# Software Topological Message Routing and Aggregation Techniques for Large Scale Parallel Systems

Lukasz Wesolowski

Preliminary Examination Report

## 1 Introduction

Supercomputing networks are designed to minimize message latency. The focus on low latency implies a best effort attempt to deliver each message injected onto the network as soon as possible. Ostensibly, prioritizing message latency is important, and there are numerous examples of applications that benefit from it, but while this latency-centric view of a network seems logical, it often leads to unintended consequences.

It may at first appear surprising that one would find fault with the idea of trying to deliver messages as soon as possible, but if we take a broader view of communication and consider modern world systems for delivery and transportation, we will find that the micro-management of the latency-centric approach is hardly the norm. As an example, consider an urban rail transit system. Traveling by train requires being aware of the schedule and sharing travel space with other passengers. The time to reach the destination may be higher on average than by car, as it involves waiting at the station and changing trains if the destination does not happen to lie along the route of the initially boarded train. Unless one is traveling in a densely populated area, a train may not even be an option. The various restrictions and limitations, which are a source of inconvenience for the passengers, also allow trains to function more efficiently. In many ways a public transit system is an example of a system that deemphasizes latency, that is each individual's travel time, to improve bandwidth, the number of people transported to their destination per unit time.

Compared to the bandwidth-centric approach of a rail system, roads and highways are decidedly latency-centric. Driving a car involves few of the limitations of a public transit system, and under normal conditions leads to significantly shorter travel times. Despite the higher convenience of cars, public transportation remains an indispensable piece of infrastructure in large cities. Allowing large numbers of people to travel using a minimal real estate footprint makes public transit systems a key prerequisite for supporting dense population concentrations of large cities, where office and living space are stacked vertically, while transportation is often limited to a fraction of the space along the two dimensions of the street level. If instead of using public transportation, everyone chose to drive to work in these areas, traffic congestion would increase travel times several fold, and other than the very earliest travelers, most people upon arrival at their destination would find that they have nowhere to park.

As latency-optimized systems, supercomputing networks are designed like highways for the data traveling between the increasingly densely populated computing resources of a large system. This work explores the idea of reorganizing the network in software to act more like the public transportation system in order to significantly improve the efficiency of handling a high volume of communication. Toward this goal, we introduce the Topological Routing and Aggregation Module (TRAM), a software communication system for efficiently handling high volume communication patterns, particularly ones involving a high volume of fine-grained communication.

## 2 Related work

Aggregation and routing of messages over mesh topologies has been explored before in specific contexts. With a simplified performance model and experimental results, [17] demonstrated improved performance of all-to-all and many-to-many personalized communication using 2D and 3D virtual meshes for routing and aggregation. Analysis of the reduction in the number of messages when 2D and 3D virtual topologies are used to aggregate messages for all-to-all personalized communication was presented in [19]. For aggregation of non-collective communication, [19] also described a library which used a two-step routing and aggregation approach within a two dimensional virtual mesh mapped over the processors. The above work did not consider the importance of a good match between virtual and physical network topologies, however. Another example of aggregation and routing of messages using a library is [27], although here aggregation was done by separately buffering the data for every destination and sending buffers that fill up directly. A more recent effort by the same authors utilized virtual topologies to improve communication performance on infiniband clusters [28]. In contrast to our approach, the latter work did not involve matching the virtual topology to the physical network topology, and used a hypercube virtual topology, which on mesh and torus interconnection networks such as on Blue Gene/P will lead to higher overhead and non-minimal routing.

Other efforts have focused on improving performance of specific benchmarks and libraries on systems with topologically assigned job partitions. [9] demonstrated improved performance of the HPC Challenge Random Access benchmark on a Blue Gene/L system. [21] used topological routing and aggregation to improve performance of All-to-All and FFT on the Blue Gene/L.

In our performance analysis we consider the impact of network topology and congestion on communication performance. Other research on communication over mesh and torus topologies which took into account issues of network congestion includes work on MPI collectives [15], [24], topology-aware mapping [5], and network saturation on Blue Gene/P [2]. Our approach to analyzing message aggregation and determining a good message size also shares common elements with work on message strip-mining [14].

Although MPI collectives are typically meticulously optimized, taking advantage of their performance requires adhering to a rigid set of communication patterns. As a result, applications with sparse, irregular, or dynamic communication patterns are typically relegated to using point-to-point communication. Non-traditional forms of collective communication have recently been gaining attention, including work on dynamic sparse data exchange [13] and neigborhood collectives [12]. We believe TRAM can be used to improve performance of both traditional and exotic classes of collectives.

# Preliminary Work and Results

## 3 TRAM

Topological Routing and Aggregation Module is a library for optimization of fine-grained units of communication in parallel applications. The library performs topological routing and aggregation of network communication in the context of a virtual mesh topology comprising the processes involved in the parallel run. Currently, the choice of topology is limited to an N-dimensional mesh, where the number of dimensions and their sizes are specified by the user when initializing an instance of the library.

### 3.1 Background Information

TRAM is implemented as a module of the Charm++ parallel runtime system for distributed object-oriented parallel programming [18]. For simplicity of discussion, the runtime system can be assumed to consist of a process on every core involved in the parallel run. These processes, called Processing Elements (PEs), are globally ranked. Expressing parallelism in this system typically involves creating collections of globally accessible objects called *groups* and *arrays*. Groups map a single runtime-system-managed object to every core in a parallel run. Arrays, on the other hand, may contain an arbitrary number of such objects, which are assigned to physical cores by the runtime system based on predefined or custom mappings. A parallel program begins from one or more objects marked as being initial, which in turn create groups and arrays and invoke methods on these objects to continue the parallel program. Functions on individual member objects of groups and arrays can be invoked from any core using globally unique identifiers. If the invoked object is not local to the core where the call is made, an asynchronous message is sent by the runtime system to the appropriate destination, where the message is received by the scheduler for the local instance of the runtime system. Functions invoked by the scheduler are non-preemptible. When a scheduler picks a message from its queue and calls the corresponding function, it becomes inactive until the function it called returns. TRAM is implemented as a group, so an *instance* of TRAM has one object on every PE used in the run. We use the term *local instance* to denote a member of the TRAM group on a particular PE.

Collective communication patterns typically involve sending linear arrays of a single data type. In order to more efficiently aggregate and process network data, TRAM restricts the data sent using the library to a single data type specified by the user through a template parameter when initializing an instance of the library. We use the term *data item* to denote a single object of this data type submitted to the library for sending. While the library is active (i.e. after initialization and before termination), an arbitrary number of data items can be submitted to the library at each PE.

Due to the underlying assumption of a mesh topology, the benefits of TRAM are most pronounced on systems with a mesh or torus network topology, where the virtual mesh of PEs is made to match the physical topology of the network. The latter can easily be accomplished using a separate module of the runtime system called Topology Manager [6].

The next two sections explain the routing and aggregation techniques used in the library.

### 3.2 Routing

Let the variables $j$ and $k$ denote PEs within an N-dimensional virtual topology of PEs and $x$ denote a dimension of the mesh. We represent the coordinates of $j$ and $k$ within the mesh as $(j_0, j_1, \ldots, j_{N-1})$ and $(k_0, k_1, \ldots, k_{N-1})$. Also, let

$$f(x, j, k) = \begin{cases} 0, & \text{if } j_x = k_x \\ 1, & \text{if } j_x \neq k_x \end{cases}$$

$j$ and $k$ are *peers* if

$$\sum_{d=0}^{N-1} f(d, j, k) = 1. \tag{1}$$

When using TRAM, PEs communicate directly only with their peers. Sending to a PE which is not a peer is handled inside the library by routing the data through one or more *intermediate destinations* along the route to the *final destination*.

Suppose a data item destined for PE $k$ is submitted to the library at PE $j$. If $k$ is a peer of $j$, the data item will be sent directly to $k$, possibly along with other data items for which $k$ is the final or intermediate destination. If $k$ is not a peer of $j$, the data item will be sent to an intermediate destination $m$ along the route to $k$ whose index is $(j_0, j_1, \ldots, j_{i-1}, k_i, j_{i+1}, \ldots, j_{N-1})$, where $i$ is the greatest value of $x$ for which $f(x, j, k) = 1$.

Note that in obtaining the coordinates of $m$ from $j$, exactly one of the coordinates of $j$ which differs from the coordinates of $k$ is made to agree with $k$. It follows that m is a peer of $j$, and that using this routing process at $m$ and every subsequent intermediate destination along the route eventually leads to the data item being received at $k$. Consequently, the number of messages $F(j, k)$ that will carry the data item to the destination is

$$F(j, k) = \sum_{d=0}^{N-1} f(d, j, k). \tag{2}$$

### 3.3 Aggregation

Communicating over the network of a parallel machine involves per message bandwidth and processing overhead. TRAM amortizes this overhead by aggregating data items at the source and every intermediate destination along the route to the final destination.

Every local instance of the TRAM group buffers the data items that have been submitted locally or received from another PE for forwarding. Because only peers communicate directly in the virtual mesh, it suffices to have a single buffer for every peer of a given PE. For a dimension $d$ within the virtual topology, let $s_d$ denote its *size*. Consequently, each local instance allocates up to $s_d - 1$ buffers per dimension, for a total of $\sum_{d=0}^{N-1}(s_d - 1)$ buffers. Buffers are of a constant size. The ability to specify this size gives the user direct control over the maximum memory footprint of TRAM's buffer space.

Sending with TRAM is done by submitting a data item and a destination identifier, either PE or array index, using a function call to the local instance. If the index belongs to a peer, the library places the data item in the buffer for the peer's PE. Otherwise, the library calculates the index of the intermediate destination using the previously described algorithm, and places the data item in the buffer for the resulting PE, which by design is always a peer of the local PE. To prevent memory allocation of buffers for peers which rarely or never communicate, TRAM allocates buffers on demand as soon as a data item is submitted for a particular peer. Buffers are sent out immediately when they become full. When a message is received at an intermediate destination, the data items comprising it are distributed into the appropriate buffers for subsequent sending. In the process, if a data item is determined to have reached its final destination, it is immediately returned to the user through a virtual function reserved for that purpose.

The user has the option of specifying a total buffering capacity, which may be reached even when no single buffer is completely filled up. If that is the case, the buffer with the greatest number of buffered data items is sent out.

Figure 1 shows an example of aggregation of messages by TRAM in a 3D topology.

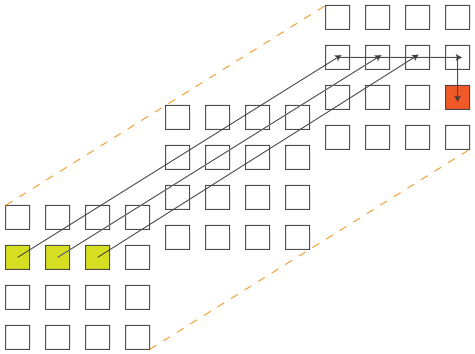


Figure 1: TRAM routes messages along the dimensions of a virtual topology, using intermediate destinations for increased aggregation. The three separate messages in this example are combined at an intermediate destination and delivered in a single message to the destination.

### 3.4 TRAM Usage Scenario

A typical usage scenario for TRAM involves a start-up phase followed by one or more *communication steps*. Usage of the library normally follows these steps:

1. **Start-up** Creation of a TRAM group and set up of client arrays and groups
2. **Initialization** Calling an initialization function, which returns through a callback
3. **Sending** An arbitrary number of sends using the insertData function call on the local instance of the library
4. **Receiving** Processing received data items through the virtual *process* function which serves as the delivery interface for the library and must be defined by the user
5. **Termination** Termination of a communication step
6. **Re-initialization** Object migration, potentially followed by re-initialization of TRAM (step 2) for a new communication step

### 3.5 Periodic Message Dispatch and Termination

Users of TRAM have the option of enabling a message dispatch mechanism which periodically checks for progress in the library and sends all buffers out if no sending took place since the last time the check was done. The period at which checks are performed is left up to the user, but typically it should be done infrequently enough so that it does not impede aggregation performed by the library. A typical use case for the periodic dispatch is when the submission of a data item A to TRAM from user code depends on the delivery of another data item B sent using the same TRAM instance. If B is buffered inside the library and insufficient data items are submitted to cause the buffer holding B to be sent out, a deadlock could arise. With the periodic dispatch mechanism, the buffer holding B is guaranteed to be sent out eventually, and deadlock is prevented.

Termination of a communication step occurs through an ordered dispatch of messages along dimensions from highest to lowest. Termination requires each contributor to a communication step to specify that it has finished submitting data items. The underlying runtime system can determine the number of contributors that are present at each PE and activate the termination mechanism when all contributors at a given PE are done. During termination, each local TRAM instance ensures that it has finished receiving messages along a higher dimension before sending out its buffers for the next dimension. This check is made by comparing

the number of messages received along a given dimension with the sum of the total message counts that are sent out by the peers which finished sending along that dimension. These counts are a minor source of overhead in TRAM messages.

# 4 Theoretical Results

Network communication analysis is a challenging task. Attempts at high fidelity communication analysis invariably require complexity which can only be managed through simulation. On the other hand, simplifying the network model to manage complexity runs the risk of yielding analysis which fails to predict observations. Our approach will be to use a simplified theoretical model, but to identify the assumptions of our analysis at the outset to avoid misrepresenting our results.

## 4.1 Analysis of Aggregation for Nearest Neighbor Communication

In this section we aim to show that most of the improvement in performance due to message aggregation can be explained by a combination of two effects:

- reduction in costs due to per message processing overhead as a consequence of decreased message counts
- reduction in network bandwidth consumed due to message envelope data

We will use hop-bytes [5] as a metric of bandwidth consumption. Hop-bytes for a message are defined as the product of the number of bytes injected onto a network and the number of network links traversed by the message. A message consumes the bandwidth of each link it traverses along the route to its destination, so hop-bytes are a more accurate measure of system bandwidth consumption than the size of the sent buffer. Using hop-bytes will allow us to take into account the impact of network topology and contention in our analysis. For example, hop-bytes capture the benefit of near-neighbor communication as compared to communication between nodes which are several links apart.

We will use the following parameters for our analysis:

- $l$: number of links; the number of network hops between the source and destination
- $m$: payload, or size of application buffer to be sent
- $e$: constant per message data overhead in bytes (due to envelope, CRC, etc.)
- $g$: combining factor; the number of messages combined per send
- $o$: per-message overhead incurred by the runtime system and network

We begin with the simple case of a sequence of $N$ messages sent from a source A to a neighboring node B within the network topology, so that $l = 1$. Each message consists of $m$ bytes of payload data and $e$ bytes of envelope data.

For simplicity, let us assume a fictitious system where per message data overhead is constant regardless of message size, and no additional overhead due to packetization exists. We will later consider a more realistic scenario.

The total hop-bytes when sending the messages individually are:

$$h_{indiv} = N \times (m + e). \tag{3}$$

Combining the payload data of individual messages in groups of g, assuming for simplicity that g divides N exactly, reduces the hop-bytes to:

$$h_{aggr} = N \times (m + e/g). \tag{4}$$

The above leads to several observations. First, messages with payloads that are small relative to the constant envelope size drastically increase hop-bytes. Sending 8 bytes of payload in a message with a 72 byte envelope leads to only 10% of the hop-bytes accounting for payload. If the number of such messages is large, combining may lead to a reduction of close to 90% in hop-bytes.

The second observation is that while increasing $g$ reduces hop-bytes, the effect is diminished as $g$ increases. Considering our example from above, a $g$ of 9 yields a speedup of 5 over the original, a $g$ of 36 a speedup

of 8, and the overall speedup with further increases to $g$ is bounded by 10. This implies that most of the maximum benefit of small message-combining can be effectively exploited with a relatively small combining factor. This works in favor of our approach. Messages which can potentially be combined are rarely sent in close succession, so getting enough messages requires buffering individual payloads until $g$ messages are available. The effect is that in any practical application message combining will *increase* the latency of individual messages. We will assume that performance is bandwidth-limited, but in practice, preventing messages from being sent for too long could stall progress in the application, and without new data injected onto the network, would invalidate the assumption of a bandwidth bottleneck. The fact that a relatively small combining factor gives a substantial benefit implies that low values of $g$ will suffice, limiting the time messages are buffered and allowing the use of relatively little buffer space to good effect.

Equation 4 also shows us that large messages will not significantly benefit from this aspect of combining, as the envelope will account for an insignificantly small fraction of total message size.

**Finding a Good Aggregation Buffer Size** Earlier we proposed that performance improvements due to message aggregation could be explained through (1) lower overhead as a result of fewer messages, and (2) conservation of network bandwidth. We set out to verify this conjecture, as well as to determine an appropriate aggregation buffer size, using tests of point-to-point communication on a pair of neighboring nodes of the Surveyor Blue Gene/P system at ALCF.

Communication performance sometimes benefits from pipelining, where the sending of messages in sequence leads to overlap between phases of transmission and processing of different messages. We decided to incorporate pipelining into our test to show that the benefit due to pipelining does not significantly help to improve poor performance of fine-grained communication.

For our first test, we compared the time for sending a buffer of data using a single message to the time for sending an equivalent amount of data using a sequence of smaller-sized messages, effectively pipelining the communication. Each sequence of sending was immediately followed by an identical sequence in the reverse direction to obtain a "round-trip time." By dividing twice the buffer size by the round-trip time, we obtained an effective bandwidth - the rate of delivery of data when taking into account all overhead. Figure 2a plots the effective bandwidth relative to maximum link bandwidth for this experiment (425 MB/s).

The first thing to notice from the figure is the effect of short messages on bandwidth utilization. Sends of 32 byte buffers utilize at most 2.6% of the network bandwidth, even when multiple messages are sent in sequence to benefit from pipelining. Messages of at least 2KB are required to achieve 50% of the theoretical link bandwidth. The effective bandwidth peaks at around 82% of the maximum link bandwidth, due to message processing costs and runtime system overhead as well as communication overhead incurred on the network.

Sending short messages clearly has a large detrimental effect on communication performance on this system, but this leaves the question of what message size is large enough for good performance, and whether there is a threshold at which pipelining helps rather than hurts performance compared to using a single send. Figure 2b shows the results of the experiment when pipelining in chunks of 8 KB or more. Results show that on Blue Gene/P, sends in chunks of 16 KB or more come within 1% of the peak observed bandwidth for a particular message size. Compared to sending a single message, pipelining helps only for messages of 1 MB or more, and even then the benefit is limited. Also, due to the small decrease in performance of individual sends of buffers over 1 MB, using chunks of 1 MB or more slightly degrades performance.

For the purposes of our library then, the experiment shows 8 or 16 KB to be a good value for the size of TRAM's aggregation buffers. While using larger chunk sizes yields a small additional increase in bandwidth, we don't believe this benefit would justify the cost of significantly increased latency for buffered messages.

**Predicting the Impact of Aggregation** Figures 2a and 2b also provide evidence for the mechanisms responsible for performance improvement due to message aggregation. Based on the results we can rule out that one or the other mechanism we identified at the beginning of this section is the *sole* determining factor for performance improvement due to aggregation. Given the envelope size of 88 bytes for our runtime system, if bandwidth consumption due to envelope data could explain poor performance of fine-grained communication,

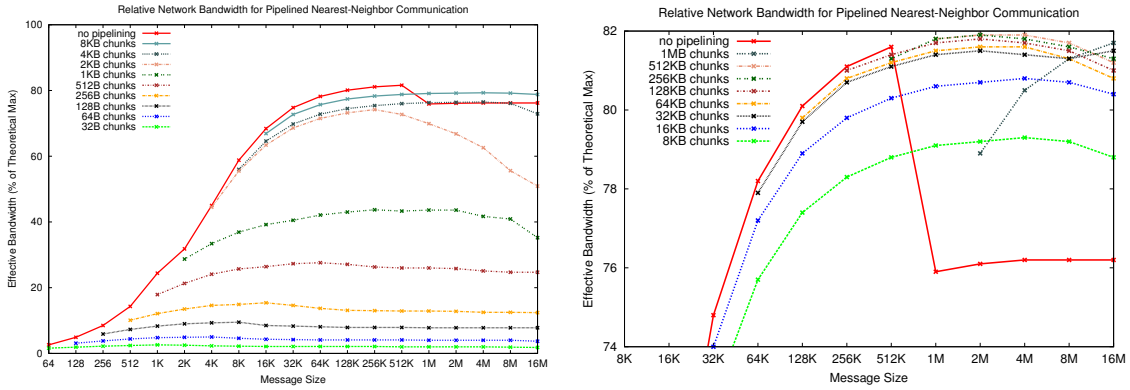



Figure 2: (a) A plot showing effective bandwidth of nearest-neighbor communication on Blue Gene/P when sending data using chunks of small size. Results show that fine-grained communication can only utilize a few percent of the available network bandwidth. (b) Effective bandwidth of pipelined sends on the Blue Gene/P when using chunks of medium to large size. Messages between 16 KB and 512 KB lead to good bandwidth utilization. Sends of 1 MB or more suffer from a slight performance degradation.

32-byte sends should not slow down execution time by more than a factor of $(32+88)/32 = 3.75$. In fact, the effective bandwidth when sending in chunks of 32 bytes is more than 30 times lower than our observed peak. We will soon see that network implementation details further increase data-related overhead of sending, but certainly not by enough to explain a factor of 30 difference in performance of fine-grained communication compared to sends of medium-sized messages.

Similarly, one should not expect constant overhead due to message processing in the runtime system and network to fully explain the performance difference between single and chunk sends, as it ignores any impact that message size has on performance, which is clearly unrealistic.

To establish the combination of the two factors as a good indicator of messaging performance, we used linear regression analysis, with the number of sent messages and hop-bytes as independent variables for predicting round-trip time. As our model does not take into account pipelining, we did not expect a perfect match between the model and data, but due to the limited effects of pipelining we had observed, we felt the model should do well enough to be usable.

We soon realized, however, that our approach to calculating hop-bytes required refinement. The issue at hand is that fine-grained communication is sensitive to additional sources of overhead due to network communication. We have already seen envelope cost as a significant source of overhead. In addition to the envelope attached to a message by the runtime system, the communication interface for the network uses its own headers for routing and error checking, and there are other significant sources of overhead due to packetization [1].

Before a message can be sent on the Blue Gene/P torus network, it is split into *packets*, each consisting of 1 to 8 *chunks* of 32 bytes each. The first eight bytes of a packet are reserved for link-level protocol information such as routing and destination information, and the last 4 bytes of every packet are used for error correction through a cyclic redundancy check (CRC) and an indicator of validity. There are several performance implications of this packetization for fine-grained communication. First, it leads to at least 12 bytes of overhead in every 256 bytes sent (4.7% overhead). Secondly, a chunk is the smallest unit of data that can be injected onto the network, so short messages for which the last chunk of a packet is not filled completely with data will suffer additional overhead. Third, each packet requires processing overhead due to routing and other network-level processing. Crossing a message size threshold which requires injection of two rather than one packet can noticeably affect performance.

Based on the details of Blue Gene/P network implementation, we calculated the number of packets and chunks that would be required for sending an 8 MB buffer using messages of various sizes. We picked a large buffer size so that pipelining could be expected to affect execution time, as we wanted to see how well our model could match the results in the presence of pipelining effects.

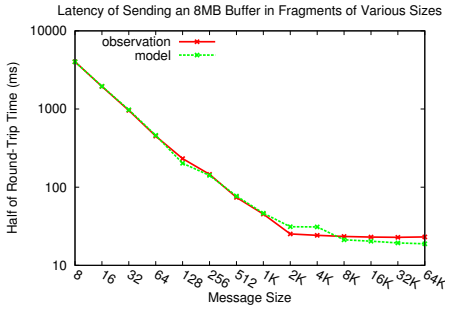



Figure 3: Latency of a pipelined send of 8 MB on the Blue Gene/P. Increasing the size of the messages used for sending markedly improves performance up to a message size of 2 KB, beyond which bandwidth saturation limits further improvements. The observed results show non-regularities, such as at 256 bytes. Using a least-squares regression model with the number of messages, number of packets, and hop-bytes as the independent variables, we were able to closely match the observed results, showing that together these variables are good predictors of communication time.

| **Msg Size** | **#Msgs** | **#Packets** | **Hop Bytes** | $\frac{1}{2}$ **RTT (ms)** |
|---|---|---|---|---|
| 8 | 1M | 1M | 128M | 4030 |
| 16 | 512K | 512K | 64M | 1945 |
| 32 | 256K | 256K | 40M | 960 |
| 64 | 128K | 128K | 24M | 450 |
| 128 | 64K | 64K | 16M | 232 |
| 256 | 32K | 64K | 12M | 146 |
| 512 | 16K | 48K | 10M | 74 |
| 1K | 8K | 40K | 9.25M | 45.3 |
| 2K | 4K | 36K | 8.89M | 25.2 |
| 4K | 2K | 36K | 9.13M | 24.2 |
| 8K | 1K | 34K | 8.5M | 23.4 |
| 16K | 512 | 34K | 8.45M | 23.0 |
| 32K | 256 | 33.8K | 8.42M | 22.8 |
| 64K | 128 | 33.6K | 8.41M | 23.1 |

Table 1: Packet and hop-byte counts for pipelined sends of 8 MB using various numbers of equal-sized messages. The hop-bytes represent the actual number of bytes injected onto the network based on calculations taking into account Blue Gene/P network implementation.

Table 1 presents total packet counts, hop bytes, and execution time when sending the buffer using a set number of equal sized messages. Hop bytes represent the total number of bytes injected onto the network. We did not include the number of chunks, as these are simply equal to the hop bytes divided by 32. We also plotted the execution time, representing half of the round-trip time, in Figure 3.

A first look at the figure implies a range of exponential decrease in execution time for fragment message sizes from 8 bytes to 2 KB, followed by only relatively minor further decrease in time for higher message sizes. Nonetheless, one can see from the curve and the table, particularly for messages of size 256 bytes, that the exponential decrease is non-uniform. Our calculations of the number of packets required to perform the sends were able to clarify much of the non-uniformity. For example, sending a message of size 256 bytes requires 2 packets, while 128 byte messages can be sent using a single packet. As a result, using messages of size either 256 bytes or 128 bytes leads to an injection of an equivalent number of packets onto the network when sending the full 8 MB buffer. [1] Packets represent a considerable source of overhead on the network due to routing. We believe that the equivalent packet counts for 128 and 256 byte messages are responsible for the relatively low speedup when using 256 byte messages.

Figure 3 also shows the time predictions of our regression model, which estimates execution time based

[1]Note that the hop byte counts are not equivalent for the two cases. The total overhead due to the runtime system envelope is two times higher when using 128 byte messages.

on a linear combination of three variables: number of messages, number of packets, and number of hop bytes. The regression coefficient of .99995 for our model indicates that overall it represent a good fit for the experimental data. As such, we think this experiment provides strong evidence that the performance impact of message aggregation can be predicted based on the two factors we identified at the beginning of this section.

### 4.2 Virtual to Physical Topology Mapping

While the choice of a virtual topology to be used for TRAM is left to the user of the library, we believe that *matching* the physical topology typically leads to the best performance. In this section, we explain why this is the case.

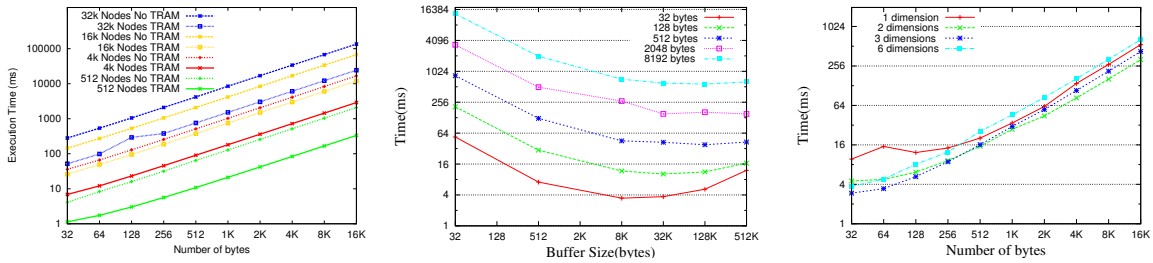
Consider a data item sent over an N-dimensional mesh or torus using TRAM, and let a = F(j,k) from eq. 2 be the number of messages required to deliver it to its destination. As noted in Section 3, every intermediate message along the route makes positive progress toward the destination along a single dimension of the virtual topology. If the virtual topology is identical to the physical topology, then as long as the network's dynamic routing mechanism does not make an intermediate message take a non-minimal path, the route to the destination determined by intermediate messages will travel over the least number of hops between the source and the destination.

It is also possible to reduce the number of dimensions in the virtual topology as compared to the physical topology while preserving minimal routing. The key is to notice that any two dimensions that are "adjacent" (within the ordering of dimensions used for determining ranks in the topology) can be joined into a single dimension while maintaining minimal routing. For example, a 4 x 4 x 8 topology can be reduced to a 16 x 8 topology by joining the first two dimensions. Compared to a virtual topology that is an exact replica of the physical topology, a data item sent within the reduced topology obtained in this way will follow the same route as in the full topology while skipping over some intermediate destinations. Each intermediate destination leads to processing overhead and the need to re-inject the data onto the network. In return, intermediate destinations allow aggregation of messages which would otherwise be sent separately. It is conceivable, then, that in cases where messages are sufficiently large so that extensive aggregation is not strictly necessary for performance, a lower-dimensional virtual topology may be better than a higher dimensional one.

On the other hand, assume a random mapping between the nodes in the virtual topology and the physical topology, and let $s_d$ be the size of dimension d in the physical topology. On average, each intermediate message will travel the average distance between pairs of nodes in the physical topology, which is $\sum_{d=0}^{N-1} s_d/4$ for a torus and $\sum_{d=0}^{N-1} s_d/3$ for a mesh. This is clearly undesirable, and leads to a significant increase in congestion. Note that this congestion grows not just in proportion to the size of the physical topology, which affects the number of hops traveled by each intermediate message, but also in proportion to the number of dimensions in the virtual topology, which affects the average number of intermediate messages required to deliver a data item. In fact, an intermediate message when using a random mapping will on average travel as many hops as a full route would when using a virtual topology that matches the physical topology.

## 5 Experimental Results

There is a common misconception that fine-grained communication is always the result of poorly written parallel code. According to this school of thought, it is the programmer's responsibility to ensure a proper grain size such that messages are of a large enough size to yield good performance. While there are certainly cases where grain size control in an application can lead to good results, sometimes this becomes very tedious, and at other times difficult. In particular, for a distributed object-oriented system like Charm++, one would like to be able to have objects in the parallel program represent the players and entities of a parallel simulation, rather than collections of these items which bear no similarity to any physical or abstract entity. Using a library like TRAM allows a programmer to maintain the illusion of direct communication between fine-grained entities, which may lead to a cleaner expression of a given algorithm.




(a) All-to-all benchmark on Blue Gene/P (b) Effect of aggregation buffer sizes (c) Effect of virtual topologies dimensionality

Figure 4: All-to-all benchmark: (a) Blue Gene/P with and without TRAM. (b,c) Effect of various parameters on Blue Gene/Q with TRAM.

A second reason to take fine-grained communication seriously is that there are situations which require its use. Small control messages and acknowledgements are very common in a parallel setting and rarely lend themselves to grain control. Further, improving parallel application performance for a given data set typically requires increasing the number of cores on which the application is executed, which typically leads to partitioning of the data set into increasingly fine pieces. By improving fine-grained communication performance, TRAM may allow for increased scaling of some classes of parallel applications. We next take a look at some example benchmarks and applications whose performance improved from using TRAM to aggregate fine-grained communication.

The results presented in this section demonstrate the effectiveness of TRAM in improving performance of both clear-cut and subtle sources of fine-grained communication in benchmarks and applications.

## 5.1 Benchmarks

### All-to-All

We ran tests of an all-to-all benchmark on four different partition sizes of the ALCF Intrepid Blue Gene/P system:

- 8 x 8 x 8 = 512 nodes
- 8 x 16 x 32 = 4096 nodes
- 16 x 32 x 32 = 16384 nodes
- 32 x 32 x 32 = 32768 nodes

For these tests, we used a single process per node of Blue Gene/P. The TRAM version of the benchmark used data items of size 32 bytes, while the version without TRAM used point to point sends of 32 bytes. For TRAM runs we set the size of aggregation buffers to 2 KB. To lessen communication bottlenecks, we used a standard optimization of randomizing the order in which sends are performed for both versions of the benchmark. All-to-all of more than 32 bytes per destination was performed in rounds (for both versions of the benchmark) by looping over all the destinations and sending 32 bytes to each destination in each round.

Results are shown in figure 4a. Using TRAM led to speedups of between 3.5 and 6.25. The speedups for the 512 node run show the most variability with the amount of bytes sent per destination. Speedup compared to not using TRAM for that run was 3.65 when sending 32 bytes per destination, and increased gradually to 6.25 for sends of 16KB per destination. For the other runs, speedup was more uniform, at around 5 to 6 x, with the exception of the outlier for the 128-byte data point at 32K nodes, for which speedup was 3.59. Overall, the results indicate a clear performance advantage to using TRAM for fine-grained communication.

We ran further tests using this benchmark on the Vesta Blue Gene/Q system at ALCF, with the focus of showing the impact of varying the size of the aggregation buffers and the number of dimensions in the virtual topology. The 5D network topology on Blue Gene/Q made it a particularly suitable system for the latter tests. The Blue Gene/Q tests were carried out by Yanhua Sun.

Figure 4b shows the results of the all-to-all benchmark using TRAM on 64 nodes using various aggregation buffer sizes, with a 3D virtual topology. Results confirm our prior observations for nearest-neighbor communication. Smaller buffer sizes, representing less aggregation, produce significantly higher total execution time, even in the case when the payload per destination is just 32 bytes. With increased buffer size, performance improves sharply up to a buffer size of 1 KB, and more slowly up to at least 8 KB. Beyond that, any additional improvements are minor.

Our second Blue Gene/Q experiment, the results of which are presented in Figure 4c, involved comparing the performance when varying the number of dimensions in the virtual topology using the approach described in Section 4.2. The runs were done on a 64 node partition. The difference in execution time between the best and worst topology for a given byte count can be as high as 4.4x, demonstrating the importance of taking some care in selecting a virtual topology when using TRAM. Results show a 3D virtual topology being best for low total payload sizes, and a 2D virtual topology being better for higher total payload size. As explained in Section 4.2, intermediate destinations are a source of overhead in terms of processing and injection bandwidth. At higher payload sizes, there is sufficient data available for aggregation when using lower dimensional virtual topologies, so that a 3D or higher topology is not worth the additional overhead of additional intermediate destinations. The 6D topology, obtained by using the five dimensions of the network topology and cores per node as the sixth dimension, further reinforces this conclusion, as it always leads to worse performance than the 3D topology at this node count.

### HPCC Random Access

The HPC Challenge Random Access benchmark measures the rate of processing updates to a distributed table. Each process issues updates to random locations in the table. The small size of individual update messages in this benchmark makes it prohibitively expensive to send each item as a separate message. This makes the benchmark an ideal scenario for using TRAM.

For testing on the Intrepid Blue Gene/P system, we used a 3-dimensional virtual topology comprising the processing elements (PEs) involved in the run. Using the Charm++ Topology Manager library, we made the virtual topology match the physical topology of the nodes in the partition for the run, except that we dilated the lowest dimension of the virtual mesh by a factor of 4 to account for the separate cores within a node. Random Access specifications dictate a buffer limit of 1024 items per process. To adhere to this limit, we set the TRAM buffering capacity at 1024 items per local instance of the library.

On the Vesta Blue Gene/Q system, where the network is a 5D mesh, we found that a 5 or 6-dimensional virtual topology works best, where two of the dimensions comprise the 64 processes within a node (8 x 8), and the remaining dimensions are obtained by reducing the mesh dimensions through combining of some individual dimensions according to the scheme presented in Section 4.2. We found that combining the first and second dimensions of the physical partition into a single dimension in the virtual mesh made the most sense, and likewise with the fourth and fifth. In general, we selected for combining dimensions of the lowest size. Scaling runs and tuning of dimension sizes of the virtual topology for runs on Vesta were done by Ramprasad Venkataraman. Figure 5 shows scaling plots of the results on Intrepid and Vesta.

### HPCC Fast Fourier Transform

Another HPC Challenge benchmark which greatly benefits from the use of TRAM is the Fast Fourier Transform benchmark, which measures the floating point rate of execution of a double precision complex one-dimensional Discrete Fourier Transform.

In the benchmark, processes send arrays of a single data type on the network. To simplify the user interface and remove the need for individually submitting each element of a data array to TRAM, we employed a version of TRAM that allows submitting arrays of data items. The library then automatically performs segmentation of the array into *chunks* at the source and reconstructing arrays from chunks at the receiver. Chunk size is independent of the size of the data type for individual elements, allowing us to limit processing overhead when data items are very small. Figure 6 shows a scaling plot for the benchmark on Intrepid.

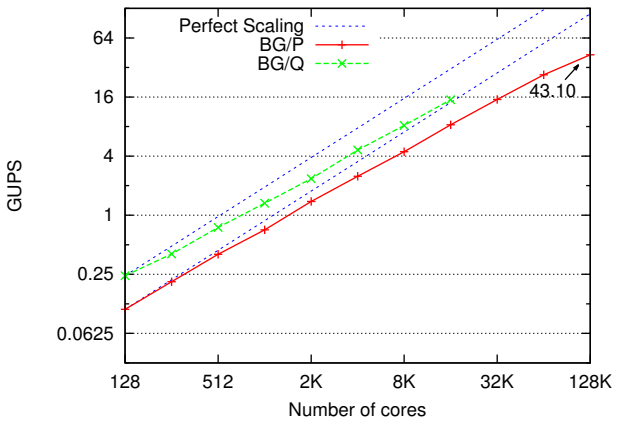



Figure 5: Results of the Random Access benchmark on Blue Gene/P and Blue Gene/Q

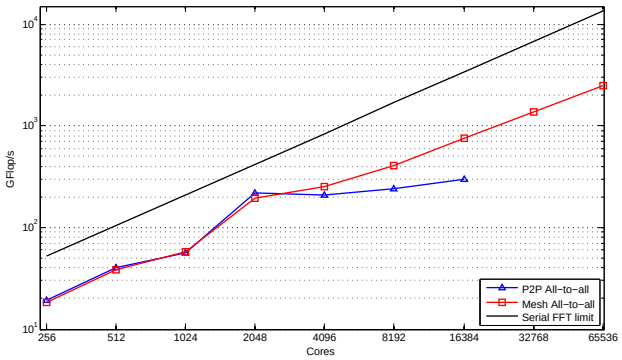



Figure 6: Results of Running the FFT benchmark on Blue Gene/P

The Charm++ FFT benchmark was written by Anshu Arya, who also did the scaling runs on Intrepid.

## 5.2 Applications

### EpiSimdemics - Simulations of the Spread of Contagion

EpiSimdemics is a parallel algorithm and simulator developed at Virginia Tech for simulating the spread of infectious disease over realistic social networks with hundreds of millions of individuals [3, 4, 8]. Originally coded in MPI, it has more recently been ported to Charm++.

EpiSimdemics employs an agent-based epidemic model. In each time step, each person object sends a message to every location object visited. The location objects in turn calculate the probability of infection and send messages to infected person objects. Messages sent to the location objects are numerous and small in size (38 bytes), so that network communication is a bottleneck to performance. These factors combine to make EpiSimdemics a good use case for TRAM.

We evaluated the performance using TRAM on Blue Waters [23] using a 2D virtual topology with the the number of nodes as the first dimension and the the number of cores per node as the second. EpiSimdemics was run using CHARM++'s Cray Gemini SMP machine layer [25, 26]. Combining messages with TRAM yields speedups of up to 4x, as shown in Figure 7. Also shown are results when using a manual message combining scheme rather than TRAM. In this scheme local messages which are to be sent to the same PE are combined, similar to using TRAM with a linear (1D) virtual topology. Compared to TRAM, the manual scheme performs slightly worse and uses more memory, as a buffer is required for each PE in the run.

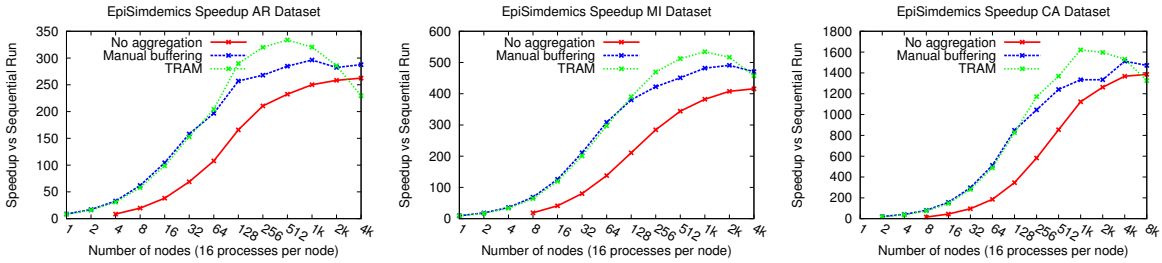



Figure 7: Performance of EpiSimdemics on Blue Waters for the AR, MI, and CA data sets

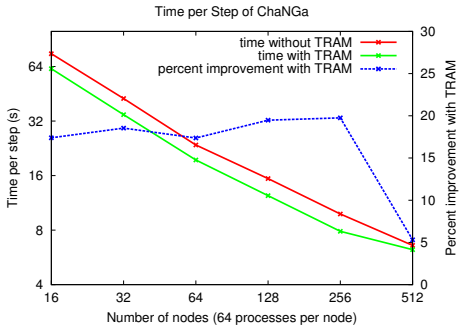



Figure 8: Performance of ChaNGa on Vesta using the 50 million particle dwarf data set

The work of incorporating TRAM into EpiSimdemics was done by Abhishek Gupta and Jae-Seung Yeom. Performance results were collected by Jae-Seung Yeom.

### ChaNGa - Cosmological N-Body Simulations

Another application which benefited from TRAM is ChaNGa [10, 16], a cosmological N-body simulator written in Charm++, based on the Barnes-Hut algorithm. Network communication structure in ChaNGa had been optimized over the course of several years with multiple levels of abstraction grouping particles in a hierarchy of structures of increasing size to balance different sources of overhead. Despite this, we still found some opportunities for improving performance with TRAM. One source of improvement was combining of messages requesting data from remote processors. These messages, each consisting of 16 bytes of payload data, were numerous, and combining them significantly lowered the total number of messages in the application. Secondly, we used TRAM for the messages that returned particle data in response to a request. While the average size of these messages was relatively large, in some cases the returned data consisted of only a few particles. We also used TRAM during domain decomposition, when particles are migrated between processes in a sparse all-to-all communication pattern.

Figure 9 illustrates the effect of TRAM on the total volume of communication in ChaNGa. Shown are histograms of messages sent in a single time step of ChaNGa, both with and without using TRAM. As can be seen, using TRAM greatly reduces the number of messages requesting data from remote processes. as shown in Figure 8, TRAM improved performance by 15 to 20% on Vesta, except at 512 nodes where bottlenecks in parts of the code not accelerated with TRAM determine performance.

For messages of sufficiently large size, further combining may not improve performance, and the delay due to buffering may in fact hurt performance. A case in point is the node reply message type in ChaNGa. These messages, while numerous, are several hundred bytes in size or larger. We found that using TRAM for these messages hurt performance.

Incorporating TRAM into ChaNGa was done in collaboration with Pritish Jetley.

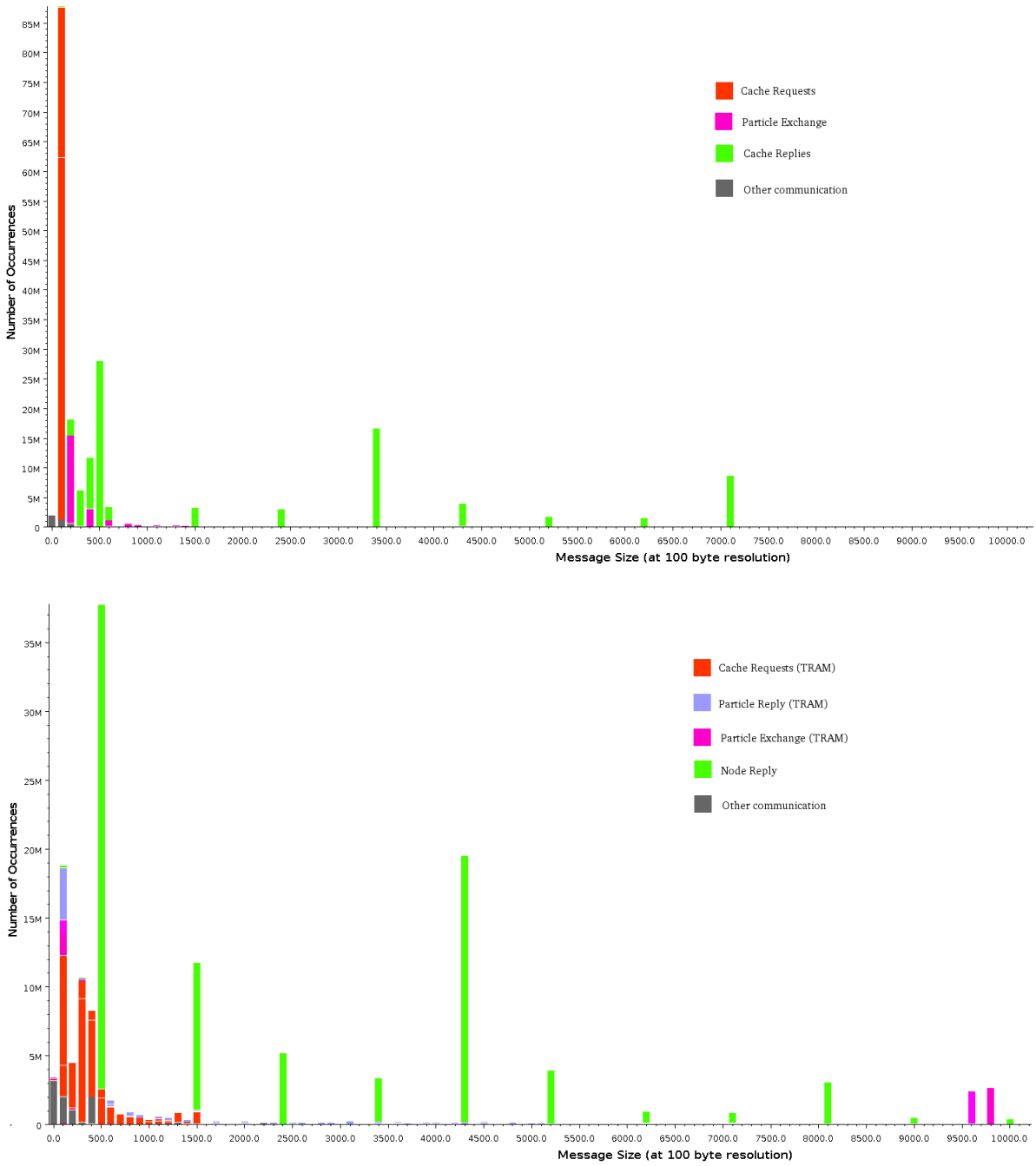



Figure 9: Histogram of messages, grouped by message size, for a time step of ChaNGa when not using TRAM (top) and with TRAM (bottom). Results were obtained on 4k nodes of the Intrepid Blue Gene/P system at ALCF, using the 50 million particle dwarf data set.

# Planned Work

The results presented in the previous sections show the utility of the topological routing and aggregation approach as employed in TRAM for improving communication and overall performance in benchmarks and large applications. Moving forward, the planned work, described in the following sections, serves to advance this work along multiple axes.

The success of a software library depends heavily on ease of use and feature set. Our existing TRAM use cases have served to identify multiple opportunities for making TRAM easier to use. Over time, we made some of these changes. Other issues, which are more involved, are part of the proposed work in the following sections. These changes will allow tighter integration of TRAM as a module and subsystem of the Charm++ runtime system. Ultimately, our goal is for every Charm++ program to benefit from topological message aggregation without requiring additional code, tuning, or input from the developer or user. This entails several challenges, which we will describe in Section 6.

Secondly, in its current form TRAM can only exploit the full benefits of topological aggregation while preserving minimal routing on mesh and torus systems which assign hyperrectangle job partitions. Currently, only Blue Gene systems satisfy this requirement. On other systems, one can still get some benefit from intranode aggregation, typically by setting up a 2D virtual topology with one dimension for the nodes in the run and one for the processes or worker threads within the node. As part of the work proposed in Section 7, we will be implementing topological schemes for non-mesh topologies, as well as mesh networks which do not assign hyperrectangle partitions.

Our theoretical study of aggregation, so far limited to nearest neighbor communication, will need to be extended to more realistic systems, taking into account both topology and network congestion. This will be explored in Section 8.

We believe TRAM can be leveraged for implementation of higher level communication libraries on top of it, particularly collective libraries. We have already implemented broadcast capabilities within TRAM. In the future, we will look at implementing other collective calls, including multicasts, reductions, scatters, gathers, allreduce, and possibly other collective operations. Section 9 describes these efforts in more detail.

In order to make TRAM competitive with the best communication libraries on supercomputing systems, we will need to exploit native communication operations. Examples of such interfaces include multicast capabilities, special auxiliary networks (such as the collective network on Blue Gene/P), and remote direct memory access (RDMA), which we can leverage through the use of an existing Charm++ persistent communication library. Effective use of shared memory within an SMP-node will also be important for getting the best possible performance with TRAM. This will be further explored in sections 10, 11, and 12.

A module that is tightly coupled within the Charm++ runtime system will not serve the needs of most of the HPC community, where MPI is the norm. With a bit of careful software engineering, it should be possible to separate out sections of the TRAM code base that are truly Charm++ dependent from the more generally applicable code for routing and aggregation. The small amount of Charm++-specific code could then be ported to other communication libraries, such as MPI. This will be further described in Section 13.

Finally, we will continue working on application and benchmark use cases for TRAM. These efforts serve a double purpose, as an evaluation tool for TRAM on one hand, and a gauge of its robustness and generality on the other hand. Also, we intend to demonstrate the use of TRAM to combine control messages at the level of the runtime system. We will describe this in Section 14.

# 6 From Library to Autonomous Subsystem

In its existing form, TRAM is a generalized library for aggregation of network communication in Charm++ applications. The current library is capable of aggregating network communication to a client chare group or array, and takes as a template parameter a data type denoting the type of data items to be communicated. To send a data item using TRAM, users call a method of the library, which takes a reference to a singleton data item as input. In addition, for group clients, there exists a "Chunk" variant of TRAM which supports sending arrays of data items.

Our initial implementation of TRAM is by design relatively simple, leaving out features which were not deemed to be necessary to establish the proof of concept, and assigning many parameter specification decisions to users of the library. We now take a closer look at the challenges involved in automating parameter selection and relaxing interface restrictions.

## 6.1 Automating Parameter Selection

Library parameters that users have to specify include:

- Size of buffers to be used for aggregation (or alternately, the total memory footprint for the buffer space)
- Virtual topology, including the number of dimensions and their sizes
- Whether to activate periodic dispatch, and if so, at what time interval

Automating the choice of each of the above parameters presents its own challenges. In the theoretical results presented in Section 4.1 we identified 2 - 16 KB as reasonable buffer sizes which lead to good bandwidth utilization while limiting the amount of time data items are buffered before being sent. However, our tests were done on a single system (Blue Gene/P) and only for nearest neighbor communication. Although we are confident the results are generally applicable, this needs to be shown concretely by repeating the test on other systems. Perhaps the most interesting result of this exercise will be to see whether the range of buffer sizes that work well for aggregation varies significantly across systems.

Automating the choice of virtual topology can be done reasonably well on systems with mesh networks by using the Charm++ Topology Manager, which returns the dimensions of the physical topology for the partition used for a run. On Blue Gene/P, using Topology Manager to match the physical topology appears to work best, while on Blue Gene/Q, using a 3D topology which combines the 5D dimensions of the job partition into two dimensions and uses the third dimension for the processes within a node appears to work well. Still, the Blue Gene/Q runs were done on the Vesta system at ALCF, which is a relatively small test system that may not accurately represent conditions on larger machines. As such, more work is needed to assess the tradeoff between aggregation and overhead that depends on the number of dimensions used for the virtual topology specification. Handling non-mesh topologies is another serious challenge, which we will describe further in the following sections.

The periodic flushing mechanism is normally used only in situations that require it to avoid deadlock, such as when a TRAM data item is submitted in response to the delivery of another TRAM data item. Automating the flushing aspect of the library may thus require code analysis to identify the potential for deadlock. Another possibility is deadlock detection and recovery. Detection could be achieved by using the Charm++ quiescence detection feature, which identifies a global idle state. Recovery could be done by dispatching all TRAM buffers. The performance of this scheme will depend on how frequently deadlock is encountered. If necessary, we will develop a more efficient scheme.

## 6.2 Generalized Aggregation

In addition to the need to specify library parameters, users of TRAM currently need to be aware of certain restrictions in usage of the library:

- Each TRAM instance is limited to supporting a single client, either group or chare array, and a single input data type.

- A TRAM instance must be initialized prior to use and terminated to mark that submission of data items is done.
- Data items must have a constant size.
- The operator = for the data item data type must be sufficient for copying the state of the data item into an aggregation buffer.
- For client chare arrays, location of chare array elements must be globally known while TRAM is active.

Most of these limitations arise from the decision to implement TRAM as a templated library that restricts data items to a single type. An alternative design is to allow sending arbitrary data buffers. We have already taken a step in this direction with a "Chunk" variant of TRAM, which allows users to submit arrays of data items. To support data items of arbitrary size, Chunk TRAM processes submitted buffers as raw data and performs segmentation automatically without regard for data type boundaries. Further generalization of this approach is required to allow sending arbitrary data, and to properly serialize/deserialize complex objects before/after transmission on the network. The Charm++ pack-unpack framework will be useful for accomplishing this task. We anticipate a significant challenge in attempting to make the generalized library perform as well in simple cases as the current, more specialized variants.

The last bullet point above refers to an esoteric performance problem that users of TRAM may encounter when using it in applications that perform migration of chare array elements, such as for load balancing purposes. When an element is migrated in Charm++, its location information is updated only at a single, globally known process, called the homePE. Other than the homePE and the two PEs involved in migration, any process that subsequently tries to send a message to that element will send to the previous location. When the message arrives at the old location, the runtime system recognizes that the object no longer resides locally and forwards the message to the homePE, from where it will be delivered to the correct destination. The homePE also performs another important function - it sends the updated location back to the original sender, so that future sends from that sender may go directly to the correct destination. Unfortunately, this process does not have the desired effect when using TRAM. When a data item arrives at an incorrect destination due to stale location data at the original sender of the data item, the rank of the process that sent the data item is unknown, as the data item may have passed through one or more intermediate destinations. As such, while the default lazy location update scheme will be sufficient to deliver the item to the correct destination by forwarding it to the homePE, the updated location data will be sent to the last sender, which is likely to be an intermediate destination rather than the original source. Hence, future sends from the original source for the same destination will be indefinitely routed to the incorrect location. In this scenario, using TRAM will likely hurt performance. To work around this problem, we implemented a scheme that updates the location of migrated array elements globally. However, this scheme is not scalable and only works for applications that employ centralized load balancing. As such, we will need to develop a new scheme with low overhead and low incidence of data items that require forwarding.

### 6.3 Automatic Aggregation

On the path to fully automating TRAM, one of the most ambitious aspects of the planned work will be to determine which messages to aggregate and which to send directly without delay. The challenge involved is substantial. Optimally, the runtime system would have to determine which messages lie on the critical path for the application in order to correctly decide when aggregation should not be attempted. Alternatively, we may employ autotuning to determine at runtime which message types are amenable to aggregation. Due to the possibly large number of different types of messages, we may also need to prune the full set of message types in the application and runtime to be considered for aggregation. One way to do so would be to determine which message types lead to substantial bandwidth use, and only consider these messages for aggregation.

If the task proves too difficult, we will consider relaxing slightly the automation requirement. For example, we may leave it to the user to specify latency-sensitive messages that should not be aggregated, or even to specify when aggregation should be attempted. This can be achieved by adding new annotation types for Charm++ interface (.ci) files, supplementing existing types that specify message-specific behavior, such as prioritizing specific messages in the scheduler.

# 7 Non-Mesh and Incomplete Mesh Topologies

In its current form TRAM employs a mesh topological aggregation scheme that works well on mesh networks with hyperrectangle job partitions. We now take a closer look at the challenges involved in extending topological aggregation to various other network types.

## 7.1 Incomplete Meshes - Cray Gemini Systems

Like Blue Gene/P systems, Cray supercomputers such as the XE series use a 3D torus network. Despite this, there are two main challenges in effectively using topological aggregation on these systems:

- some nodes within the torus topology are reserved for I/O, login, and system related tasks
- irregular job partitions - typical partitions consist of various nodes throughout the system

The issue of missing nodes, or *holes*, within an otherwise regular partition has been previously explored in [17] and [19]. There, the proposed solution was to route traffic that would normally go through the missing node to a nearby node, while ensuring that links were utilized as uniformly as possible in all-to-all scenarios.

The above scheme may be useful when running on a full Cray XE system or a large, manually selected partition from the full system. Otherwise, a different scheme is required in the more general case of a run across various nodes throughout the system. One approach we can try is to identify substructures within the list of nodes in the run using a clustering algorithm. A 2D mesh virtual topology could then be established, with one dimension for the substructures and a second for their contents. Recursive application of the clustering algorithm within each cluster could be used to identify additional dimensions for the topology. Due to the unpredictability of this scenario, adaptivity may be required for good performance, where the virtual topology would be adjusted depending on the structure of the list of nodes in the run.

## 7.2 Fat Tree Networks - Infiniband Clusters

Fat tree networks are the standard interconnection network topology for Infiniband networks, which account for 45% of the systems on the November 2012 Top 500 list. In a true fat tree the communication nodes are arranged into a tree, with progressively higher bandwidth in the upper levels, closer to the root. Theoretically, this allows for full bisection bandwidth, so that in any partitioning of the leaf nodes into pairs, all pairs can concurrently communicate at the bandwidth of the links in the bottom level of the tree. Typical network configurations are not strictly fat trees, as they have multiple nodes in the uppermost level instead of a root. However, as messages between distant nodes always pass through one of the handful of switches in the top level, the top switches can be imagined to collectively form a "root", so that the full topology resembles a fat tree. For example, Stampede, placed 7th on the November 2012 Top500 list, has a two level fat tree topology, with 8 core switches forming the top level of the tree, connected in an all-to-all manner to 640 leaf switches in the second level, each of which in turn serves a group of 20 compute nodes.

Fat tree networks are characterized by a low diameter (and hence low small message latency) and high bisection bandwidth, but due to much network traffic being routed through the core switches, they tend to exhibit considerable performance degradation in network congestion scenarios [11]. Some recent efforts to address this issue take the approach of throttling communication to prevent congestion-related performance deterioration [22]. We expect that an aggregation approach should further improve the efficacy of such schemes, by making better use of the time that messages are buffered. As such, using TRAM could make a significant difference in the network performance on these systems. This would require modifications to allow TRAM to actively throttle injection of messages onto the network. An alternative is to use TRAM as a first stage in combination with existing throttling libraries.

To date, our approach to using TRAM on Infiniband clusters has been to employ a two-dimensional virtual mesh topology, with one dimension for the cores within a node, and the second for the nodes in the run. This has led to some performance improvement, particularly for EpiSimdemics (see Section 5.2), but there is room for further improvement.

To better exploit fat tree networks, we propose new abstractions for message aggregation. The first step along this direction will be to establish a TRAM dimension for the set of nodes connected to each leaf switch.

For all-to-all communication on Stampede, this should reduce the number of messages by up to a factor of 20, while maintaining minimal distance routing. More involved schemes are possible to further increase aggregation. While employing aggregation at the core and leaf switches would likely require modifying the switch firmware and may not be possible, we could try assigning some nodes the task of aggregation across a collection of leaf switches. This last change in particular would move us away from the purely mesh virtual topology abstraction. A possible direction in this vein is a mesh of trees virtual topology model.

### 7.3 Hierarchical Clustered Networks - IBM PERCS, Cray Cascade

Some recent supercomputing networks use a multi-level, hierarchical approach that attempts to strike a balance between network performance and scalability. At the top level, these networks consist of a set of supernodes that form a dense (or complete) graph. Each supernode is in turn another dense or complete graph of substructures. Examples of such networks include the IBM PERCS network, used in its Power 775 series of supercomputers, and the Cray Cascade. These networks provide high bandwidth connections between nodes that are nearby in the topology, and progressively lower bandwidth between far-away nodes (especially in congestion scenarios, such as for bisection bandwidth). To make the discussion more concrete, we will use PERCS as a guiding example, but we expect most of the ideas to carry over to Cascade as well.

Previous work has shown that when using these networks, care is required to make good use of the numerous direct connections between clusters at each level [7]. Secondly, one must try to avoid overutilizing individual links in the higher levels of the topology, which may have lower bandwidth than links at the subcluster level. Unlike with mesh and torus networks, topology-aware mapping schemes, when used in combination with direct routing, may aggravate these problems by using only a small fraction of the links. The proposed solution was to randomize placement of tasks to physical nodes and use indirect routing, improving communication load balance among links, at the cost of increased aggregate bandwidth consumption.

The use of randomized mapping on these systems has the effect of obscuring the distinction between near-neighbor and far away communication by making it difficult to predict whether two ranks are nearby or far apart within the physical topology. This may increase the applicability of TRAM for near-neighbor communication. On mesh networks an application with strictly nearest-neighbor communication may not benefit significantly from a message aggregation approach, and would certainly not benefit from combining at intermediate destinations. With a randomized mapping on a PERCS network, the previously nearest-neighbor pattern would instead consist of mostly far away communication. As a result, aggregation at the source and intermediate destinations may lead to a significant reduction in the number of messages on the network and the bytes consumed by envelope data.

Our choice of TRAM virtual topology on these systems will be guided by the physical topology of the network. As with other network types, we do not want TRAM to cause excessive bandwidth use through software routing choices that significantly stray from a "good" path. We define the quality of a path for a software routed data item based on the maximum number of hops required to deliver a data item from the original source to its destination. To develop a scheme with relatively low maximum path length, we can take advantage of an important characteristic of these networks - messages sent between nodes on the same supernode are always routed directly [7]. This gives us control over the routing decisions within the supernode for the source and destination. The only uncertainty is whether the hardware will route a message directly to the destination's supernode or along an indirect route when sending across supernodes.

Given a source node and destination node, it should be simple to determine the path that direct routing would choose. Hence, our planned scheme is to set up TRAM to route data items along the direct path to the destination, with the nodes along that path as the intermediate destinations. If direct routing is used in hardware, the path will be no different from a direct send, and each data item will travel along at most 3 links. If indirect routing is used in hardware, the path between supernodes could conceivably take as many as 5 links, but in practice this number will be lower. In the worst case, the number of links used to deliver the data item will be 7, so the additional bandwidth use due to dilated paths will be manageable.

# 8 Analysis for Generalized Communication Patterns

With the analysis presented in Section 4.1, we have shown that medium-sized buffers (e.g. 8 - 16 KB on Blue Gene/P) achieve reasonably good utilization of network bandwidth for nearest neighbor communication. We also showed that sends of messages below 2 KB suffer due to envelope and processing overhead. We used these results to support the idea that relatively low levels of aggregation can lead to large benefits. Using medium-sized buffers directly limits the memory footprint of the aggregation buffer space and indirectly affects the time spent buffering data items before sending them out on the network.

More theoretical and empirical analysis is needed to further quantify the benefits of aggregation. To start, the experiment from 4.1 should be repeated on other platforms to verify that the same conclusions will hold for other systems and network types. Secondly, we need to evaluate aggregation in the presence of non-nearest neighbor communication, network congestion, and overlap of computation and communication.

To estimate whether medium sized buffers will work well in congestion scenarios, we will run tests of the all-to-all benchmark 5.1 with various buffer sizes on a number of systems.

Other issues which need to be examined, both analytically, and empirically include:

- Effects of varying the number of dimensions in the virtual topology
- Quantification of processing overhead due to sending a message
- Quantification of TRAM overhead due to processing at source and intermediate destinations
- Injection bandwidth overhead due to TRAM

# 9 Topology-aware Collective Communication

Up to now, we have primarily assumed communication specified as point-to-point messages, but many communication patterns are specified in terms of collective operations over a large number of processes or objects. We propose to look at how these patterns can be implemented within TRAM to take advantage of its aggregation capabilities. Our first approach will be to implement well established and highly optimized topology-aware algorithms from literature [15], [24]. Our goal is not theoretical or practical improvement on execution time of traditional collective patterns, many of which have been proven to be close to theoretical bounds on performance. Instead, rather than focusing on collective algorithms executed in isolation over the full set of processes in the run, we hope our approach may provide some benefits in less structured scenarios where collectives are executed over subsets of processes or when multiple communication patterns are active concurrently.

By constructing collective implementations on top of a single underlying aggregation library, we hope to attain a more cohesive collective ecosystem, with less code duplication and more centralized control over parameters such as branching factor and buffer size for pipelining. Another benefit of this approach is to allow simultaneous aggregation over concurrent collectives or even point-to-point communication patterns. As part of this work, we will look for applications where overlap across concurrent communication patterns can lead to performance improvement.

As proof of concept and an example of the research aspects of this angle to our work, we have implemented the broadcast collective in TRAM. For an N-dimensional virtual mesh topology, the algorithm proceeds in N stages. In the first stage, the local TRAM instance where the broadcast call was made sends the data item to each of its peers along dimension N-1. In a subsequent step x, every local TRAM instance that received the broadcast data item during steps 1 to x - 1 sends it to all its peers along dimension N - x. After N stages, the broadcast is complete. The construction employed is similar to that of a spanning tree.

Because this broadcast implementation is done within TRAM, it has desirable performance characteristics that are easily ensured. First, as long as the virtual topology matches the physical network topology, no two steps of the algorithm will use any of the the same links of the network (although there is some redundant sending/contention within a single step, see below). Secondly, the implementation allows aggregating data items from multiple concurrent broadcasts or even point to point sends. This allows an all-to-all multicast implemented as a set of TRAM broadcasts to benefit from aggregation of data from the various concurrent

broadcasts. Finally, the broadcast implementation uses most of the same code as the point-to-point TRAM code, with relatively minor extensions.

One potentially interesting aspect of this work is tuning the branching factor of the spanning tree used for the broadcast. As noted above, the algorithm is not bandwidth minimal. In each step, a given sender sends the same data to each of its peers along a dimension. These messages will travel over the same links of the network, using more bandwidth than necessary. An alternative that would minimize bandwidth use would be to replace each multicast within the above algorithm with a ring pattern within that dimension. This would lead to minimal bandwidth usage, but a significant increase in the number of messages along the critical path. An in-between scheme is also possible, where the sends along a given dimension are done using a recursive algorithm in a logarithmic number of steps.

# 10 Machine-specific Network Optimizations

Some collective operations have system-optimized native implementations that rely on specialized hardware or network-level capabilities. As an example, Blue Gene/P has a separate low-latency collective network and the ability to broadcast a packet to a line of nodes without doing store-and-forward. In some cases, optimized low level libraries provide fast implementations for communication and collective primitives. An example is the multisend capabilities on Blue Gene systems available through the DCMF library [20], which, among other things, provide an interface to the line broadcast. Multisends look particularly attractive for the TRAM broadcast algorithm described in Section 9. We plan to add support for fast machine-specific optimizations where appropriate.

# 11 SMP Optimizations

While an appropriate choice of virtual mesh topology in TRAM leads to some SMP-level message combining, the degree of aggregation can in general be further improved with separate schemes at the intranode level. For example, if $p$ processes on an SMP-node send messages to corresponding processes on a nearby SMP-node, a mesh virtual topology would typically lead to $p$ separate messages sent onto the network, as each pair of communicating processes would typically be peers within the virtual topology. Instead, we could say that a given process will be responsible for all the messages sent to a given node. In the above example, one of the processes within the node would collect all the messages, then send a single message to the destination node, where the messages would be scattered to the appropriate destination. Note that this would involve $2p - 1$ messages in total, compared to $p$ in the previous case, but only one message on the network, and so much less bandwidth consumed due to envelope data. The efficacy of such schemes will depend on intranode communication being significantly less expensive than internode communication. If, rather than having separate processes on the same node, one considers threads of the same process, a shared memory approach may be feasible, where the messages are deposited into shared buffers. When undertaking any such scheme, one would have to consider whether locking may be required to allow various threads to concurrently access the shared buffer space. If so, evaluation of the locking overhead would be required.

Another issue to consider is whether the use of a single process or thread to send and receive all the data (for a set of other nodes) may lead to bottlenecks. For example, if each node communicates with just one other node, but the traffic is heavy, then the single thread may become a bottleneck.

Thus, there is often a tradeoff between increasing the amount of combining and overhead due to copying, locking, non-minimal routing, and load imbalance. We plan to investigate these tradeoffs to determine when aggressive SMP-level combining makes sense and when a more balanced approach is preferable.

# 12 Persistent Communication

One-sided communication protocols such as RDMA can be used to send messages at a lower latency and without the copying normally incurred in the runtime system. Due to the time and memory overhead

involved in setting up a channel for one-sided communication, it is best to establish these channels once for recurring, *persistent* communication patterns, amortizing the overhead over multiple sends. When using TRAM, each PE typically communicates repeatedly with a relatively small number of peers. As a result, there is an opportunity to establish persistent channels between TRAM peers, giving additional benefits to communication using TRAM. Note that this possibility does not generally exist without a virtual topology scheme, as the total number of persistent channels required for all the processes in the run would be too high. We plan to investigate the use of one-sided communication for TRAM using an existing Charm++ API for persistent communication, and study the costs and benefits of this approach.

# 13 Generalized Communication Interface and MPI Support

As a module of the Charm++ runtime system, TRAM is currently usable only in Charm++ programs. In practice, this limits its potential set of users to the small fraction of the HPC community that is familiar with Charm++ or willing to learn it. Porting TRAM to MPI would greatly increase its potential user base. Further, we could separate out the small amount of Charm++-specific code in TRAM for communication, determining locations of chare objects, and a few other tasks, from the more generally applicable code for aggregation and routing. In the end, to use TRAM with non-Charm++ parallel applications and benchmarks, one would only need to supply an implementation of the TRAM communication interface using that language. We will furnish implementations of the interface for Charm++, MPI, as well as for LRTS, the underlying messaging layer for Charm++.

As long as TRAM remains a library, the above approach should work relatively well. Our plans to evolve TRAM into an adaptive, self-tuning system that aggregates without user involvement will be more difficult to translate into a generic library. We may attempt a port of this approach to a particular implementation of MPI, such as MPICH.

# 14 Evaluation using Benchmarks and Applications

Implementation and testing of benchmark and application use cases has been an integral part in the development process for TRAM. Benchmarks and applications have motivated the need for TRAM, influenced its design and interface, and served as the ultimate evaluation test of its effectiveness. Some of these programs revealed performance issues with the library. Others identified a need for new interfaces and functionality. The application work also demonstrated some challenges in using TRAM, motivating our plans for automation.

We will continue working with use cases as part of the planned work. In addition to the ones presented in Section 5.2, we will need to explore new use cases for some of the work, such as the collectives aspect.

Some of the use cases we have looked at are not currently scaling to full petascale systems. For EpiSimdemics, TRAM leads to improved performance up to a certain node count on Blue Waters, but then ceases to help compared to the application-specific buffering scheme. With ChaNGa, performance bottlenecks in code unrelated to TRAM prevent scaling beyond 512 nodes of Blue Gene/Q. In both cases, if we wish to see improved application performance at scale, significant work may be required, a lot of which may not directly relate to TRAM.

One area that may yield interesting use cases for TRAM is the Charm++ runtime system itself. A number of fine-grained control messages are used at the level of the runtime system, many of which are sent concurrently with the execution of the body of a program (as opposed to, for example, at startup). The functions performed by these messages include: updating location information for chares, load balancing, collectives book-keeping and corner-case handling, buffer address sends and acknowledgements for one-sided communication, fault-tolerance related traffic, and others. In addition, some use cases exist for which the main driving point is reduction in memory use. For example, for Infiniband networks, a queue pair must be established by the runtime system between every pair of processes in a $P$ process run, leading to a memory

footprint of $O(P)$. In TRAM, where only peers communicate directly, this could be reduced, for an $N$ dimensional virtual topology, to $O(\sqrt[N]{(P)}$.

# 15 Plan for Completion of Work

The work described in the previous sections will be completed over the course of the following months, with December as the tentative deadline for the final examination. Figure 10 shows the plan for completing this work within the allotted time. The plan is ambitious overall, but we have a few options at our disposal for managing the various stages and ensuring a timely finish.

First, while Figure 10 shows the total time estimates for each subproject, we anticipate that some of the components will be collaborative efforts, as they lie at the intersection of research interests and commitments of other people in the group. We have already employed this collaborative approach to obtain many of the application and benchmark results presented in Section 5, where some of the code for using TRAM in benchmarks and applications was written by people who had implemented or were more familiar with the original user-level code. Where research issues are involved, this approach works such that each member of a project emphasizes and contributes to the aspect of the work that is central to his or her research. For example, some of the planned work involves support for non-mesh networks, such as fat trees. Some members of our group are working on abstractions for providing topology-related information about the nodes involved in a job on various network types. Naturally, when porting TRAM to new network types, we will use these abstractions rather than reimplementing the topology discovery process. Another example is work on low-level messaging interfaces, where it makes sense to use existing APIs and collaborate with members of the group who have been working extensively on messaging aspects of Charm++.

Secondly, some of the subprojects in the calendar view may be modified or skipped. For example, while we intend to develop the findings of the planned work into papers, the three papers planned in Figure 10 are probably the maximum, and may be reduced to two. We believe that the most risky aspect of the planned work is automatically determining which messages to aggregate and which to send directly. For this reason, as described in Section 6.3, we have prepared somewhat less ambitious contingency plans should the task of full automation prove untenable.

Third, if absolutely necessary, we may extend the tentative December deadline by a month or more. In the end, we will prioritize and orient our efforts with an eye for what we see are the main goals of this work:

- to advance the state of message aggregation techniques for high performance computing
- to produce a stable, robust, automatic message aggregation and routing subsystem that demonstrates improved benchmark and application performance on most supercomputing systems in use today.

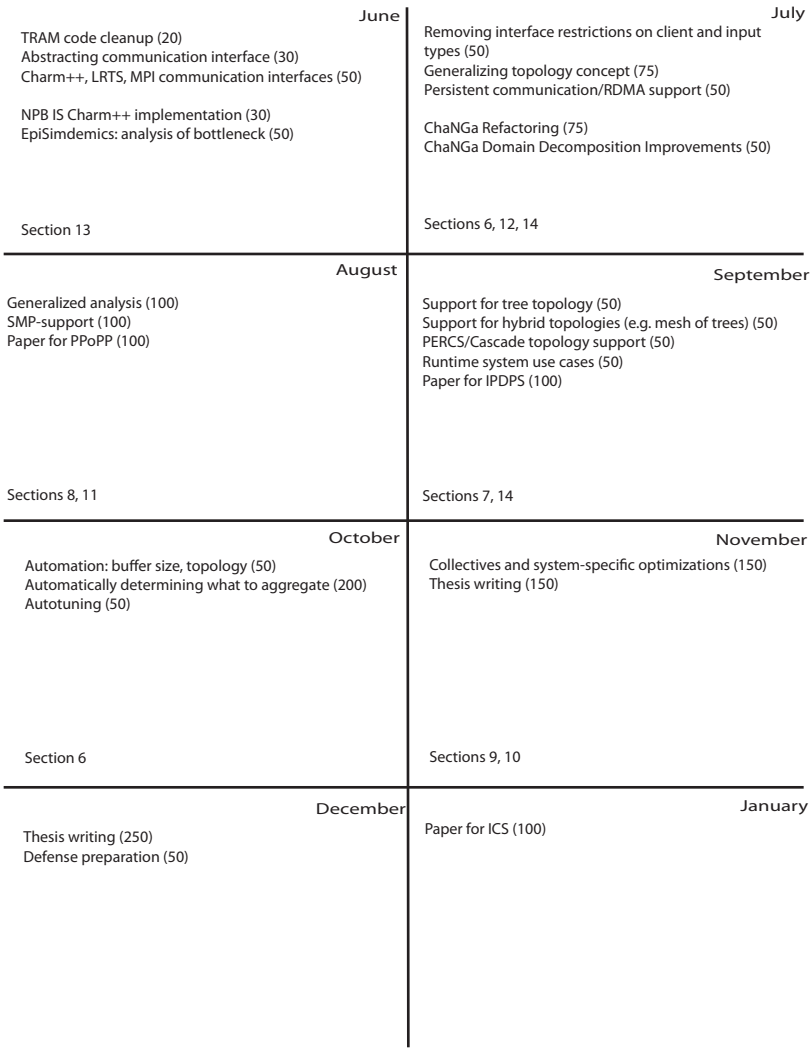



Figure 10: A calendar view of the planned work. Included are the time estimates in hours for each component and the relevant sections in the report for each month's subprojects.

# Bibliography


[1] N. R. Adiga, M. A. Blumrich, D. Chen, P. Coteus, A. Gara, M. E. Giampapa, P. Heidelberger, S. Singh, B. D. Steinmacher-Burow, T. Takken, M. Tsao, and P. Vranas. Blue Gene/L torus interconnection network. *IBM Journal of Research and Development*, 49(2/3), 2005.

[2] Pavan Balaji, Harish Naik, and Narayan Desai. Understanding network saturation behavior on large-scale blue gene/p systems. In *Proceedings of the 2009 15th International Conference on Parallel and Distributed Systems*, ICPADS '09, pages 586–593, Washington, DC, USA, 2009. IEEE Computer Society.

[3] Christopher L. Barrett, Richard J. Beckman, Maleq Khan, V.S. Anil Kumar, Madhav V. Marathe, Paula E. Stretz, Tridib Dutta, and Bryan Lewis. Generation and analysis of large synthetic social contact networks. In M. D. Rossetti, R. R. Hill, B. Johansson, A. Dunkin, and R. G. Ingalls, editors, *Proceedings of the 2009 Winter Simulation Conference*, Piscataway, New Jersey, December 2009. Institute of Electrical and Electronics Engineers, Inc.

[4] Christopher L. Barrett, Keith R. Bisset, Stephen G. Eubank, Xizhou Feng, and Madhav V. Marathe. Episimdemics: an efficient algorithm for simulating the spread of infectious disease over large realistic social networks. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 37:1–37:12, Piscataway, NJ, USA, 2008. IEEE Press.

[5] Abhinav Bhatele. *Automating Topology Aware Mapping for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois, August 2010. `http://hdl.handle.net/2142/16578`.

[6] Abhinav Bhatele, Eric Bohm, and Laxmikant V. Kale. Optimizing communication for charm++ applications by reducing network contention. *Concurrency and Computation: Practice and Experience*, 23(2):211–222, 2011.

[7] Abhinav Bhatele, Nikhil Jain, William D. Gropp, and Laxmikant V. Kale. Avoiding hot-spots on two-level direct networks. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 76:1–76:11, New York, NY, USA, 2011. ACM.

[8] K.R. Bisset, A.M. Aji, E. Bohm, L.V. Kale, T. Kamal, M.V. Marathe, and Jae-Seung Yeom. Simulating the spread of infectious disease over large realistic social networks using charm++. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2012 IEEE 26th International*, pages 507–518, 2012.

[9] Rahul Garg and Yogish Sabharwal. Software routing and aggregation of messages to optimize the performance of hpcc randomaccess benchmark. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[10] Filippo Gioachin, Amit Sharma, Sayantan Chakravorty, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Scalable cosmology simulations on parallel machines. In *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[11] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage switches are not crossbars: Effects of static routing in high-performance networks. In *Cluster Computing, 2008 IEEE International Conference on*, pages 116–125, October 2008.

[12] Torsten Hoefler and Timo Schneider. Optimization principles for collective neighborhood communications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 98:1–98:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[13] Torsten Hoefler, Christian Siebert, and Andrew Lumsdaine. Scalable communication protocols for dynamic sparse data exchange. *SIGPLAN Not.*, 45(5):159–168, January 2010.

[14] Costin Iancu, Parry Husbands, and Wei Chen. Message strip-mining heuristics for high speed networks. In *Proceedings of the 6th international conference on High Performance Computing for Computational Science*, VECPAR'04, pages 424–437, Berlin, Heidelberg, 2005. Springer-Verlag.

[15] Nikhil Jain and Yogish Sabharwal. Optimal bucket algorithms for large mpi collectives on torus interconnects. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 27–36, New York, NY, USA, 2010. ACM.

[16] Pritish Jetley, Lukasz Wesolowski, Filippo Gioachin, Laxmikant V. Kalé, and Thomas R. Quinn. Scaling hierarchical n-body simulations on gpu clusters. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, Washington, DC, USA, 2010. IEEE Computer Society.

[17] L. V. Kale, Sameer Kumar, and Krishnan Vardarajan. A Framework for Collective Personalized Communication. In *Proceedings of IPDPS'03*, Nice, France, April 2003.

[18] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[19] Sameer Kumar. *Optimizing Communication for Massively Parallel Processing*. PhD thesis, University of Illinois at Urbana-Champaign, May 2005.

[20] Sameer Kumar, Gabor Dozsa, Gheorghe Almasi, Dong Chen, Mark E. Giampapa, Philip Heidelberger, Michael Blocksome, Ahmad Faraj, Jeff Parker, Joseph Ratterman, Brian Smith, and Charles Archer. The Deep Computing Messaging Framework: Generalized Scalable Message Passing on the Blue Gene/P Supercomputer. In *Proceedings of the 22nd ACM International Conference on Supercomputing*, 2008.

[21] Sameer Kumar, Yogish Sabharwal, Rahul Garg, and Philip Heidelberger. Optimization of all-to-all communication on the blue gene/l supercomputer. In *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, pages 320–329, Washington, DC, USA, 2008. IEEE Computer Society.

[22] Miao Luo, Dhabaleswar K. Panda, Khaled Z. Ibrahim, and Costin Iancu. Congestion avoidance on manycore high performance computing systems. In *Proceedings of the 26th ACM international conference on Supercomputing*, ICS '12, pages 121–132, New York, NY, USA, 2012. ACM.

[23] National Center for Supercomputing Applications. Blue Waters project. http://www.ncsa.illinois.edu/BlueWaters/.

[24] Paul Sack and William Gropp. Faster topology-aware collective algorithms through non-minimal communication. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 45–54, New York, NY, USA, 2012. ACM.

[25] Yanhua Sun, Gengbin Zheng, Chao Mei Eric J. Bohm, Terry Jones, Laxmikant V. Kalé, and James C.Phillips. Optimizing fine-grained communication in a biomolecular simulation application on cray xk6. In *Proceedings of the 2012 ACM/IEEE conference on Supercomputing*, Salt Lake City, Utah, November 2012.

[26] Yanhua Sun, Gengbin Zheng, L. V. Kale, Terry R. Jones, and Ryan Olson. A uGNI-based Asynchronous Message-driven Runtime System
for Cray Supercomputers with Gemini Interconnect. In *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Shanghai, China, May 2012.

[27] Jeremiah Willcock, T. Hoefler, Nicholas Edmonds, and A. Lumsdaine. AM++: A generalized active message framework. *The International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 09/2010 2010.

[28] Jeremiah James Willcock, Torsten Hoefler, Nicholas Gerard Edmonds, and Andrew Lumsdaine. Active pebbles: parallel programming for data-driven applications. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 235–244, New York, NY, USA, 2011. ACM.