IExchange: Asynchronous Communication and Termination Detection for Iterative Algorithms

Dmitriy Morozov*
Lawrence Berkeley National Laboratory, Berkeley, CA
Tom Peterka†
Argonne National Laboratory, Lemont, IL
Hanqi Guo‡
Argonne National Laboratory, Lemont, IL
Mukund Raj§
Jiayi Xu¶
The Ohio State University, Columbus, OH
Han-Wei Shen‖The Ohio State University, Columbus, OH

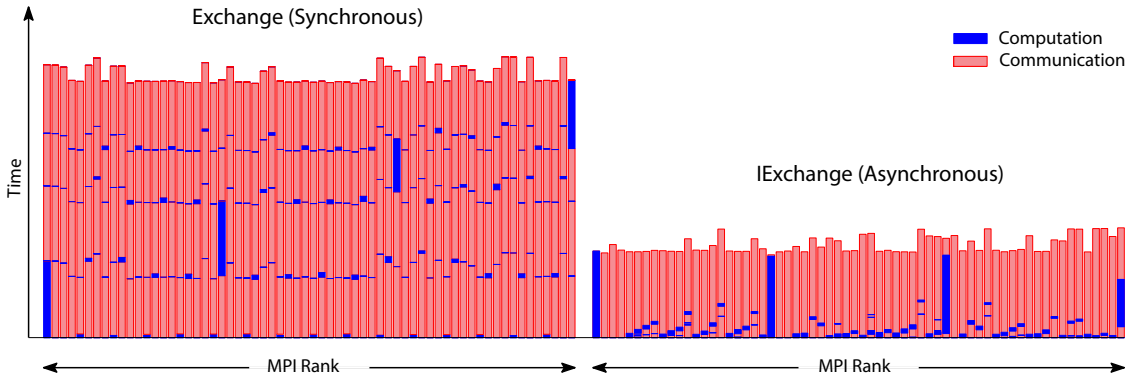

Figure 1: Comparison of synchronous `exchange` and asynchronous `iexchange` Gantt charts of particle tracing in a synthetic flow field. The left (synchronous) chart clearly shows 4 rounds in horizontal bands because each round can begin only after the slowest block in the previous round is completed. Our asynchronous method (right chart) is several times faster because communication and computation are interleaved, and blocks execute as soon as they are able. This synthetic dataset is designed such that the overall time for asynchronous iexchange is only slightly longer than the time for slowest block.

Abstract

Iterative parallel algorithms can be implemented by synchronizing after each round. This bulk-synchronous parallel (BSP) pattern is inefficient when strict synchronization is not required: global synchronization is costly at scale and prohibits amortizing load imbalance over the entire execution, and termination detection is challenging with irregular data-dependent communication. We present an asynchronous communication protocol that efficiently interleaves communication with computation. The protocol includes global termination detection without obstructing computation and communication between nodes. The user's computational primitive only needs to indicate when local work is done; our algorithm detects when all processors reach this state. We do not assume that global work decreases monotonically, allowing processors to create new work. We illustrate the utility of our solution through experiments, including two large data analysis and visualization codes: parallel particle advection and distributed union-find. Our asynchronous algorithm is several times faster with better strong scaling efficiency than the synchronous approach.

Index Terms: Computing methodologies—Parallel computing methodologies—Parallel algorithms—Massively parallel algorithms

*email: dmorozov@lbl.gov
†email: tpeterka@mcs.anl.gov
‡email: hguo@anl.gov
§email: mukundraj2@gmail.com
¶email:xu.2205@osu.edu
‖shen.94@osu.edu

1 Introduction

Iterative parallel algorithms consist of alternating rounds of computation and communication. Traditionally, HPC scientific codes implemented this pattern by synchronizing at the end of each round, often because a blocking collective operation was needed to gather results and determine whether the algorithm converged and could terminate. This pattern of bulk-synchronous parallel (BSP) programming has been in use since 1990 [27] and is still popular today. Relatively straightforward to program, the BSP model can be inefficient for algorithms that otherwise do not require strict synchronization to produce a correct result. Global synchronization, in addition to being a costly operation at scale, prohibits amortizing load imbalance over the cost of the entire execution, instead forcing all processes in all rounds to wait for the slowest computation to finish.

BSP can be especially inefficient in distributed data analytics. All but the most trivial analyses require considerable non-local communication that is often irregular and highly unbalanced. Unlike simulation codes, analytics codes often lack the computational intensity to mask communication overheads. Iterative irregular data-dependent communication is common in modern data analytics: scientific visualization, machine learning, computational topology and geometry, graph analytics, uncertainty analysis are abundant in algorithms that require flexible communication patterns.

To build these at scale, algorithm designers and software developers need high-level asynchronous communication patterns that interleave frequent communication with computation. We are interested in the following high-level algorithm, running on every process:

```
1 while not globally done do
2   dequeue incoming messages
3   perform local work
4   enqueue outgoing messages
5   exchange messages asynchronously
```

There are many ways to approach this problem. We are specifically interested in the general setting, where one cannot assume that the total amount of work in the system is known, nor that it monotonically decreases. Performing local work can spawn new work (what we refer to as nonmonotonicity throughout the paper), making the total number of iterations of the while-loop unknown. Each process only knows whether it currently has any local work to complete. The key problem is to decide when the algorithm is *globally done*, i.e., when all processes are done locally and no messages remain in flight. To be efficient and let each process proceed at its own pace, we explicitly aim to avoid synchronizing in the global termination detection.

This paper presents a solution to this problem, called `iexchange`: an asynchronous iterative message exchange protocol and an asynchronous termination detection algorithm, both built on top of MPI. The communication pattern is responsible for detecting when all processes are out of work and there are no point-to-point messages in flight. A key advantage of our approach is that porting an existing code from BSP to an asynchronous implementation is straightforward. The main abstraction remains an iterative compute-communicate pattern like BSP, familiar to parallel programmers.

After reviewing related work in Section 2, we present the design of the protocol in Section 3. It features a simple implementation of asynchronous termination detection that can be cancelled or updated by a local process at any time. Using MPI-3 nonblocking collectives enables an elegant solution to (cancellable) termination detection. Section 4 demonstrates and evaluates performance of `iexchange` in two applications: parallel particle tracing and distributed union-find. In both applications, synthetic and real datasets are used to evaluate performance and compare `iexchange` with traditional synchronous `exchange` (i.e., BSP). At scale, `iexchange` demonstrates clear performance advantages.

2 Related Work

2.1 Bulk-synchronous parallel programming

Traditional data-parallel codes are written in MPI [9], designed over twenty years ago for executing process-parallel bulk-synchronous distributed computing and communication. For productive programming of data analytics, higher level programming models built on top of MPI can promote modularity and reuse of frequently recurring design patterns while attaining the proven performance of MPI.

Our work builds on DIY [17], which acts both as a programming model and a runtime for block-parallel analytics on distributed-memory machines. DIY uses MPI as the underlying communication mechanism, hiding low-level details of MPI from the programmer. DIY's main abstraction is block parallelism: data are decomposed into blocks; blocks are assigned to processing elements (processes or threads); computation is described as iterations over these blocks, and communication between blocks is defined by reusable patterns. Blocks and their message queues are mapped onto processes and placed in memory or storage by the DIY runtime, enabling the same program to execute in-core, out-of-core, serial, parallel, single-threaded, multithreaded, or combinations thereof. Building on the block abstraction, communication patterns and other algorithms can be developed once and reused.

Prior to the work described in this article, DIY expected the computation to be organized in a BSP pattern, called `exchange`, where computation iterates over the blocks, followed by a communication phase. This article describes the implementation of an asynchronous communication pattern called `iexchange`, the nonmonotonic termination detection that is part of the implementation, and experimental results comparing the performance and scalability of the BSP and asynchrononous protocols in DIY. [1]

2.2 Iterative algorithms

Unfortunately, the BSP pattern, while allowing block scheduling, placement, and execution by the runtime, can be limiting for programmers faced with the kinds of data-dependent communication patterns found in modern data analytics algorithms.

Higher-level programming models than MPI that offer iterative data-dependent processing are Charm++, Legion, and Regent. Charm++ [14] has a dynamic mapping between data objects and processes (called chares) and between chares and physical processes. In Legion [2], blocks are called logical regions, and the runtime maps logical regions to physical resources. Regent [25] is a new language and compiler for the Legion runtime, which results in shorter, more readable code than the original Legion language.

Ovcharenko et al. [21] proposed a library for data-driven communication among neighboring processes, where the neighborhood is allowed to change dynamically: processes can enter and leave a communication neighborhood at will. Their implementation differs from ours in two ways. First, their programming model is process-parallel—computation and communication is programmed in terms of MPI processes rather than blocks. Second, they use a blocking allreduce operation to synchronize computation each time consensus on group membership in the neighborhood needs to be determined; we rely on non-blocking consensus.

[1] The work described in this article is available in the latest version of the open-source DIY software. https://github.com/diatomic/diy

2.3 Consensus protocols

2.3.1 Client-server distributed protocols

In general, monotonicity makes consistency in distributed systems easier. The CALM (Consistency as Logical Monotonicity) Theorem states that monotonic programs have consistent and coordination-free distributed implementations [11]. Our problem is not monotonic, however. We assume that whether a process is done can change locally at any time, while globally, the total amount of work in the system is not fixed. Hence, the CALM Theorem does not apply. Moreover, we assume these changes can happen at a high frequency, meaning that the cost of achieving consistency through distributed consensus can impede performance and scalability.

Distributed consensus protocols such as Paxos [15] and Raft [20] rely on quorum voting protocols for determining ordering or group membership and are not designed for high-frequency changes. Distributed database transaction protocols generate serializable ACID (Atomic, Consistent, Isolated, Durable) transactions in order to guarantee correctness, at the cost of coordination using blocking communication. To minimize the costs of coordination, some systems relax consistency by minimizing coordination [1, 28] or eliminating it altogether [32].

Neither the quorum nor the transaction consensus protocols are appropriate for our case. First, MPI has strict and static membership, so that we always know that all group members (i.e., processes) are alive. Second, the state of our consensus (in our case the global work left in the system) can change rapidly. It is more appropriate to build a protocol over the strong guarantees and the low latency provided by MPI.

2.3.2 Termination Detection in MPI Applications

Blocking collectives such as allreduce[2] are not appropriate because our objective is to avoid synchronization in the rounds of an iterative algorithm. Likewise, a single instance of a nonblocking collective such as iallreduce does not support nonmonotonicity because according to the MPI standard [5], nonblocking collectives are not allowed to be cancelled.

MPI distributed consensus protocols for sparse data exchange such as nonblocking consensus [12] assume that the local work decreases monotonically. Once a process enters the nonblocking barrier, it cannot leave and do any additional work.

A simple implementation of a nonmonotonic consensus is a single global work counter accessed through a one-sided MPI window. Implemented using atomic increment and decrement operations, the counter can increase and decrease as needed and can be accessed by all processes. Our early experiments (not shown here) using this method confirmed that the problem with this approach is scalability. Contention for access to the counter increases with number of processes. Furthermore, hosting the shared counter occupies so many CPU cycles that the same process cannot effectively perform the same work as the other processes.

A scalable tree version of a global counter was developed by Sinha et al. [24] that counts local work, which is allowed to increase or decrease, and accumulates the global counts up a tree. Charm++ uses this algorithm for termination detection, but requires producers to indicate when they are done adding work to the system, making the total work after this point decrease monotonically.

Active Pebbles [31] is another framework that incorporates several termination detection algorithms, all more restrictive than our algorithm. The key distinction of our termination protocol is what Willcock et al. [31] call depth limitation. They assume limits on the amount of new work or nonmonotonicity, which makes termination detection easier, allowing for reduce-scatter patterns to be used, such as the PCX protocol [12]. Our approach, in contrast, is unlimited-depth by design. This means that `iexchange` will iterate until it converges.

Dathathri et al. [3] describe a bulk-asynchronous parallel system, Gluon-Async, for distributed graph analytics that takes advantage of the graph algorithms' robustness to stale reads. Their termination detection relies on broadcasts of consecutive snapshots of the state of the system across all data blocks. While sharing some similarities with Gluon-Async, our solution is not tailored to a particular set of algorithms such as graph analytics. Besides supporting a wider class of algorithms, our termination detection uses two non-blocking collectives to transition through the underlying state machine vs. three snapshots used by Dathathri et al.

We present here a simple and effective implementation of a nonmonotonic consensus protocol out of nonblocking ibarrier and nonblocking iallreduce calls. The result is a cancellable protocol that allows the amount of work to change independently on each rank, terminating only when all blocks have simultaneously exhausted all their local work.

3 IEXCHANGE ALGORITHM AND TERMINATION DETECTION

The main goal of our algorithm is to seamlessly interleave computation and communication, while presenting a simple interface to the user. We accomplish this by requiring the user to provide a callback that the algorithm may call arbitrarily often. The callback is supposed to dequeue any incoming data, perform local computation, and enqueue any outgoing data. Algorithm 1 provides a pseudo-code sketch of a callback. We emphasize that this sketch is simplified for clarity: dequeuing, local computation, and enqueuing can be performed in arbitrary order. The salient point of the pseudo-code is that the user works with intermediate data queues for communication; the actual exchange of the data between MPI ranks is taken care of by our algorithm. The return value of the callback signals whether any local work still remains (set to false, for example, if the work cannot be completed because extra information is required).

Algorithm 1: Callback structure

```
1 foreach incoming queue q do
2   dequeue data from q
3 perform local computation
4 foreach data to send do
5   enqueue outgoing data to its recipients
6 return whether any local work remains
```

Algorithm 2 outlines the `iexchange` algorithm itself. While

[2]Here and in pseudocode, we typeset MPI commands in sans-serif, but skip the prefix, to avoid clutter. In other words, we write allreduce instead of MPI_Allreduce.

global termination condition is not reached, the algorithm cycles over its local blocks. It sends and receives the message queues, which store arbitrary data at the granularity chosen by the user — from individual particles to large subsets of grids — and executes the user-supplied callback. Crucially, it maintains the count of local *work*. Initially, this is set to be the number of local blocks. When a block indicates that it has finished all of its local computation, the *work* counter is decremented. The global termination is reached when *work* counters on all MPI ranks drop to 0.

Algorithm 2: iexchange(`f`)

```
1  finished = false
2  state = 0
3  work = number of local blocks
4  set b.done = false for all blocks b
5  while not finished do
6    foreach block b do
7      send outgoing message queues (Algorithm 3)
8      check incoming message queues (Algorithm 4)
9      done = block callback function f(b)
10     work += number of outgoing queues filled by the
         callback
11     if done and not b.done then
12       decrement local
13     b.done = done
14   finished = all_done(work, state) (Algorithm 5)
```

The local work counter keeps track of more than just the number of incomplete blocks. It is incremented for every outgoing queue and decremented once the queue is received by its target. To make sure this happens synchronously and work is never lost in the system, we use MPI's nonblocking synchronous issend operation. When the request returned by this operation is complete, MPI guarantees that the message has been received by its target. This allows us to increment the work counter on the receiver before it is decremented by the sender, and therefore total work in the system does not drop to 0. Algorithms 3 and 4 spell out the details of this exchange.

Algorithm 3: Send outgoing

```
1 foreach outgoing queue q to rank r do
2   request = issend(r, q)
3   append request to requests
4 foreach request ∈ requests do
5   if test(request) then
6     decrement work
7     delete request
```

The key problem for termination detection is how to decide when everyone is done with an irregular distributed computation whose global amount of work cannot be assumed to monotonically decrease. We developed a scalable implementation of a nonmonotonic consensus protocol out of a pair of MPI nonblocking collective calls, namely by interleaving ibarrier and iallreduce operations. Together they make up a cancellable protocol that allows work to be globally agreed upon, while local work may grow and shrink. The protocol terminates only when all ranks have simultaneously exhausted all their local work.

Algorithm 4: Check incoming

```
1 while r = iprobe do
2   increment work            // "inflight" count
3   q = recv(r)
4   b = determine which block the queue is for
5   if b.done then
6     increment work
7     b.done = False
8   decrement work            // undo "inflight" count
```

The crux of the problem is reaching consensus about global termination: when all local work is finished and no messages remain in-flight. Our algorithm accomplishes this by a call to `all_done`, listed in Algorithm 5. Figure 2 lists the underlying state diagram. Initially, every rank starts in State 0 and remains there while its local work is non-zero. When the local work is exhausted, it transitions to State 1, activates a non-blocking ibarrier, which will detect when all ranks have entered State 1, and initializes a *dirty* flag, which indicates whether this rank has seen any work since entering the new state. Although not explicitly listed in Algorithms 2, 3, 4, they set the *dirty* flag to `true` any time they increment local work from 0. In State 1, once the non-blocking ibarrier succeeds, the ranks transition to State 2 by initiating a non-blocking global reduction of the *dirty* flags, using a logical `or`: if any rank has seen any work since activating its ibarrier, the termination detection needs to reset. This check is the only function of State 2. If the iallreduce returns `true`, we can successfully terminate: all ranks entered the ibarrier and haven't seen any work since. If it returns `false`, we reset to State 0.

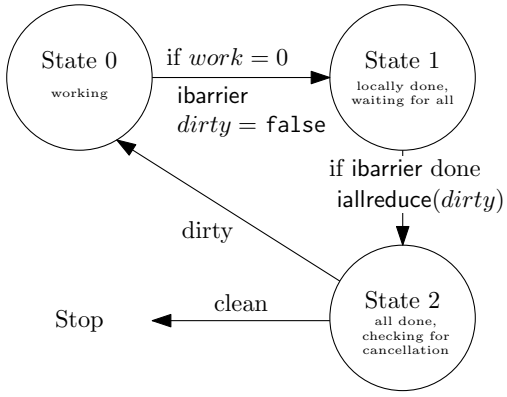

Figure 2: State diagram for `all_done`, Algorithm 5.

Because we use a synchronous issend to exchange the data between ranks in Algorithm 3, we can guarantee that termination is not reached prematurely. If the receiver is in State 1, while the sender is still in State 0, then before the sender transitions into State 1 (after decrementing its local *work*), the receiver will detect the new message via an iprobe in Algorithm 4, increment its local *work* and therefore set its *dirty* flag to `true` before the ibarrier succeeds and iallreduce is activated.

Algorithm 5: all_done(*work*, *state*)

```
1  if state = 0 and work = 0 then
2    ibarrier_request = ibarrier
3    dirty = false
4    state = 1
5    return false
6  if state = 1 then
7    if test(ibarrier_request) = true then
8      iall_reduce_request = iallreduce(all_dirty ← dirty,
         or)
9      state = 2
10     return false
11 if state = 2 then
12   if test(iall_reduce_request) = true then
13     if all_dirty then                          // done
14       return true
15     else                                       // reset
16       state = 0
17       return false
```

4 Applications of Iexchange and Comparison with Exchange

We evaluate the iexchange protocol on two classical algorithms in large-scale parallel visualization and analysis: parallel particle advection and distributed union-find. Both applications are iterative, and their computation and communication loads are highly irregular and data-dependent.

4.1 Particle Advection

Particle advection is a fundamental procedure in the visualization and analysis of vector flow fields such as those in computational fluid dynamics (CFD) simulations. Massless particles are seeded in initial positions in a vector field and are then advected over a number of integration steps. The trajectories that the particles follow can be visualized or used for other analysis, such as segmenting the field based on its divergent and convergent behavior [8]. Parallel particle tracing in distributed memory has traditionally been difficult to scale because the communication volume is high, the computational load is unbalanced, and the I/O costs are prohibitive.

Parallel methods typically follow one of three models. Task-parallel algorithms decompose the problem along different seed particles and their trajectories. Each process is responsible for some number of seeds over their lifetime [18]. Data-parallel methods decompose the problem over the vector field, where each process is responsible for the particles as they come and go through the subdomain of the field [22]. Hybrid algorithms combine the two approaches, parallelizing over particles but transferring particles when workload becomes unbalanced [23].

We implemented the data-parallel method of Algorithm 6 using DIY's blocks as the units of domain decomposition, each block containing a spatial subdomain of the input vector flow field. We wrote two variants of the algorithm; one using synchronous nearest-neighbor exchange and synchronous termination detection using a global collective in each round, and the other using the asynchronous iexchange communication protocol of Algorithm 2 and asynchronous termination detection of Algorithm 5. The synchronous exchange version is also the algorithm in [22], meaning that the following experiments both benchmark the difference between the exchange and iexchange protocols, and at the same time, they compare our asynchronous implementation with a state-of-the-art particle tracing algorithm.

Algorithm 6: Parallel particle tracing

```
1 while not globally done do
2   local particles ← dequeue incoming particles
3   foreach local particle p do
4     while p ∈ local block bounds do
5       advect p with Runge-Kutta scheme
6     if p ∈ global domain bounds then
7       enqueue p to neighboring block
8     else
9       retire p
```

We used the Theta supercomputer at the Argonne Leadership Computing Facility for the following tests.[3]

4.1.1 Synthetic Data

We designed a synthetic flow field, shown in Figure 3 to compare the performance of the synchronous exchange protocol with iexchange. The dataset is intentionally constructed with load imbalance that drifts through the field as the particle advection progresses. We hypothesize that this type of dataset should favor the iexchange protocol, which can amortize the cost of heavily-loaded blocks with the lightly-loaded ones, whereas the strict synchronization of the exchange protocol forces each advection round to take as long as the slowest block. Velocity is in the $+x$ direction. All blocks have velocity $fast\_vel$, except the blocks on the 3-d diagonal, which have velocity $slow\_vel$. Both parameters are adjustable, allowing us to change the ratio between fast and slow velocities. We used $fast\_vel = 10$ and $slow\_vel = 1$ in the following tests. A 512^3-voxel grid is seeded with 4 particles per voxel in the y and z dimensions at the far left plane of the domain.

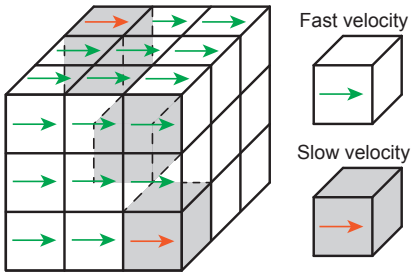

Figure 3: Synthetic flow test generates load imbalance by modifying the velocity magnitude in the blocks that are positioned on the 3-d diagonal of the lattice of all the blocks.

[3]Theta is a Cray XC40 machine with 4,392 nodes. Each node has one Intel Xeon Phi Knights Landing 64-core CPU, 16 GB high-bandwidth MCDRAM, 192 GB DDR4 RAM, and 128 GB of SSD storage.

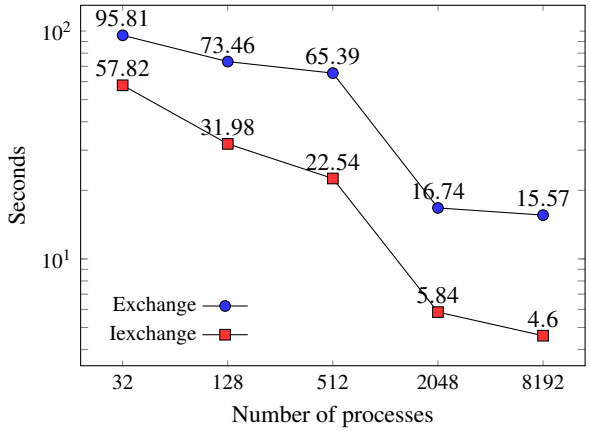

Figure 4: Time to compute particle advection using `exchange` at 2 blocks per process and `iexchange` at 16 blocks per process over a synthetic vector field of 512^3 vectors that are unbalanced by a factor of 10:1. Particles are seeded at a rate of one per every 0.25 grid points in y and z dimensions in leftmost plane of the domain.

Determining number of blocks per MPI process Table 1 shows the result of an experiment to determine how many blocks to create for each MPI process. For each number of processes the best (lowest) time for `exchange` and for `iexchange` is highlighted in bold font. We see that the performance is sensitive to the number of blocks, and the two protocols have different optimal number of blocks per process. `Exchange` tends to favor smaller numbers of blocks. The optimum is either 2 or 4 blocks per process; we selected 2 blocks per process as the best setting. `Iexchange`, on the other hand, favors larger numbers of blocks per process. The optimal number of blocks increases with scale, with 16 blocks per process being best at 2,048 processes, and nearly best at 8192 processes. We selected 16 blocks per process as the setting for `iexchange`.

Strong scaling experiment Figure 4 shows the timing for a strong scaling experiment using the synthetic dataset described above. The ratio of *fast_vel* to *slow_vel* is 10:1. Particles are seeded at a rate 0.25 (4 particles per voxel). The reported time, shown in log-log scale, is the average over three trials. Based on the result of the previous experiment, we used two blocks per MPI process for the synchronous `exchange` protocol and 16 blocks per process for asynchronous `iexchange` Comparing the time for `exchange` with `iexchange`, we see that `iexchange` is up to 3 times faster and has 2 times higher strong scaling efficiency (the slope of the curve) than `exchange`.

To help understand the results, Figure 1 visualizes time traces collected on a Linux workstation with dual-socket Intel Xeon Gold 6230, with total of 40 physical cores and 80 hyper-threads. 64 MPI ranks were used; the domain was decomposed as a 4^3 grid of blocks, meaning there were 4 blocks with *slow_vel*. In this test, $fast\_vel = 10$ and $slow\_vel = 0.1$. Computation (particle advection) is in blue; communication (particle exchange and termination detection), in red. The four slow blocks correspond to the long blue lines. In `exchange`, the four rounds of advection are clearly visible, with synchronization at each round: each slow block is processed in a separate round, which is responsible for the long cumulative time. In contrast, `iexchange` has interleaved computation and communication. The particles reach each slow block faster, without having to synchronize across blocks. As a result, the cumulative time is only a little slower than the time to process a single slow block.

4.1.2 Computational Fluid Dynamics Data

The Nek5000 dataset shown in Figure 5 is a 3-d vector field representing the numerical results of a large-eddy simulation of Navier-Stokes equations modeled by the Nek5000 simulation code [4] for the MAX experiment [16]. The dataset represents the turbulent mixing and thermal striping that occurs in the upper plenum of liquid sodium fast reactors. The data have been resampled from their original topology onto a 512^3 regular grid.

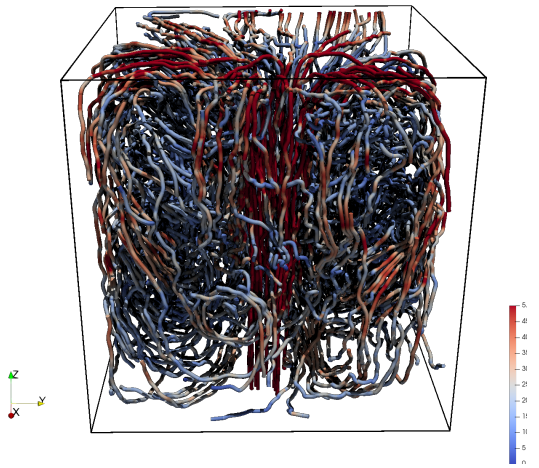

Figure 5: Streamlines traced in Nek5000 dataset indicate a high degree of turbulence in the flow field.

We conducted a similar experiment as shown in Table 1 to find the best number of blocks per MPI process for `exchange` and `iexchange` for the Nek5000 dataset as for the synthetic data. The data, not shown here to save space, resulted in the same conclusion: two blocks per MPI process for `exchange` and 16 blocks per process for `iexchange` produced the best performance. We continued to use these settings in the tests below.

Figure 6 shows the timing for a strong scaling test advecting particles through the Nek5000 dataset. The time, shown in log-log scale, initially shorter for `exchange`, is shorter for `iexchange` beyond 512 processes. There are two reasons for this improvement. Typically, load imbalance worsens as the number of blocks and MPI processes increases because blocks become smaller, isolating local fluid flow effects such as vortices. As the previous experiment showed, `iexchange` can amortize some of those effects. Also, the higher number of blocks affords more opportunities for `iexchange` to find work to do because it does not have to wait and synchronize on each round. This is true during message transmission as well as termination detection, both being synchronous in `exchange` and asynchronous in `iexchange`. Similar to the synthetic benchmark, at scale

Protocol	Blocks per Process	Time (s) for 32 Processes	Time (s) for 128 Processes	Time (s) for 512 Processes	Time (s) for 2048 Processes	Time(s) for 8192 Processes
Exchange	1	**93.027**	79.390	19.931	17.330	15.846
	2	95.807	**73.461**	65.394	**16.740**	15.567
	4	285.032	74.465	62.390	59.530	**15.239**
	8	281.349	244.787	**63.319**	58.225	56.280
	16	290.736	243.664	228.618	58.785	55.583
Iexchange	1	96.403	81.911	20.377	17.074	15.583
	2	64.308	45.683	36.211	9.145	7.986
	4	107.284	**27.348**	20.046	17.066	**4.277**
	8	73.196	47.373	**12.249**	9.561	8.123
	16	**57.816**	31.981	22.537	**5.839**	4.601

Table 1: Performance with Varying Number of Blocks per MPI Process

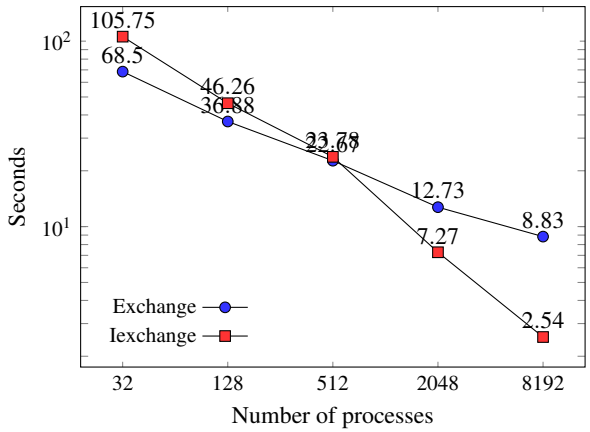

Figure 6: Time to compute particle advection of over Nek5000 dataset of 512^3 vectors. Particles are seeded at a rate of one per every 4 grid points in each dimension. Two blocks per MPI process were used for `exchange` and 16 blocks per process for `iexchange`.

`iexchange` is up to 3 times faster than `exchange`, with 5 times higher strong scaling efficiency.

4.2 Distributed Union-Find

Union-find is a classical algorithm for connected component labeling of undirected graphs using a disjoint set data structure [7, 26]. Algorithm 7 outlines the serial union-find algorithm.

Algorithm 7: Serial union-find

```
1 foreach vertex v ∈ V do
2   make singleton set(v)
3 foreach edge (u,v) ∈ E do
4   if find_set(u) ≠ find_set(v) then
5     union(u, v)
```

Several distributed parallel algorithms for union-find of disjoint sets have appeared in various contexts. Harrison et al. [10] identify disjoint sets locally and then merge local sets into global ones using synchronous global communication. Iverson et al. [13] combine local computation and global synchronous communication round-by-round to find unions of disjoint sets. Friederici et al. [6] implemented distributed union-find, in order to analyze percolation of turbulent fluid flows, by sending all local updates to a single global process. Nigmetov and Morozov [19] find unions of distributed disjoint sets in the course of computing connected components in parallel merge trees. They compute the unions over multiple rounds until convergence, using DIY's synchronous `exchange` for nearest-neighbor communication and `reduce` for global reduction.

Xu et al. [33] compute connected components with a distributed union-find algorithm for the purpose of tracking features such as trajectories of critical points over time-varying data. Like Nigmetov and Morozov, they compute unions locally, exchanging only information with nearest neighbors, but they use DIY's new `iexchange` protocol for asynchronous exchange and termination detection.

For the purpose of the following experiments, we ran the asynchronous algorithm of Xu et al., along with a second version using synchronous nearest-neighbor `exchange` and synchronous collective `reduce` to detect termination, with the other local operations being identical. The time reported is only for the distributed union-find, i.e., connected component labeling, which is one step of a larger feature tracking pipeline in the paper of Xu et al. As in the previous particle tracing experiments, the objective of these tests is to compare the performance of the synchronous and asynchronous protocols. The synchronous and asynchronous algorithms in [33] are representative of the state of the art, scaling to 8,192 ranks. This means that the following experiments both benchmark the difference between the `exchange` and `iexchange` protocols, and they demonstrate the state of the art in distributed union-find algorithms.

Each MPI process executes Algorithm 8.[4] The operation `set`(u) returns the label of the set containing vertex u, and the operation `process`(u) is the rank of the neighboring MPI process owning the subdomain containing vertex u.

We used the same Theta supercomputer as for particle advection. For the following tests, we used only one block per MPI process because the code of Xu et al. was not written for multiple blocks per process.

[4] Algorithm 8 is a high-level outline of the algorithm of Xu et al., with more details in [33].

Algorithm 8: Distributed parallel union-find

```
1 while not globally done do
2   dequeue incoming vertices and labels of sets
3   perform local serial union-find (Algorithm 7)
4   foreach edge (u, v) do
5     if process(u) ≠ process(v) then
6       if set(u) changed then
7         enqueue [u, set(u)] to process(v)
8       if set(v) changed then
9         enqueue [v, set(v)] to process(u)
```

4.2.1 Synthetic Data

We used the synthetic benchmark dataset developed by Xu et al. [33] for these experiments. The dataset consists of a regular grid mesh of two spatial dimensions and 1 temporal dimension (3-d in total) and has a variable number of critical points whose trajectories move in the space dimensions as a function of time to generate spiral-like trajectories in space-time. The trajectories of the critical points in space-time are the connected components (i.e., disjoint sets) that are being identified with Algorithm 8.

For the following experiment, we selected 128×128 spatial resolution $\times 128$ time steps, and we tracked three types of critical points (minima, maxima, and saddles). The resulting total number of components is 280. These components on average have 337 number of elements, ranging from 4 to 574.

Figure 7 shows the result of a strong scaling experiment comparing `exchange` with `iexchange`, plotted in log-log scale. The synchronization in `exchange` stops scaling beyond 128 MPI processes, whereas `iexchange` continues to scale to 2,048 processes. The connected component statistics above highlight the severe load imbalance between the smallest and largest component. Moreover, the load changes dynamically as the union-find algorithm progresses through its iterations, and components grow in size. The global amount of work in the system also fluctuates dynamically, and `exchange` must do many rounds of blocking global collectives to detect termination.

4.2.2 Experimental Data

Using an exploding-wire experiment, scientists can generate high-temperature microparticles of different types, including electrons, ions, atoms, and molten dust [29, 30]. High-speed cameras capture the movement of these particles and produce high-resolution images. Tracking these particles (Figure 8) in the images helps researchers understand properties of microparticles and high-temperature plasmas, study their interactions, and develop advanced techniques for plasma fueling. Moreover, the imaging data can aid scientists to enhance theoretical models for simulations.

The following experiment measures the performance of the distributed union-find algorithm while tracking particles over 4,745 time steps of exploding wire images, each 384×384 pixels. The total number of components is 52,978. The minimum number of elements per component is 1; the maximum number of elements per component is 11,086, and the average number of elements per component is 60.

Figure 9 shows the strong scaling results plotted in log-log scale. We see similar trends as in the synthetic data. The

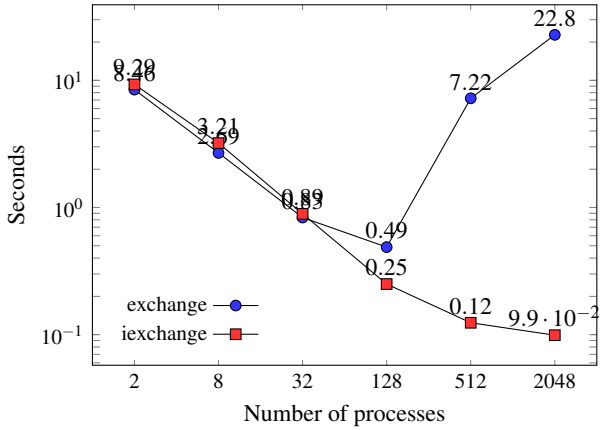

Figure 7: Time to compute distributed union-find of connected components of critical points in a synthetic field of 128^2 scalars over 128 time steps. The number of blocks per MPI process is 1.

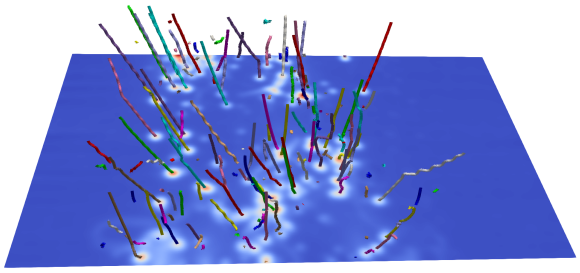

Figure 8: Microparticle tracking in exploding-wire experiments is one application of the distributed union-find algorithm.

`exchange` protocol stops scaling at 128 MPI processes, while `iexchange` continues to scale to 8,192 processes. At scale, `iexchange` is over 130 times faster than `exchange`, for the same reasons as the synthetic data: severe, dynamically changing workload over a large number of iterations.

5 Conclusion

Bulk-synchronous parallel programming can be inefficient for iterative algorithms that do not require strict synchronization to produce a correct result. As an alternative to BSP, we presented an asynchronous communication protocol, for exchanging MPI messages in iterative algorithms, that efficiently interleaves communication with computation. The protocol includes an algorithm that detects global termination without obstructing computation and communication between individual nodes. The computational primitive supplied by the user only needs to indicate when it is done with the local work; our algorithm detects when all processors reach this state. Crucially, we do not assume that global work decreases monotonically, and we allow for processors to create new work, including for other processors.

We demonstrated the practical utility and efficiency of our solution in several synthetic and real-world examples. Our experiments featured two data analysis codes: parallel particle

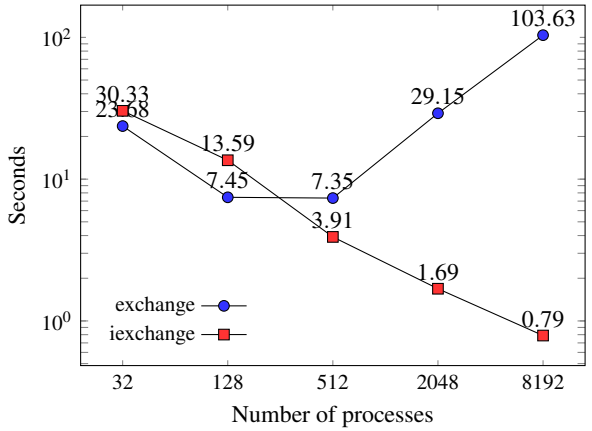

Figure 9: Time to compute distributed union-find of connected components of critical points in high-speed images of 384^2 pixels over 4,745 time steps. The number of blocks per MPI process is 1.

advection in fluid flow analysis, and distributed union-find in connected component labeling. The performance tests in the previous section showed that our asynchronous algorithm is several times faster and has better strong scaling efficiency than the conventional synchronous approach. The results demonstrated that `iexchange` can effectively hide load imbalance that occurs in algorithms where the computational and communication workloads are data-dependent. This is particularly the case at scale, when load imbalance becomes more acute, and the ratio of communication to computation increases.

Acknowledgments

This work is supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contracts DE-AC02-05CH11231 and DE-AC02-06CH11357. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. The authors wish to thank Zhehui Wang of Los Alamos National Laboratory for use of the high-speed imaging dataset.

References

[1] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment*, 8:185–196, 2014.

[2] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 66. IEEE Computer Society Press, 2012.

[3] R. Dathathri, G. Gill, L. Hoang, V. Jatala, K. Pingali, V. K. Nandivada, H. Dang, and M. Snir. Gluon-Async: A Bulk-Asynchronous system for distributed and heterogeneous graph analytics. In *28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 15–28, Sept. 2019. doi: 10.1109/PACT.2019.00010

[4] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale Algorithms for Reactor Hydrodynamics. *Journal of Physics Conference Series*, 125(1):012076–+, July 2008. doi: 10.1088/1742-6596/125/1/012076

[5] M. P. I. Forum. *MPI: A Message Passing Interface Standard: version 3.1; Message Passing Interface Forum, June 4, 2015*. University of Tennessee, 2015.

[6] A. Friederici, W. Köpp, M. Atzori, R. Vinuesa, P. Schlatter, and T. Weinkauf. Distributed Percolation Analysis for Turbulent Flows. In *2019 IEEE 9th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 42–51. IEEE, 2019.

[7] Z. Galil and G. F. Italiano. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Computing Surveys (CSUR)*, 23(3):319–344, 1991.

[8] C. Garth, F. Gerhardt, X. Tricoche, and H. Hans. Efficient Computation and Visualization of Coherent Structures in Fluid Flow Applications. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1464–1471, 2007.

[9] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, and T. Skjellum. MPI-2: Extending the Message-Passing Interface. In *Proceedings of Euro-Par'96*, 1996.

[10] C. Harrison, J. Weiler, R. Bleile, K. Gaither, and H. Childs. A Distributed-Memory Algorithm for Connected Components Labeling of Simulation Data. In *Topological and Statistical Methods for Complex Data*, pp. 3–19. Springer, 2015.

[11] J. M. Hellerstein and P. Alvaro. Keeping CALM: When Distributed Consistency is Easy. *arXiv:1901.01930*, 2019.

[12] T. Hoefler, C. Siebert, and A. Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *ACM Sigplan Notices*, 45(5):159–168, 2010.

[13] J. Iverson, C. Kamath, and G. Karypis. Evaluation of Connected-Component Labeling Algorithms for Distributed-Memory Systems. *Parallel Computing*, 44:53–68, 2015.

[14] L. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, ed., *Proceedings of OOPSLA'93*, pp. 91–108, September 1993.

[15] L. Lamport et al. The Part-Time Parliament. *ACM Transactions on Computer systems*, 16(2):133–169, 1998.

[16] E. Merzari, W. Pointer, A. Obabko, and P. Fischer. On the Numerical Simulation of Thermal Striping in the Upper Plenum of a Fast Reactor. In *Proceedings of ICAPP 2010*, 2010.

[17] D. Morozov and T. Peterka. Block-Parallel Data Analysis with DIY2. In *2016 IEEE 6th Symposium on Large Data Analysis and Visualization (LDAV)*, pp. 29–36. IEEE, 2016.

[18] C. Müller, D. Camp, B. Hentschel, and C. Garth. Distributed Parallel Particle Advection using Work Requesting. In *2013 IEEE Symposium on Large-Scale Data Analysis and Visualization (LDAV)*, pp. 1–6. IEEE, 2013.

[19] A. Nigmetov and D. Morozov. Local-Global Merge Tree Computation with Local Exchanges. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2019.

[20] D. Ongaro and J. Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*, pp. 305–319, 2014.

[21] A. Ovcharenko, D. Ibanez, F. Delalondre, O. Sahni, K. E. Jansen, C. D. Carothers, and M. S. Shephard. Neighborhood Communication Paradigm to Increase Scalability in Large-Scale Dynamic Scientific Applications. *Parallel Computing*, 38(3):140–156, 2012.

[22] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In *2011 IEEE International Parallel & Distributed Processing Symposium*, pp. 580–591. IEEE, 2011.

[23] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber.

Scalable Computation of Streamlines on Very Large Datasets. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–12, 2009.

[24] A. B. Sinha, L. V. Kalé, and B. Ramkumar. A Dynamic and Adaptive Quiescence Detection Algorithm, 1993.

[25] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. Regent: A High-Productivity Programming Language for HPC with Logical Regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 81. ACM, 2015.

[26] R. Tarjan. *Data Structures and Network Algorithms*. Siam, 1983.

[27] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[28] A. Verbitski, A. Gupta, D. Saha, J. Corey, K. Gupta, M. Brahmadesam, R. Mittal, S. Krishnamurthy, S. Maurice, and T. Kharatishvilli. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *Proceedings of the 2018 International Conference on Management of Data*, pp. 789–796. ACM, 2018.

[29] Z. Wang, Q. Liu, W. Waganaar, J. Fontanese, D. James, and T. Munsat. Four-Dimensional (4D) Tracking of High-Temperature Microparticles. *Review of Scientific Instruments*, 87(11):11D601, 2016.

[30] Z. Wang, J. Xu, Y. E. Kovach, B. T. Wolfe, E. Thomas Jr, H. Guo, J. E. Foster, and H.-W. Shen. Microparticle Cloud Imaging and Tracking for Data-Driven Plasma Science. *arXiv:1911.01000*, 2019.

[31] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine. Active Pebbles: Parallel Programming for Data-Driven Applications. In *Proceedings of the international conference on Supercomputing*, pp. 235–244. ACM, 2011.

[32] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A KVS for any Scale. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[33] J. Xu, H. Guo, H.-W. Shen, M. Raj, X. Wang, X. Xu, Z. Wang, and T. Peterka. Asynchronous and load-balanced union-find for distributed and parallel scientific data visualization and analysis. *IEEE Transactions on Visualization and Computer Graphics*, 27(6):2808–2820, 2021.