Neighborhood Communication Paradigm to Increase Scalability in Large-Scale Dynamic Scientific Applications

Aleksandr Ovcharenko[a], Daniel Ibanez[a], Fabien Delalondre[a], Onkar Sahni[a], Kenneth E. Jansen[b], Christopher D. Carothers[c] and Mark S. Shephard[a]

[a] Scientific Computation Research Center (SCOREC), Rensselaer Polytechnic Institute, 110 8th St, Troy, NY 12180, USA
email: {shurik, dibanez, delalf, osahni, shephard}@scorec.rpi.edu; phone: +1-518-276-6795

[b] Department of Aerospace Engineering Sciences, University of Colorado at Boulder, Boulder, CO 80309, USA
email: jansenke@colorado.edu; phone: +1-303- 492-4359

[c] Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th St., Troy, NY 12180, USA
email: chrisc@cs.rpi.edu; phone: +1-518-276-2930

Abstract

This paper introduces a general-purpose communication package built on top of MPI which is aimed at improving inter-processor communications independently of the supercomputer architecture being considered. The package is developed to support parallel applications that rely on computation characterized by large number of messages of various sizes, often small, that are focused within processor neighborhoods. In some cases, such as solvers having static mesh partitions, the number and size of messages are known *a priori.* However, in other cases such as mesh adaptation, the messages evolve and vary in number and size and include the dynamic movement of partition objects. The current package provides a utility for dynamic applications based on two key attributes that are: (i) explicit consideration of the neighborhood communication pattern to avoid many-to-many calls and also to reduce the number of collective calls to a minimum, and (ii) use of non-blocking MPI functions along with message packing to manage message flow control and reduce the number and time of communication calls. The test application demonstrated is parallel unstructured mesh adaptation. Results on IBM Blue Gene/P and Cray XE6 computers show that the use of neighborhood-based communication control leads to scalable results when executing generally imbalanced mesh adaptation runs.

Keywords

Asynchronous communication; MPI; Dynamic data migration; Parallel algorithms; Overlapping communication and computation

1. Introduction

A number of parallel computing applications devote substantial execution time in data communication between processors, which leads to degradation of parallel performance, especially on very large processor counts. In such applications load imbalance coupled with global communication is a key factor in loss of scalability. Moreover, exchanging a large number of small messages magnifies latency costs. One approach to reduce communication cost is to limit the number of collective calls. In a majority of scientific applications this is possible by effectively accounting for the fact that a substantial portion of the communication is within neighborhoods of processors.

Consider applications with the distributed data where computation and decision making on each processor depends on the information available on another one. In order to provide the locally missing data, communication with possible data movement is required. Inter-processor decision making and data migration is based on the knowledge of the initial communication links (neighbors) between the set of processors requesting and receiving the data. Thus, it is possible to localize data exchange to only those processors within the neighborhood. A property of many of these computations is that as the total number of processors increases, the number of neighbors any processor will exchange local data with becomes bounded by a constant independent of the total number of processors.

Maintaining balance of workload per processor is an obvious initial requirement for scalable parallel algorithms. There are many applications where the workload associated with the individual processors is either initially out of balance, or becomes unbalanced due to dynamic changes in the structure of the computation. To effectively continue the computation these applications must perform a substantial number of communications and determine how to improve load balance, typically by moving data between processors using appropriate migration routines. Data migration is dominated by irregular communications of (small) messages. Fast process migration and execution recovery while maximizing the task balance are critical in such applications, for example, parallel discrete-event simulation [1], distributed virtual environment [2], trading models [3], divide and conquer decomposition algorithms [4], dynamic load balancing [5, 6], adaptive mesh modification [7, 8, 9], etc. Another characteristic of applications with dynamic processes is the fact that much of the communication and migration is local in the sense that one processor does most, or all, of its communication to a set of neighboring processors.

As an example of operations of the type just indicated, consider adaptive mesh modifications [7] on unstructured meshes that include mesh coarsening and entity swapping. The parallel algorithms for the execution of these operations require the migration of mesh entities between neighboring processors [10, 11]. Figure 1 illustrates the basics of mesh migration in a simple two-dimensional setting with three parts where the elements shown are to be migrated among processors. The picture on the left represents an initial mesh with solid dots marking mesh entities which are requested to be migrated to P0, while open circles indicate entities to be migrated to P1. The picture on the right shows the mesh distribution after migration.

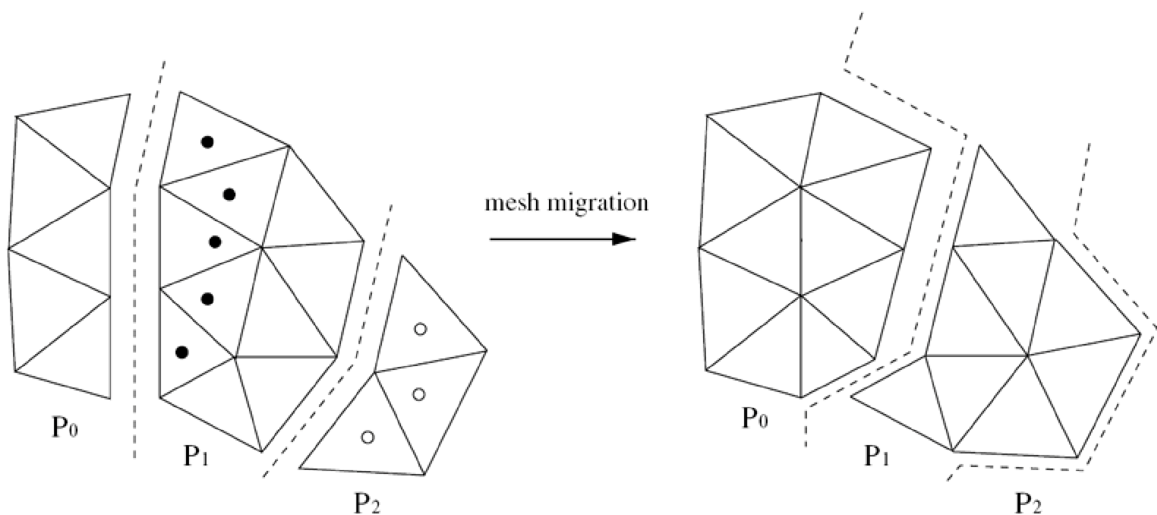

Figure 1. A 2D illustration of parallel mesh migration. Dashed lines indicate inter-part boundaries. Solid dots indicate mesh entities to be on P0 after migration. Circles indicate entities to be migrated to P1.

2. Related work

Previous work has investigated various techniques to improve point-to-point communication performance. One important technique for extracting better application performance is overlapping communication with computation whenever possible to help hide communication delays [12, 13]. This approach works well irrespective of network topology since the major benefit of potential overlap is provided with computation which is independent of pending messages. The use of MPI [14] non-blocking functions increases hardware parallelism by supporting communication and computation overlap.

One-sided communication [15] also supports overlap and lower overhead. It can avoid the protocol overhead in additional memory copying, matching of send and receive operations and

remote process dependency. However, with dynamically changing processes and irregular data migration, it is difficult to efficiently compute the remote processor's address space where the data could be modified to synchronize updates such that memory resources are properly managed and not overly allocated. Otherwise, the application might end up in uncontrolled memory growth during a communication step.

The approach to exploit reusable communication patterns in order to not reset them during every communication step has been introduced in [16] and successfully utilized by the Zoltan team [17]. Each processor has some objects to send to other processors, but no processor knows what messages it will receive. The Zoltan's unstructured communication package uses a data structure responsible for the encapsulation of the basic information about the communication operations. This information does not include the types of objects being transferred; only the number and destination are recorded. Thus, the same pattern can be used repeatedly to transfer different types of data as long as the number and destination of objects involved remains the same.

In applications with unstructured and irregular process computation and migration, there can be potential performance degradation due to high latency cost associated with sending a significant number of small messages. To help reduce the number of messages in such cases message packing strategies can be applied. For example, the Autopack communication tool uses methods of packing small messages into larger packages and automatic management of sends and receives [18]. In Autopack, the application code need not be concerned with pending send requests other than to query the communication tool to determine if they have completed or to determine the number of pending requests. Although the package provides a way to determine the number of incoming messages without barrier synchronization, the implementation of the communication tool assumes the communication between all the processors during a specific communication step. Also, Autopack uses blocking receive calls while getting the incoming package.

The dynamic sparse data exchange problem is stated in [19] and emphasizes the need to provide communications within neighborhoods of processors, when possible, to increase scalability of irregular applications at large scale. The scalable sparse communication protocol is described to improve the runtime of a neighborhood data exchange. Although the method utilizes non-blocking collective operations, it uses synchronous mode sends which only complete after the message has been received. Sparse collective operations are also described in [20] where authors suggest an API and describe functionality and a possible design of neighborhood collective operations for the MPI standard. However, the discussion of optimization/schedules makes an assumption that processes arrive at the sparse collective more or less at the same time, which is not suitable for the applications considered in the current study. Additionally, it has been studied that the process arrival patterns for MPI collective operations are usually imbalanced and can dramatically affect the performance even in a simple test case with a perfectly balanced load [21]. Moreover, neighborhoods can be very diverse in number and structure which makes it significantly difficult to optimize scheduling routines for message delivery.

This paper introduces a general-purpose MPI-based communication package, the Inter-Processor Communication Manager (IPComMan), which incorporates features of the tools discussed above and further aims to improve the scalability of data exchange costs by exploiting communications of a local neighborhood for each processor. The basic idea of this architecture-independent package is to keep the message-passing within subdomains when possible, and eliminate all unneeded collective calls. The communication tool takes care of the message flow with a subset of MPI functions and takes advantage of non-blocking functions with asynchronous buffer management behavior from both sender's and receiver's sides. IPComMan automatically tracks the completion and delivery of send and receive requests posted while overlapping communication with computation. The package provides several useful features: i) automatic message packing, ii) management of sends and receives with non-blocking MPI

functions, iii) communication pattern reusability, iv) asynchronous behavior unless the other is specified, and v) support of dynamically changing neighborhoods during communication steps.

The paper is organized as follows. Section 3 describes the design of the tool to take advantage of neighborhood-based communications. Section 4 gives the implementation of IPComMan package. Section 5 presents test comparisons of simple communication routines using MPI and IPComMan obtained on the IBM Blue Gene/P and Cray XE6 systems. Section 6 describes the current application of IPComMan in parallel unstructured mesh adaptation. Section 7 discusses the results obtained with IPComMan on two different architectures.

3. Design

In a number of applications every processor involved in point-to-point communication exchanges data with only a small subset of other processors where the number of processors in the subset is not a function of the total number of processors being used. Each processor in the subset is called a neighbor, and the subdomain constructed from neighbors is called the neighborhood. Each processor has its own neighborhood, which is different from that of its neighbor(s). At the beginning of the communication phase, each processor knows its neighborhood for the messages to be sent. Communication costs can be reduced by the development of algorithms that eliminate all-to-all or many-to-many communication patterns by taking advantage of the fact that communications are limited to neighborhoods.

Neighborhoods can be represented by a graph, where each graph node is a processor and each graph edge is a communication link. Figure 2 shows an example where P0's neighborhood contains P1 and P2, P1's neighborhood consists of P0, and P2's neighbors are P0 and P3.

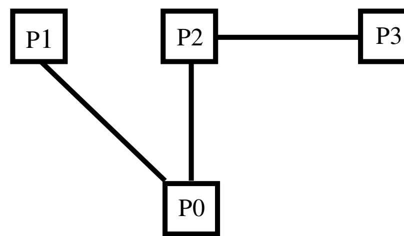

Figure 2. Example of processor neighborhoods based on point-to-point communications.

During the first communication phase the application identifies neighbors for each processor and provides this information to the communication package. With the initial neighborhoods constructed, the task is to execute the communication as efficiently as possible to reduce the communication-to-computation ratio. For these purposes knowing neighbors for each processor serves as a reusable communication pattern. The global collective calls are not needed to verify the status of the communication step, since each processor is aware of its neighbors and can control the message flow within the neighborhood. Moreover, there is no need to reset anything between communication rounds if the types of messages used are the same. If the messages are different, the communication pattern holds the same during each communication step by default and changes only with the application requirements.

In general, neighborhoods stay untouched during a specific communication step, providing data exchange within neighbors only, which number is independent of the total communication links in the domain. This also includes the ability to know the number of messages which have to be received. Since there is no way to know this information ahead of time, this data should be available on the fly to avoid deadlock situations.

There are scenarios when the migrated data impacts the information not available from the neighbors of the specific processor. In order to proceed, the processor with the requested data should join the neighborhood. Thus, the tracking and support of dynamically changing neighborhoods is also required. The application should be aware of such situations and let the communication package know that there might be new communication links created during the

specific communication step. This will switch the tracking of neighbors' communication based on the information about global sends and receives.

In most digital networks transmitting packets, the time to transmit a single packet is equal to some constant (latency) plus a factor (bandwidth) of the number of bits in the packet. Thus the time required to send multiple packets is a factor of the number of packets plus a factor of the total number of bits in those packets. The dependence of the first term on the number of packets is the reason why sending a single large packet is faster than sending multiple small packets, assuming the data is identical. Studies in this work indicate that the latency constants for the two architectures used are very significant, making small messages time-consuming.

Since it can be expensive for parallel applications to send large number of small messages, it is important to group them in appropriate sized buffers and send them out in fewer inter-processor communication steps, reducing latency, without incurring large overhead. There is no need to pack the size of each message if all messages in a buffer are of constant size during a specific communication round. Instead, the communication package should be aware of this situation to know which size of message should be extracted when the buffer is received.

After packing the messages into buffers and sending them to the designated processors it is desirable not to wait for the buffers availability in order to pack the next set of messages on the sender side. At the same time on the receiver side the messages should be extracted as soon as they are available from the first arrived buffer. This scenario hides the communication delays, delivering the application the first available message to proceed with in terms of computation while it keeps receiving other data packages. In order to do this, all the sending and receiving calls must be non-blocking and asynchronous message-passing behavior should be granted such that there is no need to wait for the correspondent receive operation after the buffer is sent.

All considerations given above are accounted for the implementation of the communication package described in the next section.

4. Implementation

In order to build an architecture independent package, IPComMan is written on top of MPI functions. It provides the necessary API to applications in order to utilize the burden of message passing calls which are wrapped into fewer ones. IPComMan is a portable interface which works with any MPI implementation. It is assumed that MPI's operations are well optimized for the underlying network on a specific machine.

Based on the design aspects, there are several issues to address the implementation of the communication package. The major ones include: i) efficient buffer management considering automatic message packing mechanism and memory optimization due to asynchronous non-blocking communication, ii) a different communication behavior for a fixed neighborhood scenario as compared to a dynamically changing one, and iii) communication optimization and overlapping with computation.

4.1. Buffer Memory Management

IPComMan assembles messages in pre-allocated buffers for each destination, and sends each package out when its buffer size is reached. Upon arrival, the package is unpacked, and individual messages are extracted. This allows the user code to be written in a natural style while achieving higher performance without knowing anything about message packing or the MPI interface.

One might be interested in why there is a need to pack and unpack messages as it seems to enforce an additional copy on the application side. MPI is designed to avoid this copy by offering a mechanism to manipulate data directly from the application buffers that don't have to be contiguous. Derived MPI datatypes allow applications to avoid explicit packing and unpacking. The application specifies the layout of the data to be sent or received, and the communication library directly accesses a noncontiguous buffer [14]. However, in certain types of applications the data required for communication is scattered between several data structures

and some computational complexity is required to obtain the desired information in one place. Moreover, multiple functions on one processor may request the same data, and buffering allows duplicate messages to be sent with less communication overhead while eliminating the cost of checking for data duplication.

IPComMan takes care of memory allocation for both sending and receiving buffers, and manages the ones that can be reused without additional allocation. The communication package stores messages going to the same processor in contiguous memory. Thus, when sending or receiving a buffer, no additional memory copying is needed. To avoid copying messages to the buffer during the send procedures, the message to be filled is given to the application as a pointer to the part of the buffer where the message begins, based on the amount of the message size requested. The application is responsible for not exceeding the message size it requested when putting the data in the buffer. At the same time, if the message size requested results in the buffer overflow, IPComMan sends the buffer to its designated processor and based on the buffer status either reuses it or allocates another one to fit the message.

The user may specify whether the size of each message is constant or arbitrary during a specific communication step. The fixed message size is taken by the package and used while extracting the messages. The arbitrary message size is put together with every message into the buffer to correctly unpack the messages upon arrival. There is no specific pattern the application should follow in order to be able to send messages of arbitrary size. IPComMan treats each message as a separate instance and decides how to correctly manage one in a sending buffer based on its logic. Figure 3 illustrates the buffer view with a fixed message size and one with arbitrary size.

Figure 3. Buffer view with different messages inside: **(a)** message size is fixed, **(b)** message size is arbitrary. NM is the number of messages in the buffer, NB is the total number of buffers put in the last buffer to be sent, *msg* denotes a message body and *msize* stands for its size.

All message send and receive functions are non-blocking. When the package is sent (using *MPI_Isend()*) and another buffer is requested for the next set of messages, the already sent buffers are checked for their requests (using *MPI_Request()*). If any of them is satisfied, it is possible to reuse the correspondent buffer to fill it with new messages rather than allocating another one. If none of the buffers were completely sent, a new buffer is allocated, increasing the total number of packages, and the message is placed inside the allocated buffer. After several allocations, an optimal ring of buffers is formed, where, after the package is sent, there is always at least one buffer available for reuse.

The same philosophy is used on the receiving side. When a new package arrives, the processor's correspondent buffers are checked for the one from which all the messages were extracted. If there is such a buffer, it is overwritten by the new package. If all the arrived packages were not processed yet, another buffer is allocated for receiving the newly arrived one. It may take some time for the newly arrived package to be completely received. With non-blocking receive operations (using *MPI_Irecv()*) the messages continue to be extracted from fully available buffers only, hiding the communication time required to completely receive a buffer.

4.2. Communication Paradigm and Termination Detection

There are two modes of operation of IPComMan during a step of communication: fixed neighborhoods and dynamically changing neighborhoods. At the beginning of each communication step the application can specify the termination detection method which allows IPComMan to finish its job correctly and receive all the messages appropriately. They are: i)

neighborhood communication validating the number of sends and receives within neighborhood, and ii) allreduce step to check the global number of sends and receives. The first method, which does not include any collective call, is more efficient and recommended when the neighbors are known and fixed during the communication step, whereas the second one must be used when there is a possibility of changing neighborhoods during a communication step. These termination detection modes are referred to as Neighborhood and Allreduce, respectively.

The default specification of the package implies that neighboring processors are known by the application, provided to IPComMan and do not change within a communication round. For example, in case of a mesh partition neighbors are figured out by the application based on the information about entities on the inter-part boundaries and which processors they are shared with [11]. From that point, IPComMan concentrates on delivering messages between processor and its neighbors only, not touching other processors of the domain. There are no all-to-all and global synchronization calls during each communication round needed when the neighbors are fixed.

There are situations when it is not possible to *a priori* identify all the neighbors for processors, i.e., new neighbors may be encountered during a communication step. In this scenario, the default strategy can not be used and another termination detection mode which requires additional checking and communication needs to be defined. This new mode of operation can be implemented in many different ways. The simplest option, which is the one that has been carried out, is to perform a global synchronization step (using *MPI_Allreduce()*) at the end of the communication round to figure out whether there are any messages to be received from unidentified neighbors.

Consider the communication pattern presented in Figure 4. Processor P0 contains processors P1 and P2 in its neighborhood. P3 has P2 as the only neighbor, but after it has sent all the messages to P2 it finds out that there are some messages to be sent to P0. P3 includes P0 in its list of neighbors and begins to send packages to it. An increment of time before that, say P0 finished sending to and receiving all the messages from P1 and P2, extracted them and proceeded to the next communication step. In that case packages from P3 to P0 would be lost, which will result in incorrect program behavior. To avoid such situations, the application must state the need for an Allreduce termination detection of sends and receives during a communication step. Using this method, one synchronization step at the end of communication round is performed to verify whether the global number of sends and receives match. In case they do not match, IPComMan continues to receive the messages identifying the new neighbors, since all the packages from existent neighbors are already received.

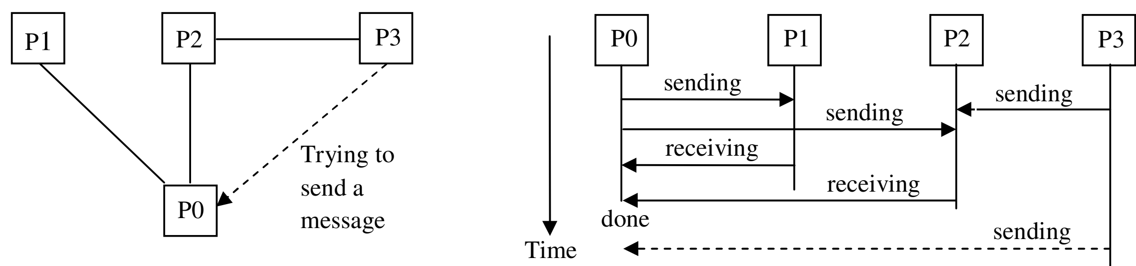

Figure 4. Processor P3 tries to send messages to processor P0, when P0 already received all the messages from its neighbors.

Once encountering a new member, the processor sends a message stating that it is becoming a new neighbor with another processor. The new neighbor receives the message about the additional member of its neighborhood, and becomes aware that it should check the sends and receives, if any, for the additional processor in the current communication round. This mechanism allows figuring out new neighbors before the end of the communication round, with no more than one synchronization step for message validity.

At any point of time the application is able to set up neighbors for a specific processor using a correspondent IPComMan function which has the same functionality as

MPI_Dist_graph_create() [14]. However, IPComMan adds neighbors independently from the communicator and manages them based on neighborhood interactions. The application is responsible for providing correct neighborhoods such that the connection graph is valid in terms of if processor *i* is a neighbor with processor *j*, the opposite also holds true. This will not be an issue if allreduce termination detection is used since IPComMan automatically catches new neighbors and adds them to the correspondent neighborhoods. At the same time, incorrectly set up neighborhoods and neighborhood termination detection will most likely lead to deadlock situations.

4.3. Communication Optimization

Dynamic and irregular computation often results in an unpredictable number of messages communicated among the processors. Using IPComMan, there is no need for the application to additionally verify and send the number of packages to be received at the end of the sending phase. The last message sent from the processor to a neighbor contains the number of buffers expected to be received by the neighbor. The neighbor then continues to receive buffers from the processor until it gets all the designated packages. If the processor has nothing to send to its neighbor during a particular communication round, a constant is sent to the neighbor notifying that the communication between those two processors is done. The control to the application is returned every time the message is extracted from a buffer assuming that the receiving loop is originated from the application side (see the example below). While the application processes the currently available message, it exploits communication to computation overlap which gives time to satisfy requests for other buffers being received. When the processor receives all the messages from its neighbors, IPComMan finishes the communication step by returning the last message, cleaning up all the allocated internally data for buffers and requests, and notifies application about the completion with a special constant.

IPComMan does not follow the message order rule where for a given set of messages sent among the processors, the destination processors may not receive the messages in the same sequence. Instead, to save communication time, the package acts asynchronously and processes the first available buffer. When buffers have different size, which is often the case (especially with the last buffers whose size can be significantly different from other completely filled ones), the smaller buffer request for being received is satisfied earlier, and it can be processed earlier than the first buffer sent from the same processor. If there is a need to exploit message ordering with respect to each neighborhood, IPComMan provides a convenient way to guarantee the same sending and receiving sequence of packages, although it incurs additional memory and performance costs.

Message flow control in IPComMan is mostly based on the interaction of the following MPI functions: *MPI_Iprobe()* checking for the next available package to be received and *MPI_Test()* letting to know if a specific request is satisfied. Each new request posted for the buffer is added to the back of the request queue, such that the earlier added request is the first candidate for being satisfied. There is also a unique mapping between buffers being processed and their requests for both send and receive phases. Once satisfied, the request can be reused for the next operation with the first available buffer, and its correspondent buffer gets in the high priority buffer queue to be rewritten with new messages in the sending step or for message extraction in the receiving phase. This way IPComMan strives to fill in and send buffers without additional allocation, quickly find available buffers to reuse, and at the same time deliver messages to the application from the first available buffer.

To avoid confusion when receiving a message, the information of what the application should to do with the data needs to be embedded in the message itself. When the message is extracted, its size is returned along with the data. If message size is not the key thing to understand how the massage should be processed, the application should put specific information into the message about what should be done with it upon its receival. IPComMan is not responsible for message recognition since it is the application's need, but is rather concerned

about proper packing and fastest ways of message delivery. IPComMan sending and receiving mechanism covers all the suggested neighborhood communication functions described in [20] except for *MPI Neighbor reduce()*. The default behavior of the communication package corresponds to functions *MPI Neighbor_alltoall(), MPI_Neighbor_alltoallv()* and *MPI_Neighbor_alltoallw()* depending on the number and size of messages sent. Since IPComMan manages message flow internally, it provides the application with only one function for sending and one function for receiving of each message. Such operations as neighborhood broadcast, scatter and gather are easily described following the same send and receive pattern. Moreover, the application should never be concerned about the type of neighborhood communication operation since IPComMan automatically tracks neighbors which do not send and / or receive a message during a specific communication round and appropriately notifies those processors that no communication will be involved. Also, the application should not have *a priori* information about how messages follow one after another and try grouping them based on message sizes as IPComMan takes care of message packing and unpacking. Ultimately, the communication package allows the application code to be written in a natural style and achieve higher communication performance without knowing anything about message packing or the MPI interface.

4.4. IPComMan Usage and Example

The initialization of the communication object is as follows:

IPComMan::IPComMan(*comm*, *nbrs*, *tag*, *max_buf_size*),

where *comm* is an MPI communicator (handle), *nbrs* - list of neighbors, containing processors with which a given processor is going to communicate, *tag* – initial tag value for messages, *max_buf_size* – desired buffer size (in bytes) the package has to reach by putting messages into it, before sending to the designated processor.

The communication package is built on top of MPI functions. The basic IPComMan API consists of three major functions: *IPComMan::send()* for sending a message, *IPComMan::finalize_send()* for closing and sending final buffers, and *IPComMan::receive()* for receiving the message. The call to *IPComMan::finalize_send()* function is needed at the end of a communication round after the last message is sent, as the buffer containing the last message includes the total number of outgoing buffers to the designated processor. The destination processor extracts this number, and by evaluating the amount of processed buffers, it knows when it is done with extracting all the messages without further communication. Additional functions of IPComMan are auxiliary ones for tuning up the correct behavior of the package. Among them, for example, *IPComMan::set_Nbrs()* for setting up a neighborhood of a specific processor, *IPComMan::set_fixed_msg_size()* for adjusting a fixed or an arbitrary size of a message and *IPComMan::set_comm_validation()* for choosing a termination detection mode. The example in Figure 5 explains the use of IPComMan with two different communication protocols: one with the fixed message size and neighborhood, and another with messages of arbitrary size and dynamically changing neighborhood during communication step. With the initial neighborhood set for each correspondent processor, the IPComMan object is allocated. The first communication round is provided for the fixed neighborhoods, therefore no collective calls are needed. The application sets the correspondent value for the neighborhood mode of message validation. It also lets the communication package know that all messages within the current communication step will have the same size. After that the application starts filling in and sending the messages to the designated processors. When all sends are done, the function for finalizing sends is called, and each processor starts receiving and processing messages. The communication round ends on a specific processor when a variable responsible for the received message size indicates that all messages have been received. The other type of communication protocol has a similar flow of function calls. The difference is that since the neighborhoods are dynamically changing, the allreduce communication validation mode is used. Also the

application does not send fixed message sizes anymore and has to explicitly request allocation for each message using the desired message size.

```
…
nbrs = {pid 1, pid 2, pid 3};                                // initial set of neighbors for the specific processor
IPComMan CM = new IPComMan(MPI_Comm_Current, nbrs, tag1, max_buf_size);  // initialize IPComMan object
…
// Communication step with fixed neighborhoods where collective calls are not needed
CM->set_comm_validation(IPComMan::Neighbors);      // communication is done when all the messages from neighbors are received
CM->set_fixed_msg_size(msg_size);                    // setting the fixed size of the message
for (some computational work)
{
      msg_send = CM->get_msg_ptr(pid);                // get the pointer to the part of the buffer to place the message
      fill in msg_send;
      CM->send(pid, msg_send);                        // sending the message
}
CM->finalize_send();                                  // finalizing sends
recv_bytes = CM->receive(msg_recv, pid_from);         // get the pointer to the part of the buffer with the received message
                                                      // (mesg_recv) and the number of bytes in the message (recv_bytes)
while (recv_bytes != All_Messages_Received)
{
      process msg_recv;
      recv_bytes = CM->receive(msg_recv, pid_from);   // no collective call is performed while receiving messages
}
…
// Communication step with dynamically changing neighborhoods
CM->set_comm_validation(IPComMan::All_Reduce);    // communication is done when global number of sends and receives are equal
CM->set_tag(tag2);                                    // use different tag from the last communication step
CM->set_fixed_msg_size(0);                            // setting the arbitrary size of the message
for (some computational work)
{
      msg_send = CM->get_msg_ptr(pid, msg_size);   // get the pointer to the part of the buffer to place the message
      fill in msg_send;
      CM->send(pid, msg_send, msg_size);              // sending the message
}
CM->finalize_send();                                  // finalizing sends
recv_bytes = CM->receive(msg_recv, pid_from);         // get the pointer to the part of the buffer with the received message
                                                      // (mesg_recv) and the number of bytes in the message (recv_bytes)
while (recv_bytes != All_Messages_Received)
{
      process msg_recv;
      recv_bytes = CM->receive(msg_recv, pid_from);   // a collective call is performed after all the messages from neighbors are received
}
…
delete CM;
…
```

Figure 5. IPComMan usage example.

5. Performance of simple communications with MPI implementation vs IPComMan

To measure IPComMan performance and understand if it is a good candidate for generally unbalanced simulations, simple communication tests have been studied on two supercomputers with different CPU and network architectures: Shaheen IBM Blue Gene/P [22] at King Abdullah University of Science and Technology and Hopper Cray XE6 [23] at National Energy Research Scientific Computing Center. Blue Gene/P is a 32-bit architecture with four 850-MHz PowerPC 450 CPUs per node, with 3 cache levels, 4 GB DDR2 SDRAM per node. It has five different networks: 3D torus with 425 MBps in each direction, global collective with 850 MBps, global barrier and interrupt, JTAG, and 10 Gigabit Ethernet (optical). Hopper is configured with 2 twelve-core AMD 'MagnyCours' 2.1 GHz processors per node, with separate L3 caches and memory controllers, 32 GB or 64 GB DDR3 SDRAM per node. Hopper has Gemini interconnect with 3D torus topology.

The Blue Gene has a separate high-bandwidth hardware-supported global collective network. Its MPI implementation uses the Deep Computing Messaging Framework (DCMF) [24] as a low-level messaging interface. DCMF takes advantage of the direct memory access (DMA) hardware to offload message passing work and achieve good overlap of computation and

communication. Also, the Blue Gene/P has a 3D torus topology for point-to-point communication and its MPI implementation supports three different protocols depending on the message size [22].

Each dual-socket node of the Cray XE6 is interfaced to the Gemini interconnect through HyperTransport™ 3.0 technology. An internal block transfer engine is available to provide high bandwidth and good overlap of computation and communication for long messages [23]. The 3D torus topology provides powerful bisection and global bandwidth characteristics as well as support for dynamic routing of messages.

All available cores per node were requested on both machines during the runs, and 1024 cores were used to introduce diverse neighborhoods and keep relatively high density of message flow. The simple tests are designed to simulate data exchange during one communication round when each processor sends to and receives multiple small messages of the same size from its neighbors. MPI sparse exchange pattern is implemented with the use of non-blocking send and receive calls for each processor's list of neighbors in a similar way as in [20]. There is a barrier at the end of the communication step for both MPI implementation and IPComMan tests such that the maximum communication time is measured. MPI neighborhood data exchange pattern implementation is shown in Figure 6.

```
...
for (i = 0; i < no of iterations; i++)
{
 for (j=0; j<no of neighbors; j++)
  {
   msg number = j + i*no of neighbors
   MPI_Isend(sendbuf + msg number, 1, MPI_DOUBLE, neighbor[j], tag, comm, send_request + msg number);
   MPI_Irecv(recvbuf + msg number, 1, MPI_DOUBLE, neighbor[j], tag, comm, recv_request + msg number);
  }
}

MPI Waitall(no of neighbors * no of iterations, send_request, MPI STATUSES IGNORE);
MPI Waitall(no of neighbors * no of iterations, recv_request, MPI STATUSES IGNORE);
MPI_Barrier(comm);
...
```

Figure 6. Generic MPI implementation of neighborhood data exchange.

It is usually the case that each supercomputer has its own MPI implementation in order to optimize a specific architecture network performance which can significantly vary across the platforms. For the same reason, each MPI implementation has a different set of limitations on key variables internal to the library, like the number of allocated requests. After receiving errors when running a test case allocating more than 130k requests for the same amount of messages, the authors contacted Hopper technical support. The reply stated that it is possible to modify MPI implementation maximum number for requests, number of buffers, buffer space, etc.; however one should be very careful when doing that since the defaults were chosen for a reason and tuned, thus changing the numbers might lead to an incorrect behavior of a program. It was also stated that "sending small messages is also an inefficient way to send data because of the large start-up cost of sending any message. It is better to send large messages than small messages. Additionally, if your program relies on sending many non-blocking small messages you might find that your program is not very portable across systems." IPComMan's ability to reuse satisfied requests allowed it to handle more sent and received messages. However, to make fair conclusions and in order to understand IPComMan's performance and its impact with different number of neighbors, the tests in this section compare pure MPI calls with IPComMan wrapped MPI function manipulations to manage the data flow of small messages within neighborhoods. Real simulation test cases and their scalability with the use of IPComMan and its full functionality are presented in Section 7.

In the first test each processor sends messages of 8 bytes to two of its neighbors such that it also receives identical messages from them. If the rank of a processor is *mypid*, then the

neighborhood of *mypid* consists of {*mypid-1*, *mypid+1*}. The base test run (*base_run*) repeats 1024 sends and receives for each neighbor. At the end of the test, *MPI_Waitall()* performs termination detection (see Figure 6). The time is measured from the beginning of the data exchange loop until after the call to *MPI_Barrier()* to measure the maximum communication time spent during a tested communication round. IPComMan's buffer is set to 8 bytes such that it does not do any additional buffering. It is expected that IPComMan behaves slower and takes more time to complete the data exchange tasks with the increase of the number of iterations for sending and receiving messages since without buffering it carries out computational overhead in tracking of message delivery. Figure 7 presents results for the simulation described, where time scale is normalized by the base time of first 1024 sends and receives divided by the factor multiple of 1024 iterations (*normalized_time[i] = time[i] / (base_run_time * iterations[i]/1024)*) for the correspondent architecture and implementation. This normalization gives an estimate of how much time it needs for each 1024 iterations with a growing number of total iterations. Ideally, if there were no penalties in the network for number of sends, receives and requests being satisfied, the graph with the described normalized time should give a straight line parallel to the *x* axis (number of sends and receives) where each *normalized_time[i]* equals to 1.

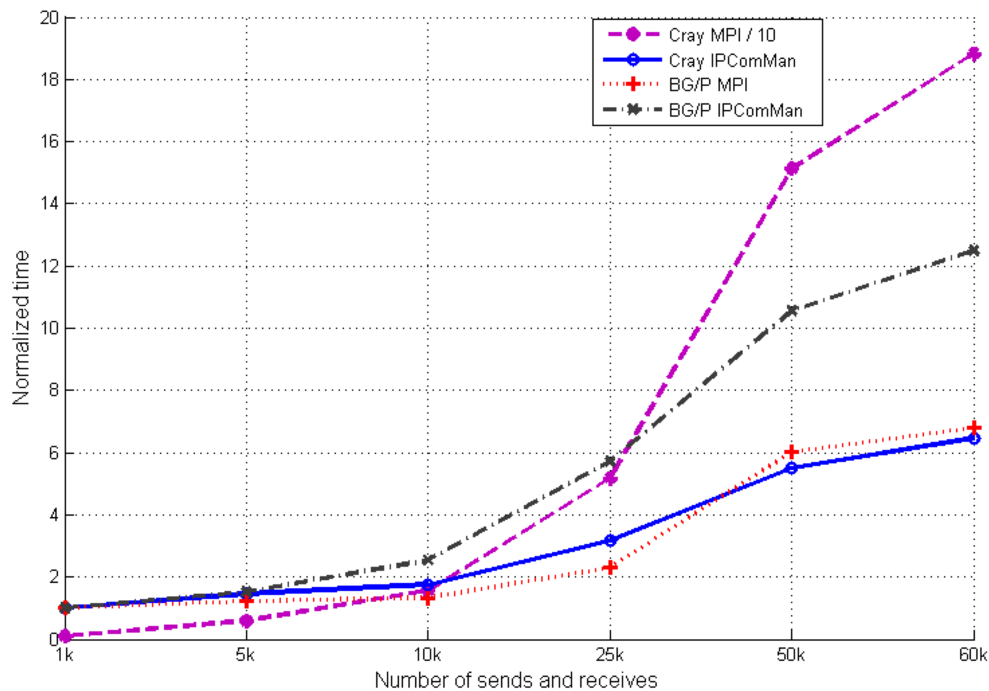

Figure 7. Sending 8 byte messages to neighborhood of each processor consisting of two neighbors. The time is normalized by the run time with 1024 iterations of sends and receives.

IPComMan is 1.5 times faster than pure MPI implementation on Cray in the base test of 1024 send and receive iterations, and 43.5 times faster for 61440 iterations. On the contrary, IPComMan is 2.75 times slower than pure MPI implementation on Blue Gene/P for 1024 iterations and 5 times slower for 61440 iterations. Note that normalized time indicated for Cray pure MPI implementation is 10 times smaller in Figure 7 than the original time indicators such that its performance amplitude does not dominate other results. It can be inferred from the figure that pure MPI implementation on Cray takes about a factor of 190 times more run time for each 1024 send and receive iterations with a total number of 61440 sends and receives. Meanwhile, MPI test on Blue Gene/P has a time penalty factor of 7. IPComMan's performance does not vary across architectures significantly, and the difference between the biggest times is a factor of 2.

The second test case gives an understanding about IPComMan's ability to save communication time with automatic message packing and unpacking support. The number of neighbors is the same as in the previous test and the number of iterations is fixed to 61440 since it resulted in the longest time execution for the test case described above. Figure 8 shows how much time is saved when using message buffering with a correspondent buffer size for each architecture. In other words, time factors in the figure reflect how much faster it takes for the test to be finished with the same amount of sent and received data.

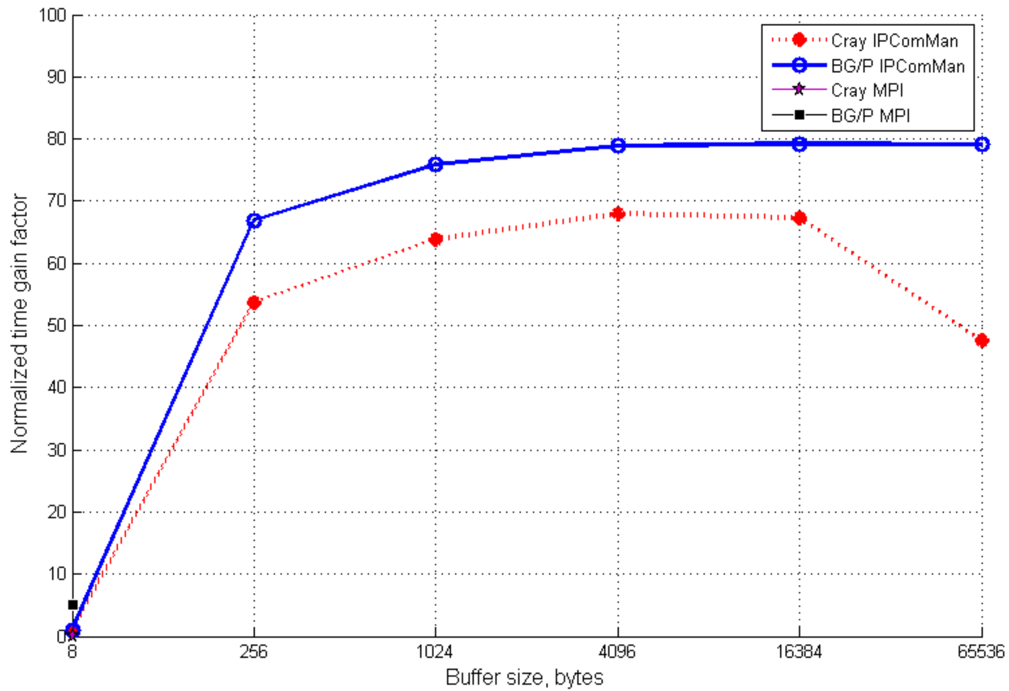

Figure 8. Using IPComMan's buffering ability to understand time savings for the same amount of data being transferred.

It can be found from Figure 8 and previous test description that initial time for pure MPI test implementation on Blue Gene/P is roughly 5 times faster than the one with IPComMan's implementation and buffer size of 8 bytes which is equal to a message size. However, increasing buffer size to 256 bytes gives a factor of 67 times speedup which is roughly 13.5 times faster than pure MPI implementation. Continuing to increase the buffer size up to 64 kbytes does not produce much more speedup and comparing to the initial run with 8 bytes buffer runs 79 times faster. Cray experiences bigger overhead when the buffer size is 64 kbytes, but it should be noted that the simulation time for 64 kbytes buffer takes an average of 0.091 seconds and there is hardly any computation involved to hide communication delays by communication to computation overlap.

IPComMan's message packing mechanism gives a significant time improvement to a simple small data exchange simulation and at the same time keeps the speedup factor consistent across architectures such that the difference in time gain between Blue Gene/P and Cray does not exceed a factor of 2. Additional studying of the communication package buffering mechanism advantage for real applications is done in Section 7 where message sizes are not equal and more than one communication step is involved in the simulation.

The next simulation is designed to test the message flow control in a pseudo-unstructured neighborhood environment such that the communication across the domain is imbalanced. For these purposes a tiling mechanism is used which defines a semi-regular connectivity graph.

Semi-regular definition means that the connectivity within neighborhoods is irregular enough that optimal scheduling is very difficult, while at the same time being regular enough such that each processor can compute its neighborhoods without global communication or reading a file.

The mechanism of tiling is as follows: the connectivity graph is defined as a set of identical tiles connected together in a 2D grid. The tiles and the resulting grid are square for simplicity. The tiles being tested contain 4 x 4 or 16 processors, but in general any square is usable. A single tile is defined by hand, constructing a moderately complex connectivity graph. Authors also define the connectivity between a tile and its adjacent tiles. This definition is small enough to be easily stored within the executable code. Each process then computes the position of its local tile using its processor rank and the total number of processors. The processor can then compute its neighborhood based on the tile definition.

All of the tasks related to the tile-based definition are handled by a single class which stores the tile definition and provides neighborhoods. The neighborhood returned by this class can then be passed directly to IPComMan or used in the pure MPI implementation. Figure 9 shows neighborhoods defined for each processor on a tile diagonal. The rest of the neighborhoods are calculated knowing the tile pattern. Dashed lines in Figure 9 represent a tile boundary. The correspondent processor is depicted as a dotted circle, solid circles define its neighbors, and non-neighbors are represented as hollow circles.

The tile pattern presented results in seven different neighborhood sizes. If the number of processors in the domain is equal to *N*, the distribution of the neighborhoods is the following: *N/4* processors has 2 neighbors, *N/8* processors has 3 neighbors, *N/4* processors has 4 neighbors, *3N/16* processors has 5 neighbors, *N/16* processors has 9 neighbors, *N/16* processors has 14 neighbors, and *N/16* processors has 36 neighbors. The described neighborhood density results in imbalanced communication load which is difficult to optimize scheduling for.

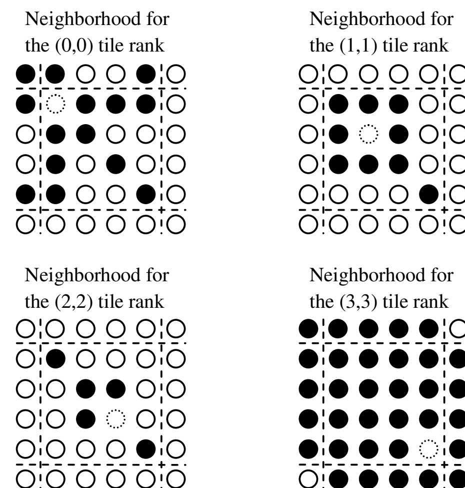

Figure 9. Neighborhoods for the diagonal processors of a tile. Note that 3 neighbors are not shown for the (3,3) tile rank as they are located in further locations of adjacent tiles.

To make fair comparisons and measure IPComMan's ability to improve data flow control with neighborhood connectivity imbalance, its buffer size is set to 8 bytes which is equal to the message size being sent and received. The test case follows the same pattern as the first one presented in this section and runs a number of data exchange iterations. The base test run has a loop over 64 sends and receives for each neighbor in the correspondent neighborhood. The base Figure 10 shows results of a tiling simulation where time scale is normalized the same way it is described in the first test, namely, by the base time of first 64 sends and receives divided by the factor multiple of 64 iterations.

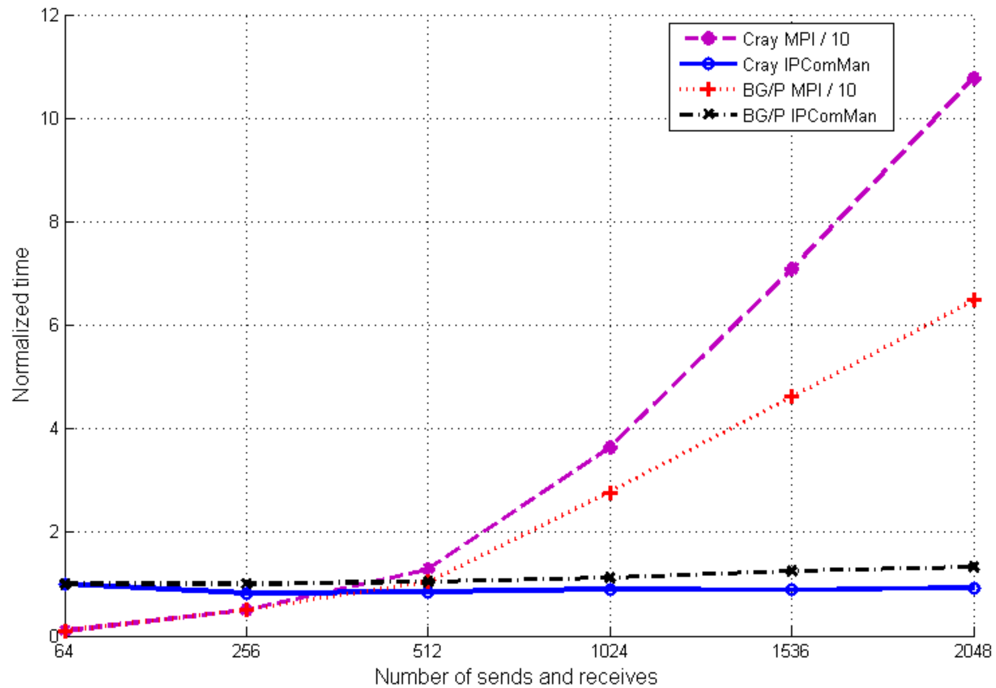

Figure 10. Sending 8 byte messages to neighborhood of each processor consisting of number of neighbors defined by the tiling algorithm. The time is normalized by the run time with 64 iterations of sends and receives.

IPComMan is 1.25 times faster than pure MPI implementation on Cray in the base test of 64 send and receive iterations, and 145 times faster for 2048 iterations. On Blue Gene/P, IPComMan is 2.1 times faster than generic MPI implementation on Blue Gene/P for 64 iterations and 102 times faster for 2048 iterations. Note that normalized time for Cray and Blue Gene/P pure MPI implementation is 10 times smaller in Figure 10 than the original time indicators such that their performance amplitude does not dominate other results. It can be inferred from the plot that generic MPI implementation on Cray takes a factor of 108 more time for each 64 send and receive iterations with a total number of 2048 sends and receives, whereas Blue Gene/P performs with a factor of 65 slower for the same amount of send and receive iterations. Meanwhile, IPComMan's performance incurs negligible penalty on Cray with the increase of transferred data, and experiences a factor of 1.33 time increase on Blue Gene/P with 2048 send and receive iterations. It also should be noted that IPComMan's time increase factors are relatively similar across architectures, and the difference between the biggest of them is a factor of 1.5.

The tests provided in this section show that IPComMan is able to reduce communication time with automatic buffering taking care of smaller messages, and perform a neighborhood optimization with the use of internal data flow control described in Section 4. It appears that the communication package achieves efficient data exchange independently of the architecture chosen. The last test results are a promising indication that IPComMan is able to successfully deal with neighbors which are not structured in any way, completely dictated by the application, with a connectivity graph of a very high complexity such that structured scheduling of data exchange is nearly impossible for communication purposes. The next two sections define in detail problems having highly unstructured neighborhoods and continue discussion about IPComMan's performance on real test scaling studies run on generally imbalanced routines.

6. Mesh Adaptation and Migration

The application of adaptive finite elements for reliable numerical simulations requires they be executed in an automated manner with explicit control of the approximations made [25]. In these applications, meshes are used to discretize the problem domains. Since there are no reliable *a priori* methods to control the approximation errors, adaptive methods must be applied [7, 26, 27, 28]. Adaptive meshing procedures provide a powerful tool for attacking problems such as fluid flows that can develop highly anisotropic solutions and can only be located and resolved through adaptivity [25, 8]. These problems can involve complicated geometries and complex physics resulting in discretizations so large that only large-scale simulation (e.g., petascale) offers the resources required for obtaining a solution in a relevant time frame. Adaptive methods modify the mesh and dramatically increase and/or decrease the number of mesh entities over portions of the problem domain [29, 30] to be sure the mesh has a nearly optimal layout. To run adaptive analysis in parallel, both the analysis step and mesh adaptation steps must be run in parallel on distributed meshes partitioned into parts over multiple processors [7].

The function of mesh adaptation is to convert a given mesh into the desired mesh consistent with the adaptive mesh improvement information provided. A mesh modification based procedure has been developed that accepts a mesh size field which is defined by entity level error estimates or correction indicators evaluated on the previous solution step. The mesh size field is used as an input to initiate mesh refinement and coarsening operations on the mesh to yield a mesh that satisfies the requested mesh size field [10, 8, 29].

Adaptive mesh modification operations increase the number of entities on some processors while reducing the number on other processors. To make mesh entity parts balanced, dynamic load balancing including appropriate mesh migration is required [31]. At the same time, mesh migration can take a significant fraction of the entire mesh adaptation time especially in cases where coarsening and/or swapping are the dominant operations as compared to other mesh modification operations. Thus, in mesh adaptivity software, a good initial partition is not sufficient to assure performance. Since the operations just indicated involve irregularly structured messages of a small size, scalability depends on effectively controlling the underlying message-passing processes.

Figure 11 illustrates the migration process for edge collapse operation. Assume an edge around the vertex indicated with solid dot (on processor P0) is to be collapsed. P0 requests from processors P1-P3 all the faces/triangles (regions in 3D) connected to the circled vertex. The appropriate mesh entities are migrated to P0 which can then collapse an edge around the circled vertex. In mesh modification cases dominated by coarsening, mesh migration operations can take up to three quarters of the entire mesh adaptation time.

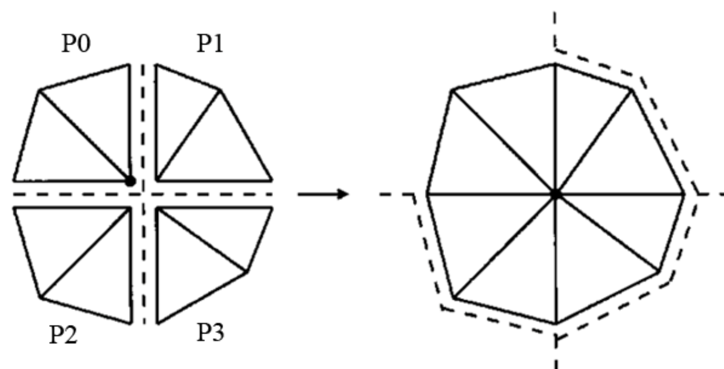

Figure 11. Mesh migration to support distributed coarsening.

During mesh migration the communication can be unbalanced. Some processors may exchange a substantial number of mesh entities, others do not need to communicate at all. Each processor does not have to exchange data with everybody to result in the up-to-date state of the

mesh validity. To quantify in advance the amount of computation and communication associated with mesh migration would consume time equivalent to the execution time of the entire procedure, therefore increasing the load imbalance and communication. Thus efforts in this area need to focus on making the communication as efficient as possible and reducing synchronizations that magnify the load imbalance.

Since mesh migration uses information about shared entities, the maximum number of neighbors for any processor is the union of the processors sharing at least one mesh entity with it. Therefore, a subdomain communication pattern can be applied to minimize or eliminate synchronization steps. As the mesh migration process changes the boundary of parts, processors which were not neighbors before can begin to share mesh entities, thus become neighbors that require communication. This is why the ability of IPComMan to dynamically track and update new neighbors described in Section 4 is a critical functionality.

There are several communication steps within a migration routine. Some of them require entity updates and unification across the inter-part boundary and the Neighborhood method should be used to minimize the communication costs. At the same time, during the entity migration phase new neighbors might be introduced to the neighborhoods, thus the Allreduce verification must be applied to complete all the needed communication correctly and not lose data while there is a possibility of dynamic change in neighborhoods. Non-blocking sends and receives, and the ability of IPComMan to extract the message from the first available buffer helps hide communication delays and tolerate the execution imbalance caused by irregular computation and communication within mesh migration routines.

7. Application results

The mesh modification procedure is applied to two examples to measure the performance of mesh adaptation using the IPComMan communication package. The first example considers a mesh size field that represents a planar shock on cube geometry (CUBE). The second example includes a mesh size field that represents the motion of air bubbles in a uniform flow (BUBBLE). These tests involve substantial coarsening and associated mesh migration.

Figure 12 shows the CUBE test with the initial uniform mesh of 136 million tetrahedra, and the final mesh of 10 million tetrahedra. Figure 8 represents the BUBBLE test, involving movement of five air bubbles by a distance of 1/5th of their radius, with the initial mesh having 165 million tetrahedra, and the final mesh having 188 million tetrahedra.

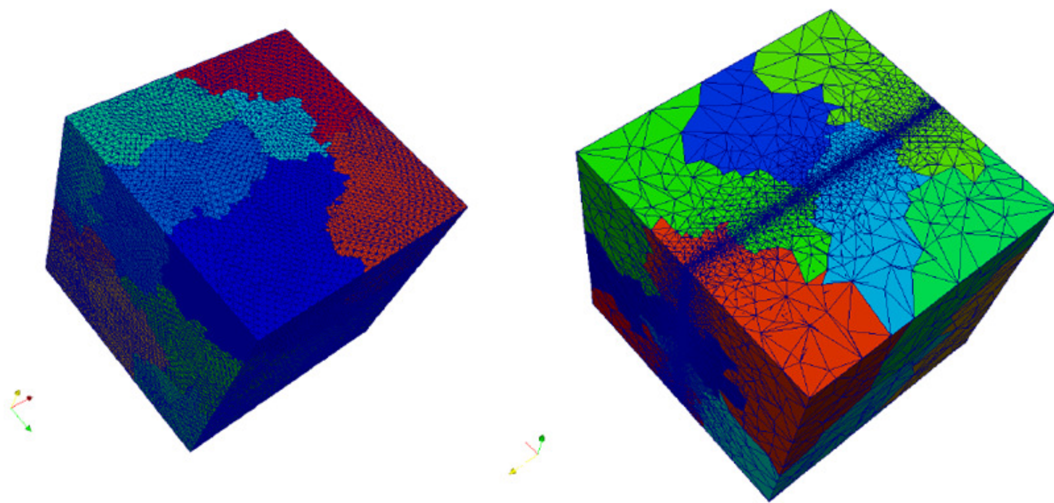

Figure 12. Parallel mesh adaptation on cube geometry.

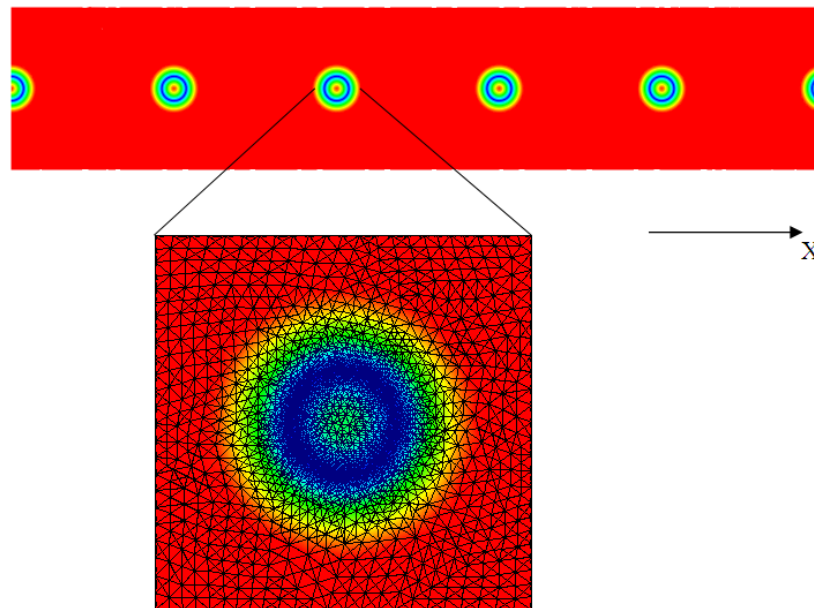

Figure 13. Parallel mesh adaptation for moving air bubbles.

The same set of supercomputers described in Section 5 is used to measure mesh adaptation performance results. All available cores per node were requested on both machines during the runs. For the strong scaling studies, the same buffer size for the message packing (10 kilobytes) is used. The tests were executed on 1,024 to 32,768 processors, where scaling is based on the execution time on 1,024 processors with respect to each supercomputer and defined as ($n_{proc\text{-}base}$ * $time_{base}$) / ($n_{proc\text{-}test}$ * $time_{test}$). Figure 14 demonstrates the scaling of mesh adaptation for the CUBE test case and the corresponding run times are given in Table 1. Figure 15 depicts the scaling for the BUBBLE test case for both architectures with the execution times represented in Table 2.

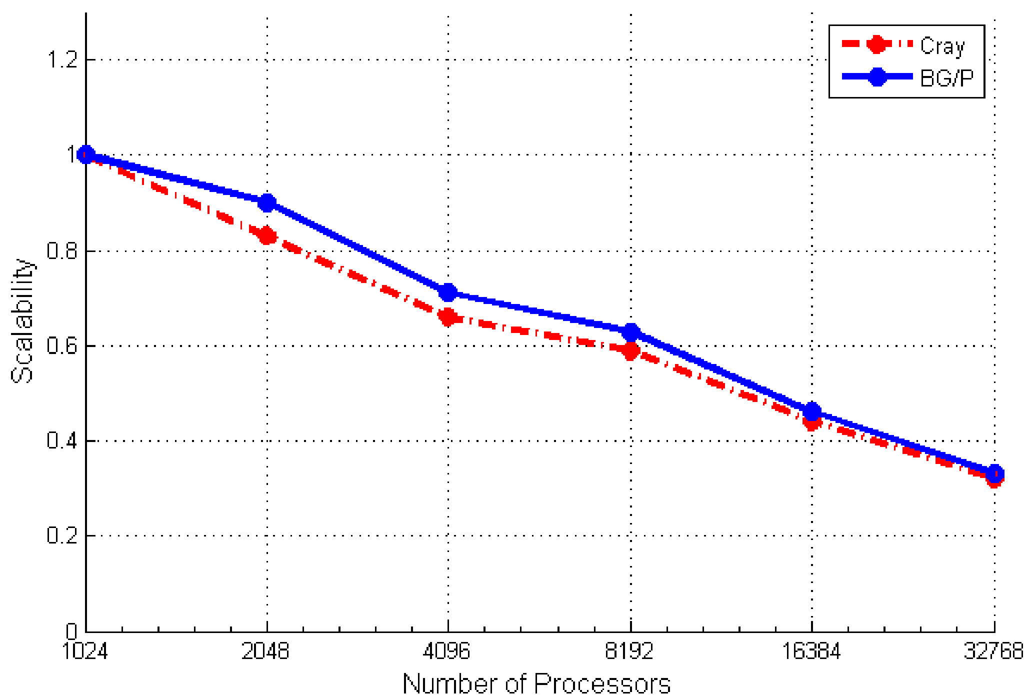

Figure 14. Mesh adaptation scaling results for the CUBE test case.

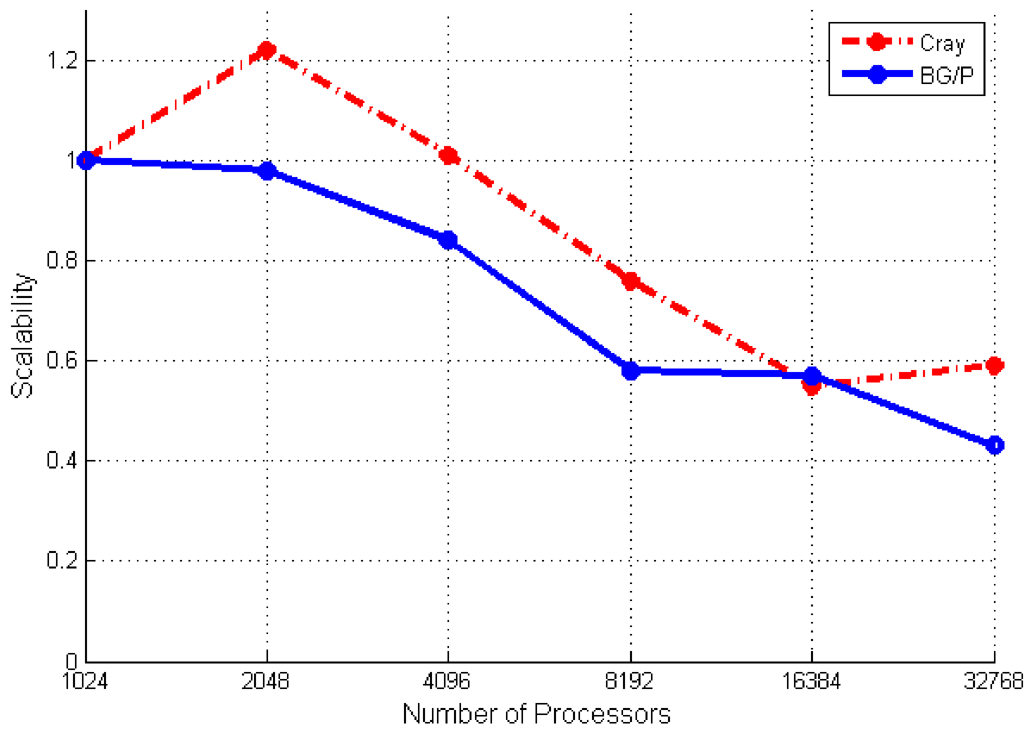

Figure 15. Mesh adaptation scaling results for the BUBBLE test case.

Table 1. Mesh adaptation time and scaling results for the CUBE test case.

Architecture	N/proc	1024	2048	4096	8192	16384	32768
BG/P	Time	470.06	262.57	164.59	93.91	63.36	44.38
	Scaling	1	0.90	0.71	0.63	0.46	0.33
Cray	Time	126.30	76.38	47.55	26.97	17.99	12.52
	Scaling	1	0.83	0.66	0.59	0.44	0.32

Table 2. Mesh adaptation time and scaling results for the BUBBLE test case.

Architecture	N/proc	1024	2048	4096	8192	16384	32768
BG/P	Time	368.22	188.10	109.98	79.36	40.08	26.57
	Scaling	1	0.98	0.84	0.58	0.57	0.43
Cray	Time	138.36	56.59	34.40	22.65	16.82	7.27
	Scaling	1	1.22	1.01	0.76	0.55	0.59

Figure 14 shows that the CUBE test case, which is highly loaded with mesh migration, behaves in a similar way for both considered architectures. IPComMan performs better on Blue Gene/P with smaller processor count, but becomes almost the same with Cray on higher numbers of cores. The Cray was able to perform superlinear scaling for two steps in the BUBBLE test case. This happened due to the fact that the mesh was repartitioned in a way where a larger part of the communication intensive process appeared to be on the same nodes resulting in the faster inter-node data exchange.

To understand the advantage of packing smaller messages into pre-allocated buffers, predictive load balancing routines [32] were tested with different buffer sizes. Predictive load balancing was used to repartition the mesh in the BUBBLE test case such that each mesh part has approximately the same number of entities at the end of mesh adaptation. The rebalancing

procedures resulted in a migration of 60 million entities with the average of 8Mb of data sent and received by each processor. 1,024 cores were used to test the IPComMan's performance with respect to the buffer size since the smaller number of cores results in a bigger amount of data exchange between processors.

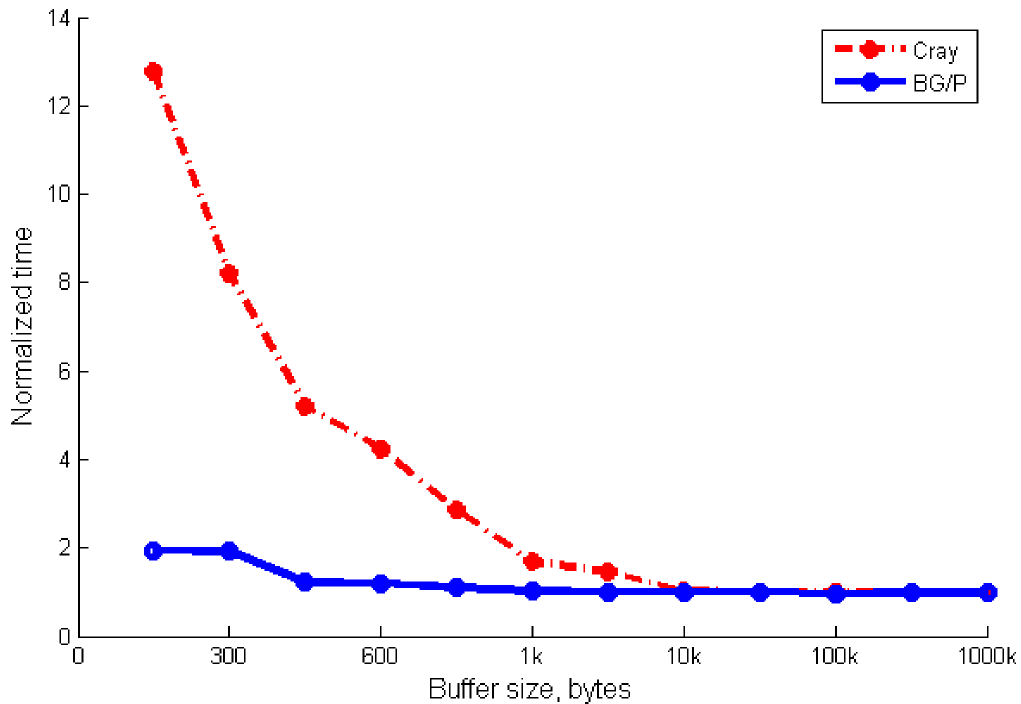

Figure 16. Predictive load balancing routines time normalized by the run time with 10kbytes buffer size.

Figure 16 shows the timings for the predictive load balancing routines applied on the BUBBLE test case with different buffer sizes. The time is normalized by the run time of the test with 10kbytes buffer size for the correspondent architecture. It can be found that the run time decreases significantly on Cray architecture (13 times) as the buffer size increases. The load balancing run times become almost equal on the correspondent supercomputer with the buffer size above 5kbytes. As indicated before, Blue Gene/P and Cray have different networks and message routing protocols. Blue Gene/P has three different protocols to optimize a delivery of smaller messages. However, the ability to pack messages into buffers and control of the buffer size allows IPComMan to unify network features and provide desirable performance on different architectures without being dependent on their message delivery strategy.

Regardless of the architecture, the IPComMan's ability to localize communication to neighborhoods and being independent of the total number of processors, buffer size control and its use of non-blocking functions allows it to provide good results as the number of processors increases, for generally computationally imbalanced procedures. However, since the tests provided address a fixed size problem, with the core count increase the larger percentage of the mesh entities are shared between inter-part boundaries. This leads to the growing number of entities to be migrated during mesh adaptation procedures and more inter-processor communication is needed, whereas the computational work is reduced with mesh parts becoming less loaded. This explains why the scaling for the process does fall with increased processor count. It is also important to note that the mesh adaptation process is not well balanced from the very beginning in terms of the communication load per part and neighborhood. Even though the

number of entities on each part is balanced, the difference of computational work can easily be more than a factor of two.

8. Closing Remarks

The need for IPComMan emerged with the dramatic increase in processor count on today's parallel computers, where the approach relying on the global synchronous communication is not acceptable anymore. IPComMan is a general-purpose MPI-based communication utility aimed at improving inter-processor communications regardless of the architecture by using explicit consideration of the neighborhood communication pattern. The results obtained with the use of the package in mesh adaptation routines indicate that the neighborhood approach together with automatic message packing and non-blocking functions leads to excellent results on two different supercomputers. With a fixed size problem even though the communication load rises substantially on each part with the processor count increase, the ability not to use global collectives when neighborhoods do not change, preserves the program runs from significant growth in execution time.

It is shown that packing of smaller messages in buffers is essential in order to optimize the communication layer features on different supercomputer architectures. This is beneficial in both reducing the number of packages sent and resulting in communication and computation overlap when performing non-blocking function calls. IPComMan takes full control of message receiving while providing the first extracted message from the buffer to the application.

Additional efforts on the scalability of specific applications using IPComMan are desired. The motivation is to continue to investigate the communication and computation overlap tradeoff from the communication package standpoint. This includes automatic change of buffer size for messages individually on each processor, according to the architecture chosen, and a processor loading in terms of number of messages for each communication step. Another significant aspect for improving the scalability is making IPComMan completely free of the Allreduce method, which is applied when the dynamic change of neighborhoods take place. At a minimum, *MPI_Iallreduce()* [33] should be used to take a full advantage of non-blocking MPI functions. However the scenario of changing neighbors should be taken care of without putting a burden on the application side such that it decides which method of message control should be used. At the same time, the free of Allreduce neighborhood approach must be designed in such a way that it does not introduce redundant neighbors while trying to catch new processors coming to the neighborhood. The neighborhoods should be limited to the number of actively communicating processors only such that they are independent of the total number of processors within the domain.

9. IPComMan source code availability

IPComMan is an open source software and its code is available for download under the following link: http://redmine.scorec.rpi.edu/anonsvn/ipcomman/trunk. The communication package webpage is located at the following address: http://www.scorec.rpi.edu/IPComMan. IPComMan has a BSD license for academics, US government labs and researchers. Commercial use requires a license that can be discussed by contacting Scientific Computation Research Center office at office@scorec.rpi.edu.

10. Acknowledgments

This work is supported by the National Science Foundation under Grant No. 0749152, and by the U.S. Department of Energy under DOE Grant No. DE-FC02-06ER25769. Computing support is provided by King Abdullah University of Science and Technology for granting access to the Blue Gene/P and by National Energy Research Scientific Computing Center for granting access to the Cray XE6 supercomputers.

The authors would like to thank Professor William D. Gropp for the valuable comments and clarifications towards the improvement of this study.

References

[1] R.M. Fujimoto, Parallel and Distributed Simulation Systems, John Wiley and Sons, Inc., 2000.

[2] D. Lee, M. Lim, S. Han, K. Lee, ATLAS: a scalable network framework for distributed virtual environments, Presence: Teleoperators and Virtual Environments 16 (2007) 125-156.

[3] S. Thomas, Load balancing in CORBA: a survey of concepts, patterns, and techniques, J. Supercomput. 15 (2000) 141-161.

[4] P. Czarnul, Programming, tuning and automatic parallelization of irregular divide-and-conquer applications in DAMPVM/DAC, Int. J. High. Perform. Comput. Appl., 17 (2003) 77-93.

[5] M. Zhou, O. Sahni, T. Xie, M.S. Shephard, K.E. Jansen, Unstructured mesh partition improvement for implicit finite element at extreme scale, In preparation.

[6] K. Barker, A. Chernikov, N. Chrisochoides, and K. Pingali, A load balancing framework for adaptive and asynchronous applications, IEEE T. Parall. Distr., 15 (2004) 183-192.

[7] M.S. Shephard, J.E. Flaherty, C.L. Bottasso, H.L. de Cougny, C. Ozturan, M.L. Simone, Parallel automated adaptive analysis, Parallel Comput., 23 (1997) 1327–1347.

[8] X. Li, M.S. Shephard, and M.W. Beall, 3D anisotropic mesh adaptation by mesh modifications, Comp. Meth. Appl. Mech. Eng., 194 (2005): 4915-4950.

[9] R.H. Stogner, G.F. Carey, B.T. Murray, Approximation of Cahn-Hilliard diffuse interface models using parallel adaptive mesh refinement and coarsening with C1 elements, Int. J. Numer. Methods Eng., 76 (2008): 636–661.

[10] H.L. de Cougny, M.S. Shephard, Parallel refinement and coarsening of tetrahedral meshes, Int. J. Num. Methods Eng., 46 (1999) 1101-1125.

[11] E.S. Seol, M.S. Shephard, Efficient distributed mesh data structure for parallel automated adaptive analysis, Eng. Comput., 22 (2006) 197-213.

[12] V. Subotic, J. C. Sancho, J. Labarta, M. Valero, A simulation framework to automatically analyze the communication-computation overlap in scientific applications, IEEE International Conference on Cluster Computing (2010) 275-283.

[13] T. Hoefler, A. Lumsdaine, Overlapping communication and computation with high level communication routines, In Proc. of the 8th IEEE Symp. on Cluster Computing and the Grid, (2008) 572–577.

[14] MPI: A Message-Passing Interface Standard Version 2.2, MPI Forum (2009).

[15] W. Gropp, E. Lusk, and A. Skjellum, Using MPI-2: Advanced Features of the Message-Passing Interface, 2nd edition, (1999), MIT Press.

[16] R. Ponnusamy, J. Saltz, A. Choudhary, Runtime compilation techniques for data partitioning and communication schedule reuse, In Proceedings of the 1993 ACM/IEEE conference on Supercomputing, (1993) 361-370.

[17] Zoltan Unstructured Communication Utilities, http://www.cs.sandia.gov/Zoltan/ug_html/ug_util_comm.html, last accessed on 09/08/2011.

[18] R. Loy, Autopack user manual, Science Division Argonne National Laboratory, http://www.mcs.anl.gov/research/projects/autopack/autopack_manual.pdf, last accessed on 09/08/2011.

[19] T. Hoefler, C. Siebert, A. Lumsdaine, Scalable communication protocols for dynamic sparse data exchange, PPoPP '10 (2010).

[20] T. Hoefler and J. L. Traeff, Sparse collective operations for MPI, In Proc. of the 23rd IEEE International Parallel & Distributed Processing Symposium, HIPS'09 (2009).

[21] A. Faraj, P. Patarasuk, X. Yuan, A study of process arrival patterns for MPI collective operations, Int. J. Parallel Prog., 36 (2008) 543-570.

[21] IBM guide to using the Blue Gene/P.
http://www.redbooks.ibm.com/abstracts/sg247287.html, last accessed on 09/08/2011.

[22] Cray XE6
http://www.cray.com/Products/XE/CrayXE6System.aspx, last accessed on 09/08/2011.

[23] S. Kumar, G. Dozsa, G. Almasi, D. Chen, P. Heidelberger, M. E. Giampapa, M. Blocksome, A. Faraj, J. Parker, J. Ratterman, B. Smith and C. J. Archer, The deep computing messaging framework: generalized scalable message passing on the blue gene/P supercomputer, ICS'08 (2008).

[14] O. Sahni, J. Muller, K.E. Jansen, M.S. Shephard, C.A. Taylor, Efficient anisotropic adaptive discretization of the cardiovascular system, Comp. Meth. Appl. Mech. Eng., 195 (2006) 5634-5655.

[25] M.S. Shephard, Meshing environment for geometry-based analysis, Int. J. Numer. Meth. Eng., 47 (2000) 169-190.

[26] M.S. Shephard, M.W. Beall, R.M. O'Bara, and B.E. Webster, Toward simulation-based design, Finite Elem. Anal. Des., 40 (2004) 1575-1598.

[27] X. Li, J.F. Remacle, N. Chevaugeon, and M.S. Shephard, Anisotropic mesh gradation control, In 13th International Meshing Roundtable, (2004) 401-412.

[28] F. Alauzet, X. Li, E.S. Seol, M.S. Shephard, Parallel anisotropic 3D mesh adaptation by mesh modification, Eng. Comput., 21 (2006) 247–258.

[29] T.J. Baker, Mesh adaptation strategies for problems in fluid dynamics, Finite Elem. Anal. Des., 25 (1997) 243–273.

[30] M. Dindar, M.S. Shephard, J.E. Flaherty, and K.E. Jansen, Adaptive CFD analysis for rotorcraft applications, Comp. Meth. Appl. Mech. Eng., 189 (2000) 1055-1076.

[31] M. Zhou, T. Xie, S. Seol, M.S. Shephard and O. Sahni and K.E. Jansen, Tools to support mesh adaptation on massively parallel computers, Eng. Comput. (2011), in press.

[33] MPI: A Message-Passing Interface Standard Version 3.0, MPI Forum (2010).

Vitae

Aleksandr Ovcharenko received the BS (2004) in Applied Mathematics and Computer Science from Novosibirsk State Technical University, Russia. His Bachelor's thesis was on data analysis for the vector finite element methods for the 2D case. He received the MS (2006) in Computer Engineering and IT from the University of Ulsan, South Korea. His Master's was on resolution of a front-back confusion problem for the 3D sound localization. In 2007, he began studying toward the PhD degree in Computer Science at Rensselaer Polytechnic Institute. His current research areas include parallel mesh adaptation, parallel processing and distributed mesh database.

Dan Ibanez is an undergraduate student at Rensselaer Polytechnic Institute. He will graduate with a dual degree in Computer Science and Computer Systems Engineering in 2013. Since 2010 he has worked as a research student at Resselaer's Scientific Computation Research Center (SCOREC). He is an intern in the Blue Waters Undergraduate Petascale Institute Program. His work focuses on distributed finite element fields and multiscale engineering simulations.

Dr. Fabien Delalondre is a Senior Research Associate at SCOREC at Rensselaer Polytechnic Institute (RPI) at Troy, NY. Prior to that he was a Postdoctoral Research Associate at SCOREC. He received his PhD in Computational Mechanics in 2008 from Mines ParisTech in Paris (France) and his MSc in Mechanical Engineering in 2003 from the Institute of Engineering Sciences and Technology of Lyon – Ecole Polytechnique Universitaire (ISTIL EPU) in Lyon (France). His research interests include parallel computing, adaptive modeling and multiscale simulation of Engineering and Bioengineering applications.

Dr. Onkar Sahni is currently an Assistant Professor in the Department of Mechanical, Aerospace and Nuclear Engineering (MANE) at Rensselaer Polytechnic Institute (RPI). Prior to his current position he was a research scientist/engineer at the Center for Predictive Engineering and Computational Science (PECOS) located in the Institute for Computational Engineering and Sciences (ICES) at the University of Texas (UT) at Austin. He received his PhD degree from Rensselaer in 2007 and Bachelors degree from Indian Institute of Technology-Bombay (IIT-B) in 2002. He has over 10 years of experience in the field of CFD. He has developed and applied parallel adaptive methods to over 100,000 processor-cores for simulation of fluid flows in various real-world application areas, for example, aerodynamics, cardiovascular flows, two-phase flows.

Professor Kenneth E. Jansen received his Ph.D. in Mechanical Engineering with a minor in Aeronautical Engineering in 1993 under an Office of Naval Research Fellowship. He then joined the Center for Turbulence Research, a joint NASA-Stanford program, where he was awarded a three year post-doctoral research fellowship. In 1996 he became a Professor in the Department of Mechanical, Aerospace and Nuclear Engineering at RPI. Since 2010, he has been a Professor of Aerospace Engineering Sciences at University of Colorado at Boulder. His research interests are computational mechanics with emphasis on fluid dynamics, turbulence theory, simulation, modeling and parallel computing.

Christopher D. Carothers is a Professor at RPI in the Department of Computer Science. His research interests are massively parallel processing, parallel I/O and parallel discrete-event simulation approaches and methods.

Professor Mark S. Shephard is the Samuel A. and Elisabeth C. Johnson, Jr. Professor of Engineering at Rensselaer Polytechnic Institute. He holds appointments in the departments of Mechanical, Aeronautical and Nuclear Engineering; Computer Science; and Civil Engineering.

He is the director of Rensselaer's Scientific Computation Research Center and does research on automated and adaptive analysis methods, parallel adaptive simulation technologies, and multiscale computational methods. He is a fellow and past president of the US Association for Computational Mechanics, a fellow and member of the General Council of the International Association for Computational Mechanics and a fellow of ASME.

List of figures

Figure 1. A 2D illustration of parallel mesh migration. Dashed lines indicate inter-part boundaries. Solid dots indicate mesh entities to be on P0 after migration. Circles indicate entities to be migrated to P1.

Figure 2. Example of processor neighborhoods based on point-to-point communications.

Figure 3. Buffer view with different messages inside: **(a)** message size is fixed, **(b)** message size is arbitrary. NM is the number of messages in the buffer, NB is the total number of buffers put in the last buffer to be sent, *msg* denotes a message body and *msize* stands for its size.

Figure 4. Processor P3 tries to send messages to processor P0, when P0 already received all the messages from its neighbors.

Figure 5. IPComMan usage example.

Figure 6. Generic MPI implementation of neighborhood data exchange.

Figure 7. Sending 8 byte messages to neighborhood of each processor consisting of two neighbors. The time is normalized by the run time with 1024 iterations of sends and receives.

Figure 8. Using IPComMan's buffering ability to understand time savings for the same amount of data being transferred.

Figure 9. Neighborhoods for the diagonal processors of a tile. Note that 3 neighbors are not shown for the (3,3) tile rank as they are located in further locations of adjacent tiles.

Figure 10. Sending 8 byte messages to neighborhood of each processor consisting of number of neighbors defined by the tiling algorithm. The time is normalized by the run time with 64 iterations of sends and receives.

Figure 11. Mesh migration to support distributed coarsening.

Figure 12. Parallel mesh adaptation on cube geometry. (Color on the Web only)

Figure 13. Parallel mesh adaptation for moving air bubbles. (Color on the Web only)

Figure 14. Mesh adaptation scaling results for the CUBE test case. (Color on the Web only)

Figure 15. Mesh adaptation scaling results for the BUBBLE test case. (Color on the Web only)

Figure 16. Predictive load balancing routines time normalized by the run time with 10kbytes buffer size. (Color on the Web only)

List of tables

Table 1. Mesh adaptation time and scaling results for the CUBE test case.

Table 2. Mesh adaptation time and scaling results for the BUBBLE test case.