A SIMPLE C API FOR DYNAMIC SPARSE EXCHANGE [*]

DAN IBANEZ[†]

Abstract. We present a minimal set of C functions which form an interface for dynamic sparse exchange built on an MPI interface. Such exchanges can effectively handle the complex communication patterns of unstructured parallel codes. By allowing gradual packing and unpacking of messages, we simplify user-level source code. By implementing a scalable exchange algorithm and having the interface reflect the structure of that algorithm, we make it easier to create scalable programs. Example uses including source code and performance results are also presented.

Key words. MPI, nonblocking, massively parallel, parallel programming

AMS subject classifications. 15A15, 15A09, 15A23

1. Introduction.

Since the largest supercomputers to date are designed as distributed memory architectures, message passing between processes is the programming model of choice for programs which run at this large scale.

MPI is a standard interface for message passing which has simplified parallel programming for years by providing a consistent API across all machines of interest. The capabilities built into MPI continue to develop along with its specification, but the most complex communication patterns of real applications continue to require significant boilerplate code for MPI users, and often less-than-optimal algorithms are implemented by users for complex tasks.

We present a very simple API that gives users access to a powerful exchange mechanism, reducing code size, increasing readability, and improving scalability. This API (with trivial variations) has been used successfully to build several complex codes including the PUMI [10] set of tools for parallel unstructured mesh simulations, including adaptive finite element simulations. The interface was inspired in part by the IPComMan interface which used to provide the same functionality [13].

Programmers should have a very easy to use interface backed by the most scalable algorithms for common and difficult parallel programming tasks. Hand-coding a communication pattern with non-blocking sends and receives is more error prone that using a higher-level collective functionality [5]. The interface presented here should be ideal for applications engaged in dynamic sparse exchange as defined in Section 2.1.

2. Background.

2.1. Definition of Exchange.

An Exchange is a parallel operation in which a set of processes send messages to one another.

The first restriction that defines an exchange will be that each process must have ready all the messages it is going to send; they cannot depend upon data to be received during the exchange. Thus if one message will affect the contents of another message, those messages must be part of different exchanges.

Hoefler et al. define a Sparse Exchange as one in which each process sends to and receives from $\mathcal{O}(\log(P))$ neighbors, where P is the total number of processes [8]. The

[*] This material is based upon work supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research, under award DE-SC00066117 (FASTMath SciDAC Institute).

[†]Scientific Computation Research Center, Rensselaer Polytechnic Institute, Troy, New York, 12180 (ibaned@rpi.edu).

critical part of this definition is that the neighborhood size is asymptotically much smaller than P, and hence sparse.

A Dynamic Sparse Data Exchange (DSDE) is one in which each process knows only the processes it will send data to, not the ones it will receive data from. Thus the difficult part of such an exchange is detecting when all messages have been received.

The MPI standard is starting to include higher-level functionality for collective exchanges, most notably the neighborhood collectives [9]. These can efficiently execute non-dynamic sparse exchanges, but require pre-defined communication graphs. Our API is designed to be much easier to use, at the cost of deriving the communication graph and not performing any complex optimizations [7]. However, we automatically do message coalescing as explained in Section 3.2.

2.2. Nonblocking Consensus. Hoefler et al. further propose a highly scalable algorithm for carrying out a DSDE, which they term "non-blocking consensus". Their algorithm is repeated here because it is so fundamental to our work.

Algorithm 1 Non-blocking Consensus

```
function EXCH(List $I$ of destinations and data)
    Empty list $O$ for received data
    done=false
    barr_act=false
    for $i \in I$ do
        start nonblocking synchronous send to process dest($i$)
    end for
    while not done do
        msg = nonblocking probe for incoming message
        if msg found then
            allocate buffer, receive message, add buffer to $O$
        end if
        if barr_act then
            comp = test barrier for completion
            if comp then
                done=true
            end if
        else
            if all sends are finished then
                start nonblocking barrier
                barr_act=true
            end if
        end if
    end while
    return $O$
end function
```

In short, what Algorithm 1 does is execute two things concurrently: a loop that continuously probes for and accepts incoming messages, and a termination detection process for sent messages. The termination detection algorithm is quite natural. All outgoing messages are sent using a "synchronous" protocol, which will acknowledge receipt of the message back to its sender. Once a process receives acknowledgements about all messages it sent (meaning they were all received), it enters a barrier. Once the barrier is complete, by definition, all messages sent by all processes have been received. Such proofs of termination detection can be explained by the basic causality of message passing described by Mattern [12] and others. The trick is to execute these concurrently, which requires nonblocking synchronous sends and a nonblocking barrier.

2.3. Importance of Guarantees. The reason non-blocking consensus is so important is that it solves the DSDE problem with a guaranteed $\Theta(\log(P))$ runtime and constant memory requirements. The DSDE problem shows up very frequently in unstructured parallel programs, and there is great risk of sub-optimal algorithms being used because they are closer within reach in several ways.

For example, several packages in the popular Trilinos framework [6], including STK [3] and Zoltan [2] solve the DSDE problem using the equivalent of the algorithm termed "personalized consensus" by Hoefler et al., which has $\Theta(P)$ memory and time requirements. This is usually implemented by using MPI_REDUCE on an array of size P to compute the number of messages each rank will receive based on how many it will send to each rank, then using MPI_SCATTER to redistribute that information to the ranks.

STK does this in the

```
comm_recv_procs_and_msg_sizes
```

function of their CommSparse system, which has an API quite similar to the one presented here, with the added benefit of C++ type safety but the non-scalability of length P arrays.

Zoltan does it in the

```
Zoltan_Comm_Invert_Map
```

function of their communication utilities.

The PETSc package [1] actually has important communication functions which can use either personalized consensus or non-blocking consensus, including

```
PetscCommBuildTwoSided
```

and certain vector and matrix assembly functions. Although the scalable algorithm is implemented for these functions, it is not as readily available even for cases in which it runs faster.

Personalized consensus is not scalable in theoretical terms, but it is used in many codes as a solution to the DSDE problem because it has reasonable runtime for small values of P and it simply comes to mind more readily given the basic tools of MPI-1, specifically reduction. Non-blocking consensus is a large creative leap from there, especially since non-blocking barriers are not yet a familiar concept.

As machines such as Argonne National Laboratory's Mira IBM Blue Gene/Q are reaching core counts in the millions, we should prepare for parallel programming with P in the millions. In this case, storing and transmitting arrays of size P becomes very costly and will soon become impossible: as P increases and memory per core decreases, there may not even be room for P integers per core.

3. The Programming Interface.

3.1. Function Signatures. There are nine total functions in our exchange system. Two of them, COMM_INIT and COMM_FINALIZE, are used to set up the underlying implementation before and after all communication takes place:

```
void comm_init(MPI_Comm comm);
void comm_finalize(void);
```

Notice that the system is initialized with an MPI communicator, and will use duplicates of this communicator for all its work. The next two functions are convenient access for the common queries of rank and size on that communicator:

```
int comm_rank(void);
int comm_size(void);
```

Then we have the functions which actually comprise the DSDE operation. COMM_PACK adds some data to the buffer that will be sent to a particular neighbor, and has a C macro form which automatically provides the size argument.

```
void comm_pack(void const* data, unsigned size, int to);
#define COMM_PACK(o,to) comm_pack(&(o),sizeof(o),to)
```

After packing is complete, the entire non-blocking consensus exchange is triggered by COMM_EXCH:

```
void comm_exch(void);
```

Non-blocking consensus requires that messages be received until termination is detected. Thus there is COMM_RECV, which returns nonzero until termination is detected:

```
int comm_recv(void);
```

Messages are received, unpacked, and discarded as a stream until termination. The COMM_UNPACK function extracts data from this stream and has a macro form just like COMM_PACK:

```
void comm_unpack(void* data, unsigned size);
#define COMM_UNPACK(o) comm_unpack(&(o),sizeof(o))
```

It is also useful to know the sender of the message currently being unpacked, this can be queried at any time using COMM_FROM:

```
int comm_from(void);
```

3.2. Benefits. The key benefit of this API is the ability to gradually pack and unpack messages, in the same way that complex file formats are gradually read and written throughout a portion of code. This can be contrasted, for example, with APIs that are moving into the MPI standard for making use of sparse exchange, for example MPI_INEIGHBOR_ALLTOALLW, which requires more boilerplate code to use [4]. In fact, one can see our API as one level higher; COMM_EXCH could be implemented based on the neighborhood MPI calls.

We don't use MPI3 functions in our implementation. We implement non-blocking consensus directly, using a custom implementation of non-blocking barriers which calls only non-blocking point to point functions. This means our implementation can be compiled using most MPI installations as opposed to the most recent ones.

Note that COMM_INIT takes an MPI communicator argument, which is duplicated (not used directly) and the duplicate(s) are used for all message passing in the implementation. In addition, the implementation is such that one may call COMM_INIT after COMM_FINALIZE, essentially changing the underlying communicator. This makes it easy to implement algorithms based on changing communicators, something which will be emphasized in Section 5.2.

Finally, note that by a lack of mechanism for specifying otherwise, there is a single message going to each destination. A call to COMM_PACK just appends data to this message, creating it if necessary. This is actually a benefit in that we can avoid large numbers of non-blocking point-to-point MPI requests being created for large numbers of small messages. It has been measured that performance is greatly improved by sending a few large messages instead of many small ones [13]. Seeing this message as more of a continuous data stream simplifies programming, at least in all our use of this API.

3.3. Criticisms.

The void pointer and size integer convention for passing arbitrary data is sometimes criticized as an unsafe idiom encouraged by the C language. It should be noted that the MPI standard chose this convention and enhanced it with MPI_Datatype descriptors of the content [4]. Adding MPI_Datatype descriptors to our API would not invalidate any of the described benefits, except perhaps ease of use, given that the descriptor would also have to be gradually built up. Our assumption of homogeneity with respect to things like byte ordering and integer widths has so far held up on several leadership-class supercomputers.

The function names listed here may also be seen as excessively terse for global identifiers. This is fair criticism, and if the API had a larger user group all functions could be given an additional and more original prefix without affecting any stated benefits.

4. Implementation.

4.1. Buffers.

In order to support gradual packing of messages efficiently, we implement growing buffers to contain each message. All data packed to the same destination goes in one contiguous buffer which, when it reaches capacity, is reallocated to capacity $(3(c+1))/2$, where c is the previous capacity and integer arithmetic is used.

COMM_PACK has to find the right buffer, and create one if none exists. In choosing algorithms for doing that, we should keep in mind that per Section 2.1 there should be $\mathcal{O}(\log(P))$ total destinations. As such, it is acceptable to use a linear time algorithm to find the right buffer. Otherwise, a sorted list or balanced tree may be used to look it up in $\mathcal{O}(\log(\log(P)))$ time. This also helps in case users bend the rules of sparsity and occasionally a process has many destinations.

4.2. Determinism.

One important contribution made here is to note that the only source of parallel non-determinism in this dynamic sparse exchange algorithm is the order of messages received. If Algorithm 1 is extended by simply sorting the received messages O by their sender, the result is that the contents of all O lists on all processors , including order, are directly determined by the contents of all I lists on all processors, and not at all by the unavoidable randomness of a parallel machine and network.

This is important because the exchange algorithm itself is typically the main source of parallel non-determinism in many applications. The other communication work tends to use only collectives such as MPI_ALLREDUCE and MPI_SCAN, whose outputs are also deterministic in parallel. Serial sources of non-determinism such as using random data streams or time values as input can generally be avoided, although we will see a trade-off in Section 5.1.

As such, although the implementation in Section 3 was described as though it directly executes non-blocking consensus, it can be modified such that COMM_EXCH waits for all messages to arrive, sorts them by sender, and then COMM_UNPACK simply traverses the sorted list of messages.

Note that there are drawbacks to sorting, namely the loss of overlapping communication and computation that naturally arises in the use of non-blocking consensus. However, parallel determinism was judged to be a greater benefit, and our implementation currently defaults to this sorting behavior. Determinism becomes very important when debugging or measuring the performance of a large parallel program. Having the guarantee that, for the same input data and partitioning, the computational states and output will be the same, makes such debugging tractable. Otherwise, a combinatorial explosion results in $\mathcal{O}(\log(P)^P)$ possible outcomes for each exchange, making certain behavior not reproducible.

4.3. Race Condition. In addition to the deterministic modification discussed in Section 4.2, there is the subtle possibility of a race condition in our implementation. Specifically, when repeated exchanges are executed, there is a chance that process A receives notification that exchange i is complete much earlier than process B does. In this case, process A may begin sending messages for exchange $(i+1)$ while process B is still receiving messages for exchange i.

We choose to add a barrier to the beginning of the exchange algorithm. Another possible solution is to use a non-blocking barrier, which a process starts after it completes the exchange, and which it waits for before sending messages for the next exchange. One can also detect such issues by tagging messages with the exchange number.

4.4. State Machine. The implementation also simulates a finite state machine as users call API functions. The purpose of this state machine is to verify that users call the API correctly. The possible states are these:

- UNINIT: not initialized
- IDLE: no send or receive buffers allocated
- PACK: send buffers only allocated
- UNPACK: receive buffers only allocated
- BOTHPACK: send and receive buffers allocated

Figure 4.1 shows the states and allowed transitions based on which API functions are called. Which transition a function takes depends on implementation logic, for example COMM_RECV goes to IDLE when all received buffers have been unpacked.

Common errors detected by the state machine include calling functions before calling COMM_INIT, starting a new exchange before the previous received buffers are unpacked, and so on. Although it may seem unnecessary to do such verification, the mistakes detected would otherwise result in very subtle and hard-to-debug symptoms at large scale.

Being able to simultaneously unpack messages from the previous exchange and pack messages for the upcoming exchange is a very useful feature. As mentioned in Section 2.2, an exchange is defined such that no message can depend on others. Real complex programs have protocols with "conversation", where messages depend on previous messages. Simultaneous packing and unpacking allows a very natural style of composing such programs without setting up temporary storage between exchanges (the communication buffers replace this temporary storage).

4.5. Final Algorithm. These modifications described in Sections 4.2 and 4.3 show up as single-line changes to Algorithm 1 and result in Algorithm 2. To demonstrate that this algorithm can be well represented in code, Listing 1 shows our current

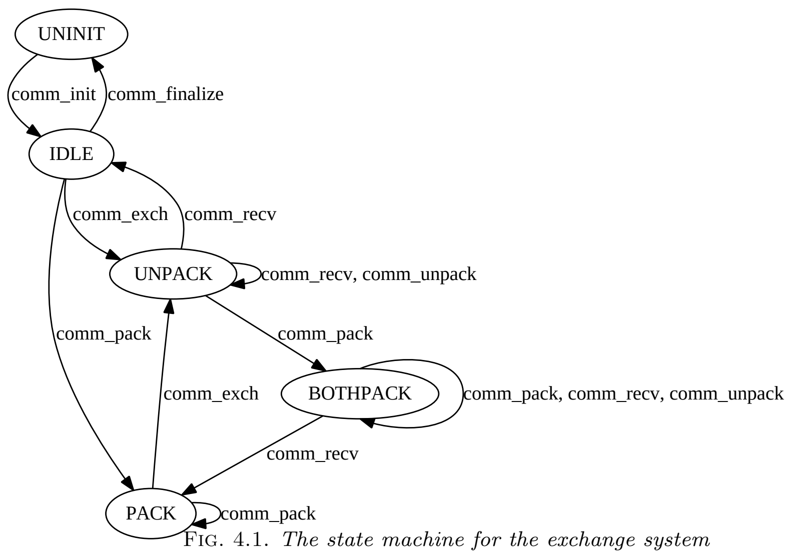

Fig. 4.1. *The state machine for the exchange system*

COMM_EXCH implementation verbatim. Although some of the specific functions may be unfamiliar, the general outline of Algorithm 2 is clearly present. Notably, we have wrapped functions such as MPI_ISSEND with lower-case variants that have simplified signatures. If nothing else, it seems to increase the readability of these listings.

5. Example Uses. Our communication API is used in a wide variety of algorithms, all of which contribute towards an unstructured mesh simulation. To avoid introducing the full complexity of such a system, we have chosen two example algorithms which can be understood on their own and are important parts of the full simulation. Performance data from both algorithms is retrieved from a scaling study performed on the full set of tools. In this study, a tetrahedral mesh is partitioned over the cores of an IBM BlueGene/Q, reaching a maximum of 16K cores on 1K nodes. Every time the number of cores is doubled, the number of elements is also doubled, resulting in 100K elements per process on average. The algorithm which adds elements also perturbs the partitioning, such that some processes have up to 120K elements (20% imbalance). The RIB algorithm described in Section 5.2 is used to restore a balance of elements before doubling the number of processes again.

5.1. Luby's Maximal Independent Set. Luby's randomized algorithm for finding maximal independent sets is notable for being well-suited to parallel implementation [11]. We will consider the simplest algorithm outlined in Luby's original work, which is based on assigning random numbers to graph nodes. It is repeated in condensed form as Algorithm 3. In our case, the graph nodes are MPI ranks, and our intention is to use this algorithm as a building block to develop a coloring algorithm to assign a color to the mesh part held by each MPI rank, such that no two adjacent mesh parts have the same color.

Listing 2 shows the C code that executes Luby's algorithm. This demonstrates the use of our API functions and their effectiveness in producing concise parallel code. We use the variable `in_Vp` to denote whether the local MPI rank is part of set V', and

Algorithm 2 Modified Non-blocking Consensus

```
function EXCH(List $I$ of destinations and data)
    Empty list $O$ for received data
    done=false
    barr_act=false
    run blocking barrier                                ▷ Race condition fix (Section 4.3)
    for $i \in I$ do
        start nonblocking synchronous send to process dest($i$)
    end for
    while not done do
        msg = nonblocking probe for incoming message
        if msg found then
            allocate buffer, receive message, add buffer to $O$
        end if
        if barr_act then
            comp = test barrier for completion
            if comp then
                done=true
            end if
        else
            if all sends are finished then
                start nonblocking barrier
                barr_act=true
            end if
        end if
    end while
    sort $O$ by sender process                           ▷ Determinism fix (Section 4.2)
    return $O$
end function
```

so on for the other relevant sets of vertices. The result of the function is whether or not the local rank is in the independent set I. A simple seed `(comm_rank() + 1)` is used for the random number generator to make the code deterministic in parallel. If one wants better randomness guarantees, a more sophisticated seed can be used, and choosing one is beyond the scope of this paper. The algorithm continues until no rank is in the set V', which is accomplished by our wrapper `mpi_max_int` and `comm_mpi`, which returns the current communicator. Most of the communication rounds pack information about the local rank to neighboring ranks, and then reducing the incoming data from neighboring ranks to compute whether the local rank is in one of the sets.

Since we assume the graph is symmetric, neighbors being received from are known and the exchange is not strictly dynamic. However, the code will only increase in size if we try to account for this explicitly.

In our scaling study, we measure the time required to choose colors for each of the processes based on mesh connectivity. This process executes Luby's algorithm multiple times. All the processes not yet colored are run through Luby's algorithm to obtain an independent set of processes, which are then given a previously-unused color. This continues until all processes are colored. Figure 5.1 shows the time for the full coloring algorithm as a function of the number of processes. We expect a

Listing 1
COMM_EXCH code

```
void comm_exch(void)
{
  msgs* o;
  unsigned i;
  if (!(global_state == PACK || global_state == IDLE))
    die("%d called exch in state %d\n", comm_rank(), global_state);
  /* prevents race condition of sending next phase while
     stragglers still receiving in previous phase */
  mpi_barrier(global_mpi);
  o = &global_out;
  /* the heart: non-blocking consensus algorithm */
  for (i = 0; i < o->n; ++i)
    o->m[i].r = mpi_issend(global_mpi,
        o->m[i].data, o->m[i].s.n, o->m[i].peer);
  while (!out_done())
    try_recv();
  free_msgs(&global_out);
  ibarrier_begin(global_ibarrier_mpi);
  while (!ibarrier_done())
    try_recv();
  /* end non-blocking consensus algorithm */
  /* sort for parallel determinism */
  sort_msgs(&global_in);
  global_state = UNPACK;
  global_idx = 0;
}
```

Algorithm 3 Luby's Maximal Independent Set algorithm

```
function LUBY(Graph $G$)
    Initialize independent set $I \leftarrow \Phi$
    Initialize active vertices $V' \leftarrow V$, edges $E' \leftarrow E$
    while $V' \neq \Phi$ do
        choose random integers $\pi(i)$ for $i \in V'$
        $I' \leftarrow \{i \in V' | \pi(i) < \min\{\pi(j) | (i,j) \in E'\}\}$
        $I \leftarrow I \cup I'$
        $Y \leftarrow I' \cup \{i \in V' | (i,j) \in E', j \in V'\}$
        remove $Y$ from $V'$ and associated edges from $E'$
    end while
    return $I$
end function
```

near-constant number of calls to Luby at each data point, so the runtime should be $\Theta(\log(P))$, which would be a straight line in this plot.

5.2. Point Cloud Bisection. When partitioning a dataset such as a mesh, one can use a Recursive Inertial Bisection (RIB) algorithm. RIB really operates on point clouds, i.e. a set of points in 3D space where each point represents some simulation object. A cutting plane is computed, which we call the median plane (`mp`). One side

Listing 2
LUBY_MIS code

```
int luby_mis(unsigned nneigh, int const neigh[], int in_V)
{
  mersenne_twister_seed((unsigned)(comm_rank() + 1));
  int in_I = 0;
  int in_Vp = in_V;
  while (mpi_max_int(comm_mpi(), in_Vp)) {
    unsigned pi = mersenne_twister();
    for (unsigned i = 0; i < nneigh; ++i) {
      COMM_PACK(in_Vp, neigh[i]);
      COMM_PACK(pi, neigh[i]);
    }
    comm_exch();
    int is_min = 1;
    while (comm_recv()) {
      int neigh_in_Vp;
      COMM_UNPACK(neigh_in_Vp);
      int neighpi;
      COMM_UNPACK(neighpi);
      if (neigh_in_Vp && (!(pi < neighpi)))
        is_min = 0;
    }
    int in_Ip = in_Vp && is_min;
    in_I = in_I || in_Ip;
    for (unsigned i = 0; i < nneigh; ++i)
      COMM_PACK(in_Ip, neigh[i]);
    comm_exch();
    int in_Y = in_Ip;
    while (comm_recv()) {
      int neigh_in_Ip;
      COMM_UNPACK(neigh_in_Ip);
      if (in_Vp && neigh_in_Ip)
        in_Y = 1;
    }
    in_Vp = in_Vp && (!in_Y);
  }
  return in_I;
}
```

of the cutting plane forms a half-space, and points are either “in” this half-space or “out” of it. We omit the cutting plane computation here because there are many variations of this algorithm, some with more sophisticated decision methods, but all of them require the data bisection operation we discuss [14].

The communication-intensive part of recursive bisection is sending all points inside the space to one half of the ranks, and all the points outside the space to the other half of the ranks. Initially the points are distributed over the ranks in a way that places roughly the same number of points per rank, but there are no real guarantees about whether the points on a rank are on one side or the other of the plane.

The way we bisect points is by first establishing a global numbering for the points on the left, and likewise for the points on the right. This can be done by using a parallel

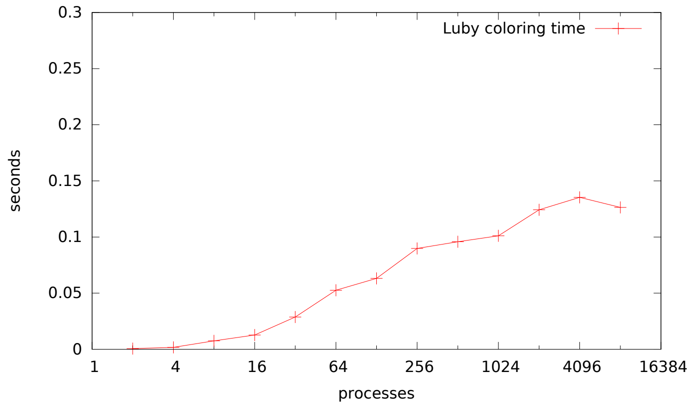

FIG. 5.1. *Time to color mesh parts*

prefix sum, also known as an exclusive scan. We then distribute the points in the half-space to the first half of the ranks by simple division, giving `(tin / ranks_in)` points to each rank, where `tin` is the total number of points in the half-space and `ranks_in` is half the number of ranks. The remainder from this division is given to the last "in" rank, and the whole process is repeated for the ranks and points which are "out" of the half-space.

Listing 3 contains the code for bisecting a point cloud, with somewhat compact formatting such that it fits on one page. The function modifies the arguments `n`, `o`, and `rc` which contain the number of points on-rank, the point coordinates array, and the identifier array, respectively. The identifiers relate a point in space back to the simulation object it originally represents, such that the code can output a useful description of where simulation objects should go.

The actual process of sending the points to their respective destinations is dwarfed by the integer arithmetic to compute the new distribution of points across the ranks, which speaks in part to the ease of use of our API. The real benefit of using the exchange API here is that there is no *a priori* knowledge of which ranks will receive points from which, making this very much a dynamic exchange. The fact that we actually compute the number of points arriving to each part (using $\Theta(\log(P))$ runtime algorithms like scans and reductions) means that we could detect termination by counting points, but we would still need the rest of the exchange system.

In our scaling study, each mesh element is converted into a point and given to an RIB algorithm to partition. The bisection function described here is run $\lceil \log(P) \rceil$ times to complete a partitioning, and we expect roughly constant runtime for each call to `bisect`. Figure 5.2 the runtime of the full RIB partitioning as a function of the number of processes. Once again, we expect runtime to be $\Theta(\log(P))$, which is a straight line.

It is important to note that we do not time the movement of actual mesh elements, this is only timing the movement of "point" objects by the RIB algorithm. The points are coordinates and a reference to the element they represent, so that the elements can be moved just once based on the final partitioning of points.

6. Conclusions. We have presented a C API for performing Dynamic Sparse Exchanges on massively parallel architectures, and included both theoretical and practical evidence that such an interface upholds code quality and program scalability.

Listing 3
BISECT code

```
static void bisect(unsigned* n, point** o, rcopy** rc, plane mp) {
  unsigned nn, i; unsigned long quo, rem, dest_i; int dest_rank;
  unsigned rank = comm_rank(); unsigned size = comm_size();
  unsigned pn = *n; point* po = *o; rcopy* prc = *rc;
  unsigned lin = count_local_in(pn, po, mp);
  unsigned lout = pn - lin;
  unsigned long tin = mpi_add_ulong(comm_mpi(), lin);
  unsigned long tout = mpi_add_ulong(comm_mpi(), lout);
  unsigned ranks_in = size / 2;
  int rank_is_in = (rank < ranks_in);
  unsigned ranks_out = size - ranks_in;
  if (rank_is_in) {
    quo = tin / ranks_in; rem = tin % ranks_in;
    nn = (rank == ranks_in - 1) ? quo + rem : quo;
  } else {
    quo = tout / ranks_out; rem = tout % ranks_out;
    nn = (rank == size - 1) ? quo + rem : quo;
  }
  point* no = malloc(sizeof(point) * nn);
  rcopy* nrc = malloc(sizeof(rcopy) * nn);
  unsigned long in_i = mpi_exscan_ulong(comm_mpi(), lin);
  unsigned long out_i = mpi_exscan_ulong(comm_mpi(), lout);
  for (i = 0; i < pn; ++i) {
    if (plane_has(mp, po[i])) { /* is point in half space ? */
      dest_i = in_i++;
      dest_rank = MIN((dest_i / quo), (ranks_in - 1));
    } else {
      dest_i = out_i++;
      dest_rank = MIN((dest_i / quo + ranks_in), (size - 1));
    }
    COMM_PACK(dest_i, dest_rank);
    COMM_PACK(po[i], dest_rank);
    COMM_PACK(prc[i], dest_rank);
  }
  comm_exch();
  while (comm_recv()) {
    COMM_UNPACK(dest_i);
    if (rank_is_in)
      i = dest_i - (rank * quo);
    else
      i = dest_i - ((rank - ranks_in) * quo);
    COMM_UNPACK(no[i]);
    COMM_UNPACK(nrc[i]);
  }
  free(po); free(prc);
  *n = nn; *o = no; *rc = nrc;
}
```

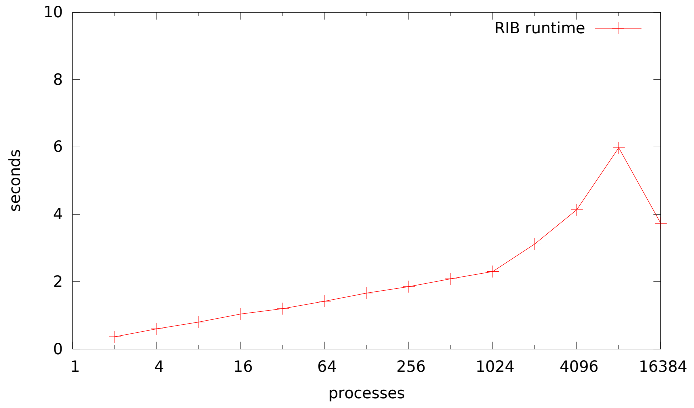

FIG. 5.2. *Time to fully partition point cloud*

Developers of non-trivial MPI programs may consider using our reference implementation directly, which is available at `http://github.com/ibaned/comm`. Our work can more likely be used as a guideline for other APIs, and if nothing else developers should know about the availability of these more scalable algorithms for non-trivial communication.

Acknowledgments. The author would like to thank Daniel Zaide, Cameron Smith, Mark Shephard, and Chris Carothers for providing feedback on early drafts.

REFERENCES

[1] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, and Hong Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2015.

[2] E. G. Boman, U. V. Catalyurek, C. Chevalier, and K. D. Devine. The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering, and coloring. *Scientific Programming*, 20(2):129–150, 2012.

[3] H Carter Edwards, Alan B Williams, Gregory D Sjaardema, David G Baur, and William K Cochran. Sierra toolkit computational mesh conceptual model. *Sandia National Laboratories SAND Series, SAND*, 1192:2010, 2010.

[4] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, 09 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[5] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 26(1):47–56, 2004.

[6] Michael A Heroux, Roscoe A Bartlett, Vicki E Howle, Robert J Hoekstra, Jonathan J Hu, Tamara G Kolda, Richard B Lehoucq, Kevin R Long, Roger P Pawlowski, Eric T Phipps, et al. An overview of the trilinos project. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):397–423, 2005.

[7] Torsten Hoefler and Timo Schneider. Optimization principles for collective neighborhood communications. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10. IEEE, 2012.

[8] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *ACM Sigplan Notices*, 45(5):159–168, 2010.

[9] Torsten Hoefler and Jesper Larsson Träff. Sparse collective operations for mpi. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[10] Daniel A Ibanez, E Seegyoung Seol, Cameron W Smith, and Mark S Shephard. PUMI: Parallel

unstructured mesh infrastructure. *ACM Transactions on Mathematical Software (submitted)*, 2015.

[11] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, pages 1–10. ACM, 1985.

[12] Friedemann Mattern. Virtual time and global states of distributed systems. *Parallel and Distributed Algorithms*, 1(23):215–226, 1989.

[13] Aleksandr Ovcharenko, Daniel Ibanez, Fabien Delalondre, Onkar Sahni, Kenneth E Jansen, Christopher D Carothers, and Mark S Shephard. Neighborhood communication paradigm to increase scalability in large-scale dynamic scientific applications. *Parallel Computing*, 38(3):140–156, 2012.

[14] Horst D Simon. Partitioning of unstructured problems for parallel processing. *Computing Systems in Engineering*, 2(2):135–148, 1991.