Scalable Communication Protocols for Dynamic Sparse Data Exchange

Torsten Hoefler
Open Systems Lab
Indiana University
Bloomington, IN, USA
htor@cs.indiana.edu

Christian Siebert
NEC Laboratories Europe
NEC Europe Ltd., Rathausallee 10
Sankt Augustin, Germany
siebert@it.neclab.eu

Andrew Lumsdaine
Open Systems Lab
Indiana University
Bloomington, IN, USA
lums@cs.indiana.edu

Abstract

Many large-scale parallel programs follow a bulk synchronous parallel (BSP) structure with distinct computation and communication phases. Although the communication phase in such programs may involve all (or large numbers) of the participating processes, the actual communication operations are usually sparse in nature. As a result, communication phases are typically expressed explicitly using point-to-point communication operations or collective operations. We define the dynamic sparse data-exchange (DSDE) problem and derive bounds in the well known LogGP model. While current approaches work well with static applications, they run into limitations as modern applications grow in scale, and as the problems that are being solved become increasingly irregular and dynamic. To enable the compact and efficient expression of the communication phase, we develop suitable sparse communication protocols for irregular applications at large scale. We discuss different irregular applications and show the sparsity in the communication for real-world input data. We discuss the time and memory complexity of commonly used protocols for the DSDE problem and develop $\mathcal{NBX}$—a novel fast algorithm with constant memory overhead for solving it. Algorithm $\mathcal{NBX}$ improves the runtime of a sparse data-exchange among 8,192 processors on BlueGene/P by a factor of 5.6. In an application study, we show improvements of up to a factor of 28.9 for a parallel breadth first search on 8,192 BlueGene/P processors.

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

General Terms Algorithms, Theory, Performance

Keywords Sparse data exchange, Irregular algorithms, Alltoall, Distributed termination, Nonblocking collective operations

1. Introduction

Parallel scientific codes based on message passing often consist of distinct communication and computation phases, where all processes are involved in either communication or computation. This *bulk synchronous* programming model [1] is a powerful organizational principle that simplifies parallel programming by providing step-wise consistency. Making the communication phase as efficient as possible is still a problem that must be solved by the programmer.

Although all of the parallel processes may be communicating during the communication phase, each process may be communicating with only a small subset of its peers. We will refer to the communication operation(s) performed during the communication phase as *data-exchange operation(s)*.

Whereas communication relations in many parallel applications with static grids and stencils do not change over time, there are an increasing number of application domains that have rapidly changing communication patterns. Many dynamic applications belong to the emerging class of irregular "informatics" applications. Currently defined static dense and sparse [2] collective operations do not match such communication patterns well.

To satisfy the requirements of irregular applications at large-scale machines, a new approach to describing and implementing data-exchange operations is required. In this paper, we consider this *sparse data-exchange* (SDE) problem. We analyze the space and time complexity of different SDE operations. In addition, we develop a new scalable algorithm—$\mathcal{NBX}$—and implementations that support sparse data exchange in large-scale runs with good performance.

1.1 Outline and Contributions

In the next section, we define the static and dynamic sparse data exchange problems. Optimizations for the static variant have been analyzed in previous research works and thus, we focus on the dynamic sparse data exchange (DSDE) with changing communication patterns. In our theoretical analysis of the communication algorithms, we use the established LogGP network model. In Section 2.2 we present a short introduction and overview of the LogGP. We then proceed to discuss bounds on the execution time of three fundamental communication functions that are used in various protocols that implement DSDE: census, personalized census, and personalized exchange. In the census function all processes agree on a globally accumulated value, the personalized census allows distinct values for each process, and in the personalized exchange, each process sends data to each other process. The established bounds will be used later in the paper to derive bounds on the runtime of possible protocols to implement DSDE.

In Section 3, we describe three commonly used protocols to implement DSDE: personalized exchange, personalized census, and remote summation. All three protocols run in two distinct phases: the first phase exchanges the meta-information about the communication peers and the second phase communicates the actual data. We analyze all three algorithms with regards to their time and memory complexity and provide lower bounds that hold irrespectively

PPoPP'10, January 9–14, 2010, Bangalore, India.

of how the used collective functions are implemented. We conclude that the needed time and memory to perform personalized exchange and personalized census grows linearly with the number of participating processes. The remote summation algorithm uses remote memory access (data accumulation) to achieve constant space and logarithmic time complexity with the number of participating processes.

After the discussion of different implementation options, we propose a new algorithm, nonblocking census, that uses nonblocking collective communication. This algorithm merges the two phases into an optimized communication protocol. We define different possibilities to implement the nonblocking census algorithm (depending on the properties of the communication network). Our algorithm does not require remote memory access and runs with constant space and logarithmic time in the LogGP model.

In Section 4, we present a parametrization of our models and an experimental evaluation on three different large-scale systems. We present parametrized LogGP models for the algorithms. We also discuss how the time complexity of the optimized algorithm changes with the communication pattern. Then we show results for a microbenchmark where each process sends to fixed number of random peers in each communication round.

In Section 5, we discuss the use of the sparse data exchange protocol and the structure of the communication topology for several dynamic applications. Our analysis of parallel graph algorithms, parallel n-body methods and sparse matrix computations shows that our new protocol can be used to improve the performance of the sparse data exchange in such applications. However, we remark that this might depend on the specific properties of the computed system (sparsity and topology) and the communication parameters (LogGP) of the used architecture.

2. Preliminaries and Background

We start with the definition of the problem and a discussion of the building-blocks which are used later in the paper.

2.1 The Sparse Data-Exchange Problem

The SDE problem is relevant to many parallel applications. It has been discussed in the context of the Message Passing Interface (MPI) to enable optimized message scheduling and process mapping on large-scale systems [2]. However, performing such optimizations requires time to compute optimized mappings as well as communication schedules, and often involves the heuristic solution of NP-hard problems (e.g., graph embedding problems or graph coloring problems for optimized communication schedules). Such optimizations are thus generally only appropriate for *static* communication schemes, communications where the same communication pattern is used multiple times. We define the SDE problem as follows:

Definition 1. *Let $\mathcal{P}$ be the set of processes of a parallel program with C communication steps and let $P = |\mathcal{P}|$ be the cardinality of this set. In each communication step $0 \leq c < C$, each process $p \in \mathcal{P}$ communicates with (sends to or receives from) a set of processes $\mathcal{N}_p^c \subseteq \mathcal{P}$, which is called the c-neighborhood of p. The communication is called a sparse data exchange (SDE) if $k = \max_{p \in \mathcal{P}, 0 \leq c < C}(|\mathcal{N}_p^c|) = \mathcal{O}(\log P)$.*

In this general definition of SDE, each process is allowed to change its communication neighborhood frequently and arbitrarily. We also generally assume that k is not known in advance and varies across communication steps.

Not all applications change communication patterns frequently. The static version of the SDE has been analyzed in previous works and possible optimizations are well documented (e.g., [3, 4]). However, the dynamic variant, which is important to many irregular applications, received less attention so far. Each process initially knows only the processes it sends messages to. However, knowledge about the whole neighborhood (also about incoming messages) must be acquired to perform the data exchange. We define the static and dynamic SDE as follows:

Definition 2. *The SDE problem on P processes is called a static sparse data exchange (SSDE) if the neighborhoods $\mathcal{N}_p^i$ ($0 \leq i < C - 1; 0 < p \leq P$) can be determined before the algorithm runs. In contrast, it is called a dynamic sparse data-exchange (DSDE) problem when processes only know the part of the neighbors that they send messages to and the neighbors to receive from need to be determined as part of the exchange.*

In this work, we analyze algorithms that implement DSDE in theory and practice. We introduce the LogGP network model in the next section.

2.2 The LogGP Model

In this paper, we analyze protocols for solving the DSDE using the LogGP network transmission model [5]. LogGP consists of the five parameters, L, o, g, G, and P, and allows us to analyze the time complexity of parallel algorithms. The parameters are:

L: Maximum latency between any two processes in the system

o: Process-local CPU overhead to send or receive a message

g: Process-local injection or reception overhead; that is, the minimum time between two message injections or receptions

G: Transmission time per byte

P: Total number of processes in the system.

The LogGP model describes a network as $P(P-1)$ independent unidirectional channels from process p_i to process p_j ($0 \leq i, j < P$, $i \neq j$). Messages are delivered in order in each channel. For example, a simple transmission of 10 bytes from one process to another would take a time of $L + 2o + 9G$ under the LogGP model (the first byte is accounted by L). After sending a message, a process has to wait $\max\{o, g\}$ before sending the next message (we assume single-CPU endpoints). We generally assume, wlog, that all processes start at $t = 0$ and an algorithm is completed when the last process finishes.

We assume that each process is running at all times (there is no operating system scheduling interference). For the purposes of space complexity analysis, we assume that an integer occupies constant space at the process where it is stored and a send operation requires $\Theta(1)$ memory at the source and target processes.

2.3 Complexity of a Census Function

Census functions are widely used in parallel programming. Informally, a census function is a function where each process receives the combination (reduction) of data items distributed among all processes. Such a function is available in MPI as MPI_Allreduce and a special version where only one process receives the final value, MPI_Reduce. Since processes can only "learn" about the values on other processes with explicit message passing (cf. [6]), there must be a chain of messages from each process to each other process.

Definition 3 (Census Function)**.** *Every process $p_0, \ldots, p_{P-1}$ has a value $v_0, \ldots, v_{P-1}$ and all processes need to receive the result $v_0 \oplus v_1 \oplus \ldots \oplus v_{P-1}$. The operation $\oplus$ is assumed to be associative and commutative for our purposes.*

Global synchronization (a barrier) is a particular example of a census function where the operation $\oplus$ itself is not important.

We assume the communication of small data for the following analysis. First we discuss an optimal broadcast algorithm where

initially a root process p_0 has a single data item and at the end of the algorithm, all processes $p_0 \dots p_{P-1}$ have this data. A census function can simulate a broadcast without additional costs if all processes but the root contribute the identity element with regards to $\oplus$ (e.g., choose addition as operation and zero as identity element). Thus, a lower bound to the broadcast problem is also a lower bound to the census problem.

The broadcast algorithm has been discussed under the postal model in [7–9]. The basic idea is that processes are arranged in a tree and each process eagerly sends to as many distinct processes as possible until a given time t is reached. We define $P(t)$ as the number of processes that can be reached in time t (including the root, which is reached at $t = 0$). In the remainder of this paper, we assume $o \geq g$ for simplicity (a similar method can be applied for the case $g > o$).

The root sends a new message to a child every o cycles and the first child itself starts sending at $2o + L$, the second child at $3o + L$ and the n^{th} child at $(n+1)o+L$. Thus, $P(t) = 1$ iff $t < 2o+L$ and the following recurrence counts the number of reached processes:

$$P(t) = \begin{cases} 1: & t < 2o + L \\ P(t-o) + P(t-L-2o): & \text{otherwise.} \end{cases} \quad (1)$$

We now present bounds for the number of reached nodes that we can use to bound the running time.

Lemma 1. *The total number of nodes that can be reached at time t is bounded by*

$$2^{\lfloor \frac{t}{L+2o} \rfloor} \leq P(t) \leq 2^{\lfloor \frac{t}{o} \rfloor}.$$

Proof. The evaluation of recurrence (1) can be done by creating a full binary tree $\mathcal{T}$ recursively (each inner node of $\mathcal{T}$ has two children). The construction of $\mathcal{T}$ starts with the root of weight t. If $t \geq 2o + L$, then two children with weights $t - o$ and $t - 2o - L$ are added. The algorithm is applied recursively to the two children until no children can be added. The number of leaves in $\mathcal{T}$ equals to $P(t)$. A complete binary tree of height h has 2^h leaves (a single root has $h = 0$). The number of children ($P(t)$) in $\mathcal{T}$ is now bounded between the number of children of the smallest complete binary subtree of $\mathcal{T}$ and $2^{h_\mathcal{T}}$. The height $h_\mathcal{T}$ of $\mathcal{T}$ is the longest path from the root to any leaf, which is $\lfloor \frac{t}{o} \rfloor$. The shortest path from the root to any leaf, and thus, the height of the smallest complete subtree, is $\left\lfloor \frac{t}{L+2o} \right\rfloor$. The bounds of $P(t)$ follow from this observation. □

The number of processes that can be reached at time t bounds the running time of the broadcast operation. After applying logarithms to the bounds in Lemma 1 we get

$$\left\lfloor \frac{t}{L+2o} \right\rfloor \leq \log_2(P) \leq \left\lfloor \frac{t}{o} \right\rfloor$$

which yields the following Corollary.

Corollary 1. *Let $T_{BC}(P)$ be the time to broadcast a small message to a set of P processes in the LogGP model, then*

$$\log_2(P) \cdot o \leq T_{BC}(P) \leq \log_2(P) \cdot (L + 2o),$$

and $T_{BC}(P) = \Theta(\log(P))$.

Bar-Noy et al. show similar asymptotic bounds in the Postal Model in [10].

2.4 Complexity of Personalized Census

A personalized census function is a special case of the census function where the resulting data is distributed over the processes. Thus, each process not only contributes a separate data item, but it also receives a distinct result. Such functions are used for parallel matrix multiplication and in various communication protocols. A personalized census function is available in MPI as MPI_Reduce_scatter.

Definition 4 (Personalized Census)**.** *In a personalized census function, each process i initially has data elements $v_{i,j}$ ($0 \leq i, j < P$). After the operation, each process, i, holds a single value $x = v_{0,i} \oplus v_{1,i} \oplus \dots \oplus v_{P-1,i}$.*

Lemma 2. *Let $T_{RS}(P)$ be the time to perform a personalized census over small data on a set of P processes, then*

$$T_{RS}(P) \geq G(P-1) + (L + 2o - G) \cdot \lceil \log_2 P \rceil,$$

and $T_{RS}(P) = \Theta(P)$

Proof. $T_{BC}(P)$ is a lower bound to this problem because each process needs to communicate one data item to all processes. However, since the items are personalized and each process must submit $P - 1$ values, this lower bound is not tight. Iannello shows a lower bound of $(P - 1) + (\min\{L', g'\} - 1) \cdot \lceil \log_2 P \rceil$ in the simplified LogGP model [5] in [11] (Lemma 7). The simplified LogGP model uses G to model the length of a cycle and thus $t(P) = \frac{T_{RS}}{G}$, $L' = \frac{L+2o}{G}$, and $g' = \frac{g}{G}$. If we substitute the parameters in Lemma 7 in [11] with the original LogGP parameters and assume $o \geq g$, we get the bound of Lemma 2. The upper bound can trivially be achieved with an algorithm where each process sends to and receives from all other $P - 1$ processes and then computes the necessary sum s_i, thus, $T_{RS}(P) = \mathcal{O}(P)$. □

The lower bound is tight for $L \approx g$. Iannello presents an optimized algorithm in [11].

2.5 Complexity of Personalized Exchange

Another important global function is the personalized complete exchange, in which each process has a vector of $P - 1$ elements to send to every other process. In MPI, this operation is available with MPI_Alltoall.

Definition 5 (Personalized Exchange)**.** *In a personalized exchange, each process i initially has data elements $v_{i,j}$ ($0 \leq j < P$). After the operation, each process i holds the data elements $v_{0,i}, v_{1,i}, \dots, v_{P-1,i}$.*

Lemma 3. *Let $T_{PE}(P)$ be the time to perform a personalized exchange of small data on a set of P processes, then*

$$T_{PE}(P) \geq T_{RS}(P) = G(P-1) + (L + 2o - G) \cdot \lceil \log_2 P \rceil,$$

and $T_{PE}(P) = \Theta(P)$.

Proof. Due to the independence of the send and receive channels, T_{RS} is also a good lower bound to this problem. All assumptions from [5, 11] remain valid and the lower bound follows. The upper bound can trivially be achieved with an algorithm where each process sends to and receives from all other $P - 1$ processes. □

An algorithm that reaches the lower bound asymptotically was proposed by Bruck et al. [12].

3. Protocols for DSDE

A specialization of the problem in MPI is the generalized all-to-all communication that is available with MPI_Alltoallv. However, this operation requires the knowledge of all source processes and data sizes at each process, which is often not available in computations. The sparse communication can be implemented by using MPI_Alltoall to communicate the sizes per host, followed by an MPI_Alltoallv to communicate the actual data.

In fact, most protocols can be divided into two phases: The first phase exchanges meta-information either about the communication neighborhood, the data sizes, or both. The second phase is usually the communication of the user data. In this work, we focus on the efficient implementation of the communication of the metadata which we expect to dominate large-scale runs. First, however, we develop bounds for the second phase when each process has global information about its neighborhoods.

Lemma 4. *Let T_{DT} be the time to perform the second phase of SDE in which each process has complete knowledge about all neighbors and data sizes. $T_{DT} \geq T_{BC}(k)$ and $T_{DT} = \mathcal{O}(\log P)$.*

Proof. If k is the maximum size of the neighborhood $\mathcal{N}_p$ of any process, then, $T_{BC}(k)$ is a lower bound to completion of the algorithm because some data needs to be sent to each of the k processes. The number of neighbors k in a sparse data exchange is limited by $\mathcal{O}(\log P)$ (Definition 1). Each process could simply send to and receive from all its neighbors and $T_{DT} = \mathcal{O}(\log P)$. □

Typical applications (cf. Section 5) often use frequent small message exchanges at large scale. If such small messages are sent to all peers in a trivial (linear) way, the time to finish the data transmission is dominated by $k \cdot o$.

In the following sections, we analyze the space and time complexity for three different protocols—$\mathcal{PEX}$, $\mathcal{PCX}$, $\mathcal{RSX}$—that use message passing as their underlying communication layer. Based on this analysis, we present $\mathcal{NBX}$, a new scalable protocol that uses novel semantic features of MPI to solve the DSDE problem in Section 3.4. We use the symbols S and T to represent space at each process and total runtime, respectively. Our analyses of spatial complexity are limited to the necessary storage for the protocol itself and do not include the size of the user data.

3.1 Algorithm $\mathcal{PEX}$—Personalized Exchange

In the alltoall exchange algorithm, $\mathcal{PEX}$, each process writes the data sizes to send to each peer in a vector with P elements and redistributes the vector with a personalized exchange as defined in Section 2.5 (e.g., MPI_Alltoall).

In a second step, each process reserves the required receive memory and posts (nonblocking) receive operations for each remote process that has a nonzero sendcount. Then all processes post their respective send operations. The operation is finished after all sends and receives are satisfied. Alternatively, the second step can be performed with a generalized exchange (for example MPI_Alltoallv).

Theorem 1. *The space needed by $\mathcal{PEX}$ on P processes is given by $S_{\mathcal{PEX}}(P) = \Theta(P)$. The time needed to complete $\mathcal{PEX}$ on P processes is $T_{\mathcal{PEX}}(P) = T_{PE}(P) + T_{DT}(P) = \Theta(P)$.*

Proof. $\Theta(P)$ bytes of buffer space are needed for the exchange operation. The needed time is the time that is required by the exchange operation (Lemma 3) plus the data transmission (Lemma 4). □

The personalized exchange of the metadata (first phase) will dominate the actual SDE in small neighborhoods in highly scalable systems because $T_{\mathcal{PEX}} = \Theta(P)$ while $T_{DT} = \mathcal{O}(\log P)$.

3.2 Algorithm $\mathcal{PCX}$—Personalized Census

In the personalized census algorithm, $\mathcal{PCX}$, each process writes '1' at position (i,j) of a global column-wise distributed $P \times P$ table and '0' otherwise. Then, the table is globally reduced row-wise and each process i receives the sum s_i of row i which is essentially the number of processes that send data to process i. This reduction could be performed with the MPI operation MPI_Reduce_scatter. In the next step, all processes start sending all data without blocking. Then, each process enters a loop with s_i iterations which probes for wildcard receives, allocates memory and receives the data as shown in Algorithm 1.

Algorithm 1: $\mathcal{PCX}$—Personalized Consensus.

```
Input: List I of destinations and data
Output: List O of received data and sources
allocate local table with P entries, initialize all entries to '0';
foreach i ∈ I do
    set row target(i) in local table to '1';
s_i = global sum of my table row;
foreach i ∈ I do
    start nonblocking sends to dest(i);
for round = 1..s_i do
    msg = blocking probe for incoming message; allocate
    buffer, receive message msg, add buffer to O;
```

Theorem 2. *The space needed to perform $\mathcal{PCX}$ on P processes is given by $S_{\mathcal{PCX}}(P) = \Theta(P)$. The time needed to complete $\mathcal{PCX}$ on P processes is $T_{\mathcal{PCX}}(P) = T_{RS}(P) + T_{DT}(P) = \Theta(P)$.*

Proof. Similarly to $\mathcal{PEX}$, a table of size $\Theta(P)$ is needed at each process for the exchange of the metadata. The needed time $T_{\mathcal{PCX}}(P)$ follows from Lemmas 2 and 4. □

The main difference from the personalized exchange protocol is that the metadata is reduced in the collective operation and each process only receives the number of neighbors (messages) to wait for. However, $T_{\mathcal{PCX}}(P) = \Theta(P)$ is asymptotically not smaller than $T_{\mathcal{PEX}}(P)$.

3.3 Algorithm $\mathcal{RSX}$—Remote Summation

If the communication environment supports remote data accumulation (MPI offers MPI_Accumulate), each process could increase a counter s_i (originally zero) on each of its target processes i. After a global synchronization step (e.g., MPI_Win_fence), each process, j, posts s_j (nonblocking) wildcard receive operations and sends all its messages. The algorithm is very similar to algorithm $\mathcal{PCX}$ (cf. Algorithm 1). The difference is that s_i is not computed in a global operation but with remote memory accesses. This results in a sparse communication pattern.

Theorem 3. *The space needed to perform $\mathcal{RSX}$ on P processes is given by $S_{\mathcal{RSX}}(P) = \Theta(1)$. The time needed to complete $\mathcal{RSX}$ on P processes is $T_{\mathcal{RSX}}(P) = L + k \cdot o + T_{BC}(P) + T_{DT}(P) = \Theta(\log P)$.*

Proof. Only a single atomic counter is needed to perform protocol $\mathcal{RSX}$. The time needed for protocol $\mathcal{RSX}$ involves k accumulation messages and the global synchronization. The global synchronization is bounded by $T_{BC}(P)$ (cf. Corollary 1). □

$\mathcal{RSX}$ is, with $T_{\mathcal{RSX}}(P) = \Theta(\log P)$, asymptotically fastest so far.

3.4 Algorithm $\mathcal{NBX}$—Nonblocking Consensus

Now, we propose a new scalable algorithm, $\mathcal{NBX}$, for the SDE problem. This method utilizes nonblocking collective operations, a novel technique that is part of the upcoming MPI-3 standard. Nonblocking collective operations are structurally similar to the communication operations that we discussed above. This means, that

all discussed lower bounds consistently apply, however, the difference is that nonblocking operations can be started and completed independently (like nonblocking point-to-point operations in MPI).

Nonblocking operations enable additional functional parallelism where the main CPU can continue to compute while the collective communication runs. This overlap potential of computation and communication has been analyzed by Hoefler et al. and Hoefler and Lumsdaine [13, 14]. However, nonblocking collective operations also provide semantic advantages that allow separate start and completion of a globally synchronizing operation.

We utilize a nonblocking barrier to set a *distributed marker* such that each process starts the barrier after it finishes its local part and serves the requests of other processes until it detects global termination (i.e., the barrier is reached by all nodes and completes). We also use send operations that only complete after the message has been received (synchronous mode send, cf. MPI_Ssend). Algorithm 2 shows pseudocode for such a two-phase implementation.

Algorithm 2: $\mathcal{NBX}$—Nonblocking Consensus.

```
   Input: List I of destinations and data
   Output: List O of received data and sources
1  done=false;
2  barr_act=false;
3  foreach i ∈ I do
4      start nonblocking synchronous send to process dest(i);
5  while not done do
6      msg = nonblocking probe for incoming message;
7      if msg found then
8          allocate buffer, receive message, add buffer to O;
9      if barr_act then
10         comp = test barrier for completion;
11         if comp then done=true;
12     else
13         if all sends are finished then
14             start nonblocking barrier;
15             barr_act=true;
```

Theorem 4. *The space needed to perform $\mathcal{NBX}$ on P processes is given by $S_{\mathcal{NBX}}(P) = \Theta(1)$. The time needed to perform $\mathcal{NBX}$ on P processes is $T_{\mathcal{NBX}}(P) = T_{BC}(P) + T_{DT}(P) = \Theta(\log P)$.*

Proof. Except for variables of constant size, algorithm $\mathcal{NBX}$ does not require any additional space to administer the communication. The time needed to perform the data exchange is simply the time needed to perform the data movement (Lemma 4) and an additional barrier (census, Corollary 1) operation. □

This algorithm scales well with $T_{\mathcal{NBX}}(P) = \Theta(\log P)$ and thus solves the SDE problem optimally. In other words, it meets the lower bound $T_{BC}(P)$ imposed by the necessary detection of termination.

Consistency This approach requires a barrier that completes only after all messages have been received. This is guaranteed if any of the following conditions is met:

1. The network is synchronous and the barrier message is only injected after all data messages have been injected.
2. Each network channel is strictly FIFO and the barrier algorithm uses all $P - 1$ outgoing channels. (Such an $\Omega(P)$ barrier algorithm obviously could not meet the lower bound, T_{BC}.)
3. The network can be drained. (This would imply a special network operation which effectively replaces a barrier).
4. The sender can check if a message reception has at least been started at the receiver side.

Options 1–3 are relevant to systems with special hardware support which is not offered by our message passing model or the MPI standard. Option 4 can be implemented with a simple protocol where the receiver acknowledges the (start of) reception of each message to the sender. The MPI-1 standard offers such a functionality with a *synchronous mode send* (e.g., MPI_Ssend), which offers multiple avenues for optimization.

Theorem 5. *Algorithm $\mathcal{NBX}$ implements SDE correctly if at least one of the above consistency requirements is fulfilled.*

Proof. Options 1, 2, and 3 guarantee that the barrier completes only after all messages are sent and have been received because messages in one channel cannot pass each other in our model. All processes check for incoming messages and receive all arriving messages before the barrier completes. The strict ordering guarantees that all messages are received at all processes. Option 4 replaces the global ordering requirement with local ordering. A process p starts the barrier operation only after all the messages it sent were received. The process continues receiving messages until the barrier completes globally hence the algorithm does not deadlock and when the barrier completes globally, all messages sent have been received.

□

Overhead of Synchronous Mode Sends A trivial implementation of synchronous mode send would effectively double the number of messages and add another $L + k \cdot o$ to the running time of T_{NBX} in the worst case. Unfortunately, many MPI libraries implement this simple protocol because of the rare usage of MPI_Ssend in applications. However, it is possible for this reception notification to be embedded into the usual reliability protocol for data transmission.

Nevertheless, we assume the worst-case for our model, and will present benchmark results of actual overheads in a later section. Thus, we model the worst-case runtime of $\mathcal{NBX}$ with explicit point-to-point synchronization as

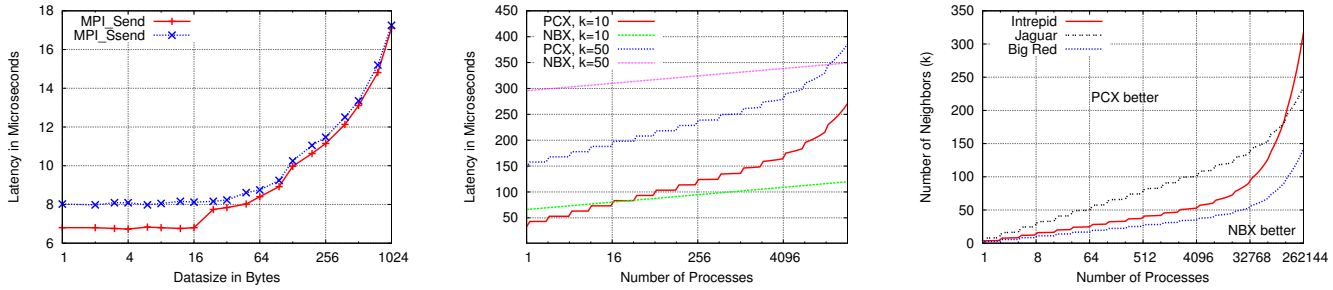
$$T'_{\mathcal{NBX}}(P) = T_{\mathcal{NBX}}(P) + L + k \cdot o. \quad (2)$$

4. Implementation and Validation

We used three different systems for our evaluation, the Jaguar system at Oak Ridge National Laboratory, the Intrepid BlueGene/P system at the Argonne National Laboratory, and the Big Red system at Indiana University to cover a wide range of different CPU and network architectures. All systems are equipped with four cores per node and we ran four MPI processes per node in all our experiments.

Jaguar, a Cray XT-4/XT-5 system with 150,152 2.3 GHz Opteron compute cores, ranks among the worlds fastest computers. Connected with the SeaStar 2 network in a three-dimensional torus topology, the network bandwidth is approximately twice as high as the injection bandwidth per node, reducing the effects of network congestion, especially for small messages. Jaguar runs Compute Node Linux 2.1 with the Cray Message Passing Toolkit 3. We used the XT-4 partition for our experiments.

The Intrepid system is a BlueGene/P and comprises 163,840 850 MHz PPC450 cores. We used virtual-node mode in our experiments. The network consists of multiple special-purpose networks. The collective network supports several collective operations including simple reduction operations. The barrier network is essentially a global OR which enables fine-grained synchronization of



(a) Synchronous send overhead on Big Red.

(b) Predicted ideal running times for $\mathcal{PCX}$ and $\mathcal{NBX}$ on Intrepid.

(c) Algorithm switching points for all systems as a function of P and k.

Figure 1. Visualization of different modeling parameters.

thousands of nodes in less than one microsecond [15]. The general purpose network is a high-bandwidth three-dimensional torus network. Intrepid runs Compute Node Kernel and a customized MPICH2-based MPI library.

The Big Red system consists of 3,072 PowerPC 970 cores. The 768 nodes are connected with a fat-tree Myrinet 2000 network. Big Red runs Linux 2.6 and Open MPI 1.3.2 with network-specific configurations.

Nonblocking Collective Operations The performance of the collective algorithms is crucial for the overall performance of the protocols. We used the standard blocking MPI collective operations supplied with the system's MPI library and LibNBC [16] for nonblocking collective operations. We assume that MPI's collective operations are well optimized for the underlying network. The generic LibNBC implementation uses the dissemination barrier. Dissemination-based algorithms [17] are asymptotically optimal for census functions of small data. However, depending on the specific LogGP parameters and number of processes, other algorithms might be faster with the same asymptotic bound.

BlueGene/P has a special hardware-supported barrier network. Therefore, on the Intrepid system, we used the deep-computing messaging framework [18] function `DCMF_GlobalBarrier` to register a callback for the nonblocking barrier that uses the hardware support.

4.1 Bounds on Real Systems

We assume in the following that all communication algorithms are implemented optimally in the LogGP model. We measured the LogGP parameters of all test systems as described in [19, 20]. Table 1 shows the parameters for all systems.

System	L (μs)	o (μs)	g (μs)	G (μs)
Intrepid	4.29	2.87	2.04	0.00267
Jaguar	9.90	1.75	3.40	0.00058
Big Red	7.08	11.78	4.19	0.00414

Table 1. LogGP parameters for all machines.

We also analyzed the implementation quality of synchronous sends. We used a standard ping-pong benchmark to compare MPI_Send with MPI_Ssend. We observed that there is generally a penalty for the latency of small messages while the transmission of large messages is not affected. This is because eager and rendezvous protocols in MPI implementations. Small eager messages do not include a handshake hence, the handshake has to be added which might require another roundtrip. The rendezvous protocol includes a handshake and effectively implements MPI_Ssend. The transmission curves for MPI_Ssend in the Big Red system are shown in Figure 1(a). For the other systems, we compare the 1-byte latency L_s of the synchronous-mode send operation MPI_Ssend with the normal MPI_Send latency (see Table 1). In addition, Table 2 reports the corresponding data size s_s where both send modes converged (i.e., latency difference is smaller than 1%). The ratios

System	L_s (μs)	L_s/L	s_s (kiB)
Intrepid	5.04	1.17	12
Jaguar	25.40	2.57	132
Big Red	8.02	1.13	1.5

Table 2. Synchronous send overheads for all machines.

for L_s/L vary from 1.13 to 2.57 in our test systems. While the measured ratios are only valid for ping-pong measurements with exactly one message in the network, we found that they represent real application behavior well. In the following theoretical analysis, we explicitly account for the handshake overhead of synchronous mode send (cf. Equation (2)).

4.2 Influence of the Number of Neighbors

The performance of the different algorithms depends on P, k, the LogGP parameters, and the overhead of synchronous mode send. We compare algorithms $\mathcal{PCX}$ and $\mathcal{NBX}$ as examples for the discussion of the influence of k. $\mathcal{NBX}$ is more scalable than the other algorithms; however, the penalty of the required point-to-point synchronization accumulates linearly with k. Thus, we expect that this algorithm performs best for sparse exchanges in large-scale systems. Figure 1(b) compares the LogGP-predicted idealized running time for algorithms $\mathcal{PCX}$ and $\mathcal{NBX}$ for 10 and 50 neighbors on Intrepid. It is clear that $\mathcal{NBX}$ is asymptotically faster than $\mathcal{PCX}$, however, the crossover point depends on k. Thus, we analyze the behavior of this switching point, that is, the scaling of k (we omit P, where it is obvious).

$$\begin{aligned} T'_{\mathcal{NBX}} &\leq T_{\mathcal{PCX}} \\ T_{BC} + T_{DT} + L + k \cdot o &\leq T_{RS} + T_{DT} \\ k &\leq \frac{T_{RS} - T_{BC} - L}{o} \end{aligned}$$

We see that the new algorithm has significant advantages over the other algorithms even if k scales linearly with the total number of processes. However, it strongly depends on the ratio of the LogGP parameters. An algorithm that selects the fastest among $\mathcal{NBX}$ and $\mathcal{PCX}$ must consider all LogGP parameters and k. Figure 1(c) shows the threshold for k, where the two algorithms would perform equally fast.

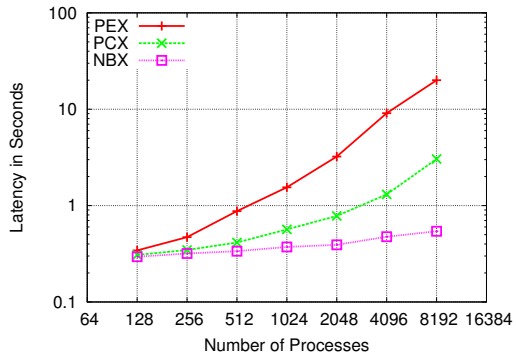


Figure 2. Algorithm comparison on Intrepid for $k = 6$.

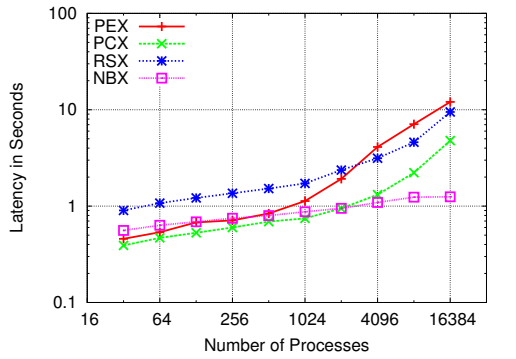


Figure 3. Algorithm comparison on Jaguar for $k = 6$.

4.3 Microbenchmark Results

We first propose a microbenchmark that allows us to measure the communication performance in isolation. The benchmark uses the Mersenne Twister [21] random number generator to select k different target processes and random message sizes between 1 and 1024 bytes at each process. For our experiment, we execute the benchmark for 1000 rounds and report the accumulated time. This simulates a bulk-synchronous application without computation.

We executed the microbenchmark with $k = 6$ on all systems with varying results. Figures 2 and 3 show the results for different process counts in Intrepid and Jaguar, respectively. Protocol $\mathcal{RSX}$ could not be evaluated on Intrepid because the one-sided implementation caused deadlocks. The three other protocols scale as predicted with $\mathcal{NBX}$ as the best protocol for all investigated process counts. This is mostly due to the optimized barrier and the relatively low overhead of synchronous-mode sends.

Jaguar shows the interesting feature that $\mathcal{PEX}$ and $\mathcal{PCX}$ perform faster at smaller process counts but are eventually surpassed by $\mathcal{NBX}$ at 2,048 processes. The low performance of $\mathcal{RSX}$ indicates optimization potential in the one-sided communication layer. LibNBC is not optimized for the underlying torus topology. Thus, we expect that $\mathcal{NBX}$ would perform faster with an optimized barrier implementation specific to the torus network.

Big Red showed a behavior similar to that of Jaguar. $\mathcal{PEX}$ and $\mathcal{PCX}$ perform faster for smaller process counts but are surpassed at about $P = 400$ by $\mathcal{NBX}$.

We see that the performance on the different systems and scales varies considerable. However, the benchmark data supports our analysis that $\mathcal{PEX}$ or $\mathcal{PCX}$ perform good at small scales but are outperformed by $\mathcal{NBX}$ at larger scales. The specific crossing points are system dependent. Thus, an adaptive library implementation of the protocols might be beneficial.

5. Applications and Parallel Algorithms

In this section, we discuss different parallel algorithms and methods that use DSDE and can thus utilize the discussed protocols. We discuss three different application domains: graph computations, n-body methods, and sparse matrix computations. We describe optimized algorithms to solve the respective problems and their relation to DSDE.

5.1 Parallel Graph Algorithms

Parallel graph algorithms are often used in large-scale network analysis and data mining and impose several hard problems [22] such as the inherent irregularity and lack of structure due to their data-driven nature and poor locality. Complex graph algorithms are often built upon level-synchronous breadth-first search (BFS) traversals [23, 24]. Thus, we choose BFS as an example to discuss the properties of the underlying communication. However, our results also apply to many other parallel graph algorithms such as shortest-path problems, connected components and betweenness centrality.

Algorithm 3 shows the pseudocode for parallel level-ordered BFS. The call $globalsum$ is a synchronizing operation that computes a global census function, equivalent to MPI_Allreduce with MPI_LOR as operation. This operation is used in Algorithm 3 as a global OR in order to check for termination of the algorithm. The call $DSDE(R, Q, d)$ starts a sparse exchange of all vertices in the set R, and all received vertices are inserted into Q with their received $dist$. If an edge is remote, then it carries the address of the destination process as a property.

Algorithm 3: Parallel BFS Algorithm.

```
   Input: Distributed Graph (V, E), root vertex r, dist[v] = ∞
   Output: Distance dist[v] from r to all vertices v ∈ V
1  Q ← empty queue; R ← empty list; d = 0;
2  if r is local then push r → Q;
3  while true do
4      if Q not empty then
5          pop v ← Q;
6          if dist[v] > d then
7              globalsum(1);
8              d = dist[v];
9              DSDE(R, Q, d);
10         foreach (v, w) ∈ E do
11             if w is local then
12                 enqueue w → Q;
13                 dist[w] = d + 1;
14             else enqueue w → R;
15     else
16         if R ≠ ∅ then globalsum(1);
17         else if globalsum(0) = 0 then break;
18         DSDE(R, Q, d);
```

The communication pattern in the BFS algorithm depends on the initial graph structure and the evolution of the computation. Many real-world problems can be represented by scale-free graphs with a very small number of high-degree vertices and a sparse network of low-degree vertices that often form clusters. An efficient partitioning algorithm is necessary to achieve a good work distribution. We generated well-partitioned (balanced) test graphs, as de-

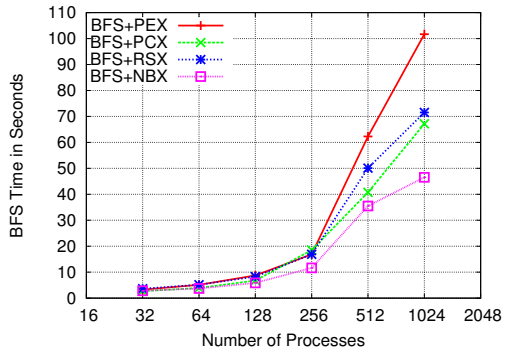


Figure 4. BFS times on Big Red with $k = 6$.

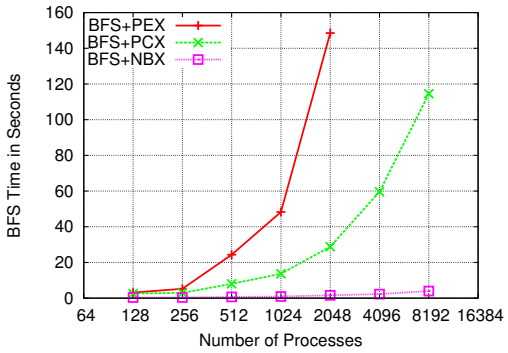


Figure 5. BFS times on Intrepid with $k = 6$.

scribed in the next section, for our experiments to overcome this problem.

5.1.1 The Balanced Graph Scenario

In this study, we utilize a random Erdős Rényi graph [25] local to each process. Each of the node-local vertex pairs (u, v) $u, v \in V_i$ $0 \leq i < P - 1$, is with probability p in the edge set E_i. Additionally, each process is a source of k remote edges that point to random vertices in remote processes. Edges are only known to the source node. A Hamiltonian cycle is added to the graph to ensure that it is connected. Our model resembles the main properties of a well-distributed Watts Strogatz model [26].

To draw a realistic scenario, we used $p = 0.03$ with 15,000 vertices per process on Big Red. The resulting $6.75 \cdot 10^6$ edges nearly fill the available 1 GiB per core. We added $k = 6$, 26 or 79 random remote edges per process. All protocols were tested with the same random graph after one warm-up round (where MPI initialization occurs) for each setting. Multiple computations have been performed, showing an acceptably low variation ($< 5\%$). We report the average running times, checking the outcome for correctness.

Figure 4 shows the timing of a parallel level-synchronous BFS search with 15,000 vertices per process and $k = 6$ random remote edges per process. The gap between BFS+$\mathcal{NBX}$ and the other algorithms is slightly bigger than in the microbenchmark run. We conjecture that this is due to lower cache pollution because $\mathcal{NBX}$ does not need to build lists of size $\Omega(P)$ before communicating. It also achieves efficient communication-communication overlap of the barrier operation and the data transmission, which is important in the irregular case. Additionally, $\mathcal{NBX}$ performs much faster when the neighborhood sizes are unequal (we used static sizes in the microbenchmark) as in BFS where the exchange is started when the local queue runs empty, which depends on the input graph. We repeated the benchmark with $k = 26$ and $k = 79$. The shape of the curves is similar in all cases and beginning with 128 processes, BFS with $\mathcal{NBX}$ was the fastest on Big Red.

We ran a similar experiment on the Intrepid system. However, we could only use 10,000 vertices per process ($3 \cdot 10^6$ edges at $p = 0.03$) due to the larger scale and the $\Omega(P)$ memory consumption of algorithms $\mathcal{PEX}$ and $\mathcal{PCX}$. On this system, we have a "constant-time" nonblocking barrier operation available that synchronizes all processes in less than one microsecond independent of the scale. Figure 5 shows the scaling of the level-synchronous parallel BFS on Intrepid. We observe that the time with all three protocols is nearly equal with 128 processes. However, as we scale up, $\mathcal{PEX}$ and $\mathcal{PCX}$ quickly pass $\mathcal{NBX}$ which stays nearly constant due to the fast barrier. $\mathcal{PEX}$ did not run above 2,048 processes because of memory constraints. Benchmarks with $k = 26$ and $k = 79$ show very similar curves. The BFS algorithm with $\mathcal{NBX}$ on 2,048 processes performs 15.8 and 14.1 times faster than with $\mathcal{PCX}$, respectively. Jaguar showed similar results that are omitted due to space constraints.

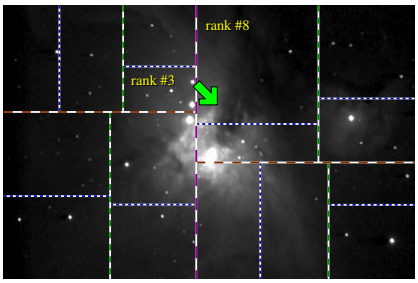
5.2 Parallel Barnes–Hut N-Body Methods

N-body simulations are typically used in cosmology to study the physical behavior of massive numbers of particles, as in galaxies or dark matter. The simulations track the particle movements under the influence of gravity. Because the number of involved particles is usually large (often millions of bodies), the computation of all N^2 direct particle–particle interactions becomes infeasible. Tree methods such as a *Barnes–Hut* simulation overcome this issue by dividing the volume (e.g., a universe) into cubic cells by forming an octree (for 3 dimensions, or quadtree for 2 dimensions). Only particles from nearby cells need to be handled individually, while particles in distant cells can be treated as a single large body that is located at its center of mass [27]. This method provides a huge improvement because only $\mathcal{O}(N \cdot \log N)$ interactions need to be considered.

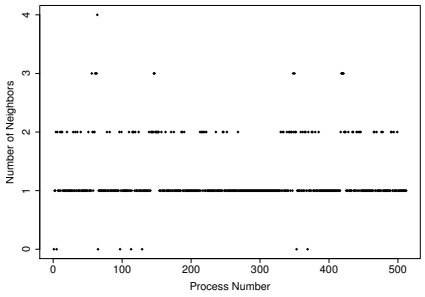
Contrary, to the slow direct method where all force calculations take similar amounts of time, a Barnes–Hut simulation has the drawback that the time to calculate the resulting force on each particle depends on its location (more precisely, the number of "near" particles) and is therefore highly irregular. For this reason, the primary challenge for a parallel simulation is to find a good mapping of particles to the processes [28]. A good way to achieve such a mapping is to use geometric partitioning [29]. One such method is *orthogonal recursive bisection* (ORB), which divides the space orthogonal to the longest dimension into two pieces that generate the same load. This procedure is repeated until the desired number of partitions (e.g., the number of processes) is obtained. The example shown in Figure 6(a)[1] shows such a two-dimensional subdivision for 16 processes. Lines show the splittings, and their colors as well as dot lengths indicate the depth of the recursive bisection.

The difference between individual time steps of a simulation are usually so small that it is more efficient to apply ORB only when the actual process skew exceeds a certain threshold (often 5%). In one ORB phase, it can happen that some particles leave the cell of one processor and need to be migrated to the processor responsible for the new location. The (green) "force" arrow on the example image shows such a situation where a particle leaves the area of influence of one process (rank #3) to become the responsibility of another process (rank #8) in subsequent simulation steps. Because force calculation and updating of the resulting positions is done in

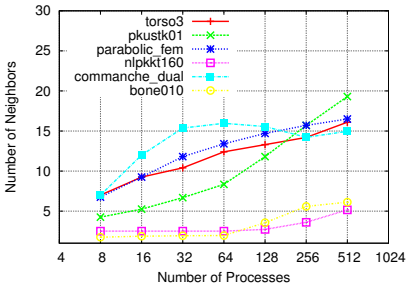
[1] Credit: NASA and The Hubble Heritage Team.



(a) Orthogonal recursive bisection (ORB) for 16 processes. (The nebula in the constellation Orion.)



(b) Maximum k_p per process in an N-body simulation with $2 \cdot 10^6$ particles.



(c) k for sparse matrix–vector multiplication of different real-world input problems.

Figure 6. Number of Neighbors k for Different Applications.

a distributed manner, only the source process knows when particles leave its cell. When this happens, it needs to move all the information of these particles to the proper processes. This is done after every simulation step (more precisely after the new positions have been calculated) in a dynamic sparse exchange operation.

The most interesting parameter of this application with regards to DSDE protocols is the maximum number of neighbors k for different numbers of processes. This parameter is specific to the application and determines the performance of DSDE. The ORB splitting based on the imbalance in previous force computations creates rapidly changing process neighborhoods. Experiments with two galaxies consisting of 4 million particles showed that most processes only need to migrate a small number of particles, whereas there are some processes that send up to a thousand particles. We were able to reproduce this behavior in a strong scaling experiment on 1,024 processes and observed rapidly changing neighborhoods between simulation rounds. Nevertheless, the maximum number of neighbors k was always below 5 in our tests (Note that the communication neighborhood of a cell is a subset of the geometrically adjacent cells). Figure 6(b) shows the maximum number of neighbors per process during a full run with 512 processes.

5.3 Parallel Sparse Matrix Computations

Sparse matrix computations are used in many scientific applications, such as computational fluid dynamics, finite element method computations, and molecular dynamics. The key operations of many such computations are (sparse) matrix–vector multiplications and dot products (e.g., in Krylov-subspace methods). The communication pattern and volume depends on the distributions of the matrix and the vector. The resulting communication is often static and can be optimized statically. However, changes to the modeled system (e.g., deformations) also change the sparse structure and the communication neighborhoods and volumes.

We already showed that the performance of the algorithms depends on the size of the neighborhood. Instead of presenting more benchmark results, we chose to analyze realistic problems regarding the size of the communication neighborhood, which mostly depends on the structure of the sparse matrix. Thus we analyzed six real-world matrices from the University of Florida sparse matrix collection [30]. We used a load-balanced row-wise distribution of the matrix elements and a block distribution of the dense vector. More elaborate partitioning methods can be used to further optimize balance and reduce communications. However, fully distributed partitioning methods to optimize such problems at large scale are still a field of active research.

We used six of the largest datasets in the matrix collection, performed a sparse matrix–vector multiplication and recorded the average and maximum number of neighbors for each process. Table 3 shows the number of rows, nonzero elements and k for a 512-process run. Details about the input matrices can be obtained from [30].

Name	# rows	# elements	k
torso3	259,156	4,429,042	27
pkustk01	22,044	979,380	76
parabolic_fem	525,825	3,674,625	79
nlpkkt160	8,345,600	225,422,112	6
commanche_dual	7,920	31,680	28
bone010	986,703	47,851,783	8

Table 3. Key properties of the used matrices and k for $P = 512$.

We observe that both the maximum and the average size of the neighborhood grow very slowly with the number of processes. We plot the average values for different process counts in Figure 6(c) because the maximum is often strained by single high-degree vertices.

Those results show that the number of neighbors grows very slowly with the number of processes in large-scale matrix problems. Thus, we conclude that the discussed optimized algorithms for DSDE could be used to optimize parallel sparse matrix computations where the system changes dynamically during the computation.

6. Conclusions and Outlook

In this work we define the static and dynamic sparse data-exchange ([S,D]SDE) problems for parallel applications. We show three different practical algorithms that can be easily implemented in today's parallel computing models such as the Message Passing Interface (MPI). In addition, we propose a new algorithm that leverages the new semantic features of nonblocking collective operations in the upcoming version of the MPI standard.

By just splitting the collective operation into a start and an end, we have the possibility to set a collective marker indicating that the local active part of the process is done while the process can still perform other tasks (e.g., receiving messages) until a globally consistent state is reached. This process-local marker is part of the global state and the collective operation will finish after all processes set their markers. We showed one concrete example for the use of those semantics that efficiently solves the DSDE problem.

We believe that the semantic advantages of nonblocking collective communications offer a huge potential, especially for irregular applications. It seems also interesting that an advanced one-sided algorithm can be replaced with a faster algorithm based

on nonblocking collective and point-to-point operations. We expect that this technique is applicable to other application domains and other collective operations (such as the census operations MPI_Allreduce, MPI_Reduce_scatter, or MPI_Alltoall).

Our evaluations of the different algorithms on three different supercomputer systems showed that there are several trade-offs to be made to choose the best algorithm, especially at smaller scales. The proposed algorithm, however, is, with a bound of $\Theta(\log P)$ asymptotically faster than all established point-to-point algorithms ($\Theta(P)$) and outperforms each for sparse exchanges at large scale. The new algorithm performs in microbenchmarks about 5.6 times faster than the best known algorithm on 8,192 BlueGene/P processors and about 3.8 times faster on 16,384 Jaguar CPUs.

We also discuss several applications for the problem and show application benchmark results for graph computations. The proposed algorithm is able to use BlueGene/P's hardware-optimized barrier which reduces the runtime significantly. We see a speedup of a parallel breadth-first search of up to 28.9 times on 8,192 CPUs. We discuss the size of the process neighborhoods for real-world n-body problems and sparse matrix–vector multiplications. We showed that the maximum and the average number of neighbors scales very slowly with the number of processes. Those results strongly suggest that the proposed protocol would perform well for large-scale computations of those inputs.

Acknowledgments

We thank Marc Snir (UIUC), Brad Chamberlain (Cray Inc.), and the anonymous reviewers for numerous comments that helped us to improve this article. We thank Jesper Larsson Träff (NEC) and Rolf Rabenseifner (HLRS) for discussions about sparse communications and protocols, Peter Gottschling (TUD) for comments about sparse matrix-vector multiplications, and Jeremiah Willcock (IU) and Nick Edmonds (IU) for discussions about parallel graph algorithms. Rich Graham (ORNL), Terry Jones (ORNL), and Rajeev Thakur (ANL) helped us to get access to the necessary supercomputer resources and one of the authors was supported by the Department of Energy project FASTOS II (LAB 07-23).

References

[1] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[2] T. Hoefler and J. L. Traeff. Sparse collective operations for MPI. In *Proceedings of the 23rd IEEE International Parallel & Distributed Processing Symposium (IPDPS), HIPS Workshop*, May 2009.

[3] R. Das, Y. Hwang, M. Uysal, J. Saltz, and A. Sussman. Applying the chaos/parti library to irregular problems in computational chemistry and computational aerodynamics. In *Mississippi State University, Starkville, MS*, pages 45–56. IEEE Computer Society Press, 1993.

[4] J. L. Träff. Implementing the mpi process topology mechanism. In *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–14, Los Alamitos, CA, USA, 2002. IEEE Computer Society Press.

[5] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating long messages into the LogP model. *J. of Par. and Distr. Comp.*, 44(1):71–79, 1995.

[6] K. M. Chandy and J. Misra. How processes learn. In *Proceedings of the fourth annual ACM symposium on Principles of Distributed Computing*, pages 204–214. ACM, 1985.

[7] I. Cidon, I. Gopal, and S. Kutten. Optimal computation of global sensitive functions in fast networks. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, pages 185–191, 1991.

[8] R. M. Karp, A. Sahay, E. E. Santos, and K. E. Schauser. Optimal broadcast and summation in the LogP model. In *Proc. of Symposium on Parallel Algorithms and Architectures*, pages 142–153, 1993.

[9] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. *Math. Syst. Theory*, 27(5):431–452, 1994.

[10] A. Bar-Noy, S. Kipnis, and B. Schieber. Optimal computation of census functions in the postal model. *Discrete Appl. Math.*, 58(3):213–222, 1995.

[11] G. Iannello. Efficient Algorithms for the Reduce-Scatter Operation in LogGP. *IEEE Trans. Par. Distr. Syst.*, 8(9):970–982, 1997.

[12] J. Bruck, C.-T. Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *Trans. on Par. and Distrib. Syst.*, 8(11):1143–1156, 1997.

[13] T. Hoefler, P. Gottschling, and A. Lumsdaine. Leveraging nonblocking collective communication in high-performance applications. In *Proc. of the 20th Annual Symp. on Parallelism in Algorithms and Architectures*, pages 113–115. ACM, June 2008.

[14] T. Hoefler and A. Lumsdaine. Overlapping Communication and Computation with High Level Communication Routines. In *Proc. of the 8th IEEE Symp. on Cluster Computing and the Grid*, pages 572–577, May 2008.

[15] IBM Blue Gene Team. Overview of the Blue Gene/P project. *IBM Journal of Research and Development*, 52(1/2), January 2008.

[16] T. Hoefler, A. Lumsdaine, and W. Rehm. Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In *Proc. of the 2007 Intl. Conf. on High Perf. Comp., Networking, Storage and Analysis, SC07*. IEEE Computer Society/ACM, Nov. 2007.

[17] D. Hengsen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[18] S. Kumar et al. The deep computing messaging Framework: Generalized scalable message passing on the BlueGene/P supercomputer. In *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*, pages 94–103, New York, NY, 2008.

[19] C. Bell, D. Bonachea, Y. Cote, J. Duell, P. Hargrove, P. Husbands, C. Iancu, M. Welcome, and K. Yelick. An evaluation of current high-performance networks. In *Proc. of the 17th Int. Symp. on Par. and Distr. Proc.*, page 28.1, 2003.

[20] T. Hoefler, A. Lichei, and W. Rehm. low-overhead LogGP parameter assessment for modern interconnection networks. In *Proc. of the 21st IEEE Intl. Par. & Distrib. Proc. Symp.*, March 2007.

[21] M. Matsumoto and T. Nishimura. Mersenne twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. In *ACM Trans. on Modeling and Computer Simulations*, pages 3–30, 1998.

[22] A. Lumsdaine, D. Gregor, B. Hendrickson, and J. W. Berry. Challenges in parallel graph processing. *Parallel Processing Letters*, 17(1):5–20, 2007.

[23] D. A. Bader and K. Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *Proc. of the 2006 Int. Conf. on Parallel Processing*, pages 539–550, 2006.

[24] D. Gregor and A. Lumsdaine. Lifting sequential graph algorithms for distributed-memory parallel computation. *SIGPLAN Not.*, 40(10):423–437, 2005.

[25] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[26] D. J. Watts and S. H. Strogatz. Collective dynamics of "small-world" networks. *Nature*, 393(6684):440–442, June 1998.

[27] J. Barnes and P. Hut. A hierarchical O($N \log N$) force-calculation algorithm. *Nature*, 324(6096):446–449, December 1986.

[28] U. Becciani, R. Ansalonib, V. Antonuccio-Delogua, G. Erbaccic, M. Gamberaa, and A. Pagliaro. A parallel tree code for large N-body simulation: Dynamic load balance and data distribution on a CRAY T3D system. *Comp. Phys. Comm.*, 106:105–113, October 1997.

[29] J. K. Salmon. Parallel $NlogN$ N-body algorithms and applications to astrophysics. *Compcon Spring*, (91):73–78, March 1991.

[30] T. A. Davis. University of Florida sparse matrix collection. *Submitted to ACM Transactions on Mathematical Software*, 1994.