K. Naik, and J. Saltz. Performance of distributed sparse Cholesky factorization with pre-scheduling. In *Proceedings of Supercomputing '92*, pages 52–61, 1992.

[6] S. Venugopal. *Parallelizing Unstructured Sparse Matrix Computations on Large-Scale Multiprocessors*. PhD thesis, Department of Computer Science, Rutgers University, 1993. Technical report DCS-TR-300.

[7] S. Venugopal and V. K. Naik. An environment for partitioning and run-time scheduling of parallel block sparse Cholesky factorization. Submitted to Parallel Computing.