SOFTWARE TOPOLOGICAL MESSAGE AGGREGATION TECHNIQUES FOR LARGE-SCALE PARALLEL SYSTEMS

BY

LUKASZ WESOLOWSKI

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2014

Urbana, Illinois

Doctoral Committee:

Professor Laxmikant V. Kalé, Chair
Professor William D. Gropp
Professor Josep Torrellas
Doctor Pavan Balaji, Argonne National Laboratory

Abstract

High overhead of fine-grained communication is a significant performance bottleneck for many classes of applications written for large scale parallel systems. This thesis explores techniques for reducing this overhead through topological aggregation, in which fine-grained messages are dynamically combined not just at each source of communication, but also at intermediate points between source and destination processors. The performance characteristics of aggregation and selection of virtual topology for routing are analyzed theoretically and experimentally. Schemes that exploit fast intra-node communication to reduce the number of inter-node messages are also explored. The presented techniques are implemented for the Charm++ parallel programming system in the Topological Routing and Aggregation Module (TRAM). It is also demonstrated how TRAM can be automated at the level of the runtime system to function with little or no configuration or input required from the user. Using TRAM, the performance of a number of benchmark and scientific applications is evaluated, with speedups of up to 10x for communication benchmarks and up to 4x for full-fledged scientific applications on petascale systems.

Table of Contents

List of Figures

List of Tables

List of Algorithms

CHAPTER 1

Introduction

Supercomputing networks and communication software are designed to minimize message latency. The focus on low latency implies a best effort attempt to deliver each message 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 communication 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 minimize message latencies, 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 minimum time to reach the destination may be higher 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 in uncongested 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. This shows that in situations with limited resources, the latency-centric approach may exhaust available resources, making the bandwidth-centric approach not just more efficient, but superior even in terms of average latency!

As latency-optimized systems, supercomputing networks (along with most communication libraries for such systems) 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 fine-grained communication.

1.1 Demonstrating the Effects of Fine-grained Communication

Many of the effects of fine-grained communication on performance are evident even in the simplest communication scenarios. As an example, in the ping-pong benchmark, a pair of processes, A and B, communicate over a network. Process A sends a message to process B, and after B has received the original message, it sends a message of the same size back to process A. Dividing the round-trip time at Process A by two yields the *message latency*, the time from when the message send is initiated at the sender to when the message is available at the destination. The second quantity of interest derived from this benchmark is the *effective network bandwidth*, obtained by dividing the message size by the message latency. The ping-pong benchmark is usually run across a range of message sizes to determine how message latency and effective network bandwidth utilization vary with message size.

Figure 1.1 shows the latency results of a ping-pong benchmark for Charm++ (see Appendix A) on three supercomputing systems. In the test, the communicating processes were mapped onto neighboring nodes within the physical system topology. The results follow a well-known pattern:

$$Latency = \alpha + \beta m$$

In this simple communication model, where m is the message size in bytes, the β term denotes the time to transmit a byte of data on the network at full bandwidth. For large message sizes, this term dominates the total latency.

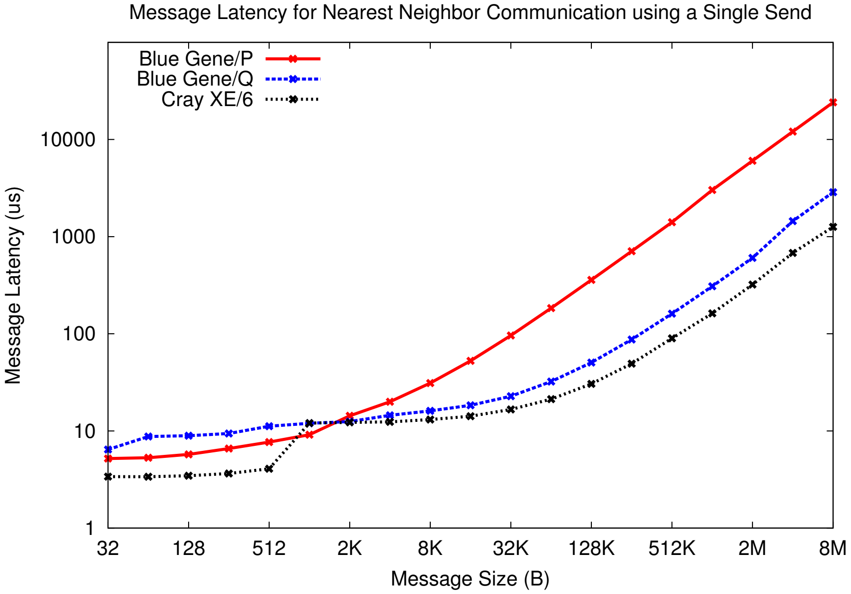


Figure 1.1: Message latency for nearest-neighbor communication on three supercomputing systems.

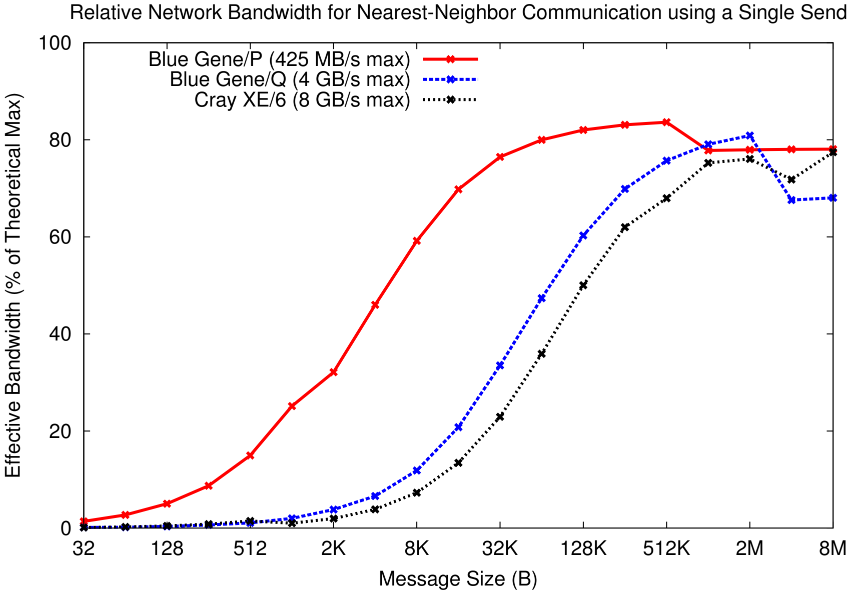


Figure 1.2: Effective bandwidth of nearest-neighbor communication relative to peak link bandwidth on three supercomputing systems. Individual sends of small to medium-sized messages utilize at most a few percent of the network bandwidth.

Sending a message also involves fixed overhead that is independent of message size. This is captured in the model by the α term, also known as the *zero-byte latency.* As denoted by its name, *zero-byte latency* approximates the latency for an empty (or token) message, when the elapsed time consists largely of *communication overhead* incurred in sending and receiving the message. For Charm++ on the three systems chosen for this test, α ranged from 3 to 6.5 μs. Overhead in this range is rarely of concern when considered in isolation. However, when large numbers of messages are sent during the execution of a parallel program, the aggregate overhead summed over all messages can add up to a significant fraction of total execution time.

It is often helpful to consider overhead in relation to other quantities when determining its potential impact on performance. For example, an overhead of $5\mu s$ is equivalent to $10,000$ cycles on a 2 GHz processor. An even more useful comparison is in relation to network bandwidth. For example, on a network with $1GB/s$ bandwidth, a $5\mu s$ communication overhead is equivalent to transmission of more than 5000 bytes on the network at peak throughput. In other words, when sending messages smaller than 5000 bytes on this system, the communication overhead would account for more than 50% of the communication time. In applications that consist primarily of fine-grained communication with little computation, this percentage can translate to a similar proportion of total execution time.

The impact of communication overhead on performance of the ping-pong benchmark can be seen in Figure 1.2, which shows effective bandwidth utilization based on the latency results from Figure 1.1. The results in this plot are displayed as percentages of the theoretical network link bandwidth on each system. For small message sizes, the observed bandwidth utilization is as low as 0.1% of the peak, demonstrating the crippling effects of overhead in fine-grained communication scenarios. It is up to 1000 times faster to send a few large messages than it is to send an equivalent amount of data using fine-grained messages (e.g. $32B$) for the ping-pong communication pattern on these systems!

A more detailed treatment of communication overhead is presented in Chapter 2.

1.2 Utility of Fine-grained Communication

Despite its poor performance characteristics, fine-grained communication is very useful in practice. First, it is often the most direct representation of interactions between objects when using an object-oriented design, where objects (and the messages between objects) tend to be fine-grained. Secondly, the computational requirements of challenges in many scientific domains require scaling problems of constant size to run faster. For applications in these domains, decomposition into fine-grained units is necessary in order to expose enough concurrency to utilize the growing number of processing elements in capability-class supercomputing systems. Programming models are likewise evolving in order to encourage expression of fine-grained concurrency for massively parallel systems. As an example, in the Charm++ parallel programming model, *overdecomposition* of a problem into more objects than there are processing cores is encouraged, as it gives the runtime system more flexibility to hide communication latency by adaptively scheduling available work units. Fine-grained communication is also useful for implementing control messages and acknowledgements for orchestrating data movement in established messaging systems such as the Message Passing Interface (MPI) [1]. For these and other reasons, fine-grained communication is essential, at the level of application as well as the runtime-system. However, as we will see, the expression of fine-grained communication in software does not necessarily have to translate into fine-grained messages at the level of the network.

1.3 Overview of Methodology

The prevalence and high overhead of fine-grained communication combine to make it an important target for optimization. A common theme of most techniques that reduce the ill effects of fine-grained communication is to replace the large volume of small messages with a smaller number of larger messages.

Most parallel programming systems assign the responsibility for grain-size control to application developers. According to this view, a good programmer should be able to adjust the grain size or perform aggregation directly in application code to maximize performance. While grain size control of this kind is often possible, it often complicates the software design and reduces code clarity.

The premise of this work is that fine-grained communication performance can be improved through general aggregation techniques that trade a delay in latency of communication for a reduction in communication overhead. Further, it will be shown how aggregation can be applied at multiple points along the route from the source to the destination using a parameterized topological approach that reduces the number of aggregation buffers and offers further performance improvements.

The approach explored in this work aims to help developers by encouraging expression of fine-grained communication while adding abstractions at the level of the runtime system that aggregate these units into larger messages before transmission on the network. This approach has the natural advantage of increasing developer productivity by providing tools and

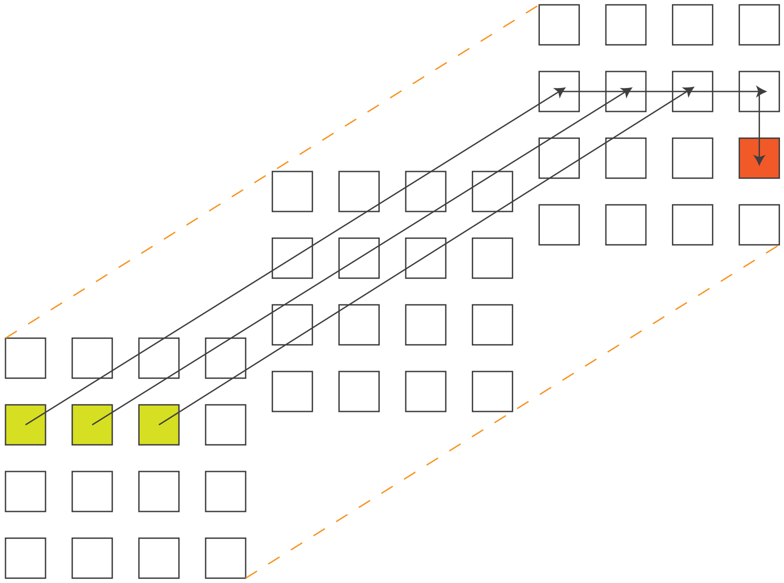


Figure 1.3: Example of routing and aggregation using TRAM. TRAM routes messages along the dimensions of a virtual topology, using intermediate destinations for increased aggregation. The three separate messages in this example are combined at an intermediate destination and delivered in a single message to the destination.

language features that remove the tedium and code bloat of application-specific aggregation approaches.

Toward this goal, we introduce the Topological Routing and Aggregation Module (TRAM) [2], a software communication system for efficiently handling high volume communication patterns, particularly ones involving a high volume of fine-grained communication. Figure 1.3 shows an example of aggregation of messages by TRAM over a 3D topology.

1.4 Organization of Content

Depending on the scenario, message aggregation may be essential, unnecessary, or even harmful. Similarly, while some situations benefit from more sophisticated forms of topological aggregation, in others simple direct aggregation methods work better. For these reasons, this work explores aggregation starting with basic concepts and approaches and building up to more complex methods. Chapter 2 presents an overview of communication overhead, the mechanisms through which aggregation seeks to reduce it, and analysis of communication scenarios that demonstrate the costs and benefits of aggregation. Chapter 3 introduces the idea of topological aggregation as a means to exploit additional opportunities for aggregation that are missed by the direct approach, and motivates the choice of generalized grid topologies for structuring aggregation routes. As an optimization that targets relatively small individual overhead costs for each message, aggregation must be done efficiently in order to yield performance improvements. This is the focus of Chapter 4, which presents the design and implementation of the Topological Routing and Aggregation Module for Charm++. Chapter 5 develops and evaluates techniques that modify the grid aggregation approach to apply additional intra-node communication and more sophisticated software routing mechanisms in an effort to aggregate more effectively. Chapter 6 presents an approach for automatic aggregation of messages based on integration of message aggregation capabilities into the runtime system.

As most of the experiments in this work were done using the Charm++ programming system, Appendix A presents an overview of Charm++ along with some details relevant to this work. As a reference, the usage manual for TRAM is also included as Appendix B. Finally, Appendix C describes the supercomputing systems used for the experiments in this work.

1.5 Summary of Contributions

The main contributions of this work are as follows:

1. **Generalized analysis of conditions required for message aggregation to improve performance**

2. **Comparison of message aggregation on generalized grid topologies to direct aggregation approach, with clearly demonstrated advantages of topological aggregation**

3. **Novel grid topological aggregation schemes that increase intra-node communication in order to aggregate more effectively for inter-node communication**

4. **An approach to automate the aggregation process through integration with the runtime system coupled with dynamic tuning**

5. **Design and implementation of Topological Routing and Aggregation Module, a production topological message aggregation library for the Charm++**

programming system based on the presented concepts, with demonstration of its utility in benchmark and application scenarios

6. **Experimental results showing speedups of up to 4x using the topological aggregation approach via TRAM for a production application, and more than a 2x speedup compared to application-specific direct aggregation at scale**

CHAPTER 2

Message Aggregation

While the concept of *message aggregation* or coalescing is readily understood to signify an optimization involving message combining, the full implications of aggregation on performance are manifold and depend on the specifics of how and when aggregation is applied. This chapter begins with a detailed look at communication overhead, which message aggregation aims to reduce. It then presents an overview of the concepts underlying message aggregation and follows with theoretical and experimental analysis showing how aggregation improves performance.

2.1 Communication Overhead

In the context of this work, messages are buffers of data sent between the processing elements in a parallel program. The purpose of message aggregation is to reduce the number of messages sent during a program's execution, which in turn leads to a reduction in *communication overhead* and an overall performance improvement for applications with fine-grained communication. In this work, communication overhead refers to any resources expended to communicate data, whether within a compute node or on the network. These resources include processing time as well as network and memory bandwidth, all of which are affected by aggregation. Communication overhead is incurred at various levels of the software stack, from the network level, through the runtime system, and even at the application level.

Communication overhead can be classified into *fixed* and *variable* types. Fixed communication overhead is incurred uniformly for every message, regardless of its size. An example of fixed communication overhead is a parallel runtime system's scheduling of an active message at the destination. On the other hand, variable communication overhead depends on the size of the message, and usually increases linearly with message size. Examples of variable communication overhead include serialization of data into a message and copying of message contents. The main benefits of aggregation come from reducing fixed communication overhead. Total variable communication overhead, by comparison, is largely unaffected by message aggregation.

The reduction in fixed communication overhead from aggregation comes at the cost of *aggregation overhead*. Aggregation overhead is the time spent organizing and buffering pending messages for this purpose. In addition, there is a secondary performance effect due to a potential increase in message latencies as a result of buffering. The trade-off between these costs and the benefits due to reduced communication overhead ultimately determines the overall performance impact of aggregation. In order to better understand this relationship, one must evaluate the individual costs and benefits and consider the factors that determine the relative magnitude of each component. The following section presents an overview of this process for communication in the Charm++ Runtime System. For a description of Charm++, see Appendix A.

2.1.1 Fixed Communication Overhead in Charm++

Coordination, data exchange and synchronization in the Charm++ programming model are expressed through communication between objects in a global space. Communication usually takes the form of asynchronous messages sent by invoking entry methods on objects in the global space.

For each entry method invocation in a parallel program, the Charm++ runtime system is responsible for determining the physical location of the destination object in the system, transmitting the message to the destination processor, and delivering it to the destination object. The communication process in Charm++ involves the following series of steps in application code and in the runtime system.

1. **Construction of a message object**

   Charm++ message objects encapsulate the data to be transmitted on the network. Construction of message objects is either done explicitly in user code or inside the runtime system, depending on how an entry method is defined. If an entry method is defined with a message type as a parameter, then the construction of the message object must be done explicitly in user code. On the other hand, if an entry method is defined with a list of parameters of arbitrary type, the construction of the message object takes place inside the runtime system.

Construction of a message object may involve significant fixed overhead in allocating a message buffer in main memory. To reduce this overhead, the runtime system generally preallocates a large pool of memory from which it issues message buffers at a significantly reduced overhead. Regardless, the overhead of message construction can be significant.

2. **Serialization of user data into the message**

   Data that is to be sent over the network must first be serialized into the message buffer. Serialization is done explicitly in user code when sending Charm++ message types, or inside the runtime system for entry methods with parameter lists. In the latter case, serialization is performed automatically through a process called parameter marshalling. Serialization for explicit messages in Charm++ generally involves less overhead than parameter marshalling.

   Serialization of objects into messages can vary in complexity from simple copying of plain old data (POD) to more elaborate schemes for objects with pointers to dynamically allocated memory. In some cases, the data members of an object reference other local objects or runtime system components and must be repopulated at the destination.

   In order to assist the runtime system in serializing complex objects, users are required to specify how the serialization and deserialization are to take place for each communicated data type using the Charm++ pack/unpack (PUP) framework.

   While the overhead of serialization depends on the data type to be serialized, it is typically of at least linear time complexity in relation to the size of the data to be serialized.

3. **Entry method invocation**

   For entry methods with explicit message parameters, invocation of an entry method represents the point in the program when a message is handed over to the runtime system in preparation for sending. For parameter marshalled entry methods, invocation precedes construction of the message object and serialization, which are performed automatically by the runtime system.

   Entry methods are called on proxy objects, whose class definitions are generated by the Charm++ interface translator for each class that implements objects in the global object space. For parameter marshalled entry methods, the proxy class is responsible for creating the message and serializing message contents. After message contents have been serialized, or if a message was passed directly from user code, the proxy object passes the message to the appropriate send function inside the runtime system depending on the type of collection for the destination object (group, nodegroup, array or singleton chare).

4. **Determining location of the destination object**

   Given the index of an object within a collection or other identifying information supplied by the proxy object, the runtime system must determine the destination processor where the message should be sent. In some cases, such as for group and nodegroup object collections, the index of the object denotes its location, as these collections have the semantics of one object created per processing element(PE) and process, respectively. Singleton chares are mapped permanently to particular PEs at the time they are created; hence, the destination PE for a singleton chare is always known to the proxy object. On the other hand, chare arrays support a variety of schemes for mapping objects to PEs, so the destination PE must either be determined by the runtime system from the index using the original mapping function or looked up in a table. Further, array elements may migrate over time, and new elements can be added to the collection. For this reason, the location of the chare array element determined by the runtime system is only a best guess for where the chare is located currently. If the guess turns out to be incorrect, the runtime system is responsible for recognizing this situation after the message arrives at the wrong processor, finding the correct location and forwarding the message there.

   For the above reasons, the fixed overhead involved in sending array messages is generally significantly higher than for groups, nodegroups, and singleton chares.

5. **Populating header fields**

   Every message sent with Charm++ contains a header at the front that includes the information required by the runtime system at the destination to make sense of the contents of the message and deliver the contained message to the appropriate entry method. Prior to sending a message on the network, the runtime system at the source must write the various fields of the header, which is a source of fixed overhead.

6. **Sending the message**

   After the destination PE is determined and the header contents have been finalized, the message is passed to the communication layer of Charm++ for transmission on the network. This layer consists of a handful of functions for sending and receiving messages and managing the required memory resources. A separate implementation of these functions exists for the various network communication libraries (MPI, verbs, PAMI, uGNI, etc.) supported by Charm++. The overhead involved in this step depends on the particular implementation and the network communication library used, and is generally close to the overhead in stand-alone programs written using the same library.

7. **Transmitting on the network**

   After an appropriate send call is issued from within the Charm++ communication layer, the message undergoes final processing by the network software and is then transmitted. Fixed overhead at this stage may involve storing the message in an injection queue prior to transmission and the sending of acknowledgment messages in one-sided protocols to synchronize between the sender and destination (e.g. to indicate when a sender buffer can be freed). Practically all network types today send messages in smaller-sized units (usually called packets). This serves to hide multi-hop latencies and on dynamically routed networks allows the various units to be separately routed to avoid congestion. At the destination, the units are collected together to form the whole message. On many network types, most of the work in this step is done by the network interface card (NIC).

8. **Receiving the message**

   Receiving of a message typically involves issuing a progress call to the network communication library to check for arrived messages. Depending on the mode in which Charm++ is used, specific threads may or may not be allocated for the purpose of sending and receiving messages. Even if a thread is allocated for communication duties, it will split its time between sending and receiving duties and will only be checking for arriving messages for some of the time. As messages are typically received one at a time in this way, there is fixed overhead involved in receiving each message.

9. **Processing received message**

   After a message is received, its header is examined to determine how to handle it. Depending on whether the message arrived from the network or from one of the other PEs within the node, and whether it was sent to a specific PE or requested to execute on an arbitrary PE within the destination node through the nodegroup mechanism, it is placed in one of four intermediate queues.

10. **Scheduling the message**

From the intermediate queues, the runtime system selects messages to schedule for execution. Unless a message is explicitly marked by the user as having high priority, a message picked from the intermediate queue by the scheduler is not executed immediately, but instead placed in a final prioritized queue, the scheduler queue. Subsequently, when a message reaches the head of this prioritized queue, it gets processed for final evaluation.

The two-level queueing mechanism employed in Charm++ for scheduling messages is a significant component of the overall fixed overhead involved in communication.

11. **Deserialization of message contents**

    Prior to further processing, the contents of the message are deserialized, typically in a reverse of the process used at the source for packing the message contents into a contiguous buffer.

12. **Determining the object to receive the message**

    In Charm++, messages encapsulate not just data, but also identifying information for the destination object and its method.

    After the scheduler has picked a message for execution and unpacked its contents, it must find the object that is to receive the message, requiring a table lookup based on the object collection and entry method identifiers. If the message is for a chare array element, determining the destination object subsequently involves a hash table lookup of the index within the collection. Occasionally, the destination object within a chare array will no longer be present locally. This will trigger a forwarding mechanism, where the misdelivered message is sent to a *home PE* responsible for knowing the location of the destination object. From there, it will be sent to the final destination. The overhead of a misdelivered message is thus high, but in practice most messages are delivered correctly on the first try. Even then, the fixed overhead involved in finding the object that is to receive the message at the destination PE is significant. This is particularly true for chare arrays, due to the hash table lookup.

13. **Extraction of entry method parameters**

    After the destination object has been identified, the message is passed to the corresponding method of the proxy object to be prepared for execution. Here, the arguments for the entry method are identified within the unpacked message in parameter-marshalled entry methods.

14. **Execution of the corresponding entry method**

    Finally, the proxy object calls the entry method on the destination object. For parameter-marshalled entry methods, the proxy object passes the unpacked arguments while for message entry methods, the unpacked message is passed directly as an argument.

Taken collectively, the above steps constitute the processing component of communication overhead (or *CPU overhead*, for brevity). CPU overhead is the amount of time a processor is busy while sending and receiving messages. It can be measured using a benchmark that consists of sending a large number of messages in sequence, when almost all processing time will be due to communication overhead. One way to determine this time is to measure the idle time and subtract it from the total execution time. And since all work passes through the scheduler in Charm++, idle time is simply the time when the scheduler's queue is empty.

CPU overhead in Charm++ depends on several factors, including the type of destination for the message, the mode of runtime system operation (see Appendix A), message size, and whether the message is for an object on another node or for a local object.

A summary of communication overhead for 16 B messages on the Vesta Blue Gene/Q system is shown in Table 2.1. Presented here are the results for chare arrays and groups for three modes of communication, for both intra-node and inter-node communication. The Charm++ PAMI network layer was used for these tests [3].

Results show that Charm++ communication on Blue Gene/Q incurs an overhead of at least 1 μs at the source and at least 2 μs at the destination PE. In addition, inter-node communication adds an overhead of 1 μs or more, though this overhead can be hidden by using a separate communication thread. If one considers the time to allocate the message at the source and deallocate at the destination, the overhead increases by a further 1 to 2 μs. In total, the per-message overhead on this system amounts to at least 5μs. This will be the target for optimization with aggregation.

Figure 2.1 and Figure 2.2 show how communication overhead varies with message size for Charm++ communication on Blue Gene/Q. While overhead tends to rise with message size, the overall effect of message size on overhead is non-linear. For example, communication overhead rises only slightly with message size up to 2 KB when using SMP mode. These results further strengthen the case for using aggregation, showing that fixed overhead is the dominant component of overall communication overhead for message sizes up to several KB.

Communication overhead, both fixed and variable, depends significantly on the hardware and/or communication libraries used. As a result, the extent to which communication overhead affects performance, and the extent by which aggregation may improve performance,

	Non-SMP	SMP	SMP with comm thread
Group inter-node	2.0 / 2.6	1.7 / 3.5	0.84 / 2.0
Array inter-node	3.3 / 3.9	2.5 / 4.4	1.5 / 2.6
Group intra-node	1.7 / 2.6	1.0 / 2.4	0.85 / 1.9
Array intra-node	2.5 / 3.7	1.8 / 3.6	1.7 / 2.9

Table 2.1: Charm++ sender/destination communication overhead (in μs) for 16 B messages on IBM Blue Gene/Q.

are system-dependent. As a case in point, Figure 2.3 and Figure 2.3 present overhead plots for Charm++ communication on the Blue Waters Cray XE6 system using the uGNI network layer of Charm++ [4]. In general, communication overhead on this system is significantly lower compared to Blue Gene/Q. Also, there is no option to run Charm++ in SMP mode without communication threads, so much of the total communication overhead is offloaded to these separate threads. These considerations will affect the conditions under which aggregation improves performance on each system.

2.2 Analysis of Aggregation

While the purpose of aggregation is a reduction in the overhead involved in communication, aggregation itself incurs some overhead through buffering of data and through delaying the transmission of buffered messages. Effectively using aggregation requires being aware of these costs and recognizing when they may outweigh the benefits of aggregation.

Toward this goal, this section presents analysis of aggregation for a simplified communication scenario to evaluate the conditions in which aggregation can be expected to have a positive performance impact. Table 2.2 lists the parameters used in the subsequent analysis. The parameters were selected to allow studying the various effects of aggregation while keeping the analysis manageable.

In the model, constant-sized communication items of size m are aggregated into buffers that hold g items of payload. Buffers are sent out as soon as they fill to completion. Each item is generated after a period of c seconds of computation at the source. Processing of each item at the destination takes p seconds. Sending a message on the network, regardless of its size, requires attaching a header (*envelope*) of size e bytes to the payload. Communication of a message involves a fixed overhead of o_s seconds at the source and o_d seconds at the destination. When aggregating, the buffering of each item incurs an overhead of o_a seconds. Network communication involves a fixed time of α seconds per message as well as a bandwidth term of β seconds per byte of data transferred. A total of z items are sent.

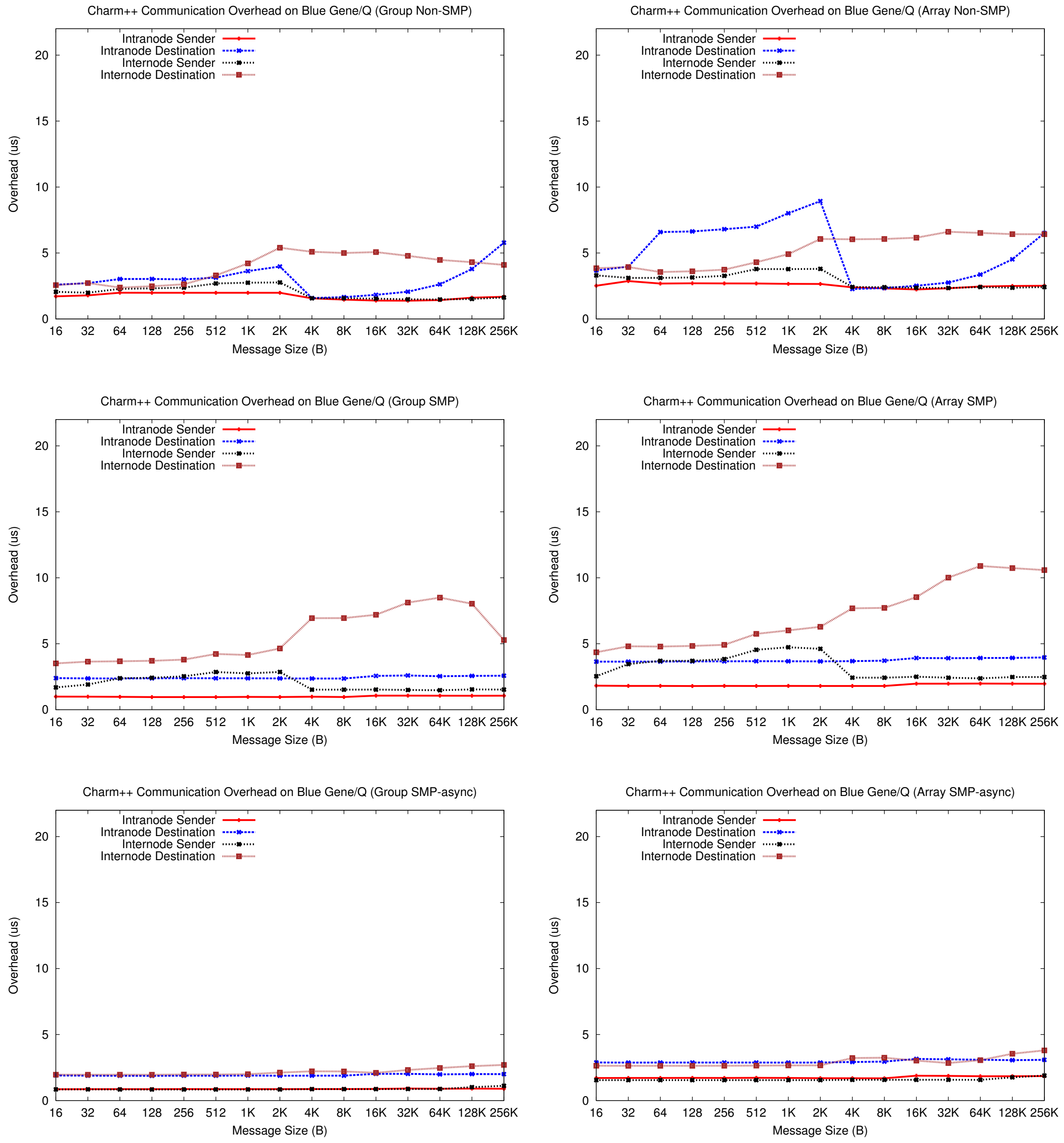


Figure 2.1: Blue Gene/Q communication overhead for Charm++ (without message allocation/deallocation).

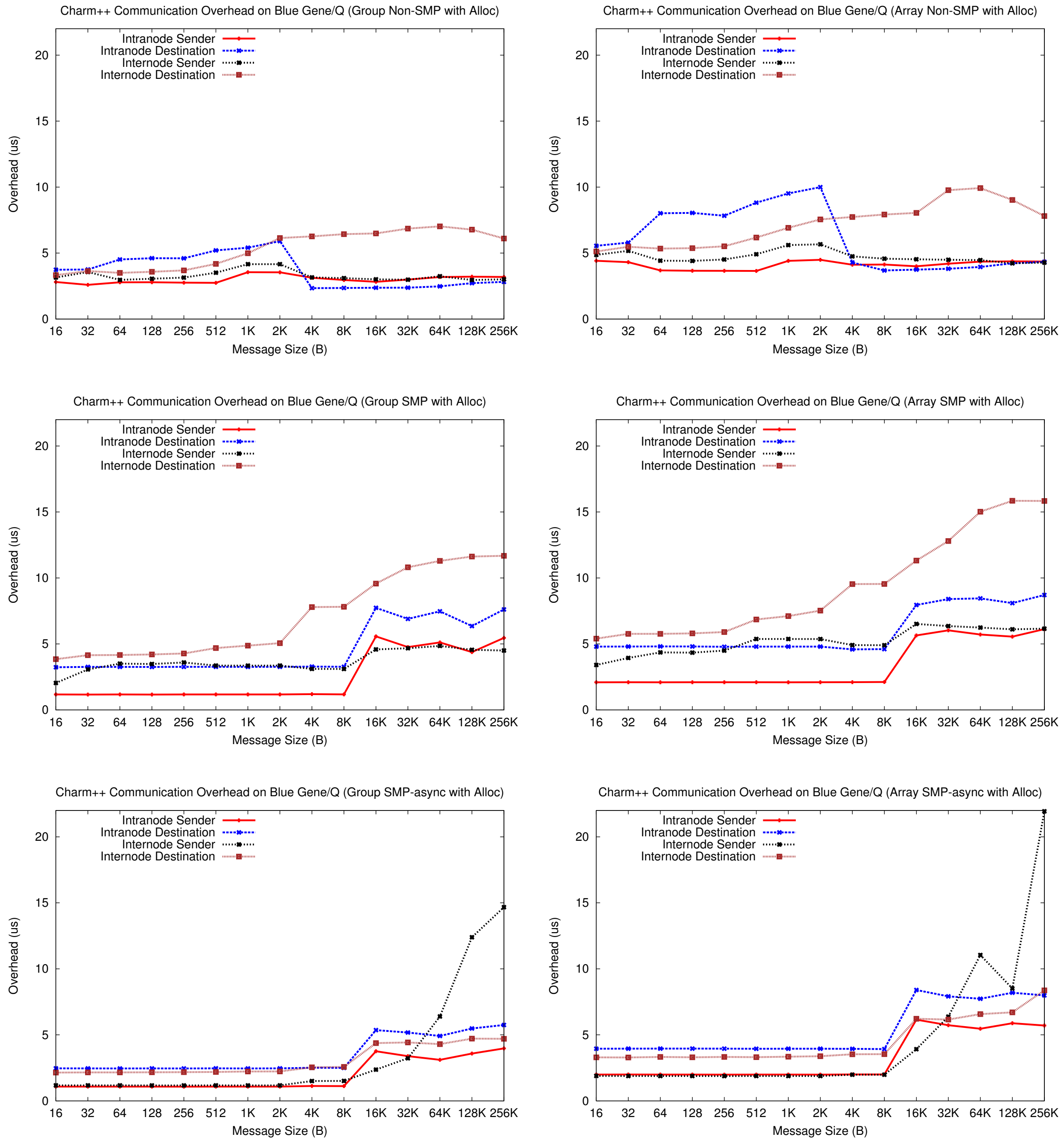


Figure 2.2: Blue Gene/Q communication overhead for Charm++ with message allocation and deallocation time included.

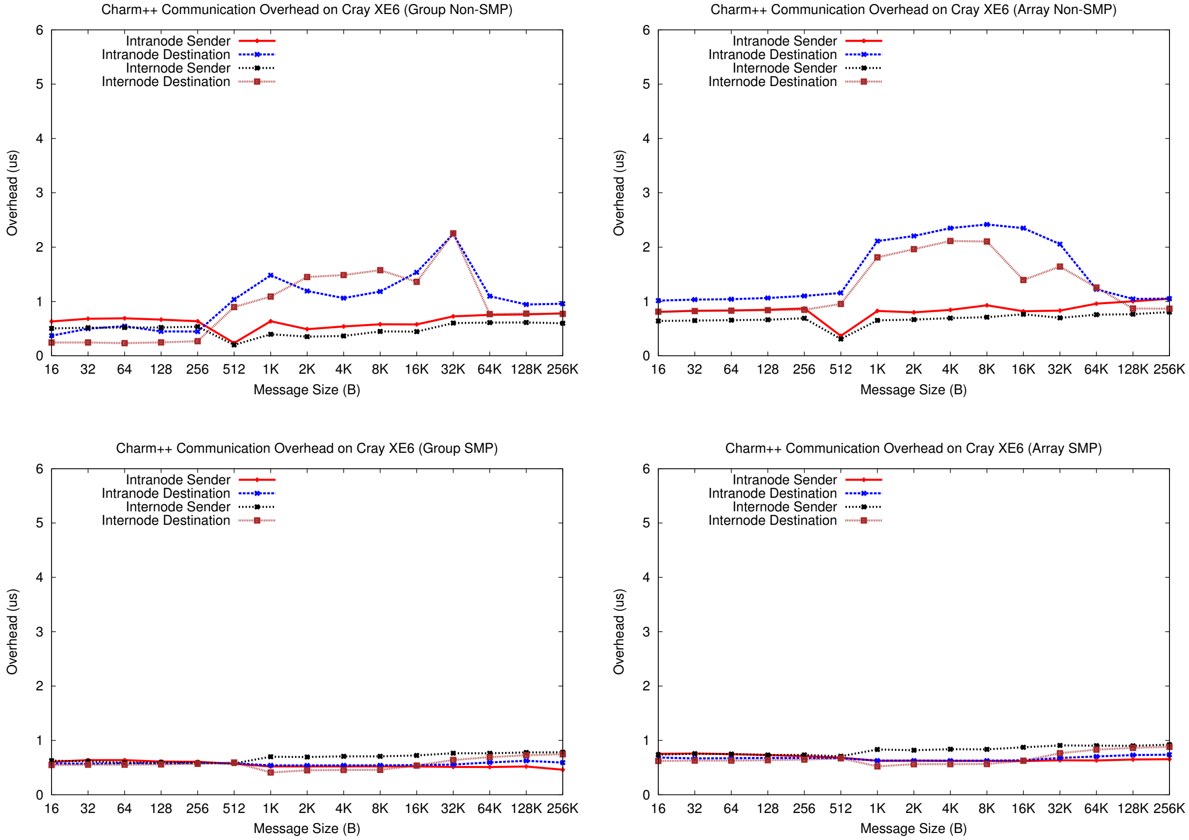


Figure 2.3: Cray XE6 communication overhead for Charm++ (without message allocation/deallocation).

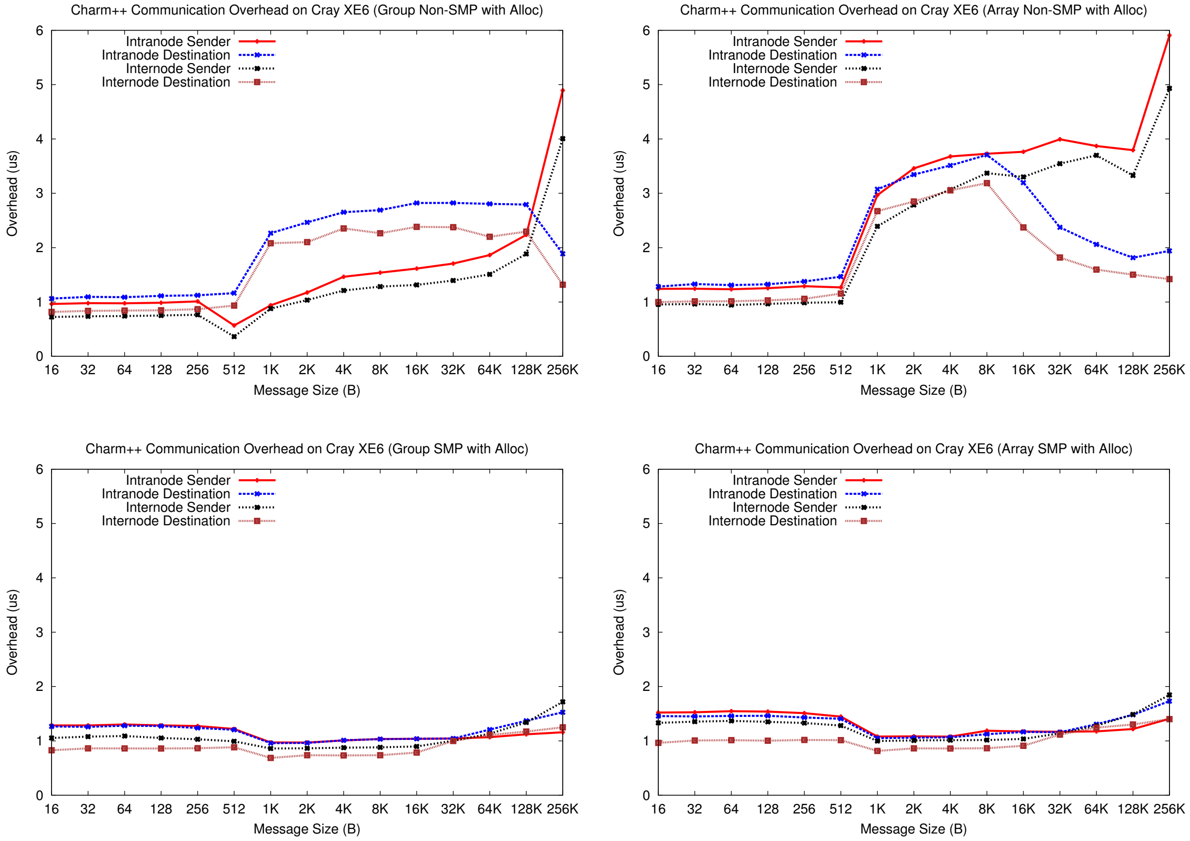


Figure 2.4: Cray XE6 communication overhead for Charm++ with message allocation and deallocation time included.

Symbol	**Description**	**Unit**
m	data item payload size	B
e	size of the message header	B
o_s	fixed communication overhead for a send	s
o_a	aggregation overhead for buffering an item	s
o_d	fixed communication overhead for a receive	s
α	zero-byte network latency	s
β	inverse of network bandwidth	$\frac{s}{B}$
g	aggregation buffer size (in units of data items)	
c	computation time to produce a data item	s
p	time to process a data item at the destination	s
z	total number of data items to send	

Table 2.2: Parameters used in the analytical aggregation model.

Stage	**Description**	**Base Time**	**Time with Agg.**	**Time Per Item (agg.)**
Source	computation to generate data items, aggregation overhead, CPU overhead for sending	$c + o_s$	$g(c + o_a) + o_s$	$c + o_a + \frac{o_s}{g}$
Network	communication time (latency + bandwidth)	$\alpha + \beta(m + e)$	$\alpha + \beta(gm + e)$	$\frac{\alpha+\beta e}{g} + \beta m$
Destination	CPU overhead for receiving, computation to process data item	$o_d + p$	$o_d + gp$	$\frac{o_d}{g} + p$

Table 2.3: Time for each of the three stages in the single source-destination streaming scenario with and without aggregation.

2.2.1 Single Source-Destination Streaming Scenario

The case of a single source PE sending a stream of messages on the network to a single destination is a simple scenario that conveys the effects of communication overhead and aggregation overhead on message latency and overall execution time.

The scenario will be modeled using a three-stage process with one stage each for the source CPU, the network, and the destination CPU. Table 2.3 shows the execution time for each of the stages for both the base scenario with no aggregation and for the case of aggregation at the source. Dividing the stage time by the number of items per message in the aggregation case gives the per item stage time, which is also presented in the table. The per item time for each of the three stages is lower when aggregating than in the base case. For the network and destination stages, this follows directly. For the sender stage, this will be true if:

$$\begin{aligned} c + o_a + \frac{o_s}{g} &< c + o_s \\ o_a &< \frac{g-1}{g} o_s \end{aligned} \tag{2.1}$$

As we will see, aggregation overhead (o_a) is almost always much lower than fixed communication overhead (o_s), so Condition 2.1 can be expected to hold in general.

The second aspect of aggregation that deserves attention is its impact on per-item communication latency. Due to the delay while items are buffered prior to sending and the significant increase in the latency of aggregated messages as compared to messages containing a single item, it is possible for aggregation to increase overall execution time even if it leads to a decrease in per item overhead for each of the stages. As an example, consider aggregating all items into a single buffer, which precludes any pipelining between the source, network, and destination stages. If the three stages take a roughly equivalent amount of time per aggregated message and the per item overhead reduction due to aggregation is low, aggregation in this case would lead to as much as a 3x increase in overall time.

The above effect can be quantified using the model by considering the time to fill and flush the three-stage pipeline as a proportion of overall execution time. Consider a task to be the sequence of operations to prepare, send over the network, receive and evaluate a single message. The complete scenario consists of a sequence of z tasks in the base case and $\frac{z}{g}$ tasks in the aggregation case. For each task, the three stages are executed in sequence. We will assume that a single stage cannot concurrently execute for multiple tasks, with the exception of the latency term in the network stage, which can overlap for consecutive events. The overall execution time (T) will correspond to the time to execute the slowest stage (t_s) z times, plus the time to fill (t_{fill}) and flush (t_{flush}) the pipeline. Further, note that $t_{fill} + t_{flush} \leq 2t_s$.

$$\begin{aligned} T &= zt_s + t_{fill} + t_{flush} \\ T &\leq (z+2)t_s \end{aligned}$$

Similarly, let t'_s be the time for the slowest of the three stages when aggregating and let T' be the overall execution time when aggregating. T' can be bounded as follows:

$$\begin{aligned} T' &= \frac{z}{g}t'_s + t'_{fill} + t'_{flush} \\ T' &\leq (\frac{z}{g} + 2)t'_s \end{aligned}$$

In order to determine the conditions under which aggregation improves performance, we evaluate the following inequality for the case of the bottleneck occurring at each of the three stages in the pipeline. We will assume that the same stage is the bottleneck in the base case as well as when aggregating, but similar analysis can be used to evaluate cases when aggregation leads to a different stage becoming the bottleneck. Simplification of the resulting expressions yields the conditions under which aggregation improves performance:

$$T' \leq T \tag{2.2}$$

Case 1 (bottleneck at stage 1): $t_s = t_{stage1}$ and $t_{fill} + t_{flush} = t_{stage2} + t_{stage3}$

$$\begin{aligned} T' &\leq T \\ \frac{z}{g}t'_s + t'_{fill} + t'_{flush} &\leq zt_s + t_{fill} + t_{flush} \\ \frac{z}{g}[g(c+o_a)+o_s] + \alpha + \beta(gm+e) + o_d + gp &\leq z(c+o_s) + \alpha + \beta(m+e) + o_d + p \\ (g-1)(\beta m + p) &\leq z(\frac{g-1}{g}o_s - o_a) \end{aligned} \tag{2.3}$$

If we look at the components of Condition 2.3, its left-hand side corresponds to the additional time required to transmit and receive a message of g items as compared to a message containing a single item. Meanwhile, its right-hand side is the total per item overhead reduction at the bottleneck stage, where for $g-1$ of every g items submitted, o_s time is saved by aggregation, while o_a time is required to aggregate each item. Condition 2.3 states that in the case of a stage 1 bottleneck, aggregation will improve performance if the overall benefit of aggregation in reducing overhead is greater than the latency cost of sending, receiving, and processing a larger message size in the final stages of the scenario (i.e. when the pipeline is being flushed).

Similarly, we can derive the following expressions for conditions under which aggregation will improve performance in the case of stage 2 and 3 bottlenecks, respectively:

Case 2 (bottleneck at stage 2): $t_s = t_{stage2}$ and $t_{fill} + t_{flush} = t_{stage1} + t_{stage3}$

$$\begin{aligned} T' &\leq T \\ (g-1)(c+p) + go_a &\leq z\frac{g-1}{g}(\alpha' + \beta e) \end{aligned} \tag{2.4}$$

Case 3 (bottleneck at stage 3): $t_s = t_{stage3}$ and $t_{fill} + t_{flush} = t_{stage1} + t_{stage2}$

$$\begin{aligned} T' &\leq T \\ (g-1)(c+\beta m) + go_a &\leq z\frac{g-1}{g}o_d \end{aligned} \tag{2.5}$$

For case 2, the cost of aggregation, expressed on the left hand side of Condition 2.4 consists in generating and aggregating enough items at the source for a full message when filling the pipeline, and processing the items in the final message at the destination when flushing the pipeline. Meanwhile, the benefit due to aggregation (right hand side) consists in reducing the number of messages by a factor of g, for a per item saving of the bandwidth required to send the message header. Note that this assumes that the separate communication items can share the same envelope data. We will later see how that can be realized in practice. The α component of communication time may also get reduced, but in practice, this time is at least partially overlapped between consecutive sends, so the overall benefit will be lower, indicated by the use of α' (s.t. $\alpha' < \alpha$) in Condition 2.4.

Finally, in the case of a stage 3 bottleneck, using aggregation will increase the pipeline fill time by the time required to compute the additional $g-1$ items and aggregate all g items in the first message, as well as by the time to transmit the additional $g-1$ items in step 2. On the other hand, the benefit will consist in lower overhead at the destination, as the fixed communication overhead for a received message (o_d) is avoided for $g-1$ of every g items received, as a result of aggregating g items per buffer.

The analysis confirms the effectiveness of using message aggregation to improve performance in scenarios with large numbers of fine-grained items sent in streaming fashion, where CPU overhead, rather than bandwidth utilization, is the bottleneck. Meanwhile, for bandwidth-limited scenarios, the reduction in header data sent due to aggregation may improve performance, especially if the size of individual communication items is very small. Finally, buffering time plays a role by increasing the latency of items at the beginning or at the end of a series of sends, as demonstrated by the fill and flush times in the analysis. This indicates that the use of aggregation requires care in situations with irregular flow of communication, when “flushes” of the pipeline may be more frequent, as well as situations where the total number of sends is small, when the fill and flush times will figure more prominently in the total time.

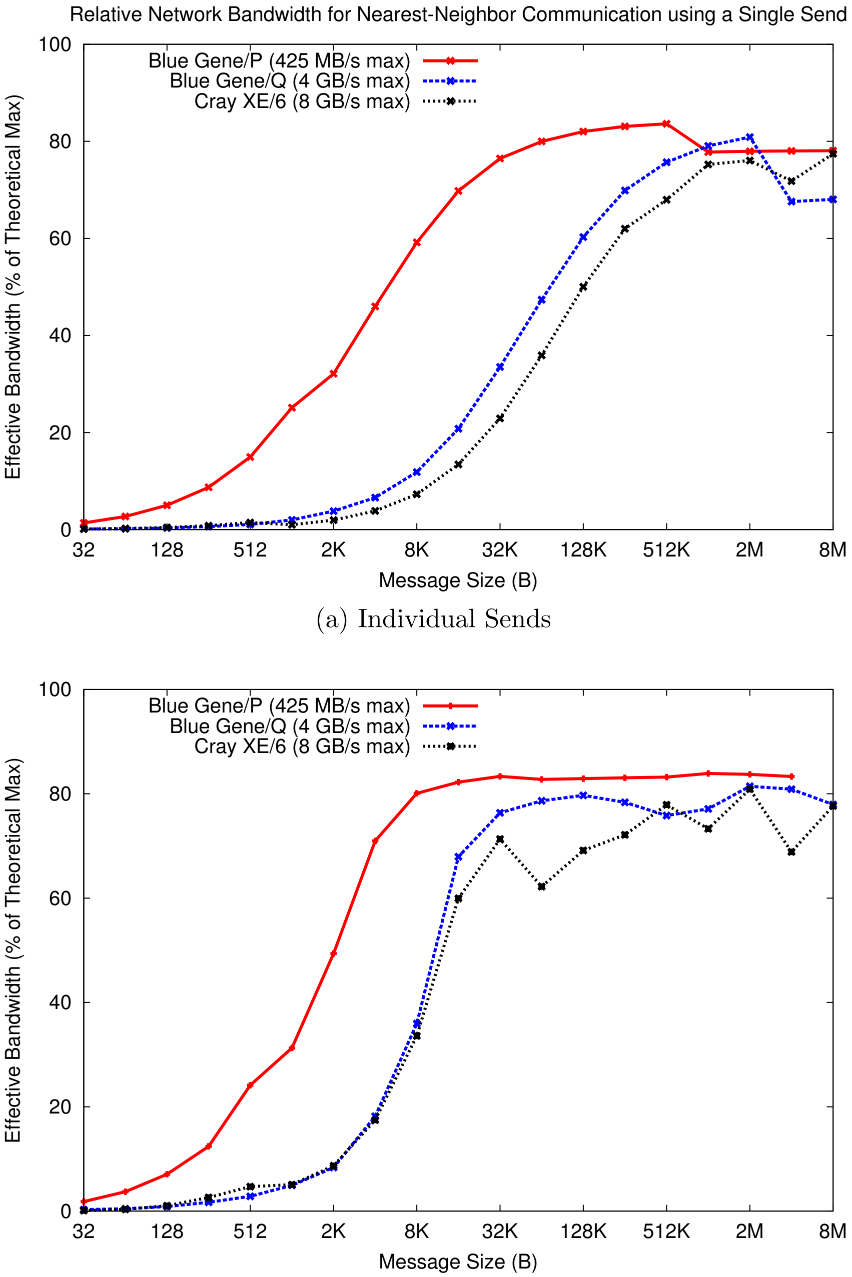
2.3 Experimental Evaluation of Aggregation

To determine appropriate aggregation buffer size and to verify the conclusions of the analysis for the single source-destination streaming scenario, this section presents experimental results for this scenario on Cray XE6, Blue Gene/Q, and Blue Gene/P systems.

2.3.1 Determining a Buffer Size for Aggregation

Figure 2.5a shows effective bandwidth utilization for single message sends of various sizes on the test systems. Results in the plot are represented as percentages of the theoretical network link bandwidth between the nodes. Bandwidth utilization is as low as 0.1% for small message sizes, and rises with increasing message size until coming close to saturating the link bandwidth. For our purposes, we will define the saturation threshold as 75% of the peak *observed* bandwidth, which is about 60% of the maximum theoretical bandwidth between the neighboring nodes on these systems. This point is reached at message sizes of 8 KB, 128 KB, and 256 KB, on Blue Gene/P, Blue Gene/Q, and Cray XE6 systems, respectively. Aggregating messages at or above this size will lead to little improvement in bandwidth utilization. In fact, very large messages may utilize less bandwidth. We believe this to be a result of cache overflow, when memory, rather than network bandwidth, may be a bottleneck. This behavior is apparent for message sizes above 1 MB on Blue Gene/P and above 4 MB on Blue Gene/Q.

A single send, taken in isolation, does not represent the typical state of a network during execution of a large application. In order to more confidently bound the region of messages that will benefit from aggregation in practice, we repeated the benchmark by sending a stream of messages of a given size between the nodes. As was shown in the analysis of the streaming scenario in Section 2.2.1, this should lead to better network utilization for small to medium-sized messages due to concurrent occupancy of network resources by data for multiple messages and pipelining of communication with the send and receive work in the runtime system. The results for this test are shown in Figure 2.5b. Compared to isolated sends, the saturation threshold was reached at lower message sizes when streaming the messages: 4 KB on Blue Gene/P and 16 KB on Blue Gene/Q and Cray XE6. Note that despite streaming, sends of messages below 128 bytes on Blue Gene/P and below 2 KB on Blue Gene/Q and Cray XE6 still attain less than 10% of the available network bandwidth. Aggregating messages of these sizes can provide significant improvements in communication and overall performance.



(a) Individual Sends

(b) Pipelined Sends

Figure 2.5: Comparison of effective bandwidth for individual and pipelined sends on three supercomputing systems. For pipelined sends on each of the three systems, messages of 16 to 32 KB come close to saturating the link bandwidth.

2.3.2 Case Study: Predicting Aggregation Performance on Blue Gene/P

This section experimentally verifies the conclusions of the analysis in Section 2.2.1:

- aggregation improves performance in scenarios with large numbers of streamed fine-grained messages
- latency effects can decrease the effectiveness of aggregation when the number of sends is low

Latency Effects

To study the latency effects of aggregation, 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 2.6a plots the effective bandwidth relative to maximum link bandwidth for this experiment (425 MB/s). As expected, sends using small chunk sizes utilize a small fraction of the network bandwidth.

In Section 2.3.1, we identified 4 KB as a message size that comes close to saturating the bandwidth on this system, so buffers of size 4 KB or more are a good choice for aggregation. However, from the latency analysis of aggregation in Section 2.2.1, we know that the effectiveness of aggregation also depends on the total number of items sent. If the number of items sent is low, so that only a few buffers are sent in total, latency effects will be a significant factor in overall performance. We can see evidence of this effect in Figure 2.6b, which plots effective bandwidth when using messages above the saturation point. For example, sending a total of 1 MB of data in chunks of 8 KB leads to an effective bandwidth of 79% of the theoretical peak, which is close to the maximum observed peak of 82%. By comparison, a single 8 KB message yields just 60% of the theoretical peak, indicating that the network is utilized less efficiently due to latency effects. While latency can definitely be seen to play a role in these results, its significance decreases sharply as the number of messages sent increases. When using $8KB$ messages, the effective bandwidth increases to 76% if just 8 messages total are sent. These results confirm that latency effects do not play a large role when streaming data as long as more than a few messages are sent in total.

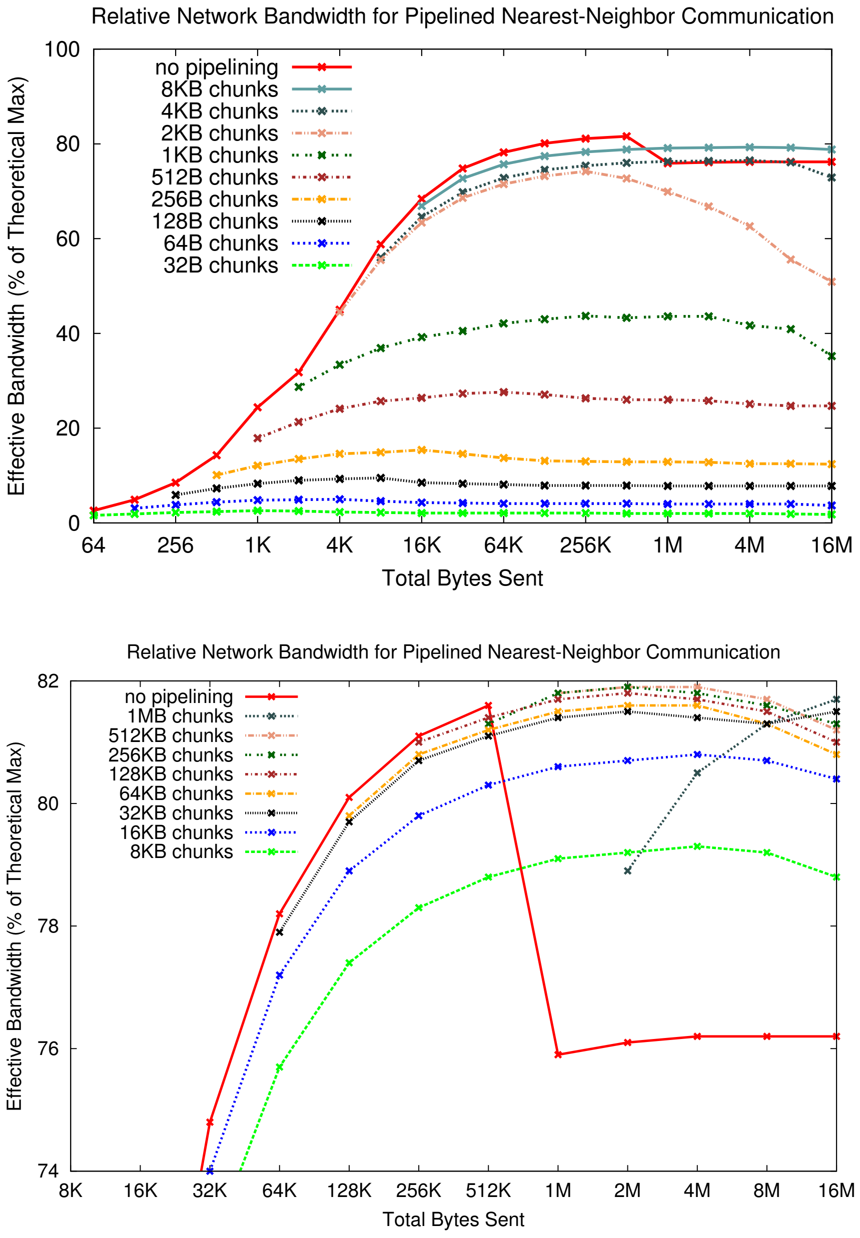


Figure 2.6: Effective bandwidth for individual and pipelined sends on Blue Gene/P. (a) Effective bandwidth of nearest-neighbor communication on Blue Gene/P when sending data using chunks of small size. Results show that fine-grained communication can only utilize a few percent of the available network bandwidth. (b) Effective bandwidth of pipelined sends on Blue Gene/P when using chunks of medium to large size. Messages between 16 KB and 512 KB lead to good bandwidth utilization. Latency effects are apparent in lower effective bandwidth utilization for sends of a small number of individual messages.

Msg Size	#Msgs	#Packets	Bytes Sent	$\frac{1}{2}$ 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 2.4: Packet and byte counts for pipelined communication of an 8 MB buffer on Blue Gene/P in segments of various size. Bytes sent represent the actual number of bytes injected onto the network based on calculations taking into account Charm++ envelope size and Blue Gene/P network implementation.

Predicting the Impact of Aggregation

To demonstrate the combination of communication overhead reduction and envelope data reduction as the two main factors responsible for performance improvement from aggregation, we used linear regression analysis, with the number of sent messages and the total bytes sent (including envelope data) as independent variables for predicting communication time. Our model does not take into account latency effects, but given previous results, it should be applicable as long as the number of messages sent is more than a few.

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 [5].

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 16 bytes of a packet are reserved for link-level protocol information such as routing and destination information, as well as the DMA header. There are several performance implications of this packetization for fine-grained communication. First, it leads to at least 16 bytes of overhead in every 256 bytes sent (6.25% 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.

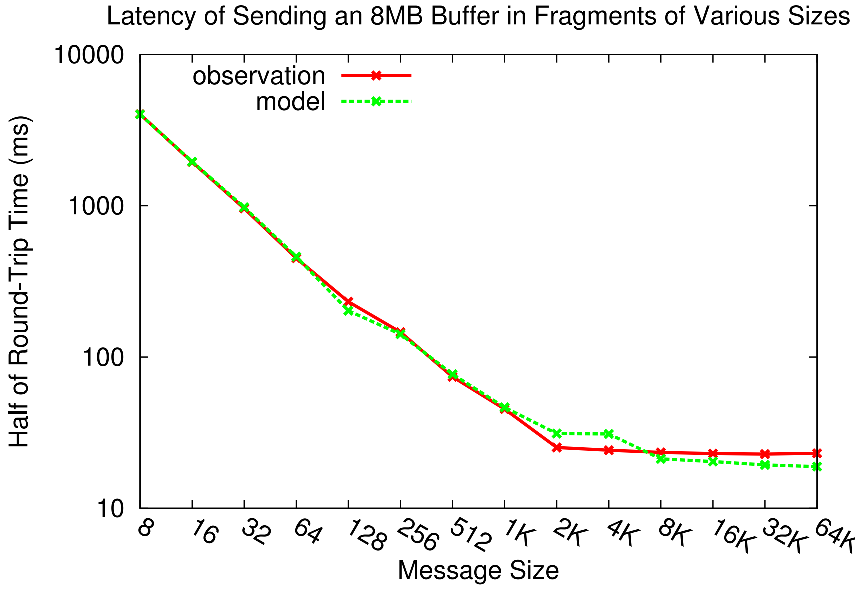


Figure 2.7: Prediction of Blue Gene/P aggregation performance using a least-squares regression model. Plotted is latency of a pipelined send of 8 MB on Blue Gene/P. Increasing the size of the messages used for sending markedly improves performance up to a message size of 2 KB, beyond which bandwidth saturation limits further improvements. The observed results show irregularities, such as at 256 bytes. Using a least-squares regression model with the number of messages, number of packets, and number of bytes sent as the independent variables, we were able to closely match the observed results, showing that together these variables are good predictors of communication time.

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.

Table 2.4 presents total packet counts, bytes sent, and execution time when sending the buffer using a set number of equal sized messages. “Bytes sent” represent the total number of bytes injected onto the network.

Figure 2.7 plots the results for this experiment on Blue Gene/P. Performance improves sharply when increasing the size of individual messages up 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 there are some irregularities in the data. 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 lower speedup when using 256 byte messages.

Figure 2.7 also shows the time predictions of our regression model, which estimates execution time based on a linear combination of three variables: number of messages, number of packets, and number of bytes sent. The regression coefficient of .99995 for our model indicates that overall it represents a good fit for the experimental data. Overall, the experiment provides evidence that the performance impact of message aggregation can be predicted accurately based on simple communication properties (message size, number of packets, bytes sent).

2.4 Summary

Communication in a parallel system, especially on the network, is a relatively complex operation requiring significant overhead. By combining multiple messages into a buffer sent as a single message, the overhead costs of communication can be significantly reduced. The extent of communication overhead (and hence applicability of aggregation) depends on the balance between the relative performance of the CPU and the network and the overhead of communication in the programming language/runtime system used. Analysis of aggregation for a stream of sends shows that the extent of improvement when using aggregation depends primarily on whether aggregation overhead is significantly lower than the per-message communication overhead at the source and destination. In addition, the latency increase due to aggregation can also be of importance, particularly in situations with very sparse or irregular communication behavior.

[1] Note that the bytes sent 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.

CHAPTER 3

Aggregation over Virtual Grid Topologies

The previous chapter presented the performance implications of aggregating at the source for communication between a pair of processing elements. These conditions allowed aggregating items into a single buffer, reducing communication overhead.

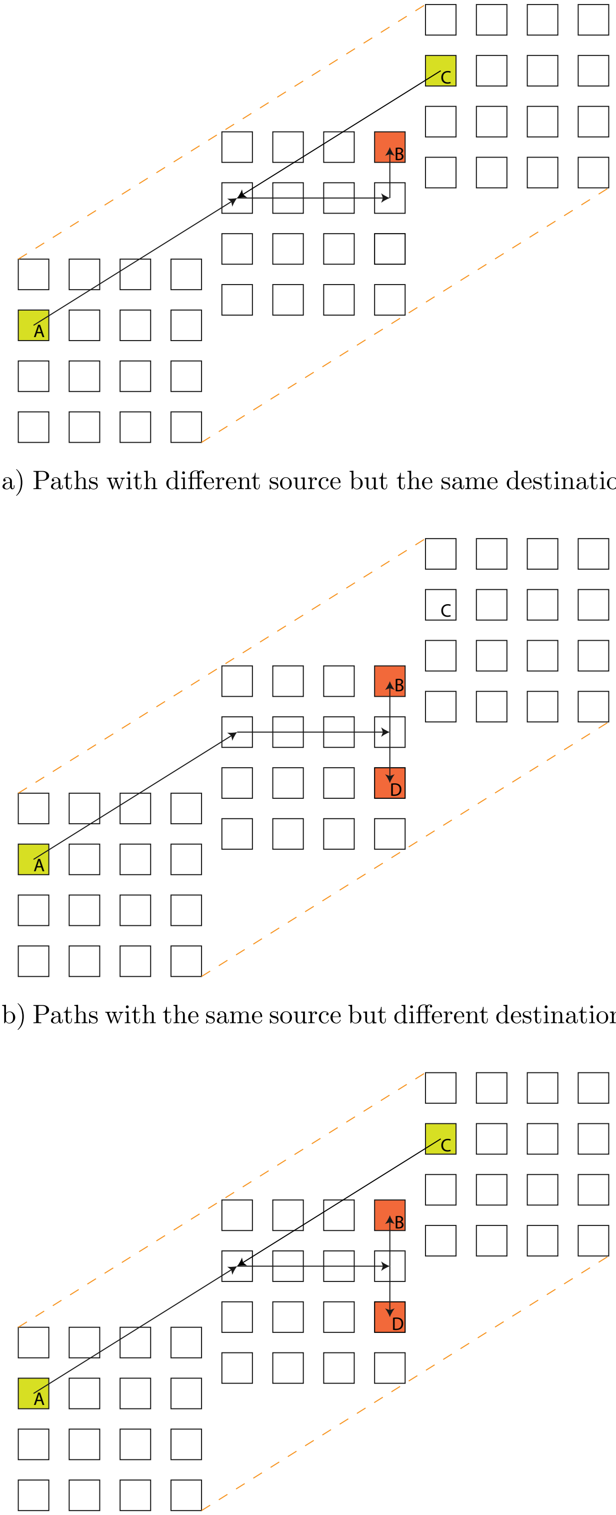
In general, however, items may originate at different source PEs but travel to the same destination, or originate at the same PE but travel to different destinations. Such messages will often pass through the same links along the route to the destination. Even items with different sources and destinations may share some links in their paths. Figure 3.1 shows examples of *partial path sharing*, where paths with different source PE, destination PE, or both, share the same links.

This chapter describes how to exploit partial path sharing by re-aggregating data items along their routes. To achieve this, the path for each item will be partitioned into sub-paths delimited by *intermediate destinations*. At the source and each intermediate destination, items will be aggregated based on the subsequent intermediate destination along the route.

3.1 Aggregation Along Shared Sub-paths

Let G be the graph of the physical topology of the network with edges between the vertices representing the physical communication links and vertices in the graph representing endpoints of communication links. Let $V(G)$ denote the set of vertices in G, and $E(G)$ the set of edges. We define a *communication path* as the series of *edges*

$$v_0v_1, v_1v_2, ..., v_{m-1}v_m$$



(a) Paths with different source but the same destination

(b) Paths with the same source but different destinations

(c) Paths with different sources and destinations

Figure 3.1: Examples of partial path sharing between messages sent on a 3D physical grid topology.

representing the communication links that a data item sent on the network passes through along the route from the source (v_0) to the destination (v_m). The *shared sub-path* for a pair of communication paths

$$w_0w_1, w_1w_2, ..., w_{p-1}w_p$$
$$x_0x_1, x_1x_2, ..., x_{q-1}x_q$$

is the set of all edges uv such that

$$\exists i \in 0...p-1, j \in 0...q-1 \text{ where } u = w_i = x_j \text{ and } v = w_{i+1} = x_{j+1}$$

Shared sub-paths quantify the maximum potential for aggregation under the assumption that every communication path is partitioned into single-edge sub-paths with aggregation applied at every intermediate destination.

Recombination of items along the route typically requires the items to be processed by the runtime system at intermediate destinations. As this leads to significant overhead, selection of intermediate destinations should be done carefully. If too many intermediate destinations are used, the overhead of delivering and re-injecting the data at each intermediate destination could outweigh any benefit due to improved potential for aggregation.

Injection bandwidth utilization is of particular importance when using intermediate destinations. As denoted by its name, injection bandwidth is the maximum rate at which a compute node can inject data onto the network, which is often limited by the processor interconnect (e.g. HyperTransport, QPI) bandwidth between the CPU and the network card. When using aggregation at intermediate destinations, the total injection bandwidth consumed per item of payload data increases linearly with the number of intermediate destinations on the route between the source and destination nodes. Hence, if an application's performance is limited by injection bandwidth consumption, using an intermediate aggregation approach could further degrade performance. These considerations will be further analyzed in subsequent sections.

3.2 Virtual Topologies

One way to formalize the process of selecting intermediate destinations is to utilize a virtual topology spanning the PEs in a parallel run. The selection of a regularly shaped virtual topology allows using the features of the topology to select intermediate destinations that improve the quality of aggregation.

3.2.1 Desired Topology Characteristics for Aggregation

A good choice of virtual topology will exhibit the following qualities:

1. **Intermediate destinations should serve to decrease the number of processing elements with which each processing element needs to communicate.**

   Large parallel runs may involve hundreds of thousands of PEs. Separately aggregating communication for each PE in this setting could fill the entire available memory for just the buffer space. Clever use of intermediate destinations provides a solution to this problem. The set of destinations that a PE is allowed to communicate with directly can be defined to be much smaller than the full set of PEs in the run. The desired effect is for the set of items to all PEs to be aggregated into the smaller set of aggregation buffers for intermediate destinations. Items whose destinations are not in the set of immediate destinations at the source PE can be sent to an intermediate destination that lies along the route to the final destination.

   The above optimization reduces the buffer space at each PE, and hence globally. Further, since items will now be aggregated into a smaller number of buffers, the fill rate of buffers will also improve, which is important in reducing the per item latency delay.

2. **The average and maximum number of intermediate destinations, over the set of all possible communication paths, should be low.**

   Because the overhead of intermediate aggregation increases linearly with the number of intermediate destinations along each item's route, it is important to establish a bound on the maximum number of intermediate destinations along any route.

3. **Each processing element should be an intermediate destination for about the same number of communication paths, leading to a roughly uniform distribution of load and traffic across the topology.**

   The work involved in aggregation and routing can be substantial, and should be distributed uniformly across the set of processing elements in a run so as not to contribute to load imbalance. Further, a uniform distribution of the routing work will also lead to a uniform distribution of the traffic on the network.

   If load imbalance is inherent in the application design, an intermediate aggregation approach may not be able to resolve the imbalance, but should not make it worse.

4. **It should be possible to express the virtual topology compactly.**

   The obvious benefit to being able to specify a virtual topology compactly is that it simplifies the interface for an aggregation library and the set up required in using the library.

   However, there are also other significant benefits. A simple, compact representation of a topology normally translates into clean expression of indices in the topology. This can greatly simplify the code for routing and reduce the overhead involved in determining intermediate destinations.

5. **It should be possible to modify the virtual topology in order to control the trade-off between overhead and aggregation potential.**

   Aggregation requirements can vary for different applications and even across runs on different data sets for the same application. In general, situations where communicated items are numerous require less intermediate aggregation. Virtual topology construction should allow adjustment in the aggressiveness of the aggregation approach so that a simpler approach with fewer intermediate destinations can be used when items are plentiful, and a more aggressive approach in situations where fewer items are communicated.

A brief evaluation of common topology choices shows that many do not adhere to the set of previously outlined characteristics.

Tree topologies work well for some communication patterns (such as broadcasting data from one PE to a set of other PEs), but have significant shortcomings in generalized communication scenarios. On a tree topology, messages are typically routed by sending them first up the tree until a node is reached that is an ancestor of the destination node, and subsequently following the path down to the destination. The logical choice for intermediate aggregation in this design is to aggregate at the nodes that are encountered along the way in this process.

It follows from this description that leaf nodes are never used as intermediate destinations. Further, the closer to the root a node is, the more routes there are for which it is an intermediate destination. For all-to-all communication, one-half of all messages sent would pass through the root node. This type of imbalance is obviously unwelcome, so the tree does not appear to be a good choice as a generalized aggregation topology.

3.3 Construction of a Virtual Topology for Intermediate Aggregation

This section presents a method for constructing topologies that adhere to the constraints listed in Section 3.2.1. The process will consist of starting with simple topologies that are trivially adherent and using them as building blocks to create more complex structures.

The topology will be constructed as a graph where each vertex represents a unique PE and an edge between two vertices indicates that the corresponding PEs can directly send messages to each other. The term *peers* will be used to indicate vertices connected by an edge. The construction will correspond to a virtual topology, so peers are not necessarily neighbors within the physical topology (for this reason, the more natural term *neighbor* was avoided). The constructed graphs in this chapter will be undirected, since an assumption is made that if A is a peer of B, then B is also a peer of A. Chapter 5 will present schemes for which the peer relation is not symmetric.

PEs which are not peers will communicate by forwarding data through vertices that lie on a path to the desired destination. Finally, the construction will be accompanied by a *routing protocol* that specifies how to select intermediate destinations when sending data between vertices that are not peers.

To begin, it is simple to see that a topology corresponding to a complete graph (or *clique*) is trivially adherent. Every PE has a direct connection to every other PE, so all items can be sent directly to the destination (after being aggregated at the source). Since the diameter of this graph is 1, no intermediate destinations are used in this topology.

Using fully connected graphs as the building blocks, we can construct a family of topologies with good aggregation properties by applying the Cartesian product operation on the set of individual component graphs. The following definition and theorem are presented as background.

Definition. The *Cartesian product* of graphs G and H, written $G \square H$, is the graph with vertex set $V(G) \times V(H)$ such that (u, v) is adjacent to (u', v') if and only if (1) $u = u'$ and $vv' \in E(H)$, or (2) $v = v'$ and $uu' \in E(G)$.

Theorem 3.1. *If G and H have diameters d_G and d_H, respectively, $G \square H$, is a graph with diameter $d_G + d_H$.*

Proof:

$d_{G \square H} \leq d_G + d_H$

Given vertices $g_1, g_2 \in V(G)$ and $h_1, h_2 \in V(H)$, a path between (g_1, h_1) and (g_2, h_2) in $G \square H$ can be constructed through (g_2, h_1). The length of the shortest path from (g_1, h_1) to (g_2, h_1) is at most d_G and from (g_2, h_1) to (g_2, h_2) is at most d_H.

$d_{G \square H} \geq d_G + d_H$

Let $g_1, g_2 \in V(G)$ be at a distance of d_G, and $h_1, h_2 \in V(H)$ be at a distance of d_H. Each edge in $G \square H$ is either an edge in a replica of G or an edge in a replica of H. By construction of the Cartesian product graph, a path from (g_1, h_1) to (g_2, h_2) must traverse at least d_G replica edges of G and d_H replica edges of H. ■

In order to simplify the conceptual and visual representation of topologies that correspond to a Cartesian product on a set of complete graphs $K_1, K_2, ..., K_N$, in the rest of this work these topologies will be presented as regular, N-dimensional *grids* constructed by reducing each component clique to a line graph over its vertices, and taking the Cartesian product over these components. To preserve the clique property, it will be assumed that vertices which lie along any line corresponding to a single dimension in the resulting topology are peers. Figure 3.2 presents an example of this process for visualizing a Cartesian product of three cliques.

A formal definition of the routing and aggregation processes on the resulting aggregation topology is presented next.

3.3.1 Routing

Let the variables j and k denote PEs within an N-dimensional virtual topology of PEs and x denote a dimension of the grid. We represent the coordinates of j and k within the grid 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

$$\sum_{d=0}^{N-1} f(d, j, k) = 1. \tag{3.1}$$

In the context of the virtual topology, PEs communicate directly only with their peers.

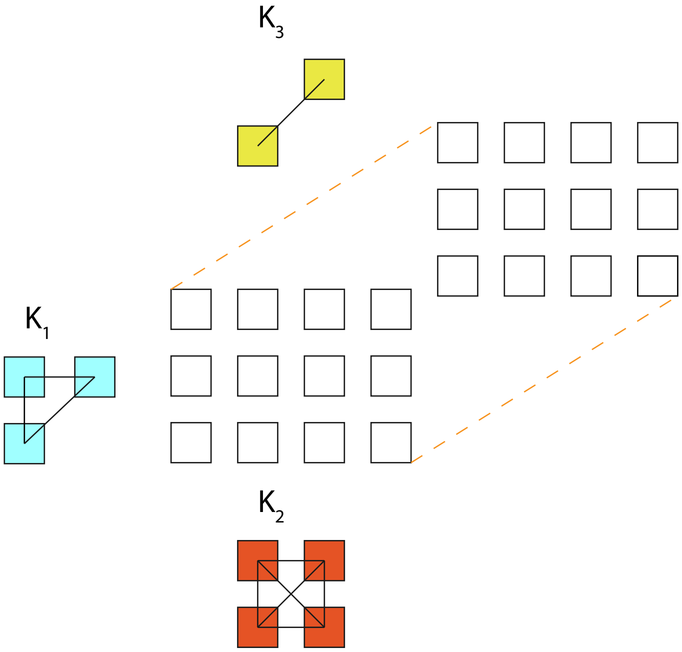


Figure 3.2: Construction of a virtual topology for aggregation. The Cartesian product of cliques can be represented by a grid where every pair of vertices that lies along a line parallel to a coordinate axis is connected by an edge. In this and future figures, these edges will be omitted for simplicity.

Sending to a PE which is not a peer is done 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 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

$$F(j, k) = \sum_{d=0}^{N-1} f(d, j, k). \tag{3.2}$$

3.3.2 Aggregation

Every PE buffers data items that have been submitted locally or received from another PE for forwarding. Because only peers communicate directly in the virtual grid, 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 PE allocates up to $s_d - 1$ constant-sized buffers per dimension, for a total of $\sum_{d=0}^{N-1}(s_d - 1)$ buffers. Note that this is normally significantly less than the total number of PEs specified by the virtual topology, which is equal to $\prod_{d=0}^{N-1} s_d$.

Each data item is submitted at a source PE to be delivered to a specified destination PE. If the destination is a peer of the source PE, the data item is aggregated in the buffer for that peer. Otherwise, an intermediate destination is determined using the previously described algorithm, and the data item is placed in the buffer for the resulting PE, which by design is always a peer of the local PE. Buffers are allocated only when needed to buffer items for a particular peer, and are sent out immediately when they become full. When a message is received at an intermediate destination, the data items it contains 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 delivered.

3.4 Analysis of Grid Aggregation

Topological aggregation with delivery at intermediate destinations involves significant overhead, and in return yields improved potential for aggregation and reduced memory usage. Here we seek to determine when multi-dimensional aggregation is worth the additional overhead compared to a direct one-dimensional aggregation mechanism.

3.4.1 Overview

We first examine the trade-offs involved in using the grid aggregation algorithm, and define bounds on how it affects various performance characteristics. We will use the following parameters in our analysis:

- m: data item payload size in bytes
- e: size of the message header in bytes
- α: constant overhead per message
- β: inverse of network bandwidth

	Direct Sends	**Grid Lower Bound**	**Grid Upper Bound**
Item Latency	$\alpha + \beta(m + e)$	$\alpha + \beta(gm + e)$	$N[\alpha + \beta(gm + e) + g/r]$
Agg. Link Usage	$lz(m + e)$	$lz(m + e/g)$	$lz(m + e/g)$
Injected Bytes	$z(m + e)$	$z(m + e/g)$	$Nz(m + e/g)$
Message Count	z	z/g	Nz/g

Table 3.1: Comparison of various performance parameters for direct sends vs. grid aggregation.

- g: aggregation buffer size (in units of data items)
- r: per buffer rate at which items are submitted for sending
- N: number of dimensions in virtual topology
- l: average number of links traversed by a message
- z: total number of data items to send

Message Latency It takes time for an aggregation buffer to fill as data items are generated by the application, submitted to the library, and copied into the correct buffer. Buffering can be treated as directly adding to the latency of each data item, or the time from submission to receipt of the data item at the destination. In addition, an aggregate message will generally take longer to arrive at the destination after being sent out compared to a single item. As a result, item latency can significantly increase when using grid aggregation. This must be taken into account when selecting the data item types to be aggregated. In particular, latency-sensitive messages along the critical path should not be aggregated.

Bytes Injected/Sent on the Network Grid aggregation typically reduces the volume of data sent on the network by decreasing the aggregate header data sent. It is important for the routing component not to dilate the path of each data item compared to a direct send, as a message that travels over multiple links consumes bandwidth on each link along the route. We will later see how this can be ensured through careful selection of the virtual topology. However, grid aggregation does typically increase the aggregate data *injected* onto the network, as it delivers and later re-injects the same data for every intermediate destination along the route of an item.

Message Count Aggregation using a multi-dimensional approach with intermediate destinations typically reduces the total number of messages by a factor of $\frac{g}{N}$. The most important consequence of the reduced message count is a corresponding reduction in aggregate message processing overhead.

Table 3.1 summarizes the effects of grid aggregation on the above performance parameters with lower and upper bounds for each quantity. The analysis assumes buffers are filled to capacity. Lower bounds correspond to the most optimistic scenario. For example, for item latency, the lower bound is an estimate for the final item inserted into a buffer before it is sent. For the number of messages and bytes injected, the lower bound assumes all data items are delivered to a peer of the source process (i.e. after a single send). The upper bounds are closer to what may be expected in practice.

Note that latency, bytes injected, and total number of messages for grid aggregation are all affected by the number of dimensions in the virtual topology, which determines the maximum number of messages needed to deliver a data item from the source to its destination. The following sections explore what the gains are in return for this significant overhead.

3.4.2 Aggregation Memory Overhead

In the direct aggregation approach, a separate buffer is allocated for each destination PE, leading to a high memory overhead for applications with dense communication graphs. By contrast, the maximum number of buffers per PE in the grid algorithm never exceeds the number of peers for the PE. For a given buffer size g, data item size m, and topology dimensions $s_d, d \in 0...N-1$, the memory footprint for the buffer space is $m \times g \times \sum_{d=0}^{N-1}(s_d - 1)$ per PE. As an example, each PE for a 16×16×16×16 virtual topology will allocate up to $15 \times 4 = 60$ buffers, which should easily fit in main memory and possibly in cache for buffer sizes which we had experimentally determined to be sufficient for good performance (e.g. 16 KB). This offers a strong reason to prefer the grid algorithm to the direct aggregation approach. In the example above, the direct approach would lead to 65535×16 KB $= 512$ MB per core, which may be too high to fit in the main memory of a compute node.

3.4.3 Message Counts

Multi-dimensional aggregation has a dual effect on communication performance. On one hand, it may lead to a faster buffer fill rate by aggregating into a smaller number of buffers. On the other hand, using intermediate destinations increases the number of messages needed to deliver items to most destinations. In the grid algorithm, a data item submitted at a PE j to be delivered to PE k will be communicated using a series of messages whose number is specified by $F(j,k)$ from Equation 3.2. Table 3.2 shows the distribution of this function for a fixed source PE j^* for two example topologies, a 32×32×32 3D topology and a 16×16×16×16 4D topology. For an N-dimensional grid where each dimension has size d, the number of

Topology	**F(j,k)**	**# Nodes**	**% Nodes**
32×32×32	0	1	3.1e-3
	1	93	.28
	2	2883	8.3
	3	29791	90.9
16×16×16×16	0	1	1.5e-3
	1	60	9.2e-2
	2	1350	2.05
	3	13500	20.6
	4	50625	77.2

Table 3.2: Distribution of the number of messages required to deliver a data item using the grid algorithm on two symmetric topologies.

values of k for which $F(j^*, k) = a$ can be counted using the expression

$$\binom{N}{a} \times (d-1)^a. \tag{3.3}$$

Items sent to the current PE are delivered immediately, so the message count for this case is 0.

Results show that a data item sent to a randomly selected PE within these topologies will typically use N messages along the route to its final destination. The implications of this are that for all-to-all communication, the average value of $F(j,k)$ will be close to N. On the other hand, the location of the destination in relation to the source in practical communication scenarios is rarely arbitrary, provided that some care is taken in mapping application-level objects to processors in a topology-aware fashion. For example, for applications with mostly nearest-neighbor communication, $F(j,k)$ will be 1 for most pairs of communicating PEs.

3.4.4 Virtual to Physical Topology Mapping

In scenarios where performance is limited by network bandwidth and communication is uniformly distributed across the network, the bandwidth utilization of each link corresponds directly to overall application performance. Hence, dilation of communication paths in such scenarios will directly increase the aggregate bandwidth use and degrade performance in proportion to the increase of the path length. For this reason, a good design choice for bandwidth-limited scenarios is to select intermediate destinations that lie along a minimal path to the destination.

Consider a data item sent over an N-dimensional grid or torus, and let $a = F(j,k)$ from Equation 3.2 be the number of messages required to deliver it to its destination. In the grid aggregation algorithm, 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 over the virtual topology will be minimal.

It is also possible to reduce the number of dimensions in a virtual topology that matches a physical topology while preserving minimal routing. This can be done by merging any two dimensions that are consecutive in the order of routing. For example, a 4×4×8 topology can be reduced to a 16×8 topology by merging the first two dimensions. A data item sent within the reduced topology obtained in this may will follow the same route as in the full topology while skipping over some intermediate destinations, but may also follow alternate minimal routes due to fewer restrictions imposed by intermediate destinations. As we have seen, intermediate destinations add overhead and require re-injecting data onto the network, but in return allow aggregation of data items which would otherwise be sent separately. In cases where the costs outweigh the benefits, a lower dimensional virtual topology that preserves minimal routing will perform better than a higher-dimensional one. Figure 3.3 shows an example of reducing a 3D topology to a 2D topology while preserving minimal routing.

In contrast to matched or properly reduced virtual topologies, topologies which do not maintain minimal routing can severely degrade performance. For example, assume a random mapping between the PEs 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 PEs in the physical topology, which is $\sum_{d=0}^{N-1} \frac{s_d}{4}$ for a torus and $\sum_{d=0}^{N-1} \frac{s_d}{3}$ for a grid. In other words, 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. Path dilation due to non-minimal routing is clearly undesirable, as it wastes link bandwidth and can increase network congestion. This effect grows in proportion to the size of the physical topology, which affects the number of hops traveled by each intermediate message.

For runs with multiple PEs per compute node, a natural extension of the approach of matching the physical topology is to use an additional dimension in the grid virtual topology for the PEs within a node. As an example, one can construct a 6D virtual topology on Blue Gene/Q, using the 5 dimensions of the network and the PEs within a node as the 6th dimension. The intra-node dimension, when specified as the last dimension of the topology, provides the benefit of some level of intra-node aggregation in the initial routing step before data is sent on the network. The intra-node dimension can also be split into multiple dimensions, as an additional intra-node communication step will not affect minimality of routing. This can be particularly useful on systems where the number of PEs per node is

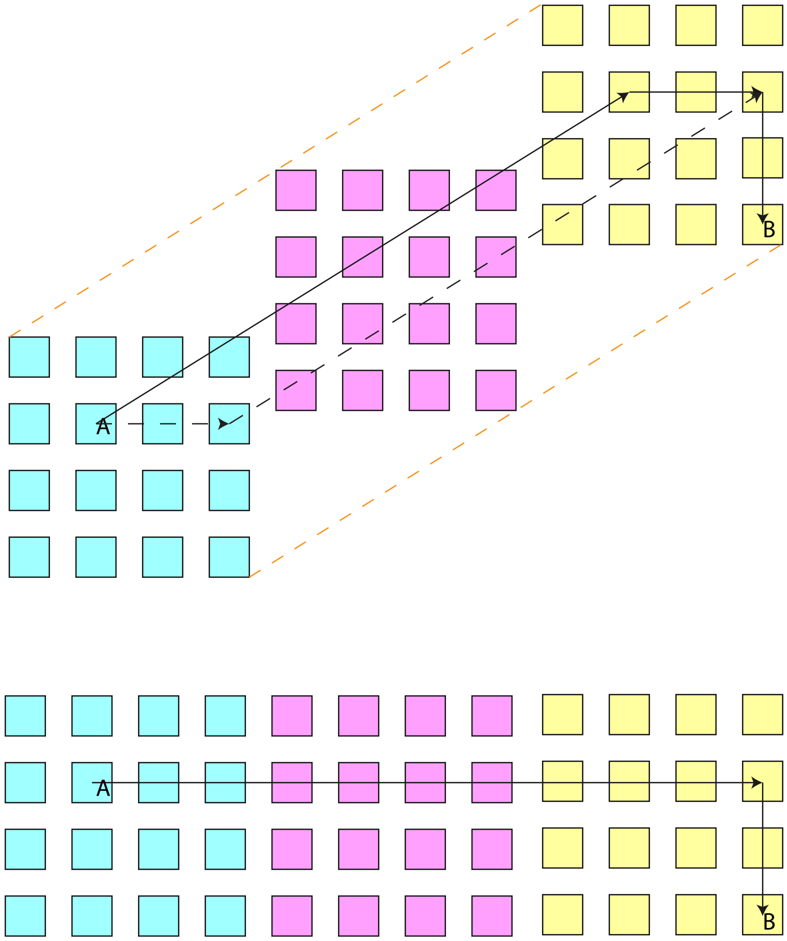


Figure 3.3: Transformation of a 3D virtual topology into a 2D topology that preserves minimal routing. For bandwidth-intensive scenarios, it is important for the intermediate destinations to lie along the minimal route on the network between the source and destination nodes. A virtual topology that matches the physical topology of the network provides a simple way to achieve this. Further, the number of dimensions in the virtual grid can be reduced while preserving minimal routing, as shown in this figure. Here, a 3D topology, whose planes are indicated by different colors, is reduced to a 2D topology by combining the dimensions corresponding to the row and plane of the 3D topology. Using the 2D topology reduces the number of intermediate destinations along most routes by one and introduces less constraints on the path of each item without violating minimal routing. The dashed line shows an alternate minimal route for the 2D topology that the network routing policy may select, which would not have been possible when using the 3D virtual topology.

high. For example, on Blue Gene/Q, where the number of PEs per node can be as high as 64, using an 8×8 virtual topology reduces the number of aggregation buffers for the intra-node dimension from 63 to 14.

For some network types, network switches form a topology that is distinct from the compute node topology. A common example are Clos networks, such as Infiniband networks. These networks have multiple levels of switches, where only switches at the lowest level are directly connected to compute nodes. Matching the virtual topology to the network topology is not possible on these systems. Our approach on these systems has been to use a 2D topology, with compute nodes as one dimension and PEs within a node as the second.

The effects of the choice of virtual topology for a topological aggregation library are examined experimentally using TRAM in Section 4.6.

3.4.5 Load Balance Across Routing Steps

Due to symmetry, the load across PEs in the topological grid aggregation scheme is perfectly balanced for all-to-all communication, even for topologies with unequal dimension sizes. In the latter case, however, the distribution of work across *routing steps* may be different. This section presents analysis of this effect for all-to-all communication. The load balance will be assessed by measuring the potential for aggregating items along an edge in the communication graph.

Let G be the communication graph for an N-dimensional grid topology with dimensions of size $s_d, d \in 0...N-1$. Assume an equivalent amount of data to be sent from each source to each destination. We first note that for a given source PE j_0 and destination j_m within the topology, the grid aggregation algorithm will always route data items along the same *communication path* from j_0 to j_m, which we define as a series of *edges* in the communication graph G

$$j_0j_1, j_1j_2, ..., j_{m-1}j_m,$$

representing messages required to deliver a data item from the source to the destination. We say two communication paths meet if they contain the same edge. Every edge where communication paths meet represents a potential for aggregating items traveling along those paths into a single buffer. We will quantify the potential for aggregation by considering how many distinct communication paths meet for a given edge in the communication graph G.

Consider the number of communication paths that meet at an edge ab along dimension x. As each source-destination pair determines a unique communication path, this can be determined by counting the number of possible source-destination pairs for paths through

ab.

Let $(a_0, a_1, \ldots, a_x, \ldots, a_{N-1})$ and $(b_0, b_1, \ldots, b_x, \ldots, b_{N-1})$ be the coordinates of a and b. Since a and b are peers, $a_i = b_i, \forall i \neq x$. From the routing algorithm presented in Section 3.3.1, the number of possible source PEs for communication paths through ab is the number of PEs whose coordinates for dimensions 0 to x match the coordinates of a. This is equal to $\prod_{d=x+1}^{N-1} s_d$. Similarly, the number of possible destination PEs for communication paths through ab is the number of PEs whose coordinates for dimensions x to $N-1$ match the coordinates of b. This is equal to $\prod_{d=0}^{x-1} s_d$. Consequently, the number of communication paths that meet at ab is equal to the product of the two quantities, which is

$$l_x = \frac{\prod_{d=0}^{N-1} s_d}{s_x} \tag{3.4}$$

Equation 3.4 quantifies the maximum aggregation load per aggregation buffer.

We can also consider the aggregate number of communication paths for a fixed choice of a, but where b can be equal to a or any of the peers of a along dimension x. This corresponds to the total routing load for dimension x at a given PE:

$$L_x = \prod_{d=0}^{N-1} s_d \tag{3.5}$$

Finally, if we add up the number of communication paths over the set of all pairs of peers that lie along a single line in the topology, we get the volume of data sent along that line. For topology-aware mappings of G to physical topology, this will correspond to the bandwidth load:

$$B_x = (s_x - 1) \times \prod_{d=0}^{N-1} s_d \tag{3.6}$$

The above set of equations serves to clarify many aspects of aggregation. First, we see that dimension-specific aggregation load, defined as the number of items sent per PE along a particular dimension, is independent of where the dimension is placed in the routing order. Further, Equation 3.5 shows that dimension-specific aggregation load is the same for each dimension of the topology, regardless of its size. However, for dimensions of higher size, the number of aggregation buffers will be larger, leading to a proportionally smaller load per buffer for dimensions with higher size, as shown by Equation 3.4. Another consequence of a higher buffer count is a larger number of sends, leading to additional communication overhead. As a result, routing steps for dimensions of higher size are likely to involve substantially more work on average. Finally, as shown by Equation 3.6, all-to-all patterns lead to more total data sent along dimensions of higher size. Though it is not a consequence of aggregation, the resulting imbalance in bandwidth load can be a significant factor in overall performance.

3.5 Summary

Direct aggregation approaches can only aggregate together items with the same source and destination. Topological aggregation addresses this deficiency through an approach that partitions paths from a source to a destination into segments and aggregates items traveling to the same intermediate destinations. When the number of processing elements in a run is large, topological aggregation can significantly reduce the buffer space used and improve the fill rate for aggregation buffers. The Cartesian product of complete graphs can be used to define a communication graph with good aggregation properties. These properties include an even balance of work among processing elements, a low degree in the communication graph, which can further be tuned by joining or splitting component cliques, and a structure that can be used to partition paths and deliver items to their destinations within a relatively small number of intermediate steps. This topology can be represented as a generalized N-dimensional grid under the assumption that any nodes that lie along a line parallel to one of the dimension axes in the grid can communicate directly.

CHAPTER 4

Design of a Parallel Grid Aggregation Library

This chapter describes the design of the Topological Routing and Aggregation Module, a parallel communication library for Charm++ based on the grid aggregation algorithm. While Chapter 3 has shown grid aggregation to exhibit good aggregation properties, low memory usage, and uniform load balance characteristics, a successful design is subject to the latency and overhead constraints presented in Chapter 2. We have seen that a message sent on the network entails an overhead of a few μs or less. It follows that the overhead due to aggregation and routing in the library must be comparatively lower in order for the approach to yield substantial performance improvements in practice. As a result, the design of TRAM seeks a balance between support for a variety of application use cases and communication patterns on one hand, and a strong emphasis on efficiency and low overhead on the other hand.

4.1 Structuring Aggregation to Improve Its Effectiveness

When aggregating messages, it is advantageous to perform the aggregation close to the level of the application, where the data to be transmitted is generated. As we will see, this allows for a subsequent reduction in communication overhead at lower levels of the stack and opens possibilities to consolidate communication of data that is common across multiple aggregated items.

All messages consist of two types of data: *header* and *payload*. Payload is the data to be delivered at the destination, while the header encodes the information needed to perform the communication and to correctly interpret the message contents at the destination. The distinction between payload and header exists only with respect to a specific layer in the system stack. For example, at the network level, the header typically consists of the handful of bytes needed by the network hardware for routing and error checking. At the level of the runtime system, the header will include additional information, such as message type, identifiers of the objects that are to receive the message, message priority, and other information needed by the runtime system to correctly interpret the message contents. Higher in the stack, at the application level, messages often include book-keeping information in reference to the data that follows. This application-level information can be viewed as an extension of the header. Figure 4.1 shows a typical layout of header and payload data within the buffer holding the contents of a message.

The size of the network header varies based on the network architecture and the communication protocol used. On Blue Gene/P systems, network header bytes consume 16 bytes of every packet, with each packet consisting of up to 256 bytes, for a 6.25% bandwidth overhead. If one also considers additional bytes for CRC and the overhead of acknowledgment packets, network overhead can reach 12% of the raw link bandwidth [6]. Similarly, on Blue Gene/Q, the overhead due to network header and protocol packet overhead consumes 10% of the raw link bandwidth [7].

The size of the runtime system header or *envelope* will depend on the programming system used. The Charm++ envelope is currently about 80 bytes, with some variation in size depending on the network layer used. This relatively high size is partly the result of the communication model that is based on delivery of messages to indexed objects of arbitrary type. By comparison, the envelope for MPI communication typically includes just the tag, communicator, and source and destination identifiers, and is 32 bytes in size or less.

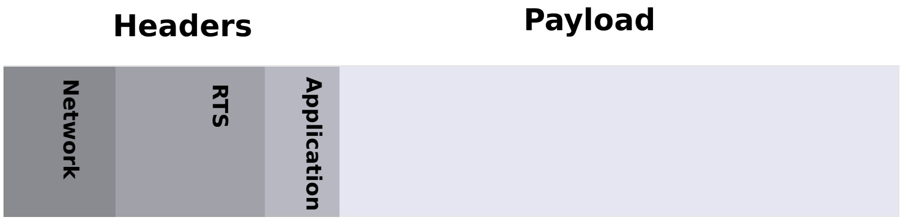


Figure 4.1: Headers and payload in a typical message.

4.1.1 Selection of Stack Layer at which to Apply Aggregation

Before a message is sent, its header data must be populated in order to ensure correct transmission and delivery of its contents. This process occurs at multiple levels of the stack, with each layer typically setting the components of the header that will be processed by the same layer of the stack at the destination. For setting some of these fields, the work involved is minimal. In other cases, this overhead may be substantial (e.g. accessing a hash table in Charm++ to determine the destination PE for a chare array element). Each layer that handles the header data in this way adds CPU overhead to the communication process.

Message aggregation provides an opportunity to remove some of this overhead, but the extent of the improvement depends on the layer of the stack where aggregation is applied. For example, by the time a message reaches the level of the network for transmission, the work involved in setting header fields at the levels of the runtime system and the application is finished, leaving only the network-level work.

In contrast, aggregating messages at the level of the application provides an opportunity to optimize out per message overheads at the levels of both the network and the runtime system. However, for increased programmer productivity, the approach should be general to any message type. Due to these considerations, the approach employed in this work involves aggregation at the level of the application using a library embedded in the runtime system.

4.1.2 Reduction in Aggregate Header Data

In addition to lowering the aggregate communication overhead for processing messages in the runtime system and on the network, aggregation at the level of the application can also be used to decrease the aggregate number of bytes sent. To see why this is the case, consider the single source-destination streaming scenario from Section 2.2.1. The information encoding the destination processor, object, and function translates directly into bytes in the header for each message. Since every item is sent to the same destination object, this information is repeated for every send when not using aggregation.

Suppose that we now decide to aggregate the items generated by the sender into a single message using an aggregation library. If the library is designed to recognize that all items are to be delivered to the same processor, object and entry method, then the redundancy of the headers can be eliminated by sending this information just once as a header in a single message, with the payload consisting of just an array of the item data. An aggregation library that functions in this way does not simply aggregate messages. Instead, it aggregates the *data items* corresponding to the payload of each fine-grained message.

By contrast, a library that treats each item as a general message may be designed to store the destination object and entry method identifier along with each item. Despite its higher overhead, this more general design would allow aggregating together items for different types of destination objects.

Are the gains from being able to aggregate together various types of communication items worth the additional overhead? In most cases, the answer is no. When aggregating items of a single type, the type can be statically specified per library instance, and the buffered data becomes a simple linear array of data items of the same type. This precludes the need to keep track of data types and offsets within the aggregation buffer, and leads to a lower aggregation overhead than in the general approach. To aggregate multiple data types with this approach, a separate library instance can be used for each data type. If the number of such data types is more than just a few *and* they are communicated concurrently, it could potentially be more efficient to aggregate together the various data types, but such cases are rare.

On the other hand, there are cases when a small additional per-item bookkeeping value yields substantial aggregation benefits. For example, in Charm++, it is common when using the chare array construct to have multiple globally addressable objects of the same type on each processing element, all receiving the same type of fine-grained message. When aggregating such items, recording the object identifier for each item allows batching together items for the various objects on the same destination processing element. In this case, the small increase in metadata per item is worth the large decrease in the number of aggregation buffers.

The above considerations form the core of the design for TRAM. These were extended to form a set of specifications for the library, which will be described next.

4.2 Specifications

This section presents the design specifications used for TRAM. Each design decision was made to (a) reduce aggregation overhead (b) improve generality and utility of the library, or both.

1. **Aggregate at the level of the application**

   Communication overhead is incurred throughout the process of sending a message. By aggregating items at the immediate level of the application, the overhead is reduced to the greatest extent possible.

2. **Aggregate into buffers of a constant size**

   As has been shown in Section 2.3.1, message size is a good indicator of the relative overhead of communication. While aggregation of tiny items into medium-sized buffers provides a large reduction in communication overhead, above a certain message size, which was experimentally determined as shown in Figure 2.6b, additional aggregation yields diminishing returns. In TRAM, constant-sized buffers are used, not just for convenience, but more importantly as a controlling mechanism for overall reduction in communication overhead.

3. **Aggregate data items, rather than messages**

   Allocation of a message is a significant component of communication overhead. Instead of allocating a different message for each item and then aggregating these messages, each item can be passed to the library directly along with an index for the destination. The library can subsequently copy the data item into an appropriate aggregation buffer.

   Explicit aggregation of messages has a second significant drawback, as it requires sending the message header data on the network for each item. For items with simple data types, the header data can be significantly larger than the item itself.

4. **Support communication of a single data type per instance of the library**

   For most applications with fine-grained communication, a small number of different data types are communicated concurrently. The restriction to support only a single data type per library instance helps to simplify the code for aggregation and reduces the metadata in the message, improving the efficiency of aggregation for the common case.

5. **Limit each instance of the library to support communication to a specific entry method of a single object collection**

   This is another example of optimizing for the common case to reduce the per-item metadata and simplify the code.

6. **For chare group and array communication alike, aggregate items based on the destination PE index**

   While aggregating separately for each index in the chare array would arguably simplify the design, it is common for multiple elements of an array to be on the same PE, so the additional complexity of this design choice is a small price to pay for a potentially large aggregation benefit.

7. **For chare array communication, determine the PE destination at the source and send it along until the destination is reached**

Due to support for migratability of objects in Charm++, the location of a chare array element in the system may change over the course of a run. Charm++ ostensibly allows determining the present PE for a chare array element based on its index by querying the Location Manager component of the runtime system. Hence, one possible design supporting aggregation of communication for chare array elements is to simply send the chare array index along with the item and determine the destination PE at each intermediate destination from the chare array index.

However, there are two good reasons for preferring a design where the destination PE index is determined once at the source and included with the item for the duration of its communication. First, determining a destination PE using Location Manager is a potentially expensive operation that normally requires a hash table lookup, so it is better not to repeat it at every intermediate destination. By comparison, buffering and sending an additional integer of data for each item entails far lower overhead.

Secondly, it is important to remember that the PE index returned by the Location Manager is only a best guess made using the information available at a given PE. When an element migrates, the only PEs that obtain the updated location information are the source and destination for the migrated element and a special *home PE* for that element, whose purpose is to always store the latest location for the elements it is assigned to track. On every other PE, when a message is subsequently sent to the migrated element, the Location Manager will return information corresponding to the element's location from before the migration. In such cases, the message will arrive at the incorrect destination PE, where the runtime system will determine that the item has migrated. From there, the message will be forwarded to the home PE. At the home PE, the Location Manager forwards the message to the object's current destination PE. Further, the home PE sends a control message to the original source of communication with the updated location, so that future communication from the PE can proceed directly to the correct destination (until the next time the item migrates).

The delivery of items at intermediate destinations presents a problem, as the Location Manager at the intermediate destination may store a different PE index than is stored at the source. In a pathological case, if the intermediate destination has stale location data, and the item's source happens to be an intermediate destination along the route to the stale destination PE, deadlock could occur, with the two PEs continually sending the item between each other. By always using the location obtained at the source PE, this problem is avoided.

8. **Use a class hierarchy to avoid code duplication between group and array versions of TRAM**

   Most of the process for grid aggregation to chare groups and arrays is the same. In TRAM, this common code is defined in a base class. The classes that define group and array aggregation inherit from the base class, and define the components of the code that are specific to each type of object collection. These components include submission and delivery of data items, transfer of item contents into a message, and, for arrays, interaction with the Array Location Manager.

9. **Define the routing and aggregation components of the library as separate classes**

   Isolating the routing and aggregation code into separate classes makes the library more extensible. It allows, for example, the definition of new routing policies without touching the aggregation code.

10. **Copy items once from the application into the aggregation buffer**

    To keep the overhead of aggregation low, it is important not to repeatedly copy item data inside the library. In TRAM, items are passed by reference inside the library and copied once from user code into the aggregation buffer at the source. Further, at each intermediate destination, items are again copied once from the buffer for the arrived message into the aggregation buffer for the subsequent intermediate destination along the route. At the destination, copying is avoided by delivering a constant reference to the object inside the message. Users who wish to store item data after delivery must make explicit copies after delivery.

    The decision to avoid copies inside the library code entailed some development work in order to allow using the same routing and aggregation code for both groups and arrays. While it would have been simple to add support for chare arrays by creating a new data item type, copying the item payload and chare array index into it, and then passing it to the same function used to aggregate group items, this was avoided, as it involved an additional copy into the temporary helper object. Instead, the base aggregator class was written to allow passing handles to item data during the routing phase, and subsequently copying the data item directly into the aggregation buffer by calling a function defined in the derived aggregator class.

    The commitment to avoiding copies influenced another design choice - the decision to aggregate items sequentially by copying the contents of each new item to the end of the current set of buffered items. As the items in an aggregation buffer can in most cases have various destination PEs, this design choice effectively required including a destination PE field for each item in the aggregation buffer. While a data structure that would store the data items grouped by destination PE would allow a potentially more compact representation, it would lead to a more expensive insertion operation, so the simple linear array of items was chosen instead.

11. **Group items and their destination indices into separate arrays**

    When aggregating items, a decision needs to be made whether to place the destination PE index for an item with the item itself or in a separate array. In TRAM, it was decided to store the two types of data in separate arrays inside the message. This was done to decrease the data overhead due to padding inserted between the elements of a data array by the compiler. As the type of destination PE is an *int*, in cases where the element payload size was a multiple of 8 bytes, placing the destination index directly after would lead to a subsequent 4 byte padding before the next item in the array. This padding is eliminated by storing the destination indices in a separate array.

    There is one case when the sending of destination PE indices can be avoided. In the routing stage for the final dimension, all items are sent to the final destination, making it unnecessary to include the array of destination PEs. For this particular case in TRAM, the destination PE array is not allocated or sent.

12. **Support sends of incomplete buffers, but avoid sending the incomplete portion of the buffer**

    When aggregating, it is sometimes necessary to send out a buffer that has not been filled to capacity. For example, at the end of an iteration of sending, when all the items have been submitted, it is likely that buffers will be filled partially. In other cases, it may be necessary to send out partially filled buffers to prevent deadlock, as explained in Section 4.5.

    When sending out partially filled buffers, it is important not to send out the data for the unfilled portion of the buffer. For data consisting of a single array, this simply involves modifying the size field in the message header so that the unfilled portion at the end is not sent out by the runtime system. However, when storing destination PE indices separately from the data items, a partially filled message has two separate gaps, one at the end of each array. As a result, it is not possible to remove all the unused space simply by cutting contents off the end.

When designing TRAM, removing the gaps due to unfilled portions of the data item and index arrays through explicit copying was considered. However, the memory bandwidth overhead involved was determined not to be worth the typically small savings in the size of data sent. The current design avoids sending the unfilled portion of the second array in the message, but does not eliminate the gap between the last element in the first array and the beginning of the second array. As data items are typically larger than the size of an integer, the data item array was placed second in the message, so that the larger of the two gaps is removed from the message in common cases.

While in pathological cases, the double-array design may cause a slowdown, as long as buffers are filled to capacity under steady state operation of the library, the overhead due to the gap will be limited. Given that in typical scenarios buffers will be filled to completion prior to sending, this design choice does not normally present significant performance problems.

13. **Send a single integer PE index per item, rather than a set of coordinates in the topology**

    One important design decision when representing and sending destination PEs is whether to use a simple integer index, or else to split the index into its coordinates along the grid and subsequently send those instead. TRAM uses the coordinate indices for routing, so it may seem that doing this work once and passing the results along subsequently would be better. However, in TRAM, the simple design of sending integer indices was chosen, which has significant advantages in terms of generality.

    First, it is difficult to tell how many bits would be needed for representing each coordinate in the index. While for specific systems and topology choices, the number of bits may be very small, more generally (such as for 2D or even 1D topologies), as well as to allow for some level of growth in the number of processing elements in future systems, at least 24 bits per coordinate may be required. In other words, the index in terms of its coordinates would likely occupy significantly more space in the general scheme.

    Secondly, the work of determining coordinates in the topology from the integer index need not be duplicated at each PE, and in fact can be partitioned so that each coordinate is determined once over the course of an item's route, as described below.

14. **When processing the destination PE index for routing purposes, perform only the work that is required to determine the subsequent destination**

    The decision to send PE indices as integers requires decoding an integer index to obtain its coordinate data in order to route an item. As determining each coordinate generally involves a division operation that can be relatively expensive on some architectures, it would be inefficient to determine the full set of destination coordinates at every intermediate destination along the route of an item. Fortunately, that is not necessary, as the routing algorithm (see Section 3.3.1) compares current and destination coordinates one at a time and selects an intermediate destination as soon as a coordinate is found that differs. This is exploited in the TRAM design by decoding and comparing individual coordinates one at a time. This can be done efficiently using a set of precomputed values for products of dimension sizes. Further, the routing algorithm always routes items along dimensions in the same order. By marking each message with a tag that indicates the dimension it was sent along, the routing process at the subsequent intermediate destination can pick up with determining only the subsequent coordinates that have not been matched at previous destinations. In this way, each coordinate in the index for an item is determined once over the whole path of an item, rather than at each intermediate destination.

15. **Buffers should be allocated as messages, so that they can later be sent without additional data copies**

    Allocation of a message and serialization of data into the message are high overhead operations. Fortunately, these costs need not be paid prior to sending an aggregation buffer. As long as the original buffer is allocated as a message of the appropriate type, it can be handed directly to the runtime system for sending.

16. **Send buffers when they fill to capacity**

    As has been shown in Chapter 2, the per item latency increase due to aggregation is a significant cost that in some cases may outweigh the benefits of aggregation in reducing communication overhead. For this reason, it is important to send out aggregation buffers as soon they reach capacity, rather than periodically in bulk synchronous fashion.

17. **Allocate a buffer only when needed to buffer items for the corresponding PE**

    Depending on the communication scenario, some peers may not be sent any items. For example, if all communication is strictly to nearest neighbors, only a subset of the peers for each PE will be sent messages. For this reason, TRAM allocates messages only when needed to buffer items for the corresponding peer.

18. **Avoid function call overhead, even for inherited classes**

The use of inheritance in TRAM serves a good purpose in improving the modularity of the design and reducing code duplication. The routing component of TRAM is encapsulated into an abstract class. This simplifies definition of new routing schemes and reduces code duplication. Unfortunately, inheritance also presents some performance challenges, of which one example is function inlining.

Inlined functions are used extensively in TRAM to preclude function call overhead. While inlining is usually straight-forward, it does not normally work with virtual functions, due to the virtual function table lookup that needs to happen in such cases. In order to allow inlining function calls to the router instances, the use of dynamic polymorphism through abstract classes with virtual functions was replaced with static polymorphism using the Curiously Recurring Template Paradigm (CRTP).

4.3 Design Overview

Topological Routing and Aggregation Module is a Charm++ library for optimization of fine-grained units of communication in parallel applications. The library implements the grid aggregation approach for communication between the processes involved in a parallel run.

A TRAM instance consists of an aggregation component and a routing component. The aggregation component exposes the application programming interface for item sending and delivery as well as initiation and termination of a communication step. The aggregation component also allocates buffers, aggregates items into these buffers, sends and receives the aggregated messages, and performs periodic progress checks, if required by the application. The routing component, on the other hand, defines the virtual topology over the processing elements and locates PEs in the virtual topology based on a linearized index of each PE. As denoted by the name, it also performs routing - given a destination PE for an index, it produces the index of a subsequent PE along the route to the destination that is a peer to the current PE.

TRAM is implemented as a Charm++ 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 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.

When running on a system with a physical torus topology, it is often a good idea to define the virtual grid topology for TRAM based on the dimensions of the physical topology of the network for the job partition. This can easily be accomplished using a separate module of the runtime system called Topology Manager [8].

4.4 Usage Pattern

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 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** After termination of a communication step, the library instance is not active. However, re-initialization using step 2 leads to a new communication step.

4.5 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 B to TRAM from user code depends on the delivery of another data item A sent using the same TRAM instance. If A is buffered inside the library and insufficient data items are submitted to cause the buffer holding A to be sent out, a deadlock could arise. With the periodic dispatch mechanism, the buffer holding A 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 in the same order as used for routing. 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 one 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.

For more details about TRAM and a description of its interface, see the manual for the library in Appendix B.

4.6 Effect of Virtual Topology

One of the key decisions required when constructing a TRAM instance is selection of a virtual grid topology for topological aggregation. As discussed in Chapter 3, the choice of a virtual topology has significant performance implications. Grid topologies with fewer dimensions are characterized by fewer intermediate destinations along the route of items, leading to a lower per-item aggregation overhead. However, as the number of peers per PE can be high in low-dimensional approaches, substantial buffer space is required for these choices of topology. Further, due to the large buffer space, cache locality when using low-dimensional grid topologies for unstructured or random communication patterns will be poor.

Higher-dimensional grid topologies, on the other hand, allow aggregating into fewer buffers, leading to a more compact buffer space and higher buffer fill rate. This comes at the cost of additional per-item routing and buffering overhead due to more intermediate destinations along the route of each item. The need to re-inject items onto the network at each intermediate destination may also cause problems in systems where injection bandwidth is relatively low compared to network bandwidth.

The above trade-offs are demonstrated in Figure 4.2 and Figure 4.3, which show the effects of the choice of virtual topology on performance for runs of the streaming all-to-all benchmark on the Vesta Blue Gene/Q system.

The streaming all-to-all test consists of each PE sending a set of communication items, each of size 32 B, to every PE in the run. The sends are done concurrently across PEs, and the order of destinations at each PE is randomized. Sends of more than 32 B are performed in rounds at each sender, where in each round a single item is sent by the sender to each

Topology	**Grouping per Dimension**				
	1	2	3	4	5
1D	ABCDET				
2D	ABCDE	T			
3D	ABC	DE	T		
4D	AB	C	DE	T	
5D	AB	C	DE	$\frac{T}{4}$	4

Table 4.1: Virtual topology specification on Blue Gene/Q. Job partitions on Blue Gene/Q systems are assigned in regular 5D grid or torus units defined by five dimensions: A, B, C, D, E. A sixth dimension, T, specifies the number of PEs per compute node. This table presents an example process for creating grids with 1 to 5 dimensions by combining dimensions in the partition topology. The process preserves minimal routing by only combining consecutive dimensions.

destination PE. The TRAM version of the benchmark aggregates and routes items using the grid aggregation approach, while the base version without TRAM simply sends each item as a message. For these tests, an aggregation buffer size of $16KB$ was chosen, which corresponds to the experimentally determined saturation point from Figure 2.6b for the Blue Gene/Q network. The test was executed on node counts ranging from 1 to 512 using the SMP mode of Charm++, with 16 PEs per node, corresponding to the number of physical cores per node.

The topologies were selected by reducing the physical network topology for a job partition in a manner that preserves minimal routing, using the process described in Section 3.4.4. Table 4.1 shows how dimensions in the job partitions were merged to arrive at each topology specification.

As shown in the figures, using aggregation for fine-grained communication on this system is essential, leading to a speedup of 10x or more. The direct, non-topological aggregation approach, with separate buffers for each destination PE, is marked as "1D" in the figures. As can be seen, 1D aggregation performs reasonably well when the number of PEs is very low or when the total amount of data sent is very high, since buffer fill rate is of secondary importance in the latter case. On the other hand, it does very poorly compared to the topological schemes for 8-node and larger runs, particularly when the amount of data sent per destination is low. For such cases, direct aggregation is not able to generate sufficient aggregation opportunities to significantly improve performance.

In general, for all-to-all tests with up to 2 KB sent per destination, the 3D virtual topology

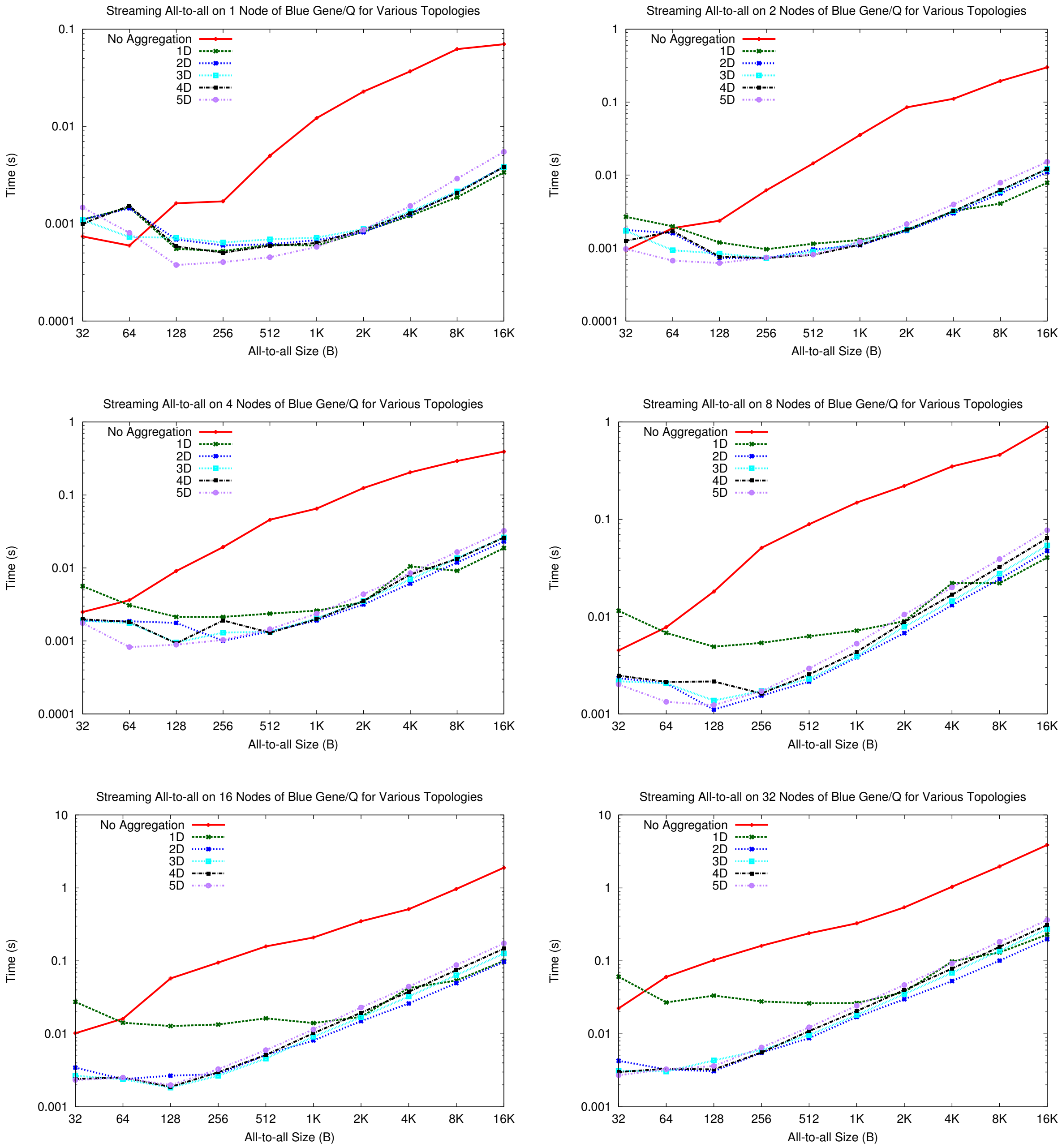


Figure 4.2: Blue Gene/Q fine-grained all-to-all performance for various grid topology aggregation schemes in runs on small node counts.

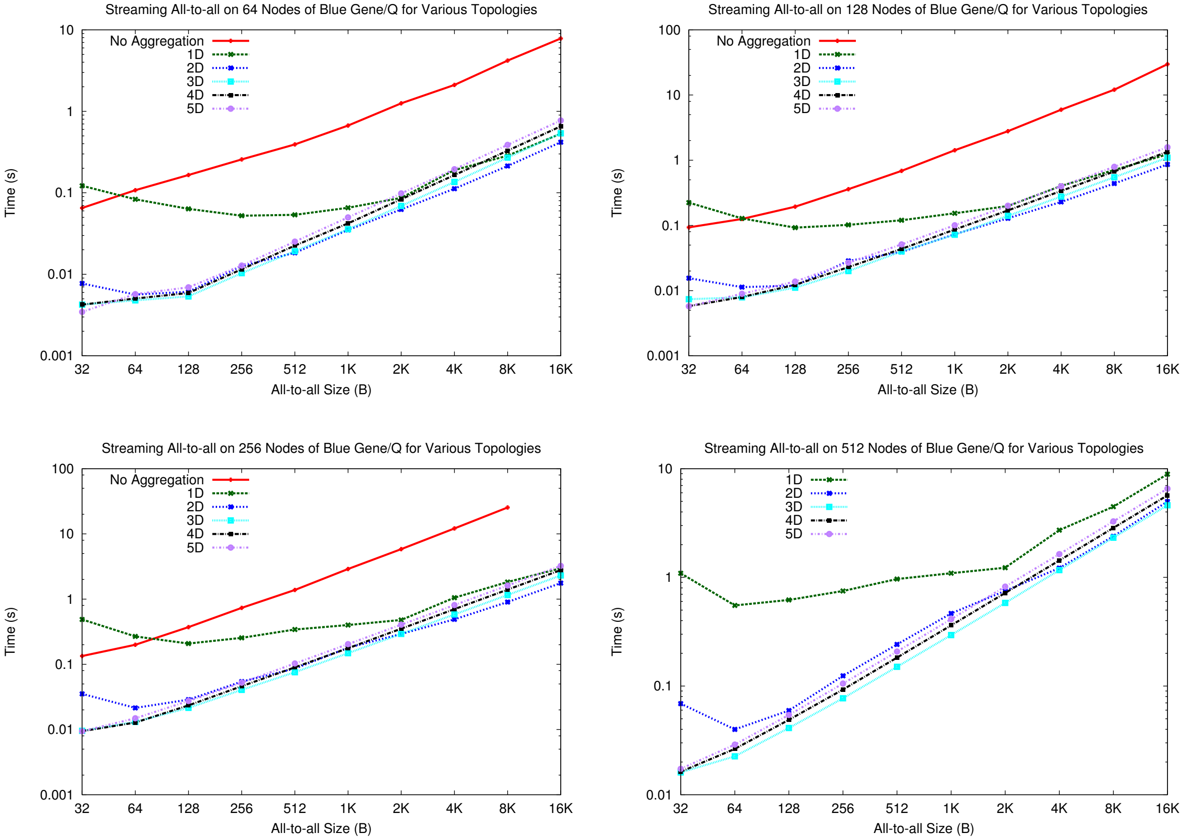


Figure 4.3: Blue Gene/Q fine-grained all-to-all performance for various grid topology aggregation schemes in runs on large node counts.

typically performed best, reflecting its balance between aggressiveness of aggregation and per-item overhead. On the other hand, a 2D virtual topology was better for high total payload size at node counts up to 256 nodes, when it could fill aggregation buffers to capacity while incurring lower per-item aggregation overhead than the 3D scheme.

It is also interesting to note that the 2D scheme for runs on more than 8 nodes outperformed direct aggregation for an all-to-all of size 16 KB, when there is sufficient data to fill buffers to capacity even when using the direct aggregation approach. The difference in performance, which was as high as 1.8x, was due to the better cache locality of the 2D scheme, whose aggregation buffer space is much smaller. For the same reason, the performance of the 3D scheme relative to the 2D scheme improves in relation to the number of nodes used in the run.

The higher concentration of traffic provided by 4D and 5D topologies typically did not justify the high aggregation overhead of these schemes at these node counts, although these schemes were competitive with the 3D topology when item counts were low in runs on large partitions. The 5D scheme was also best for some item counts in the single node test, where it was the only topology choice that split the 16 cores within a node into two dimensions (4×4).

Similar performance improvements from using aggregation can be observed on other systems. As an example, Figure 4.4 and Figure 4.5 show the results of the fine-grained all-to-all experiment on the Blue Waters Cray XE6 system. Similarly to Blue Gene/Q, topological aggregation schemes on Cray XE6 outperformed direct sending by more than 10x (and at higher node counts by more than 100x), and 2D and 3D schemes were generally better than direct aggregation. A notable difference, however, is that the message size at which 2D scheme becomes better than 3D is significantly lower on Cray XE6 compared to Blue Gene/Q. This can be explained by the fact that we were unable to guarantee minimal routing for virtual topologies with more than 2 dimensions on Cray XE6, where job partitions are not assigned as regular grids. As a result, these topologies led to increased bandwidth consumption. In the following section, we will see another reason why higher-dimensional topologies work better on Blue Gene/Q than on Cray XE6.

Finally, Figure 4.6 shows fine-grained all-to-all performance on the Piz Daint Cray XC30 system. The Cray XC30 network is configured using a high-radix, low diameter Dragonfly topology [9]. As a result of the high bandwidth and low diameter of this network, the 3D virtual topology performs comparatively better on the Cray XC30 than on the Cray XE6. In runs on higher node counts, for example, the 3D topology, despite not guaranteeing minimal routing, generally outperformed the other topological schemes for almost all data points.

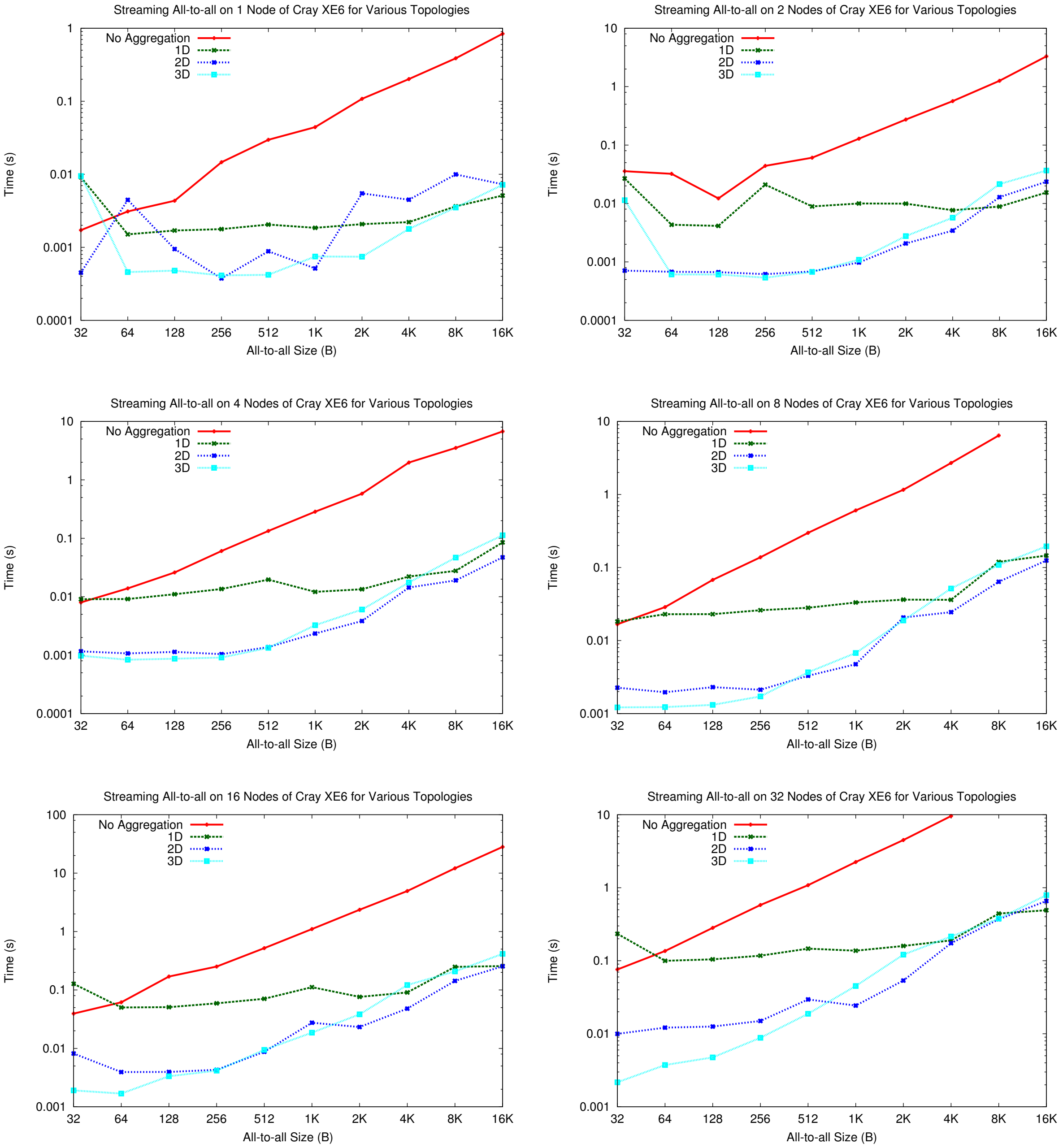


Figure 4.4: Cray XE6 fine-grained all-to-all performance for various grid topology aggregation schemes in runs on small node counts.

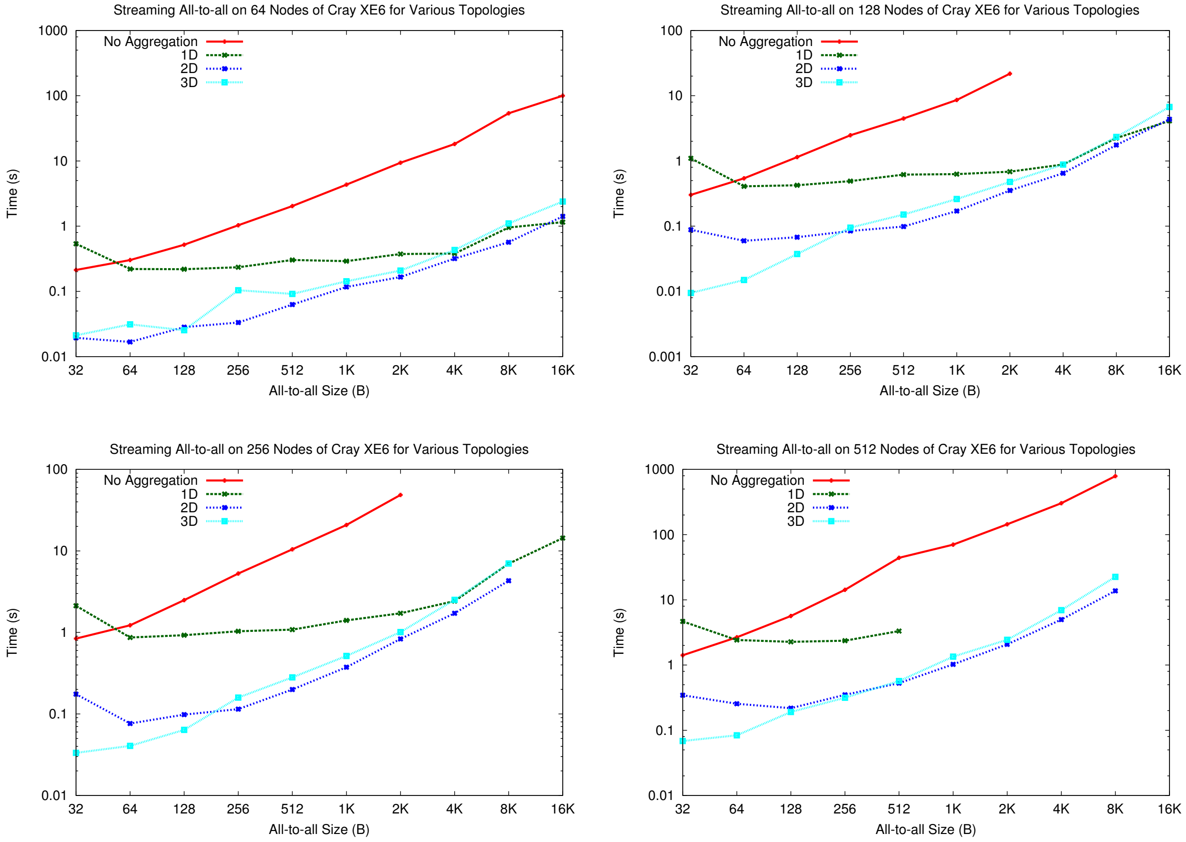


Figure 4.5: Cray XE6 fine-grained all-to-all performance for various grid topology aggregation schemes in runs on large node counts. Missing data points indicate run failures resulting from high communication load.

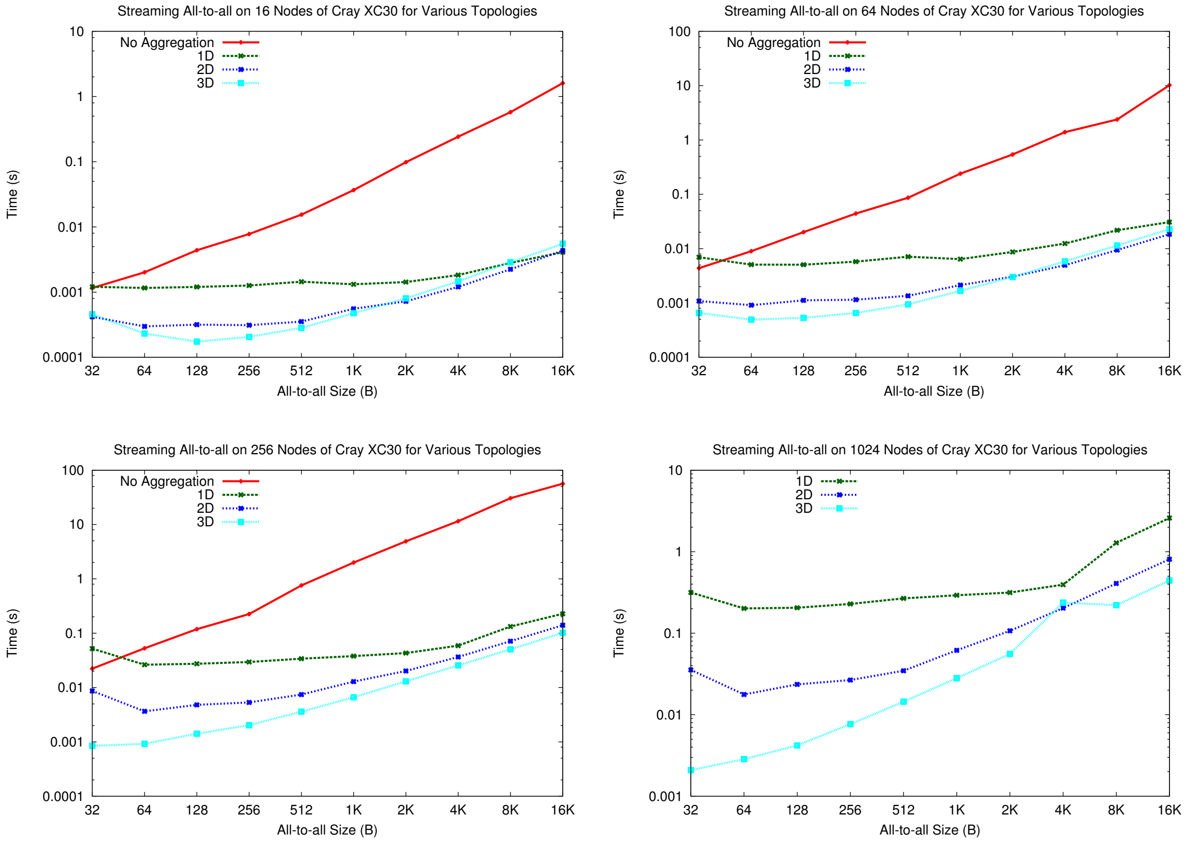


Figure 4.6: Cray XC30 fine-grained all-to-all performance for various grid topology aggregation schemes.

4.7 Aggregation Overhead

Aggregation overhead refers to the CPU and memory bandwidth resources expended along the route of each communication item. In topological aggregation, when aggregation overhead is incurred at the source and every intermediate destination, the routing step (see Section 3.3.1) is the most expensive processing task. As the benefits of aggregation stem primarily from reducing communication overhead, aggregation overhead must be significantly lower than communication overhead for an aggregation scheme to yield improved performance.

Table 4.2 presents a comparison of aggregation overhead for a single routing step of grid aggregation using TRAM on Vesta (Blue Gene/Q) and Blue Waters (Cray XE6) supercomputing systems. The results are based on a run of the streaming all-to-all of size 16 KB on 16 nodes (256 cores) using a two-dimensional 16×16 topology on Blue Gene/Q and 16×15 topology on Cray XE6 (with one thread allocated for the communication thread on each processor). Results show that mean aggregation overhead is somewhat higher on the Cray system. Further, the variation in aggregation overhead across PEs is very high on the Cray XE6. These results, combined with the fact that communication overhead is significantly lower on the Cray system (compare Figure 2.2 and Figure 2.4) indicate that Blue Gene/Q is more amenable to aggregation optimizations than Cray XE6. For example, for messages with payload 32 B, corresponding to the item size in this test, the sum of the communication overhead at the source and destination (including message allocation and freeing) is $7.2\mu s$ on Blue Gene/Q and $1.9\mu s$ on Cray XE6. Dividing the communication overhead by the aggregation overhead yields 24 for Blue Gene/Q and 5.5 for Cray XE6. These values denote a limit on the maximum number of routing steps per item for a topological aggregation scheme. On Blue Gene/Q, results confirm that the typical choice of 2 to 6 routing steps per item is viable. However, on the Cray system, a 6D aggregation scheme would lead to higher total aggregation overhead than the corresponding communication overhead saved from aggregation, resulting in a slowdown.

4.8 Effects of Non-Minimal Routing

In Section 3.4.4 it was shown that a randomized mapping of PEs within the virtual topology leads to non-minimal routing and extra bandwidth consumption. However, in practice it is always possible to construct at least a partially structured topology, where the PEs within a node are ordered contiguously to form a dimension, but where the other dimension indices

	Blue Gene/Q	Cray XE6
Min.	285	111
Max.	310	591
Mean	296	346
Std. Dev.	3.44	125

Table 4.2: Aggregation/routing overhead incurred per step of topological grid routing on Blue Gene/Q and Cray XE6. Values are in ns.

are not mapped topologically.

Figure 4.7 shows the consequences of using a partially structured topology for the fine-grained all-to-all benchmark on Blue Gene/Q. The plot compares the performance of topological aggregation for two runs using 6D virtual topologies with identical dimension sizes, but different mapping of PEs to vertices within the topology. In the first topology specification, PEs were assigned to vertices within the topology to match the location of their node within the physical job partition. As we have seen, this preserves minimal routing. In the second case, indices for the first five dimensions in the virtual topology are assigned at random to each node, but the PEs per node are mapped in contiguous fashion along the sixth dimension. In the plotted runs, when the total amount of data sent was small and the links were not saturated, performance for the randomized mapping was only about 20% worse, probably as a result of higher latency. However, when the amount of data sent was higher, leading to link bandwidth saturation, the randomized scheme performed up to 2.3x worse. By comparison, according to the analysis in Section 3.4.4, a fully randomized mapping for a topology where all dimensions have equal sizes would lead to a difference of 6x.

The experiment shows that while the practical performance implications of non-minimal routing from partially structured topologies are not as extreme as in the case of fully randomized topologies, the observed difference in performance is nonetheless still very significant. This is particularly true for bandwidth-limited scenarios, when topologies that produce non-minimal routing should be avoided. However, for small all-to-all patterns and sparse communication patterns that are not bandwidth-limited, the aggregation benefits gained through higher-dimensional partially structured topological schemes may be worth the higher latency that results from non-minimal routing.

The effects of non-minimal routing are mitigated on low diameter networks, where non-minimal routing steps consume the bandwidth of fewer links. On such systems, even dense communication patterns may benefit from additional dimensions with non-minimal routing.

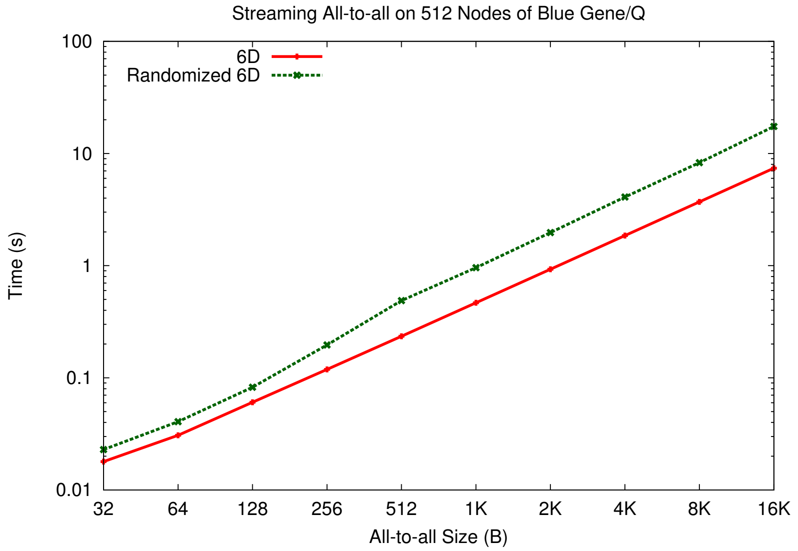


Figure 4.7: Comparison of performance for the fine-grained all-to-all benchmark between a 6D virtual topology matching the physical topology of the job partition and one obtained through a random mapping of PEs for five of the six dimensions within the virtual topology. Data is from runs on 512 nodes of Blue Gene/Q.

We saw this in runs on the Cray XC30, a system with a low-diameter, high-radix network, where at scale 3D topological aggregation was generally better than the 2D scheme (see Figure 4.6).

4.9 Summary

This chapter presented the design of TRAM, a topological message aggregation library based on the grid aggregation approach from Chapter 3. As per-message communication overhead is usually relatively low, aggregation libraries must incur very low overhead for each communicated item in order to improve performance. Along these lines, the key aspects of the design of TRAM are (a) aggregation at the level of the application that precludes the overhead incurred in message allocation and interfacing with the runtime system, and (b) reduction in per-item metadata by restricting each library instance to aggregation of a particular data type within a single collection of objects. Similarly, other aspects of the design feature a common theme: reducing aggregation overhead while preserving the generality and utility of the approach for most common scenarios.

An experimental comparison of topologies with various numbers of dimensions shows that for streaming all-to-all scenarios on Blue Gene/Q, Cray XE6 and Cray XC30, aggregation leads to a large improvement over direct sending, with the best topological schemes in many cases leading to a speedup of 10x or even 100x over no aggregation, as well as large speedups over direct aggregation. In the presented experiments, a 3D virtual topology performed best when aggregation opportunities were scarce, such as when the amount of data sent was low, while a 2D virtual topology was better when the total number of items sent was higher. The 3D scheme typically also worked best in general for higher node counts, when it benefited from a significantly smaller aggregation buffer space and hence better cache locality compared to the 2D scheme.

The optimal number of dimensions in the virtual topology also depends on system properties, and in particular on the ratio between communication overhead per message and aggregation overhead for a step of topological aggregation. As this ratio is much lower on Cray XE6 (5.5) compared to Blue Gene/Q, where it is 24, higher-dimensional topological aggregation schemes work significantly better on the latter system.

CHAPTER 5

Node-aware Aggregation

The grid aggregation algorithm presented in Chapter 3 is a significant improvement over the basic approach of direct aggregation at the source into separate buffers for each destination. By mapping the PEs in the run onto a virtual grid topology and constraining the communication graph between PEs, the grid algorithm allows aggregating items into fewer buffers, leading to lower memory usage and faster buffer fill rate.

One idea to further improve grid aggregation is to exploit the imbalance between intra-node and inter-node communication performance on modern parallel systems. Compute nodes in large-scale parallel systems today commonly comprise eight, sixteen, or more physical cores, and each core often supports multiple concurrent hardware threads through simultaneous multithreading. Cores within the same node can usually communicate at significantly lower latency and higher bandwidth than cores that are on different nodes. The idea behind node-aware aggregation is to reduce the number of aggregation buffers within a node through the use of additional intra-node communication. However, as will be shown, node-aware aggregation involves some sacrifices in terms of load balance and aggregation overhead. This chapter develops two node-aware aggregation techniques and presents analysis of their performance.

5.1 Shared Paths at the Level of Nodes

The concept behind node-aware aggregation is to consider communication on the level of compute nodes, rather than among the individual PEs. In the original grid aggregation algorithm PEs that are on the same source node typically have peers on the same destination nodes (see Figure 5.1). Each source PE uses a separate aggregation buffer, so not all items

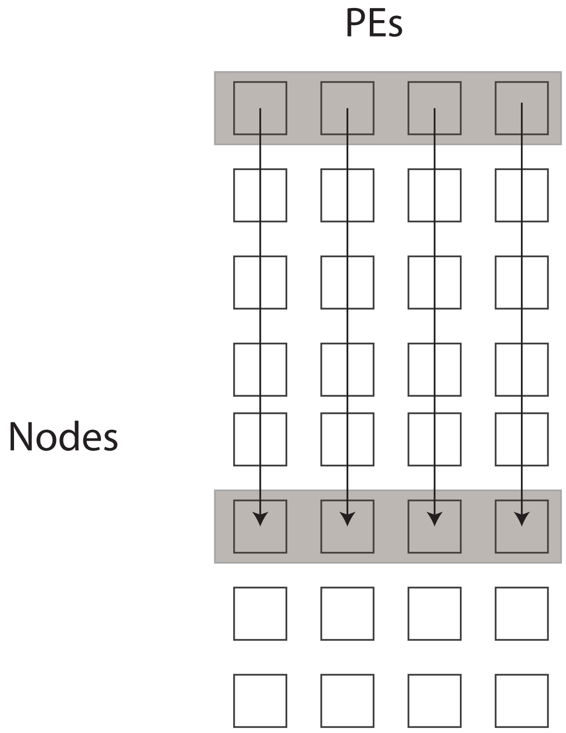


Figure 5.1: Inter-node communication in the original grid aggregation algorithm. When using the original grid aggregation algorithm with PEs per node defining a separate dimension of the virtual topology, the communication from a given source node to each destination peer node is structured such that each PE on the source node sends aggregated messages to the corresponding PE on the destination node.

sent from a given node to the same destination node are aggregated together. In contrast, the node-aware scheme views the set of aggregation buffers on PEs within the same node as belonging to a single buffer space. Additional intra-node communication, as compared to the original grid algorithm, ensures that each item is channeled to one of the PEs responsible for aggregating for the particular node to which the item needs to be sent. This leads to improved aggregation within a node prior to sending items on the network.

5.2 Dimension-based Partitioning of PEs into Teams

For a given source node in the original grid aggregation scheme, consider the set of nodes containing peers of PEs in the source node. This set can be partitioned into subsets, with each subset consisting of all the peers along a single dimension of the topology. Accordingly, the set of PEs on a node can be partitioned into *teams*, with each team assigned to aggregate items to one subset of peers. This approach forms the *base node-aware aggregation* approach presented in this section.

5.2.1 Modifications to Routing Algorithm

The routing of items in the team-based approach follows from the original grid algorithm defined in Section 3.3.1. However, since each PE in this new scheme is responsible for routing along a single dimension of the topology, additional intra-node routing steps may be required to forward items intra-node to PEs responsible for routing along the required dimension.

PEs are assigned to teams in this approach based on the last coordinate j_{N-1} in their tuple representation $(j_0, j_1, \ldots, j_{N-1})$, which by convention corresponds to the local rank within the node. Each team within a node is assigned $T = \lfloor s_{N-1}/N \rfloor$ PEs based on a block decomposition over the last dimension. If N does not divide s_{N-1}, PEs with local ranks $\lfloor N \times T, ..., s_{N-1} - 1 \rfloor$ are left without an assigned dimension.

Algorithm 1 shows the routing process for this scheme, which will be referred to as the base node-aware scheme to distinguish it from an advanced scheme with further optimizations presented later in this chapter. The algorithm works similarly to the original grid algorithm, with a few notable differences:

1. Items are routed along dimensions in the order $N-2$, $N-3$, $\ldots$, 0, $N-1$, compared to $N-1$, $N-2$, $\ldots$, 0 in the original algorithm. Because dimension $N-1$ encodes the team, the coordinate along that dimension is not finalized until the final routing step.

2. If the routing algorithm specifies that an item needs to be sent along a dimension i that is different from the current PE's assigned dimension, the item will be forwarded intra-node to a PE in team i.

3. Inter-node messages are sent to a subsequent team in the routing order. This greatly reduces the need for intra-node forwarding for most items.

Compared to the original grid algorithm, which allocates up to $\sum_{d=0}^{N-1}(s_d - 1)$ aggregation buffers per PE, the node-aware scheme allocates a maximum of $s_t - 1 + N - 1$ buffers per PE (where t is the assigned dimension). This corresponds to one peer for each coordinate along the assigned dimension and one peer for forwarding items to each of the other intra-node teams.

The benefits of the presented node-aware aggregation scheme are a reduced aggregation buffer space and greater concentration of aggregated items. These come at the cost of more sophisticated routing logic (i.e. higher aggregation overhead) and the intra-node traffic required to forward items to the correct teams. Further, because of differences in the total amount of work required to aggregate items along each dimension (see Section 3.4.5), load imbalance may result.

Algorithm 1 Routing in base node-aware scheme.

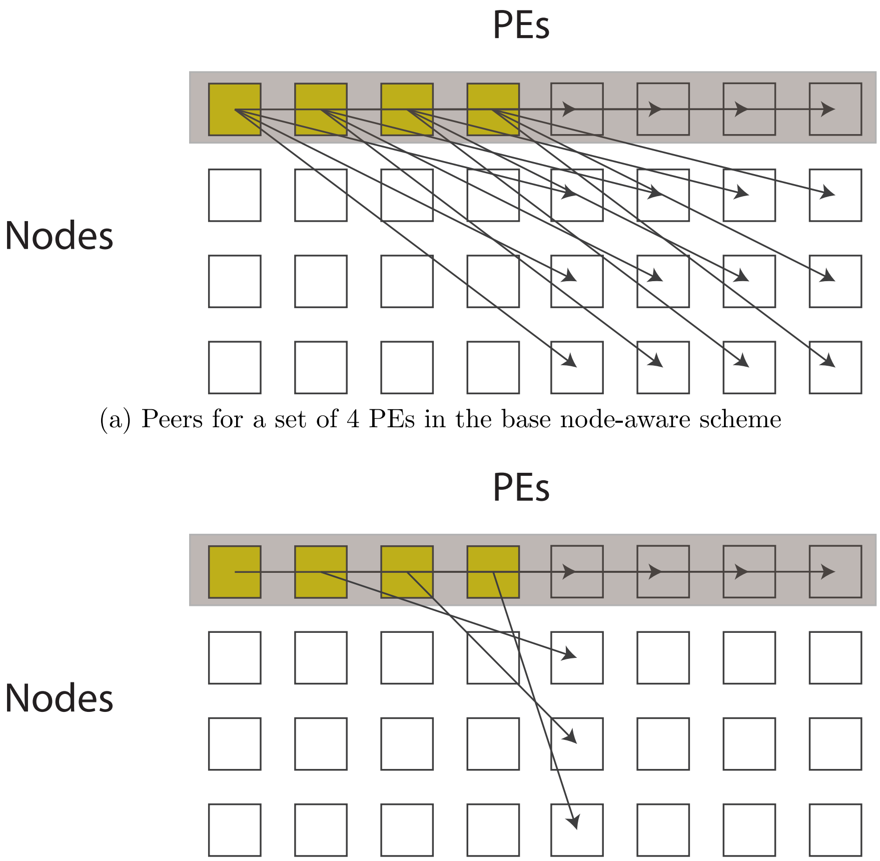
```
Inputs
$N$: number of dimensions in the virtual grid topology
$j$: current PE with index $(j_0, j_1, \ldots, j_{N-1})$
$k$: item's destination PE with index $(k_0, k_1, \ldots, k_{N-1})$
$s_n$: size of dimension $n$ in the virtual grid topology for $n \in 0, \ldots, N-1$

Output
$m$: index of next intermediate destination

$T \leftarrow \lfloor s_{N-1}/N \rfloor$                                  ▷ team size
$a \leftarrow \lfloor \frac{j_{N-1}}{T} \rfloor$                          ▷ dimension assignment for current PE
if $\exists x \in \{0, \ldots, N-2\}$ s.t. $j_x \neq k_x$ then
    $i \leftarrow \max \{0, \ldots, N-2\}$ s.t. $j_x \neq k_x$
else
    $i \leftarrow N-1$
end if                                         ▷ $i$ is the dimension that needs to be routed along
if $i = a$ then                                ▷ the assigned dimension for current PE is equal to $i$
    if $i = N-1$ then                          ▷ send to final destination
        $m \leftarrow (k_0, k_1, \ldots, k_{N-1})$
    else if $i = 0$ then                       ▷ send to last team on remote node
        $m \leftarrow (k_0, k_1, \ldots, k_{N-2}, j_{N-1} + (N-1) \times T)$
    else                                       ▷ send to subsequent team on remote node
        $m \leftarrow (j_0, j_1, \ldots, j_{i-1}, k_i, j_{i+1}, \ldots, j_{N-1} - T)$
    end if
else                                           ▷ forward item intra-node to appropriate team
    $m \leftarrow (j_0, j_1, \ldots, j_{N-1} + (i-a) \times T)$
end if
```

5.3 Specialization of PEs within a Team

While the team based approach described in Section 5.2 provides a substantial improvement in reducing the number of aggregation buffers per node and hence improving the fill rate



(a) Peers for a set of 4 PEs in the base node-aware scheme

(b) Peers for the same set of PEs in the advanced node-aware scheme

Figure 5.2: Comparison of buffer counts in the base and advanced node-aware schemes for a 2D grid topology. As there are two dimensions, PEs within a node are split into two teams of size 4 each, with the first team responsible for sending messages along the inter-node dimension, and the second team responsible for sending messages intra-node. In the base scheme, PEs within a team may send messages to destination PEs on the same node. In the advanced scheme, this issue is addressed by assigning a subset (in this case of size 1) of the destination nodes along the dimension to each PE in the team.

of buffers, it does not eliminate the problem of multiple PEs on the same source node aggregating and sending messages to peers that are on the same destination node. This is demonstrated in Figure 5.2a.

The advanced node-aware aggregation scheme, presented in this section, aims to further reduce the aggregation redundancy. It preserves the key aspect of the earlier algorithm, which assigned each team to aggregation along a single dimension of the topology, but adds on top of it further specialization of PEs within a team, as shown in Figure 5.2b.

5.3.1 Modifications to Routing Algorithm

Despite the simplicity of its premise, the advanced node-aware scheme adds substantial complexity to the routing and aggregation algorithm. The main differences compared to the base node-aware scheme are as follows:

1. Peer indices are offset based on the *index of the sender along the dimension being sent*. This is done to ensure load balance across the PEs within a team. The process is defined in Algorithm 2.

2. To reduce the number of peers for each PE, each item can be aggregated only at specific PEs within a team, based on the *index of the destination along the dimension being sent*. This requires forwarding of items within a team. While in the original node-aware scheme most items received at a given PE could be immediately aggregated (with the remainder forwarded intra-node to the appropriate team), in the new scheme arriving items need to be distributed among the PEs within the team, as shown in Algorithm 3. Section 5.3.2 describes a scheme that performs this step at low overhead.

Algorithm 4 defines the routing process in the advanced node-aware scheme.

5.3.2 Efficient Distribution of Items Within a Team

Due to the specialization of PEs within a team in the advanced node-aware scheme, items received at an intermediate destination need to be distributed among the team members for subsequent sending. Algorithm 3 defines this process. One way to implement the distribution algorithm would be to: (a) explicitly allocate buffers at every PE for each member of its team (b) copy items from intermediate messages into the aggregation buffers for the appropriate team members and (c) send buffers when they fill. This approach would generally follow what is done for inter-node aggregation. However, as team members are always within the same address space, the intra-node distribution can be made implicit to reduce its overhead. In the latter approach, a receiving team member iterates over the list of arriving items, evaluates which team member should be assigned each item, and writes the assigned indices into a local data structure of preprocessed messages. Later, when a substantial number of such assignments have been recorded for a particular team member, a lightweight message can be sent to that member with the pointer to the local data structure that holds the record of assigned items in preprocessed messages. Based on the index assignments in this data structure, the receiving team member subsequently reads the items directly from the original message and copies them into an appropriate aggregation buffer. In this manner,

Algorithm 2 Destination offset and assigned block in advanced node-aware scheme.

```
Inputs
$N$: number of dimensions in the virtual grid topology
$j$: current PE with index $(j_0, j_1, \ldots, j_{N-1})$
$s_n$: size of dimension $n$ in the virtual grid topology for $n \in 0, \ldots, N-1$

Outputs
$T$: team size
$o$: offset used for determining destination PE in Algorithm 4
$l$: least index along assigned dimension in the block assigned to this PE
$u$: greatest index along assigned dimension in the block assigned to this PE
$b$: number of destination indices assigned per team member (in small block)
$r$: number of team members with an additional destination assigned
$a$: dimension assignment for current PE

$T \leftarrow \lfloor s_{N-1}/N \rfloor$    ▷ team size
$p \leftarrow j_{N-1} \mod T$    ▷ Index of current PE within its team
$a \leftarrow \lfloor \frac{j_{N-1}}{T} \rfloor$
if $a < N$ then    ▷ does this PE have a dimension assigned?
    $b \leftarrow \lfloor s_a/T \rfloor$
    $r \leftarrow s_a - b \times T$
    if $j_a < r \times (b+1)$ then    ▷ does this PE have an extra destination assigned?
        $d \leftarrow \lfloor \frac{j_a}{b+1} \rfloor$    ▷ Index of destination within its team
    else
        $d \leftarrow r + \lfloor \frac{j_a - r \times (b+1)}{b} \rfloor$    ▷ Index of destination within its team
    end if
end if
if $a < N - 1$ then
    $o \leftarrow d - p$
else
    $o \leftarrow 0$    ▷ Do not use offset for final routing step when delivering to destination
end if
if $p < r$ then
    $l \leftarrow p \times (b+1)$
    $u \leftarrow l + b$
else
    $l \leftarrow r \times (b+1) + (p-r) \times b$
    $u \leftarrow l + b - 1$
end if
```

Algorithm 3 Team index selection in advanced node-aware scheme.

```
This algorithm is evaluated for every item arriving at a new team
Inputs
$j$: current PE with index $(j_0, j_1, \ldots, j_{N-1})$
$k$: item's destination PE with index $(k_0, k_1, \ldots, k_{N-1})$
$r$: number of team members with an additional destination assigned (from Algorithm 2)
$T$: team size (from Algorithm 2)
$a$: dimension assignment for current PE (from Algorithm 2)
$b$: number of destination indices assigned per team member (from Algorithm 2)

Output
$q$: index of PE within current team to which the item should be forwarded for further processing

if $k_a < r \times (b+1)$ then
    $q \leftarrow \frac{k_a}{b+1}$
else
    $q \leftarrow r + \frac{k_a - r \times (b+1)}{b}$
end if
```

the assignment of each item to team member can be done using a single copy as well as a single iteration through the items in the message.

Garbage Collection

When using implicit intra-node distribution of items, a message received at an intermediate destination needs to be buffered until the items it contains have been copied out by the assigned team members. To allow the PE that received the original message to recognize when it is safe to free the message, each assigned team member marks a flag in the data structure of preprocessed messages after it finishes copying out the items. The owner PE periodically checks the flags for preprocessed messages it has sent out, and frees the original message whenever its completion flags have been marked by all the assigned PEs. The flag for each PE is implemented using a 64-byte block, only 1 bit of which is used to mark completion. This is done to prevent the contention that would occur if different PEs wrote concurrently into a single block.

Due to the longer life span of received messages, transient memory use in the advanced node-aware scheme is less strictly bounded than in the previous schemes. In order to ensure that preprocessed messages get freed promptly, in the TRAM implementation of this scheme intra-node messages between team members are sent at higher priority, so that they get

Algorithm 4 Routing in advanced node-aware scheme.

```
Prerequisite
Item is at the correct index within the team (i.e. has been forwarded intra-node using Algorithm 3)

Inputs
$N$: number of dimensions in the virtual grid topology
$j$: current PE with index $(j_0, j_1, \ldots, j_{N-1})$
$k$: item's destination PE with index $(k_0, k_1, \ldots, k_{N-1})$
$s_n$: size of dimension $n$ in the virtual grid topology for $n \in 0, \ldots, N-1$
$o$: offset calculated from Algorithm 2

Output
$m$: index of next intermediate destination

$T \leftarrow \lfloor s_{N-1}/N \rfloor$                                ▷ team size
$a \leftarrow \lfloor \frac{j_{N-1}}{T} \rfloor$                        ▷ dimension assignment for current PE
if $\exists x \in \{0, \ldots, N-2\}$ s.t. $j_x \neq k_x$ then
    $i \leftarrow \max\{0, \ldots, N-2\}$ s.t. $j_x \neq k_x$
else
    $i \leftarrow N-1$
end if                                                  ▷ $i$ is the dimension that needs to be routed along
if $i = a$ then                                         ▷ the assigned dimension for current PE is equal to $i$
    if $i = N-1$ then                                   ▷ send to final destination
        $m \leftarrow (k_0, k_1, \ldots, k_{N-1})$
    else if $i = 0$ then                                ▷ send to last team on remote PE
        $m \leftarrow (k_0, k_1, \ldots, k_{N-2}, j_{N-1} + (N-1) \times T + o)$
    else                                                ▷ send to subsequent team on remote PE
        $m \leftarrow (j_0, j_1, \ldots, j_{i-1}, k_i, j_{i+1}, \ldots, j_{N-1} - T + o)$
    end if
else                                                    ▷ forward item intra-node to appropriate team
    $m \leftarrow (j_0, j_1, \ldots, j_{N-1} + (i-a) \times T)$
end if
```

processed before other messages received at a particular PE. This reduces the number of pending preprocessed messages at each PE.

5.4 Alternative Designs

In this section, a set of alternative designs for node-aware aggregation is explored, starting with simple ones and building toward the more elaborate. By demonstrating the drawbacks of these approaches, the choices made in the base and advanced node-aware schemes will be clarified.

The simplest design choice that would have the effect of node-aware aggregation is to explicitly define a single buffer space per node and perform all aggregation within the context of a single aggregator instance. As an example, when using TRAM, one could initialize a local TRAM instance on a single master PE on every node. Any communication to or from one of the other PEs on the node would go through this master PE. It should not come as a surprise that this design suffers from serious performance problems, however. First, it is entirely imbalanced, with one PE per node doing the entirety of the aggregation work. Given that aggregation is done to decrease communication overhead, concentrating the collective communication overhead for the whole node within a single PE on a node would make for a less than promising start.

This simple approach can be refined using the Charm++ nodegroup construct. Nodegroups define a single chare per Charm++ SMP node, but do not assign it to a particular PE. Instead, messages that are sent to a nodegroup element are stored in a special node queue at the destination, from which the scheduler of every PE can pull messages to execute.

By default, nodegroup semantics allow for the execution of multiple entry methods concurrently on a single element. Their entry methods can also be tagged with the keyword *exclusive*. As denoted by the name, the execution of an exclusive entry method invocation is mutually exclusive with all other exclusive entry method invocations for the local nodegroup instance. It is important to note that this is implemented through a lock, so exclusive entry method invocation incurs a higher overhead than regular entry methods.

In order to use a nodegroup for node-aware aggregation, one would either need to mark all the entry methods for the library exclusive, or else try to enforce thread safety explicitly in the library code. In the former design, the additional overhead of locking for each invocation can add up. Also, different PEs on the same node would not be able to perform the aggregation work concurrently, even when aggregating into different buffers. As different PEs should select work from the node queue with the same frequency, however, using nodegroups in this fashion would at least not induce load imbalance. In tests using the Random Access benchmark on the Blue Gene/P system, this approach was found to perform significantly worse than the original grid aggregation scheme.

The nodegroup approach with non-exclusive entry methods would allow concurrent execution on different PEs, but in practice the thread-safety requirements would require locking of aggregation buffers for every item. Further, because nodegroup entry methods may be executed on any core, they induce poor cache performance. Using the group approach of the original grid aggregation scheme, where each PE has exclusive access to its buffers without locking, avoids both of these issues, and constitutes a better starting point for the node-aware scheme, as has been demonstrated in the schemes described in previous sections.

5.5 Analysis of Node-Aware Aggregation

The higher complexity of node-aware aggregation schemes translates into additional overhead compared to the original grid aggregation approach, but also yields additional aggregation opportunities and reduces the buffer counts. This section evaluates these performance characteristics.

5.5.1 Number of Routing Steps

Node-aware grid aggregation generally requires more routing steps per item than the basic grid aggregation approach. The additional routing is a direct consequence of the fact that node-aware schemes use the coordinate of the intra-node dimension to concentrate items into fewer buffers. In the case of the base node-aware scheme, this typically translates to one additional routing step at the final destination node, as shown for an example 4D topology in Table 5.1. No additional routing steps are typically required at the intermediate destinations, as the routing algorithm delivers each message directly to one of the team members responsible for routing along the subsequent dimension in relation to the one along which the message was sent. The base node-aware scheme requires intra-node forwarding only when the coordinate of the intermediate destination matches the coordinate of the final destination along this subsequent dimension. This is in contrast to the original grid aggregation scheme where the subsequent step could be performed at the same PE.

While the base node-aware scheme requires intra-node forwarding at intermediate destinations for a small fraction of the items, the advanced node-aware scheme requires it for most items. As a result of the specialization of team members in the latter scheme, only a fraction

Step	**Original Scheme**	**Node-aware Scheme**	**Team in Node-aware Scheme**
0	a, b, c, d	a, b, c, d	$\lfloor \frac{d}{2} \rfloor$
1	a, b, c, z	$a, b, c, 4 + (d \mod 2)$	2
2	a, b, y, z	$a, b, y, 2 + (d \mod 2)$	1
3	a, x, y, z	$a, x, y, (d \mod 2)$	0
4	w, x, y, z	$w, x, y, 6 + (d \mod 2)$	3
5		w, x, y, z	$\lfloor \frac{z}{2} \rfloor$

Table 5.1: Comparison of original and base node-aware grid aggregation protocols for a 4D topology. The table displays the progression of the intermediate destinations for a data item along the route from source (a, b, c, d) to destination (w, x, y, z)

.

of items will typically be delivered to the correct team member during the inter-node routing step. In fact, most items will need to be forwarded to the appropriate team member at each intermediate destination node along the route. As a result, the total number of routing steps per item is typically almost double the number in the original grid aggregation approach for the same topology. While the intra-node distribution algorithm in the advanced scheme improves the efficiency of intra-node routing, the overhead of an additional routing step at each intermediate destination is nonetheless substantial, as will be shown in experimental results.

5.5.2 Reduction in Buffer Count

Node-aware schemes substantially reduce the number of buffers required for aggregation at each node. While the original grid aggregation approach requires up to $\sum_{d=0}^{N-1}(s_d - 1)$ buffers per PE (see Section 3.3.2), in the base node-aware scheme, each PE is assigned a particular dimension x, so it will only require $s_x - 1$ buffers for aggregating items to be sent inter-node, and an additional $N - 1$ buffers to forward items to the appropriate team intra-node, for a total of $s_x + N - 2$ buffers per PE.

The advanced scheme reduces this count further, as each PE is assigned to send to a subset of the PEs along a given dimension. As a result, the number of buffers per PE is $\frac{s_x}{T} + N - 2$, where $T = \lfloor s_{N-1}/N \rfloor$ is the team size. In addition, distribution of items within a team incurs additional memory overhead that is similar to having a buffer for each team member, so the total memory overhead for the advanced scheme is in practice equivalent to allocation of $\frac{s_x}{T} + N - 2 + T - 1$ buffers.

Team	**PE mod 8**	**Send Dim**	**Receive Dim**
0	0 - 1	0	1
1	2 - 3	1	2
2	4 - 5	2	3
3	6 - 7	3	0

Table 5.2: Summary of the dimension assignments for each team in the node-aware schemes when there are four dimensions and 8 PEs in the last dimension.

5.5.3 Load Balance

In the original grid aggregation algorithm presented in Chapter 3, the distribution of aggregation and routing work across PEs is balanced, as each PE performs routing and aggregation along each dimension of the virtual topology. As a result, unless a communication pattern is asymmetrical, the distribution of routing and aggregation work in the original algorithm is good.

By comparison, the increased complexity and specialization of PEs in the node-aware schemes unfortunately present some load balance issues. The load balance problems do not result from imbalance within a team. Good load balance within a team is guaranteed by symmetry in the node-aware scheme, while in the advanced scheme, it is ensured through the use of carefully selected offsets (defined in Algorithm 2) that control which team member receives each message.

On the other hand, the balance of work across teams can be uneven in the node-aware schemes. Each PE is constrained to sending messages along a single dimension of the topology and receiving along another dimension, as shown for an example topology in Table 5.2. The size of each dimension can be different, and PEs that send or receive messages along dimensions with more PEs will have substantially more work due to having to manage a larger number of buffers, as was shown in Section 3.4.5.

5.6 Experimental Evaluation

Figure 5.3 and Figure 5.4 present a comparison of the base node-aware schemes (marked SMP2D and SMP4D) and advanced schemes (marked SMPADV2D and SMPADV4D) to 2D and 3D schemes of the original grid aggregation approach for the streaming all-to-all benchmark on the Vesta Blue Gene/Q system. In most cases, the node-aware schemes performed worse than the original grid aggregation schemes in these runs. This is particularly evident for the advanced node-aware scheme, whose poor performance is due to the substantial additional aggregation overhead incurred by the scheme. The performance of the base node-aware scheme was comparatively better, and in some cases, such as in runs on 512 nodes, the 2D base node-aware scheme even outperformed the best original grid aggregation schemes. Overall, these experiments show that aggregation performance is very sensitive to aggregation overhead. Simpler schemes with lower overhead often work better than more complex schemes that reduce the number of aggregation buffers but incur additional overhead.

5.7 Summary

Node-aware topological aggregation schemes reduce the global number of buffers compared to the original grid aggregation approach by assigning the PEs within the node to specialized routing duties. Two node-aware schemes were explored in this chapter. In the base node-aware scheme, PEs are partitioned into teams, with each team assigned to route and aggregate items along one of the dimensions of the topology. The advanced node-aware scheme increases the specialization of PEs further by assigning each PE to aggregate items for a subset of the peer nodes along a given dimension. While the node-aware schemes are effective in reducing the number of buffers per PE, this comes at the cost of significant additional overhead due to intra-node communication. This is particularly true for the advanced scheme, which requires an additional intra-node routing step for each intermediate destination along the route of an item. On the other hand, intra-node forwarding of items in the base node-aware scheme is required less frequently. As a result, in runs of the streaming all-to-all benchmark on Blue Gene/Q, the base node-aware scheme performed better than the advanced scheme, and in some cases outperformed the original topological grid aggregation schemes.

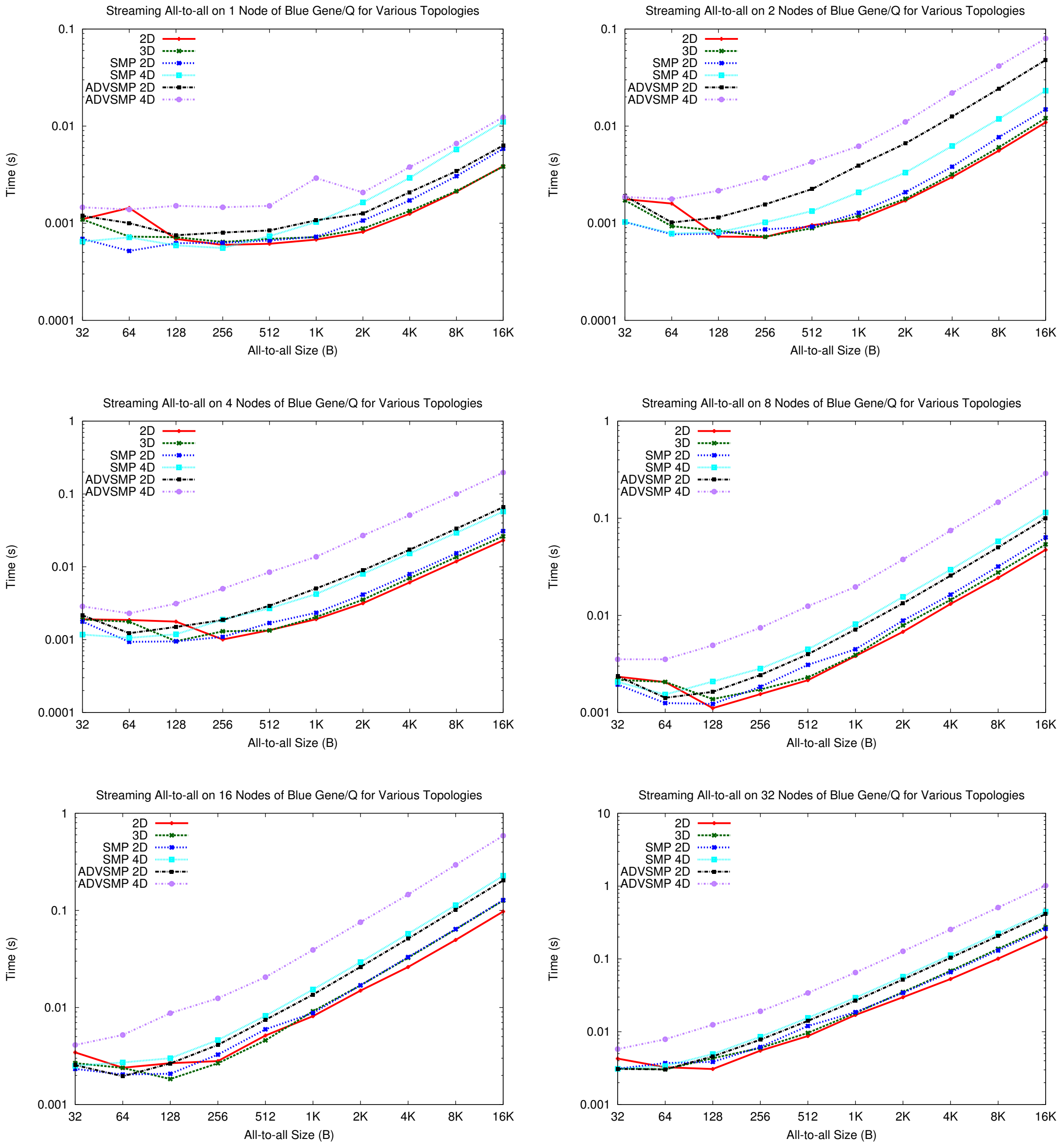
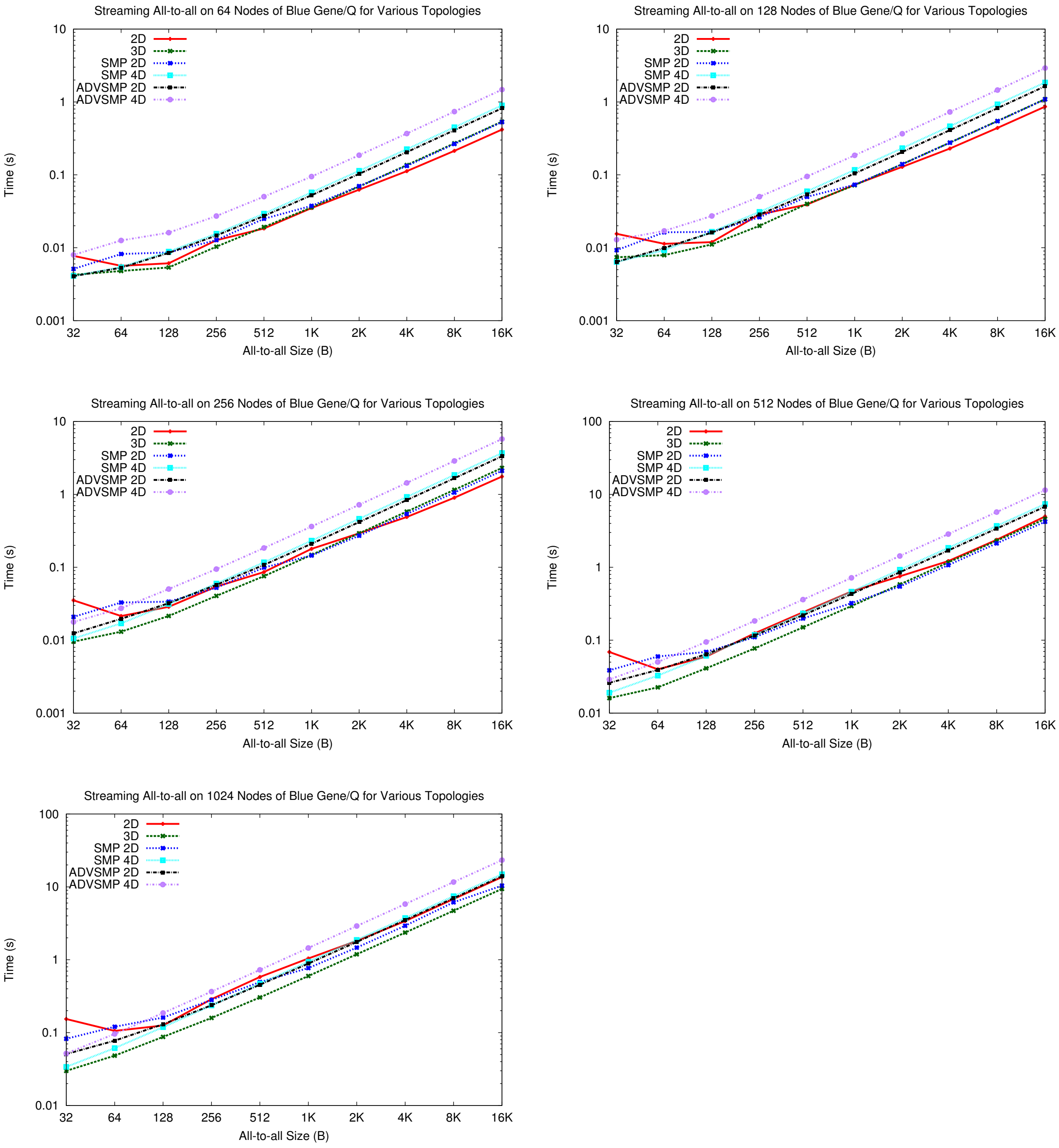


Figure 5.3: Comparison of node-aware and regular grid aggregation schemes for fine-grained all-to-all on Blue Gene/Q.


Streaming All-to-all on 64 Nodes of Blue Gene/Q for Various Topologies

Streaming All-to-all on 128 Nodes of Blue Gene/Q for Various Topologies

Streaming All-to-all on 256 Nodes of Blue Gene/Q for Various Topologies

Streaming All-to-all on 512 Nodes of Blue Gene/Q for Various Topologies

Streaming All-to-all on 1024 Nodes of Blue Gene/Q for Various Topologies

Figure 5.4: Comparison of node-aware and regular grid aggregation schemes for fine-grained all-to-all on Blue Gene/Q.

CHAPTER 6

Automating Aggregation Through Integration with the Parallel Runtime System

The previous chapters have shown how a generalized grid aggregation approach can improve performance, demonstrated the implementation of a grid aggregation library, TRAM, for optimization of fine-grained communication in Charm++ applications, and shown how to configure aggregation parameters to yield good performance for a variety of application scenarios. Although an aggregation library like TRAM simplifies the task of using aggregation, it still requires the user to perform the following steps:

1. Identify the entry methods for which communication overhead is a problem

2. Select library parameters such as topology, buffer size, progress period

3. Add the code for sending and receiving data items, and initiating and terminating streaming steps

This chapter considers how the above steps can be automated to reduce the work required by the application programmer to apply aggregation. The proposed approach will involve tighter integration of aggregation capabilities into the language and parallel runtime system. In the process, the challenges of automating various aspects of aggregation will be evaluated to identify solutions that work well while simplifying application development.

As a case study of the above approach, the integration of aggregation functionality into the Charm++ language and runtime system will be demonstrated.

6.1 Feasibility of Automation

Each of the steps involved in applying communication aggregation in a parallel application presents a different set of challenges. This section describes possible automation approaches for these issues.

6.1.1 Identifying Potential Targets for Aggregation

Given an arbitrary parallel application, how can it be determined for which messages applying aggregation will improve overall performance? When considered in full generality, the question appears difficult. For example, the answer depends on which resource is limiting application performance. The benefits of aggregation in a scenario limited by network bandwidth, where reduction in total header data sent is key, are different from the benefits in a CPU-limited scenario, where reduction in CPU communication overhead is of more importance.

Since the performance bottleneck may depend on input size and data set properties, static program analysis cannot provide a general answer to the question of when aggregation is a good idea. On the other hand, a dynamic approach based on execution statistics collected by and stored in the runtime system may do quite well. Approaches of this kind work best for iterative applications with persistent performance characteristics. Fortunately, a large class of parallel applications falls in this category. The overall idea of a profile-based approach is to use the first few iterations of an application to collect profiling information that can be used to dynamically optimize future iterations. Performance can also be tuned further periodically to support programs with dynamic behavior. The effectiveness of this approach depends on the extent to which application performance is consistent across consecutive iterations. For example, if there are multiple iteration types with different performance characteristics, the runtime system would need to be aware of the different types and the pattern of execution in order to take proper steps toward optimization. In other words, the runtime system cannot make a good optimization decision if it cannot predict within some error bound the performance characteristics of the application in subsequent iterations.

Given an iterative application with persistent performance characteristics across iterations, or one where the variation in performance occurs slowly over the course of execution, the task of the runtime system in identifying which messages should be aggregated can be split into (a) determining the performance bottleneck, and (b) estimating the effect that aggregation of various message types would have in reducing the bottleneck.

If a program's performance is limited by a CPU bottleneck, the runtime system would need to estimate the potential communication overhead reduction from aggregation for various message types, identifying the types that would reduce overhead the most. Ignoring latency effects for simplicity, the potential reduction in communication overhead per message in the original program can be estimated from Table 2.3 as

$$\frac{o_{sx} + o_{dx}}{g} - o_a,$$

where o_{sx} and o_{dx} are the source and destination CPU communication overhead for a message of type $x \in X$, and X is the set of all message types in the program. Further, if z_x is the number of messages of type x sent over the course of execution, then the maximum communication overhead reduction r_{xCPU} for messages of type x would be

$$r_{xCPU} = z_x(\frac{o_{sx} + o_{dx}}{g} - o_a) \tag{6.1}$$

Using Equation 6.1, the runtime system can estimate the reduction in overhead that would result from aggregation for each message type. It could then determine to aggregate messages where the potential improvement is above some threshold (e.g. 2% of overall execution time).

If a program's performance is limited by network bandwidth, rather than the CPU, the benefits of aggregation will instead come from the reduction in bandwidth consumed due to message header data. The potential improvement in execution time in this case, r_{xBW}, would be

$$r_{xBW} = \frac{z_x \beta e}{g} \tag{6.2}$$

Using Equation 6.2, the runtime system can approximate the maximum reduction in network bandwidth utilization that would result from aggregation of each message type.

The task of determining whether performance is limited by the network or the CPU can also be automated, so that the runtime system can correctly select when to apply Equations 6.1 and 6.2. First, if CPU utilization is high, performance is unlikely to be limited by network bandwidth. In situations where utilization is low, however, it needs to be decided whether the poor utilization is due to load imbalance or a network bandwidth bottleneck. This can be done by monitoring the amount of data injected onto each network link over time.

The above discussion assumed that information about when iterations begin and end is available to the runtime system. Automatic recognition of iterative applications and determination of time step is perhaps possible, but may be difficult in general. However, what may be difficult to the runtime system in this case is trivial to the application programmer.

By exposing an API to mark the start and end of iterations, the necessary information can be supplied to the runtime system by the application programmer at minimal overhead.

Another simplifying assumption that was made in the above discussion is that the performance profile of a time step over the course of its execution is relatively homogeneous. More commonly, a single time step may consist of multiple phases, some of which may be communication-heavy, while others are computation-bound. Further, there may not be a clear boundary between such phases, especially when using non-blocking algorithms such as is typically done in Charm++. As a result, a general solution may require an ability by the runtime system to slice time-steps into its constituent phases and analyze each phase separately, so that optimization decisions can be applied to the relevant program phases.

6.1.2 Selection of Library Parameters

After a set of candidate messages for aggregation has been identified in the program, the aggregation library needs to be configured to perform this aggregation effectively. As shown in previous chapters, the choice of parameters can have a large effect on the effectiveness of the approach.

Buffer Size Analysis

Buffer size plays a dual role in a streaming aggregation approach. First, it determines the extent of aggregation - the larger the aggregation buffer, the greater the potential impact in reducing overall communication overhead. Secondly, it affects the maximum latency delay due to aggregation, which increases with buffer size. Latency delay can be important during the initial and final stages of execution, as was demonstrated in the single source-destination streaming scenario of Section 2.2.1, or for applications with burst-like communication behavior, such as the PHOLD benchmark described in Section 7.1.3. Further, buffer size also controls the memory overhead of aggregation, and along with it cache locality effects.

Because an increase in buffer size has both positive and negative effects on different aspects of performance, an intermediate buffer size is typically the best choice. The selection process can be made systematic as demonstrated in Section 2.3.1 using a streaming communication benchmark, with the selection corresponding to the smallest buffer size that yields 75% of the peak observed network bandwidth in streaming communication scenarios.

When automating buffer size selection, there is no reason to limit testing to a single buffer size value. Since the optimal buffer size value will depend on the communication pattern and latency effects in the application, the streaming benchmark can instead be used to obtain a range of plausible values (e.g. 50 - 90% bandwidth utilization). Dynamic tuning can subsequently be used to converge to an optimal value within this range.

Virtual Topology Selection

While the number of plausible specifications of a virtual grid topology is large, in bandwidth-sensitive scenarios a topology that communicates items along a minimal route in the physical topology will work best.

The number of topology configurations that satisfy the minimal routing requirement will depend primarily on the physical network topology of the job partition, as well as the number of PEs per compute node in the system. For example, the regular 5D grid job partitions on Blue Gene/Q provide high flexibility for assigning the virtual topology dimensions. In Section 3.4.4, a process was described for combining consecutive dimensions in the physical topology to form a new topology that preserves minimal routing. Using this process, one can enumerate a full set of virtual topologies that preserve minimal routing for a given physical grid topology.

The number of PEs per node provides additional degrees of freedom for topology specification. Because intra-node communication does not affect minimal routing requirements, PEs per node can be (a) incorporated into one of the other dimensions of the topology (b) separated out into a new dimension in the virtual topology, or (c) split into multiple dimensions with various dimension sizes.

An automatic approach to virtual topology selection should first determine whether an application is limited by network bandwidth. If it is, the number of possible topology specifications that will maintain minimal routing is small enough that it can be enumerated (statically or dynamically). If it is not, the number of choices will be much larger. However, several heuristics can be used that generally improve performance:

1. A balance in the sizes of topology dimensions generally improves performance by improving load balance characteristics

2. Higher-dimensional topologies work better in scenarios where total number of items sent is relatively low, as they allow aggregating these items into fewer buffers

3. Lower-dimensional topologies may work better when the total number of items is high and buffers are generally filled to capacity

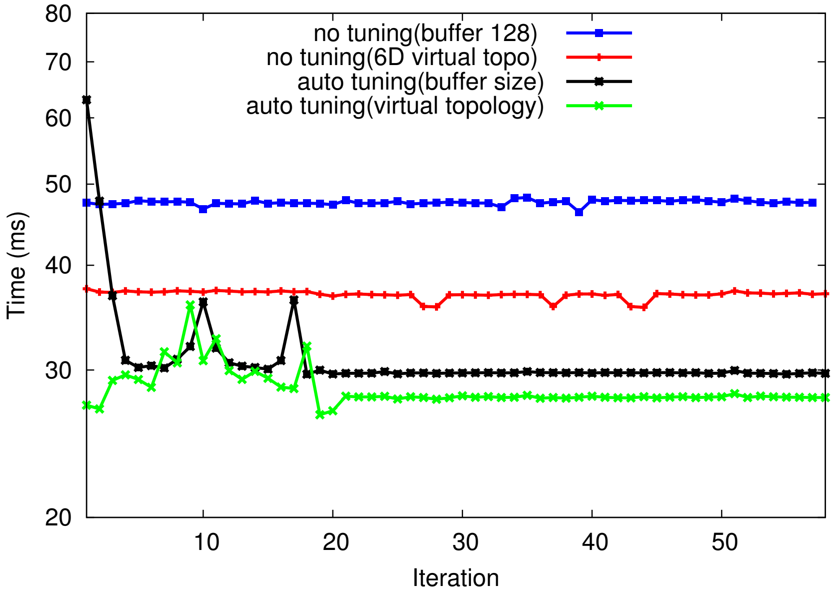


Figure 6.1: Automatically tuning buffer size and virtual topology for the all-to-all benchmark on Blue Gene/Q.

Automatic Selection of Parameters within a Search Space

After the set of possible topology and buffer size configurations has been collected, they can be passed to a control system that will automatically try various configurations and converge to the best solution.

For example, in Charm++, the Performance-Analysis-Based Introspective Control System(PICS) [10] can be used to dynamically tune performance parameters within a search space specified by the user. PICS selects a set of parameters to try for each iteration of an application, and evaluates its decision after each iteration based on the resulting performance. The control system then determines whether to adjust one or more parameters in the search space or keep the current set of parameters for the next iteration.

Our tests of TRAM using PICS show it to be effective in selecting aggregation buffer size and virtual topology specifications that maximize performance. Figure 6.1 plots the performance of runs of the all-to-all benchmark on 64 nodes of Blue Gene/Q. The Control System converged to a proper buffer size and to a good virtual topology within 25 iterations, when tuning each parameter in two separate runs.

PICS was designed and implemented by Yanhua Sun, who also performed tuning runs of TRAM using PICS.

6.2 Design of TRAM Code Generator

Based on the presented ideas, automation support for TRAM was implemented to allow developers of Charm++ applications to use aggregation with minimal effort. The approach combines automatic instantiation and configuration of TRAM libraries with user-specified selection of communication to aggregate. While the process is not completely automatic, it requires minimal input from the user and presents a significantly lower barrier to adoption of TRAM compared to explicit use of the library in application code. The remaining steps can be automated in future iterations of the implementation.

6.2.1 Charm++ Extensions

The automation approach for TRAM is based on an extension to the Charm++ interface file syntax to allow specifying where aggregation should be applied in a parallel application. This is done by tagging appropriate entry methods with the keyword `aggregate`. For each tagged entry method, the Charm++ interface translator creates an instance of TRAM for aggregating the data type corresponding to the parameter of the entry method. Further, the translator generates code for automatically using the TRAM instance when an aggregate entry method is invoked. The implementation involves simply passing the parameter as an item to the local TRAM instance. This is in contrast to the usual code generated for entry method invocation, which prepares a message, serializes the data and passes the message to an appropriate runtime system routine for sending.

Automatically generated aggregation library instances are configured to route messages over a 2D virtual grid topology with one dimension corresponding to the compute nodes in the run and the second corresponding to the PEs within a node. This conservative choice maintains minimal routing while providing significant aggregation benefits. The performance of this approach generally mirrors that of a manually defined and initialized TRAM instance using a 2D grid. Future refinement of this approach will include dynamic tuning of the library parameters based on the performance of the initially selected configuration.

6.3 Summary

This chapter presented an approach to automatically applying aggregation for appropriate communication using the runtime system. The process consisted in dynamic selection of message types for aggregation, application of aggregation using automatically generated

TRAM library instances, and dynamic tuning of library parameters based on observed performance characteristics. As a proof of concept, a scheme was implemented for automatically generating and using TRAM library instances for communication to tagged entry methods.

CHAPTER 7

Applications and Case Studies

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.

7.1 Benchmarks

We begin with a look at TRAM performance for two HPC Challenge [11] benchmarks. HPC Challenge is an annual competition to determine the top performing supercomputing systems and parallel languages for high performance computing. Charm++ won the 2011 award for productivity and performance of a parallel programming language, based on an implementation of a suite of five benchmarks [12]. In our submission, we used TRAM to improve the performance of two of these benchmarks, Random Access and FFT.

7.1.1 HPC Challenge Random Access

The HPC Challenge Random Access benchmark measures the rate of processing updates to a large 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. 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

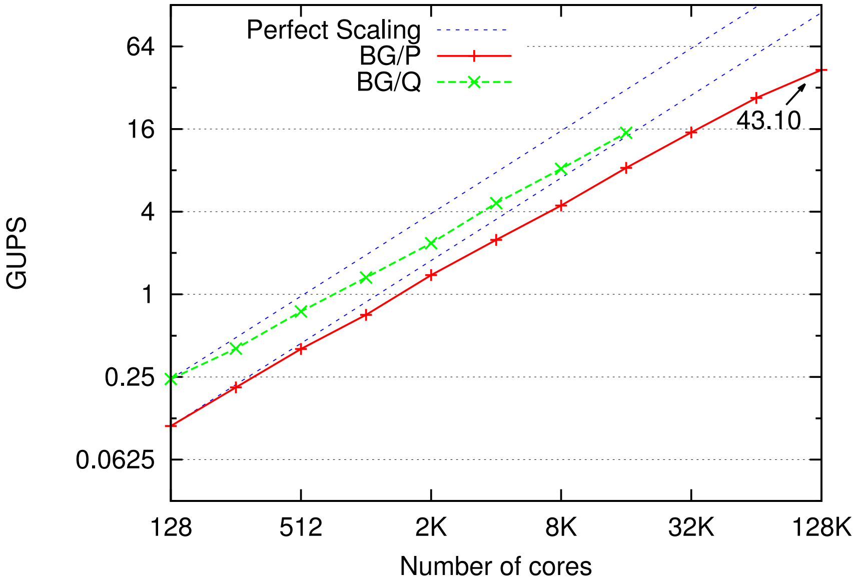


Figure 7.1: Performance of the HPC Challenge Random Access benchmark on Blue Gene/P and Blue Gene/Q using TRAM.

dimension of the virtual grid 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 grid, we found that a 5 or 6-dimensional virtual topology works best, where two of the dimensions comprise the 64 processes within a node (8×8), and the remaining dimensions are obtained by reducing the grid dimensions through combining of some individual dimensions according to the scheme presented in Section 3.4.4. When reducing the number of dimensions in the topology, we combined dimensions to form topologies with roughly equal dimension sizes. Figure 7.1 shows scaling plots of the results on Intrepid and Vesta. Scaling runs on Vesta were done by Ramprasad Venkataraman.

7.1.2 HPC Challenge 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

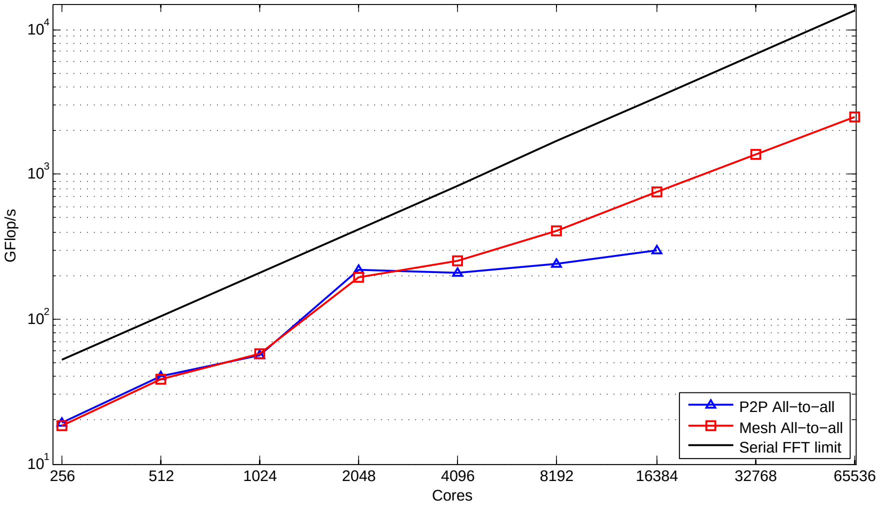


Figure 7.2: Results of Running the FFT benchmark on Blue Gene/P.

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 automatically performed segmentation of the array into *chunks* at the source and reconstructing arrays from chunks at the destination. Chunk size is independent of the size of the data type for individual elements, allowing users to limit processing overhead when data items are very small. Figure 7.2 shows a scaling plot for the benchmark on Intrepid.

The communication pattern in this benchmark is similar to an all-to-all, but as the input size is large, the benefits of aggregation are not seen until higher node counts, when individual sends become small enough that communication overhead becomes a problem. In the runs on Blue Gene/P this occurred at 4096 cores. Past this node count, using TRAM yielded a significant improvement in performance.

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

7.1.3 Parallel Discrete Event Simulations: PHOLD

Parallel discrete event simulations involve modeling a system through execution of discrete events on logical processes (LPs). Each event is marked with a virtual timestamp, and events must be executed in non-decreasing order of timestamps. *Conservative* and *optimistic* schemes have been proposed in literature for ensuring this constraint. PHOLD is a parallel discrete event simulation benchmark using the windowed conservative protocol [13].

PDES applications are characterized by a high number of fine grained events, which can be aggregated using TRAM. However, due to the use of the windowed conservative protocol,

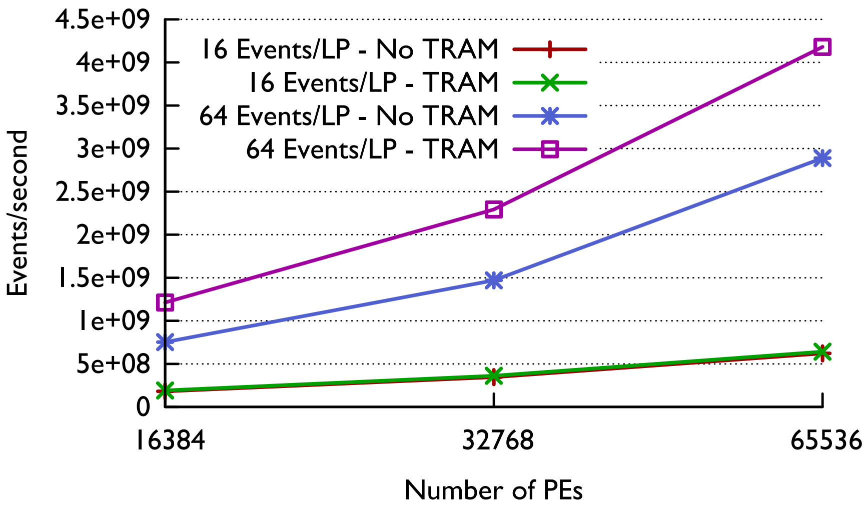


Figure 7.3: Benefits of TRAM at high event counts for a weak-scaling run of the PHOLD benchmark with 40 LPs per PE on Vesta (Blue Gene/Q).

in PHOLD these messages are sent in bursts, with synchronization after each window. As a result, latency effects are important and performance is sensitive to the progress period selected for the aggregation library. Message counts are also lower than for an all-to-all. This makes it significantly more difficult to improve performance using aggregation.

Figure 7.3 shows the impact of TRAM in runs with 40 LPs per PE and initial event loads of 16 and 64 events per LP, with 90% of the messages being local to a PE. In the case of 16 initial events per LP, aggregation opportunities were more limited and short periods of steady-state aggregation were interrupted by synchronization, so that TRAM provided only a slight benefit over direct sending of about 3% at 64K PEs. However, in the case of 256 initial events per LP, the much larger volume of messages in the system led to a more pronounced benefit from using TRAM. At 64K PEs TRAM outperformed direct sending by 40%, and reached an event rate of about 4 billion events per second [14]. Due to the burst-like flow of communication in the benchmark, optimal buffer size was uncharacteristically low, at 8 to 16 items of size 16 bytes. A low progress period of 100 to 400 μs was also required for best performance.

The PHOLD benchmark was written by Eric Mikida and Nikhil Jain. Eric Mikida performed the scaling runs on Vesta.

7.2 Applications

This section presents TRAM performance for large scientific applications, each consisting of tens of thousands of lines of code.

7.2.1 Contagion Simulations: EpiSimdemics

EpiSimdemics [2, 15, 16] is an agent-based application which simulates the spread of contagion over extremely large interaction networks. Contagion refers to transmitted phenomena such as diseases, opinions, malware propagation in computer networks, spread of social movements, etc. Here, we consider the spread of a disease over the contact network of a population.

The key object types in an EpiSimdemics simulation are Person and Location objects. Person agents send visit messages to locations they plan to visit. When a location receives all the visit messages, interactions between spatially and temporally colocated people are computed and infection messages are sent to newly infected agents. Once an agent receives all the infection messages destined for it, its health state is updated.

For this simulation, the primary communication pattern is the person to location communication. In the Charm++ implementation, there are two types of object arrays called *Location Manager* and *Person Manager* to handle collections of location and person objects. The Person Manager to Location Manager communication can be expressed as a bipartite graph. Since Person Manager contains many persons and each person can visit many locations, this communication pattern is many to many. In a typical simulation scenario, visit messages have size 36 bytes and account for more than 99% of the total communication volume.

Manual buffering vs. TRAM The initial implementation of EpiSimdemics used *manual application-specific buffering* for reducing communication overhead. This manual buffering operates at the object level, and combines the messages destined to the same Location Manager object. There is one fixed-size buffer for each pair of Person Manager-Location Manager objects, making the total number of buffers $M * N$ if there are M Person Manager objects and N Location Manager objects. The buffer size parameter is controlled through the configuration file. The application aware nature of manual buffering allows some optimizations. Since all the messages originating from an object are combined and sent to the same destination object, some fields, such as the index of the source object, need to be sent only once. In our case, this resulted in an extra 12 bytes (for a total item payload of 48 bytes) overhead per data item when using TRAM compared to application-specific buffering.

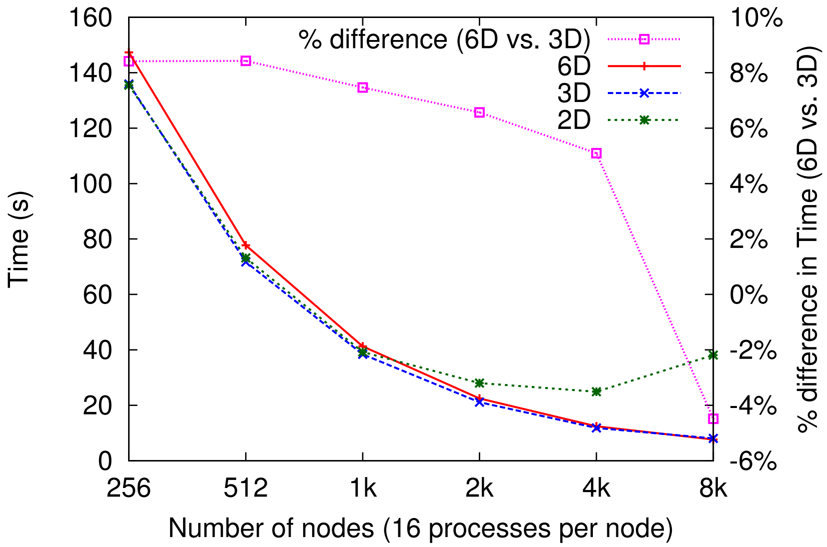


Figure 7.4: Effect of TRAM virtual topology on the performance of EpiSimdemics on Blue Gene/Q for a simulation of the spread of contagion in the state of California.

However, TRAM provides multiple benefits compared to application-specific buffering: (1) it saves programming effort since it is a library as opposed to application-specific implementations; (2) it operates at processor level, allowing it to perform more aggregation than the object-level manual buffering; (3) it is topology-aware; (4) it generally uses less memory for buffering. Manual buffering allocates a fixed buffer for each object pair, while TRAM can share the buffer space over multiple source and destination objects. Hence, per process memory requirement for manual buffering grows linearly with the number of objects, making it unscalable.

Results We evaluated the performance using TRAM on the Vulcan Blue Gene/Q system. EpiSimdemics was run using the Charm++ PAMI BlueGeneQ SMP machine layer [3]. For TRAM, we experimented with various topologies – 6D, 3D and 2D. Figure 7.4 shows the execution time for a contagion simulation on population data of the state of California for various node counts using a buffer size of 64 items. At medium scale, 3D topology marginally outperformed 6D, whereas for 8K nodes, 6D was marginally better.

Figure 7.5 compares the application speedup for three aggregation scenarios: TRAM using 6D topology, manual buffering, and direct sending without aggregation, for a contagion simulation over the populations of three different states. A buffer size of 64 items was used for TRAM and manual buffering. The speedup is calculated with respect to a base run. The base run refers to sequential execution, provided that the memory footprint of the application fit in a single node. Otherwise, base execution time was approximated by

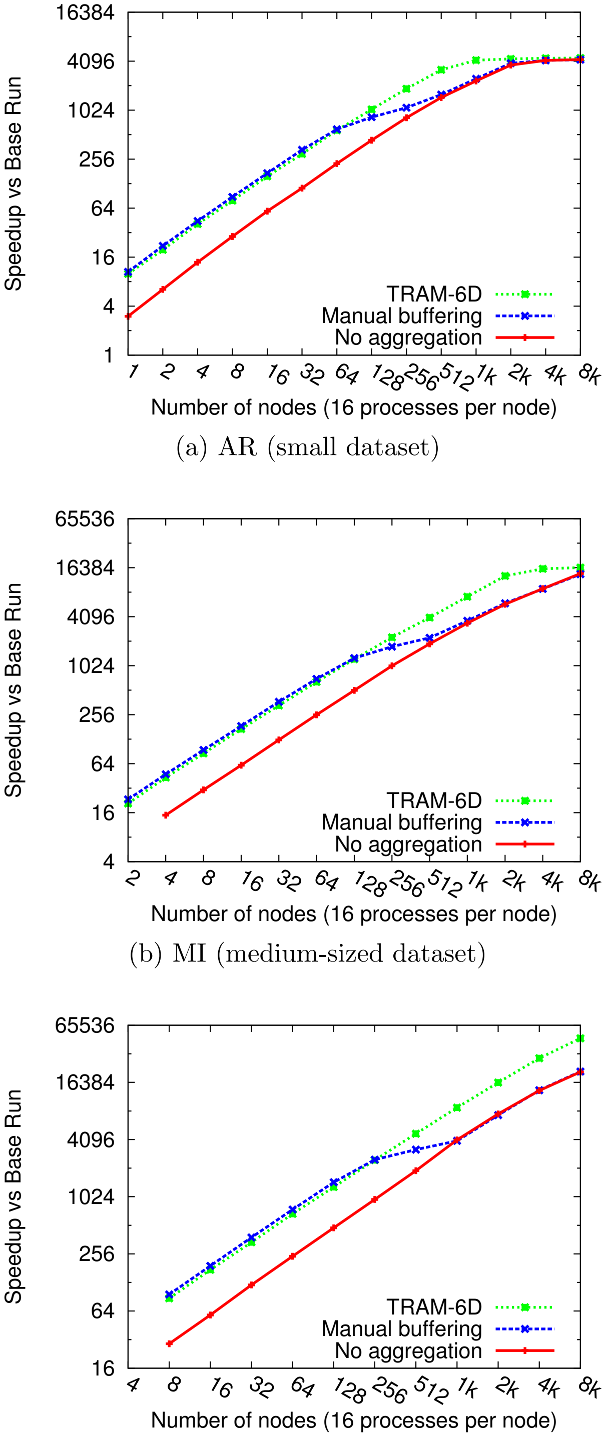
	Estimated				Observed TRAM	
#Nodes	CPU Overhead (s)		Communication (s)		Improve-	Final
	No Aggr	TRAM	No Aggr	TRAM	ment (s)	Time (s)
8	6900	646	6900	122	8470	4220
16	3450	324	3610	67.3	4200	2120
32	1720	161	1700	35.9	1960	1080
64	862	80.8	954	19.5	974	542
128	431	40.4	470	11.6	478	283
256	215	20.2	155	9.14	236	147
512	108	10.1	73.1	4.71	113	77.7
1024	53.9	5.06	42.6	2.52	49.0	41.3
2048	26.9	2.52	23.3	1.51	25.6	22.5
4096	13.5	1.26	12.6	.924	14.8	12.4
8192	6.73	.630	9.88	.462	7.13	7.69

Table 7.1: Comparison between estimated message processing and communication time for EpiSimdemics when not using TRAM to the observed improvement in execution time from using TRAM, for runs on Blue Gene/Q simulating the spread of contagion in the state of California.

running on 2 or 4 nodes and assuming ideal speedup up to that point. For small scale (1–128 nodes), TRAM performs almost as well as manual buffering and up to 4x better than without aggregation. At larger scale (beyond 256 nodes), TRAM outperforms manual buffering. For CA dataset, at 8K nodes the execution time using TRAM was 7.7s compared to 17.2s using manual buffering. Scaling runs and topology tuning for EpiSimdemics were done by Jae-Seung Yeom.

The experiments demonstrate the advantage of a topological aggregation approach over direct aggregation at the source for strong scaling, where the performance for a constant-sized data set is evaluated for runs on increasing numbers of processors. Here, as the number of nodes in a run is increased, the individual sends are distributed throughout the network, so that direct aggregation at the source for each destination fails to aggregate items into large enough buffers to sufficiently mitigate communication overhead. By comparison, a topological aggregation approach is able to maintain the advantage of aggregation to much higher node counts.

Understanding TRAM Performance We previously identified bandwidth utilization improvement and communication overhead reduction as the two main mechanisms for performance improvement from using TRAM. Using our performance model along with experimentally determined performance parameters, we can separately approximate the benefit due to each of the two mechanisms. For a given dataset, the number of visit messages in our simulations was constant. By multiplying this number by an experimentally determined value for the constant per message runtime system overhead incurred at the sender and destination (about 5 μs each), we obtained an upper bound for the communication overhead incurred in an EpiSimdemics simulation. Likewise, we approximated network communica-



(a) AR (small dataset)

(b) MI (medium-sized dataset)

(c) CA (large dataset)

Figure 7.5: Scaling EpiSimdemics up to 8k nodes (128k cores) on Blue Gene/Q using message aggregation.

tion time using the total number of bytes sent on the network for a given simulation and an experimentally determined bandwidth utilization for an all-to-all communication pattern for each node count. We also approximated the corresponding network and processing time when using TRAM.

Table 7.1 compares the aggregate time spent for message processing and network communication when not using aggregation and when using TRAM to aggregate. For reference, the total observed improvement in execution time from using TRAM is also presented. Results suggest that the overall improvement from TRAM is due to a combination of reduction in CPU overhead and improved bandwidth utilization. As expected, the processing and network time largely overlap.

7.2.2 N-Body Simulations: ChaNGa

ChaNGa [17, 18] is a code for performing collisionless N-body simulations. It simulates cosmological phenomena with periodic boundary conditions in comoving coordinates or isolated stellar systems. It uses a Barnes-Hut tree to calculate gravity, with hexadecapole expansion of nodes and Ewald summation for periodic forces. The same tree is also used for neighbor finding required by the hydrodynamics component that uses the Smooth Particle Hydrodynamics (SPH) algorithm. Timestepping is done with a leapfrog integrator with individual time steps for each particle.

Gravity is a long-range force, so ChaNGa simulations involve heavy network communication. Most of this communication is not fine grained. Although individual particles are only 40 bytes in size, a hierarchy of abstractions in the code groups the particles into structures of increasing size to control the grain size for computational and communication purposes. While the messages which communicate the data are typically not fine-grained, the messages which request the data for particles or groups of particles are very fine-grained, having just 16 bytes of payload data. These messages, typically representing 30 - 50% of the total number of messages in a run, are a good match for aggregation using TRAM.

Figure 7.6 shows average iteration time with and without TRAM for a 50 million particle simulation of a dwarf galaxy formation, using 10-iteration runs on Vesta with 64 processes per node. We found that using TRAM to aggregate the request messages improved execution time for the gravity-calculation phase of the application by 20 - 25%, leading to a 15 - 20% overall performance improvement for most node counts. At 512 nodes with this dataset, the application reached its scaling limit, where the portion of the time spent in the gravity phase was smaller, and the impact of using TRAM less noticeable as a result. For the TRAM runs,

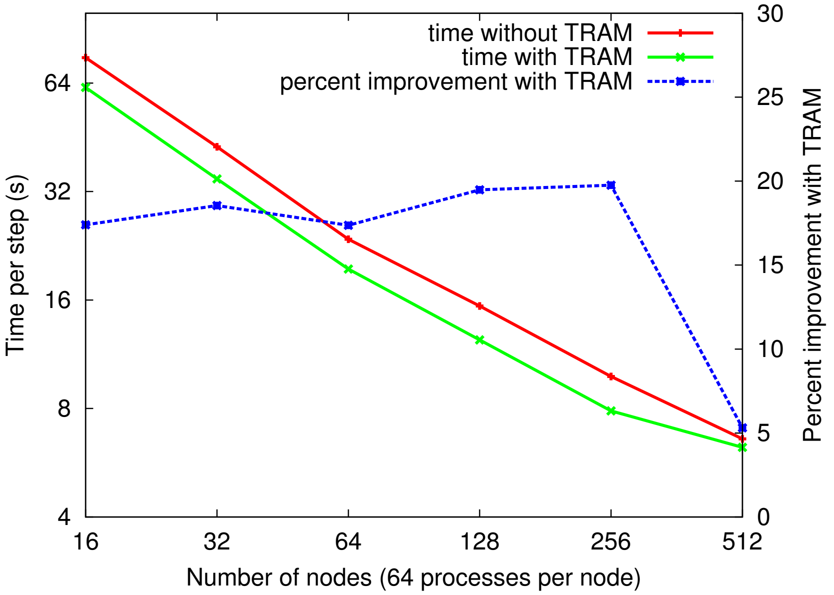


Figure 7.6: ChaNGa performance on Blue Gene/Q.

we used a 3D virtual topology and a buffering memory footprint of just 4096 data items of size 16 bytes, reinforcing the idea that a relatively small aggregation buffer space provides a large performance improvement.

Figure 7.7 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.

The use of TRAM for ChaNGa also demonstrated the limits of the aggregation approach. 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. Further, they can be latency-critical. We found that using TRAM for these messages led to worse performance.

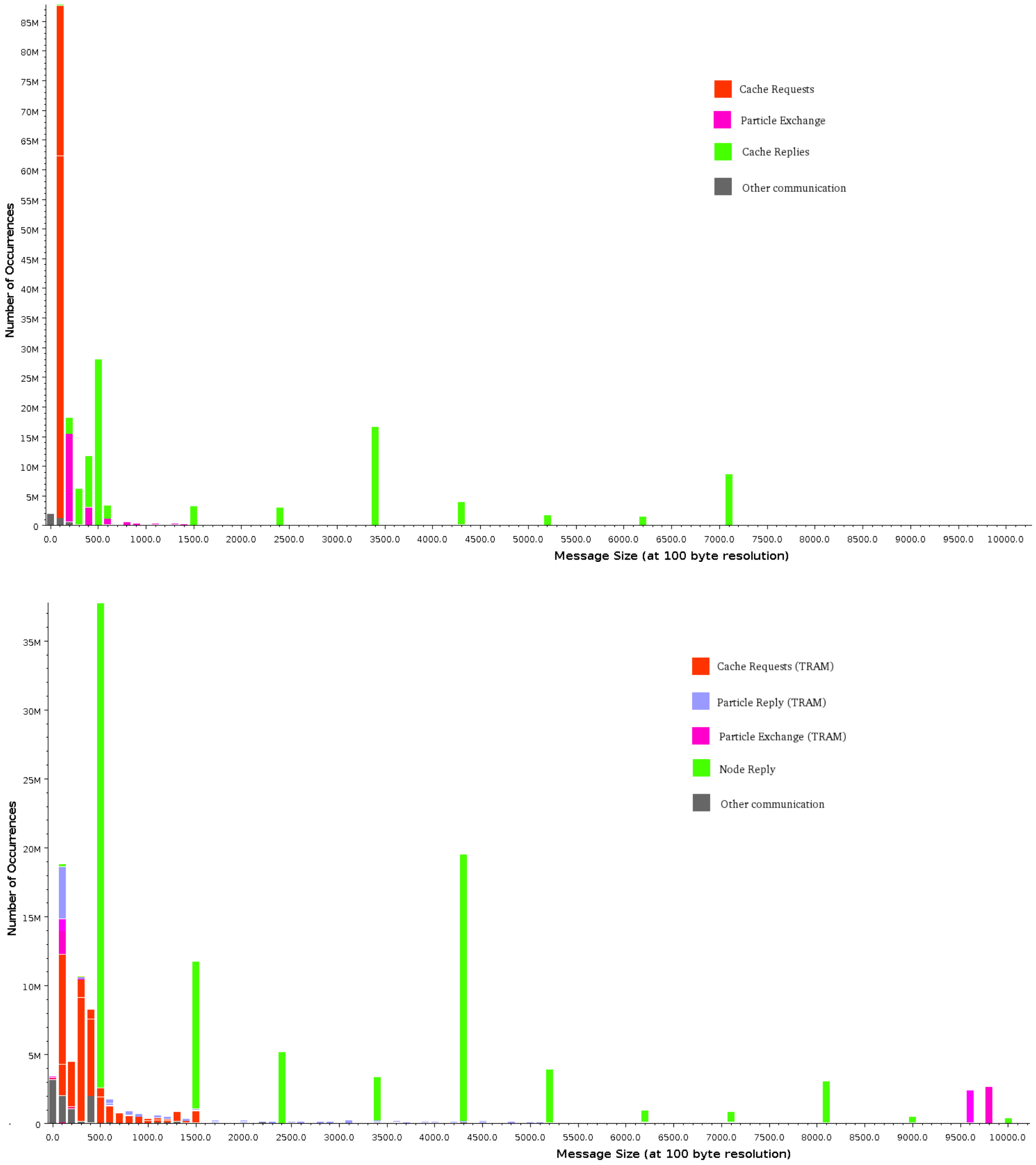


Figure 7.7: Histogram of message counts at 100-byte granularity for a time step of ChaNGa when not using TRAM (top) and with TRAM(bottom). Results were obtained on 4096 nodes of the Intrepid Blue Gene/P system at ALCF, using the 50 million particle dwarf data set.

CHAPTER 8

Related Work

The key aspects of the approach in this work are *a*) *message aggregation using a library*, *b*) *software routing*, the use of a *c*) *generalized grid virtual topology*, *d*) *streaming*, and *e*) *reduced per-item metadata by using a shared header.* This section identifies prior work that matches multiple of these key elements.

Aggregation and software routing of messages has been studied most prolifically for collective communication, particularly all-to-all, going back at least 20 years to work on indirect grid algorithms by Thakur and Choudhary [19].

Over time, the approach was generalized to other virtual topologies and a wider class of collectives. Past work from our group demonstrated improved performance of all-to-all and many-to-many personalized communication using 2D and 3D virtual grids for routing and aggregation [20]. Kumar also presented analysis of the reduction in the number of messages by using 2D and 3D virtual topologies to aggregate messages for all-to-all personalized communication [21]. Most of these algorithms did not employ streaming, however. Kumar's work did include a streaming library that used a routing and aggregation approach within a two dimensional virtual grid approximating a square, but its topology was not configurable and the work did not analyze the importance of a good match between virtual and physical topologies.

A more recent example of aggregation and routing of messages over virtual topologies using a streaming library is Active Pebbles [22]. In contrast to our approach, this work did not involve matching the virtual topology to the physical network topology to ensure minimal routing, and did not analyze the implications of virtual topology on performance.

Aggregation of communication within a node prior to transmission on the network is done in Global Memory and Threading (GMT), a runtime system library that couples software multithreading and message aggregation for a Partitioned Global Address Space (PGAS) data model [23]. GMT defines a shared aggregation buffer space per node along with a two-level aggregation approach within a node to control locking overhead. Unlike the approaches explored in our work, however, it does not consider topological aggregation for inter-node communication, which, as we have shown, can provide additional benefits.

Other research efforts have focused on application and machine-specific uses of the aggregation and routing approach. Garg and Sabharwal demonstrated dramatic improvements in performance of the HPC Challenge Random Access benchmark on Blue Gene systems [24–26]. Kumar et al. used topological routing and aggregation to improve performance of All-to-All and FFT on the Blue Gene/L [27]. Pearce et al. applied 2D and 3D message aggregation techniques for graph algorithms [28]. Similarly, Edmonds et al. have demonstrated using Active Pebbles with a 2D topological aggregation approach for graph algorithms [29].

Message aggregation and routing over virtual topologies have also each been applied in isolation to reduce runtime system overhead. For ARMCI on the Cray XT5, Yu et al. proposed virtual topologies and routing protocols to control resource management and contention [30]. For Infiniband networks, Koop et al. described a scheme for MVAPICH to dynamically coalesce messages for the same destination, with the effect of reducing memory usage and increasing the message rate [31]. A scheme like TRAM, that combines aggregation with routing over a virtual topology, could further improve the effectiveness of these approaches.

The implementation of topological aggregation in this thesis is also distinguished by precluding the need to send full header data for each communication item. This is done by constraining each library instance to aggregation of items sent to particular entry methods of a single object collection. This approach shares some similarities with work on static and dynamic communication coalescing techniques for PGAS languages such as High Performance Fortran [32], Titanium [33], and Unified Parallel C [34]. These techniques are based on the idea of applying aggregation over accesses to indices of globally shared arrays, which, similarly to our approach, allows efficient aggregation with little metadata required for each individual element. In contrast to the work in this thesis, topological aggregation has not been explored in this context.

The performance analysis in this work considers the impact of network topology and congestion on communication performance. Other research on communication over grid and torus topologies which took into account issues of network congestion includes work on MPI collectives [35], [36], topology-aware mapping [8], and network saturation on Blue Gene/P [37]. Our approach to analyzing message aggregation and determining a good message size also shares common elements with work on message strip-mining [38].

While MPI collectives are typically meticulously optimized, they often artificially limit overlap of otherwise concurrent sends through the use of a static communication schedule. Neighborhood collectives, which allow for expression of a larger range of collective operations and generation of schedules based on link load information, offer an improvement, but are still limited in dynamic scenarios by the use of a static communication schedule [39]. A topological aggregation approach with streaming routes messages over a static communication graph, but it does not follow a prescribed schedule. Instead, it streams data according to its availability using a parameterized communication grain size. As such, topological aggregation is particularly well suited for implementation of collectives for sparse, irregular, or dynamic communication patterns.

CHAPTER 9

Concluding Remarks

This work presented topological aggregation as a solution to address the problem of high overhead of fine-grained communication on large-scale clusters and supercomputers. In the course of the preceding chapters, the following findings were made:

1. **Communication overhead is a dominant component of overall communication time for fine-grained messages sent on most network types, including current-generation supercomputing networks.**

2. **Overhead depends on the relative speeds of CPU and network hardware, complexity of the underlying runtime system, and the mode of communication.**

3. **Aggregation of fine-grained messages greatly reduces total overhead, but may increase the latency of some messages, such as at the start and end of a stream of sends. This may be of concern in applications with windowed communication patterns.**

4. **Buffer size is a convenient control mechanism for aggregation. Using benchmarks of network communication, buffer size can be selected to ensure performance at a given fraction of the peak network bandwidth in continuous sending scenarios. Experiments on current generation supercomputing systems show that buffers of size** 16 **to** 32 **KB are sufficient for near-peak communication performance.**

5. **Aggregation can be used to lower the per-message metadata sent on the network by combining communication items that share a subset of the**

identifying information, effectively sending just the item payload, rather than the full message.

6. Topological aggregation reduces the number of aggregation buffers and improves aggregation throughput by restricting the destinations to which each processing element can send messages directly.

7. The use of a virtual topology simplifies the task of selecting intermediate destinations that guarantee desired performance characteristics. For example, virtual topologies can be used to limit the maximum number of messages required per item to reach the destination, and to distribute the aggregation and routing load among processing elements.

8. The Cartesian product over complete graphs is a topology with many good characteristics, when used to define the communication graph for topological aggregation. With some simplifying assumptions, the topology can be represented as a generalized N-dimensional grid.

9. In bandwidth-limited scenarios, it is important for topological aggregation to maintain minimal-routing, which means that the path of an item from the source to the destination along each intermediate destination is of minimal length.

10. A topological message aggregation library must perform routing and aggregation at very low overhead to be effective. In designs of such libraries, opportunities for limiting overhead while maintaining generality and usability must be exploited. The design of the Topological Routing and Aggregation Module for Charm++ illustrates this point.

11. The relative speed of intra-node communication (as compared to network communication) can be exploited with node-aware topological aggregation schemes that concentrate items within a node into fewer aggregation buffers. However, node-aware schemes require additional intra-node traffic to deliver the items to the correct processing element for aggregation, which generally increases per-item aggregation and routing overhead incurred. Whether node-aware aggregation improves performance over the base grid aggregation approach will depend on system and application characteristics.

12. **Topological message aggregation capabilities can be built into parallel runtime systems to minimize the developer effort required to use aggregation. An example design of this approach was demonstrated using TRAM in Charm++, where aggregation was applied automatically for specially tagged entry methods. While a good selection of library parameters in an automatic approach is important, further tuning can be done dynamically by the runtime system as the program executes, particularly for iterative applications.**

13. **The set of application classes that may benefit from aggregation is large. The effectiveness of aggregation depends primarily on the number of fine-grained messages sent in a program and on system characteristics.**

14. **A number of Charm++ applications and benchmarks were shown to benefit from aggregation using TRAM, with application speedups of up to 4x. For strong scaling runs of the EpiSimdemics application, where aggregation opportunities were limited, topological grid aggregation using TRAM achieved speedups of more than 2x compared to an application-specific direct source to destination aggregation approach.**

9.1 Success Stories

In addition to the theoretical findings of this work, the aggregation techniques presented herein have been incorporated to form the Topological Routing and Aggregation Module, a production component of the Charm++ programming system, where they continue to be used to yield significant improvements in many application scenarios. Some of the highlights of the practical benefits yielded by this work include:

1. **The 2011 HPC Challenge (Class 2) Award for Productivity and Performance of Parallel Programming Languages, presented for a suite of five Charm++ benchmarks, two of which (Random Access, FFT) used TRAM to significantly improve performance with minimal changes to the code [12].**

2. **Application speedups of up to 4x for the EpiSimdemics simulator of contagion, and up to 2x when compared against a direct application-specific aggregation approach, highlighting the benefits of topological aggregation compared to direct aggregation.**

3. **An automated approach to using TRAM that allows developers to use aggregation with fewer changes to the original code.**

4. **About a dozen different applications and benchmarks in which TRAM has been attempted, with most leading to some benefits.**

9.2 Future Work

Looking into the future, as the number of nodes and processing units in supercomputing systems continue to increase and power constraints become more restrictive, the importance of optimizations such as message aggregation that improve the efficiency of communication is likely to grow. Even the number of cores per individual node may soon reach a point at which aggregation optimizations, in software and perhaps even in hardware, would make sense.

Directions for future research that may prove fruitful in advancing the utility and performance of topological aggregation include:

1. **Techniques that improve efficiency of node-aware aggregation to make it more generally applicable.**

2. **Novel virtual topologies for new classes of networks**

3. **Applications of (topological) aggregation in hardware at the level of the network and intra-node.**

4. **Advanced adaptivity capabilities for aggregation approaches that would allow dynamic adjustments in the aggregation algorithm according to changes in application behavior.**

APPENDIX A

The Charm++ Parallel Programming System

Charm++ [40, 41] is an object-based parallel programming system comprising an *adaptive runtime system* and an *interface translator*. The adaptive runtime system implements a scheduler and subsystems for creation, mapping, localization, and management of distributed collections of objects, along with subcomponents for point-to-point as well as collective operations on sets of objects.

A.1 Parallel Execution Model

A Charm++ program consists primarily of C++ code with calls to runtime system (RTS) libraries. Programs are typically written without reference to physical system characteristics such as the number of cores or processes executing in a parallel context. Instead, objects are used to express work and data units that map to entities in the application problem domain. Interactions between objects are implemented using method invocations, similarly to how one would implement a sequential program.

Object types for which runtime system support is needed (i.e. chares), such as for receiving messages or migration, must be specified in a special interface definition (.ci) file. For each class specified in the interface description file, the user must indicate its public methods that can be invoked remotely, as well as the type of objects (singleton or collection) it will define. From the interface description file, the Charm++ translator generates class types that allow users to streamline interaction with the runtime system for purposes of communication and object management. Classes for work and data units that need to be transmitted across processes must also define a serialization operator, called pup() for Pack/UnPack. This allows transmission of objects with pointers to dynamically allocated data structures.

Chares are typically organized into indexed collections called *chare groups* and *chare arrays*. Groups map one chare object to every process in a parallel run. Arrays, on the other hand, may contain an arbitrary number of such objects, which are assigned to processes by the runtime system based on predefined or custom mappings. Chares in a group or array are always of the same type, and as such present a common interface of entry methods. A single name is used to identify the whole chare collection, and individual elements can be invoked in the program by using a subscript on the collection name. Some collectives in Charm++ are supported by extending the syntax for invoking entry method to chare collections, rather than just individual elements of a collection. For example, broadcasts to all elements of an array or group are performed by calling an entry method on the chare collection itself, without specifying a subscript. For other collectives, such as reductions, the interface translator generates the functions that need to be called on the elements of a collection to perform the collective. Results of collectives and other operations can be delivered to any object or collection in the system using a generic *callback* mechanism.

By allowing creation of work and data units independently of the notions of processes and cores, the Charm++ programming model encourages *over-decomposition*, or expression of more objects than the available number of processing cores in a parallel context. Over-decomposition allows the runtime system to hide or overlap the communication for one object with computation of concurrent work in another - while one object is waiting for a message to arrive, another can execute its pending work.

A Charm++ program begins with the execution of one or more chares marked as being initial in the interface file, 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 process using globally unique identifiers. If the invoked object is not local to the process where the call is made, an asynchronous message is sent by the runtime system to the appropriate destination, where the message is delivered to the scheduler for the local instance of the runtime system. Entry methods 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.

A.2 Communication

Communication in Charm++ is sender-driven and asynchronous. Parallel control flow is expressed through method invocations on remote objects. These invocations are generally *asynchronous* - control returns to the caller before the entry method is executed at the destination. Return values are typically not used, but the callee can send a message back to the original caller at any time, such as after receiving the caller's message. Calls can also explicitly be made synchronous, though this is often avoided to prevent idling the CPU while waiting for the return.

Invocation of entry methods is the main way of implementing dependencies between work units in Charm++. A work unit is executed in response to being invoked from another work unit that executed before it, and it in turn initiates calls to entry methods on other objects whose work is dependent on completion of its work. In this manner, the control structure for the program can be specified. Chares do not have to post receives to synchronize for message arrival. Instead, the arrival of the message triggers the entry method execution. This is known as *message-driven execution*.

A.3 Runtime System

The runtime system performs a variety of tasks relating to creation, management and scheduling of work, including:

- Mapping of objects onto the available compute resources
- Maintaining location information for objects
- Interfacing with the network to send and receive messages
- Scheduling entry methods from the set of arrived messages
- Maintaining load balance by migrating chares away from overloaded PEs

A.4 Interface Definitions

Charm++ provides a code generation mechanism that hides many of the details of chares' interactions with the runtime system. To utilize it, Charm++ programs must include one or more interface description files (named with a .ci extension) that list the classes and variables requiring runtime system support. The contents of the interface file generally include:

1. System-wide global variables, marked as readonly as they can only be set at startup
2. Message types, along with any variable-length arrays contained within, to facilitate message construction and serialization by the runtime system

3. Chare, group and array types, along with their entry method prototypes

Interface descriptions for a program can be specified in multiple modules using separate .ci files. The generated declarations and definitions for each module are placed in separate files (using .decl.h and .def.h extensions, respectively). The program's starting point(s), analogous to main() in C and C++ programs, are also indicated in the interface file using the mainchare keyword. To start the program, the runtime system passes the command-line arguments to the mainchare's constructor. There are also other annotations to be used in the interface file to specify various entry method characteristics, such as identifying the reduction functions, or for high-priority entry methods that will always be scheduled with maximum priority. The aggregate keyword described in Section 6.2.1 is another example of an entry method attribute.

User-defined message, chare, and collection classes must inherit from the corresponding classes generated by the interface translator. This is done in the pure C++ part of the code. This mechanism straps each such class with handles that enable interaction with the runtime system to support object creation, mapping, entry method invocation, collectives, load balancing, and other operations.

A.5 Modes of Runtime System Operation

Multi-core processors have in the past decade displaced the unicore processor as the de-facto standard in general purpose processing. This change has been reflected in systems for high-performance computing. While in the past a compute node in a parallel system consisted of a single processing core, compute nodes today may consist of multiple processors, each with multiple cores, all sharing access to the same memory and network resources. For increased flexibility, Charm++ provides a number of different modes of setting up the runtime system on these systems.

The first option is to run a full runtime system process on each core of a multiprocessing system, effectively treating the multiprocessor as a set of unicore processors. The advantage of this approach is its simplicity and homogeneous utilization of cores. The disadvantage is the overhead incurred from running multiple instances of the RTS.

To address the overhead and allow for more efficient communication within a node, the second mode of operation for the Charm++ RTS involves running one process over a set of cores in the same address space, with one core assigned a *communication thread*, and the remaining cores assigned *worker threads*. This mode is known as the symmetric multiprocessor *(SMP) mode*. In the SMP mode, worker threads run individual schedulers, and can efficiently communicate among themselves through shared memory. When a worker thread needs to send a non-local message, it hands over its message to the communication thread, which interfaces with the network and communicates with communication threads in other processes on the same compute node. The communication thread likewise polls the network to receive messages from other compute nodes. The SMP mode has clear advantages in reducing the latency of communication in the same address space, and in coalescing network management duties within fewer threads. The disadvantage of the SMP mode is that it typically leads to an odd, non-power-of-two number of worker threads per node, and that communication threads may become overloaded or underloaded, depending on the amount of communication in the application.

On some systems a hybrid SMP mode is also available where each worker thread performs inter-node communication duties, so that there is no separate communication thread. This mode combines the fast intra-node communication of the regular SMP mode with the homogeneity of the non-SMP mode, at the cost of a potentially higher aggregate communication overhead due to the dispersed network communication management.

APPENDIX B

Topological Routing and Aggregation Module User Manual

B.1 Overview

Topological Routing and Aggregation Module is a library for optimization of many-to-many and all-to-all collective communication patterns in Charm++ applications. The library performs topological routing and aggregation of network communication in the context of a virtual grid topology comprising the Charm++ Processing Elements (PEs) in the parallel run. The number of dimensions and their sizes within this topology are specified by the user when initializing an instance of the library.

TRAM is implemented as a Charm++ 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.

Most collective communication patterns involve sending linear arrays of a single data type. In order to more efficiently aggregate and process 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.

On systems with an underlying grid or torus network topology, it can be beneficial to configure the virtual topology for TRAM to match the physical topology of the network. This can easily be accomplished using the Charm++ Topology Manager.

TRAM supports the original grid aggregation approach as described in Chapter 3, as well as the two node-aware aggregation schemes described in Chapter 5.

B.2 Application User Interface

A typical usage scenario for TRAM involves a start-up phase followed by one or more *communication steps*. We next describe the application user interface and details relevant to usage of the library, which 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 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** After termination of a communication step, the library instance is not active. However, re-initialization using step 2 leads to a new communication step.

B.2.1 Start-Up

Start-up is typically performed once in a program, often inside the main function of the mainchare, and involves creating an aggregator instance. An instance of TRAM is restricted to sending data items of a single user-specified type, which we denote by dtype, to a single user-specified chare array or group.

Sending to a Group

To use TRAM for sending to a group, a GroupMeshStreamer group should be created. Either of the following two GroupMeshStreamer constructors can be used for that purpose:

```
template<class dtype, class ClientType, class RouterType>
GroupMeshStreamer<dtype, ClientType, RouterType>::
GroupMeshStreamer(int maxNumDataItemsBuffered,
                  int numDimensions,
                  int *dimensionSizes,
                  CkGroupID clientGID,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);

template<class dtype, class ClientType, class RouterType>
GroupMeshStreamer<dtype, ClientType, RouterType>::
GroupMeshStreamer(int numDimensions,
                  int *dimensionSizes,
                  CkGroupID clientGID,
                  int bufferSize,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);
```

Sending to a Chare Array

For sending to a chare array, an ArrayMeshStreamer group should be created, which has a similar constructor interface to GroupMeshStreamer:

```
template <class dtype, class itype, class ClientType,
          class RouterType>
ArrayMeshStreamer<dtype, itype, ClientType, RouterType>::
ArrayMeshStreamer(int maxNumDataItemsBuffered,
                  int numDimensions,
                  int *dimensionSizes,
                  CkArrayID clientAID,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);

template <class dtype, class itype, class ClientType,
          class RouterType>
ArrayMeshStreamer<dtype, itype, ClientType, RouterType>::
ArrayMeshStreamer(int numDimensions,
                  int *dimensionSizes,
                  CkArrayID clientAID,
                  int bufferSize,
                  bool yieldFlag = 0,
                  double progressPeriodInMs = -1.0);
```

Description of parameters:

- maxNumDataItemsBuffered: maximum number of items that the library is allowed to buffer per PE
- numDimensions: number of dimensions in grid of PEs
- dimensionSizes: array of size numDimensions containing the size of each dimension in the grid
- clientGID: the group ID for the client group
- clientAID: the array ID for the client array
- bufferSize: size of the buffer for each peer, in terms of number of data items
- yieldFlag: when true, calls CthYield() after every 1024 item insertions; setting it true requires all data items to be submitted from threaded entry methods. Ensures that pending messages are sent out by the runtime system when a large number of data items are submitted from a single entry method.
- progressPeriodInMs: number of milliseconds between periodic progress checks; relevant only when periodic flushing is enabled (see Section B.2.5)

Template parameters:

- dtype: data item type
- itype: index type of client chare array (use int for one-dimensional chare arrays and CkArrayIndex for all other index types)
- ClientType: type of client group or array
- RouterType: the routing protocol to be used. The choices are:
  (1) SimpleMeshRouter - original grid aggregation scheme (see Chapter 3);
  (2) NodeAwareMeshRouter - base node-aware aggregation scheme (see Algorithm 1);
  (3) AggressiveNodeAwareMeshRouter - advanced node-aware aggregation scheme (see Algorithm 4);

B.2.2 Initialization

A TRAM instance needs to be initialized before every communication step. There are currently three main modes of operation, depending on the type of termination used: *staged completion*, *completion detection*, or *quiescence detection*. The modes of termination are described later. Here, we present the interface for initializing a communication step for each of the three modes.

When using completion detection, each local instance of TRAM must be initialized using the following variant of the overloaded init function:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
init(int numContributors,
     CkCallback startCb,
     CkCallback endCb,
     CProxy_CompletionDetector detector,
     int prio,
     bool usePeriodicFlushing);
```

Description of parameters:

- numContributors: number of done calls expected globally before termination of this communication step
- startCb: callback to be invoked by the library after initialization is complete
- endCb: callback to be invoked by the library after termination of this communication step
- detector: an inactive CompletionDetector object to be used by TRAM
- prio: Charm++ priority to be used for messages sent using TRAM in this communication step
- usePeriodicFlushing: specifies whether periodic flushing should be used for this communication step

When using staged completion, a completion detector object is not required as input, as the library performs its own specialized form of termination. In this case, each local instance of TRAM must be initialized using a different interface for the overloaded init function:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
init(int numLocalContributors,
     CkCallback startCb,
     CkCallback endCb,
     int prio,
     bool usePeriodicFlushing);
```

Note that numLocalContributors denotes the local number of done calls expected, rather than the global as in the first interface of init.

A common case is to have a single chare array perform all the sends in a communication step, with each element of the array as a contributor. For this case there is a special version of init that takes as input the CkArrayID object for the chare array that will perform the sends, precluding the need to manually determine the number of client chares per PE:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
init(CkArrayID senderArrayID,
     CkCallback startCb,
     CkCallback endCb,
     int prio,
     bool usePeriodicFlushing);
```

The init interface for using quiescence detection is:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::init(CkCallback startCb,
                                           int prio);
```

After initialization is finished, the system invokes startCb, signalling to the user that the library is ready to accept data items for sending.

B.2.3 Sending

Sending with TRAM is done through calls to insertData and broadcast.

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
insertData(const dtype& dataItem,
           int destinationPe);

template <class dtype, class itype, class ClientType,
          class RouterType>
void ArrayMeshStreamer<dtype, itype, ClientType, RouterType>::
insertData(const dtype& dataItem,
           itype arrayIndex);

template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
broadcast(const dtype& dataItem);
```

- dataItem: reference to a data item to be sent
- destinationPe: index of destination PE
- arrayIndex: index of destination array element

Broadcasting has the effect of delivering the data item:

- once on every PE involved in the computation for GroupMeshStreamer
- once for every array element involved in the computation for ArrayMeshStreamer

B.2.4 Receiving

To receive data items sent using TRAM, the user must define the process function for each client group and array:

```
void process(const dtype  &ran);
```

Each item is delivered by the library using a separate call to process on the destination PE. The call is made locally, so process should not be an entry method.

B.2.5 Termination

Flushing and termination mechanisms are used in TRAM to prevent deadlock due to indefinite buffering of items. Flushing works by sending out all buffers in a local instance if no items have been submitted or received since the last progress check. Meanwhile, termination detection is used to send out partially filled buffers at the end of a communication step after it has been determined that no additional items will be submitted.

Currently, three means of termination are supported: staged completion, completion detection, and quiescence detection. Periodic flushing is a secondary mechanism which can be enabled or disabled when initiating one of the primary mechanisms.

Termination typically requires the user to issue a number of calls to the done function:

```
template <class dtype, class RouterType>
void MeshStreamer<dtype, RouterType>::
done(int numContributorsFinished = 1);
```

When using completion detection, the number of done calls that are expected globally by the TRAM instance is specified using the numContributors parameter to init. Safe termination requires that no calls to insertData or broadcast are made after the last call to done is performed globally. Because order of execution is uncertain in parallel applications, some care is required to ensure the above condition is met. A simple way to terminate safely is to set numContributors equal to the number of senders, and call done once for each sender that is done submitting items.

In contrast to using completion detection, using staged completion involves setting the local number of expected calls to done using the numLocalContributors parameter in the init function. To ensure safe termination, no insertData or broadcast calls should be made on any PE where done has been called the expected number of times.

Another version of init for staged completion, which takes a CkArrayID object as an argument, provides a simplified interface in the common case when a single chare array performs all the sends within a communication step, with each of its elements as a contributor. For this version of init, TRAM determines the appropriate number of local contributors automatically. It also correctly handles the case of PEs without any contributors by immediately marking those PEs as having finished the communication step. As such, this version of init should be preferred by the user when applicable.

Staged completion is not supported when array location data is not guaranteed to be correct, as this can potentially violate the termination conditions used to guarantee successful termination. In order to guarantee correct location data in applications that use load balancing, Charm++ must be compiled with -DCMK_GLOBAL_LOCATION_UPDATE, which has the effect of performing a global broadcast of location data for chare array elements that migrate during load balancing. Unfortunately, this operation is expensive when migrating large numbers of elements. As an alternative, completion detection and quiescence detection modes will work properly without the global location update mechanism, and even in the case of anytime migration.

When using quiescence detection, no end callback is used, and no done calls are required. Instead, termination of a communication step is achieved using the quiescence detection framework in Charm++, which supports passing a callback as parameter. TRAM is set up such that quiescence will not be detected until all items sent in the current communication step have been delivered to their final destinations.

The choice of which termination mechanism to use is left to the user. Using completion detection mode is more convenient when the global number of contributors is known, while staged completion is easier to use if the local number of contributors can be determined with ease, or if sending is done from the elements of a chare array. If either mode can be used with ease, staged completion should be preferred. Unlike the other mechanisms, staged completion does not involve persistent background communication to determine when the global number of expected done calls is reached. Staged completion is also generally faster at reaching termination due to not being dependent on periodic progress checks. Unlike completion detection, staged completion does incur a small bandwidth overhead (4 bytes) for every TRAM message, but in practice this is more than offset by the persistent traffic incurred by completion detection.

Periodic flushing is an auxiliary mechanism which checks at a regular interval whether any sends have taken place since the last time the check was performed. If not, the mechanism sends out all the data items buffered per local instance of the library. The period is specified by the user in the TRAM constructor. A typical use case for periodic flushing is when the submission of a data item B to TRAM happens as a result of the delivery of another data item A sent using the same TRAM instance. If A is buffered inside the library and insufficient data items are submitted to cause the buffer holding A to be sent out, a deadlock could arise. With the periodic flushing mechanism, the buffer holding A is guaranteed to be sent out eventually, and deadlock is prevented. Periodic flushing is required when using the completion detection or quiescence detection termination modes.

B.2.6 Re-initialization

A TRAM instance that has terminated cannot be used for sending more data items until it has been re-initialized. Re-initialization is achieved by calling init, which prepares the instance of the library for a new communication step. Re-initialization is useful for iterative applications, where it is often convenient to have a single communication step per iteration of the application.

B.2.7 Charm++ Registration of Templated Classes

Due to the use of templates in TRAM, the library template instances must be explicitly registered with the Charm++ runtime by the user of the library. This must be done in the .ci file for the application, and typically involves three steps.

For GroupMeshStreamer template instances, registration is done as follows:

- Registration of the message type:

  ```
  message MeshStreamerMessage<dtype>;
  ```

- Registration of the base aggregator class

  ```
  group MeshStreamer<dtype, RouterType>;
  ```

- Registration of the derived aggregator class

  ```
  group GroupMeshStreamer<dtype, ClientType, RouterType>;
  ```

For ArrayMeshStreamer template instances, registration is done as follows:

- Registration of the message type:

  ```
  message MeshStreamerMessage<ArrayDataItem<dtype, itype> >;
  ```

- Registration of the base aggregator class

  ```
  group MeshStreamer<ArrayDataItem<dtype, itype>,
                     RouterType>;
  ```

- Registration of the derived aggregator class

  ```
  group ArrayMeshStreamer<dtype, itype, ClientType,
                          RouterType>;
  ```

B.2.8 Automated Use of TRAM Through Entry Method Annotation

The entry method attribute aggregate can be used to instruct the runtime system to automatically generate and use TRAM instances for communication to specific entry methods. While this is much simpler than defining and using TRAM instances in application code, it precludes the ability to tune library parameters. Currently, automatically generated TRAM instances use a statically determined buffer size and topology (2D). Also, they rely on the quiescence detection termination mechanism. Future iterations of the automation approach may improve the generality of this approach through dynamic tuning of library parameters.

APPENDIX C

Experimental System Summary

The following large supercomputing systems were used for the experimental results in this work.

C.1 Blue Gene/P

The IBM Blue Gene family of systems is characterized by simple low-frequency processors coupled with fast interconnection networks arranged in a torus topology. This arrangement leads to power efficiency and a good balance of computation to communication capabilities. Blue Gene systems use lightweight custom kernels on the compute nodes, leading to very low system noise and high reproducibility of results. Job partitions on Blue Gene systems are regular grids of compute nodes. This allows using multi-dimensional topological aggregation techniques that map directly to the physical topology.

A Blue Gene/P [6] compute node features four PowerPC450 cores running at 850 MHz and 2 GB of main memory. The main interconnection network is the 3D proprietary torus network, where each node is interfaced to six bidirectional links. The peak unidirectional bandwidth on each torus link is 425 MB/s.

In this work, the 40-rack (40960 node) **Intrepid** installation and the smaller **Surveyor** Blue Gene/P systems at the Argonne Leadership Computing Facility were used.

C.2 Blue Gene/Q

Blue Gene/Q [7] is the successor to Blue Gene/P, featuring a faster 5D proprietary network and higher-clocked processors with more cores. The peak unidirectional link bandwidth on the torus network is 2 GB/s. Compute nodes consist of 16 64-bit PowerPC cores operating at 1.6 GHz, with support for up to 4 hardware threads per core. Main memory per node is 16 GB.

The **Vesta** Blue Gene/Q system at ALCF, consisting of 2048 compute nodes, was used for many experiments in this work. For some large application runs, the $24,576$ node **Vulcan** system at Lawrence Livermore National Laboratory was also used. At the time of writing, Vulcan was at #9 in the Top500 list of fastest supercomputers in the world [42].

C.3 Cray XE6

The Cray XE6 supercomputing system features a 3D torus network topology using the high-bandwidth, low-latency Gemini interconnect [43] and dual-socket compute nodes using AMD Opteron processors. Link bandwidth is up to 9.3 GB/s.

We used the $22,640$-node Cray XE6 partition of the **Blue Waters** Cray XE6/XK7 installation at the National Center for Supercomputing Applications. Each XE6 node on Blue Waters contains 64 GB of physical memory and two 8-core AMD Interlagos processors with a nominal frequency of 2.3 GHz.

C.4 Cray XC30

The Cray XC30 system is an example of current-generation supercomputing systems. Its Aries interconnect [44] employs the high-radix, low-diameter dragonfly topology. For experiments in this work, the **Piz Daint** system at the Swiss National Supercomputing Centre (CSCS) was used. Piz Daint consists of $5,272$ compute nodes, each with an Intel Xeon E5-2690v3 processor, an NVIDIA Tesla K20x accelerator, and 32 GB of host memory. Piz Daint is currently at #6 on the Top500 list.

REFERENCES

[1] "MPI: A Message Passing Interface Standard," in *MPI Forum*, http://www.mpi-forum.org/.

[2] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "TRAM: Optimizing Fine-grained Communication with Topological Routing and Aggregation of Messages," in *Proceedings of the International Conference on Parallel Processing*, ser. ICPP '14, Minneapolis, MN, September 2014.

[3] Y. S. Sameer Kumar and L. V. Kale, "Acceleration of an Asynchronous Message Driven Programming Paradigm on IBM Blue Gene/Q," in *Proceedings of 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Boston, USA, May 2013.

[4] Y. Sun, G. Zheng, L. V. Kale, T. R. Jones, and R. 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.

[5] 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*, vol. 49, no. 2/3, 2005.

[6] IBM Blue Gene Team, "Overview of the IBM Blue Gene/P project," *IBM Journal of Research and Development*, vol. 52, no. 1/2, 2008.

[7] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q interconnection network and message unit," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '11. ACM, 2011, pp. 26:1–26:10.

[8] A. Bhatele, E. Bohm, and L. V. Kale, "Optimizing communication for charm++ applications by reducing network contention," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 211–222, 2011.

[9] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," *SIGARCH Comput. Archit. News*, vol. 36, pp. 77–88, June 2008.

[10] Y. Sun, J. Lifflander, and L. V. Kale, "PICS: A Performance-Analysis-Based Introspective Control System to Steer Parallel Applications," in *Proceedings of 4th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2014*, Munich, Germany, June 2014.

[11] J. Dongarra and P. Luszczek, "Introduction to the HPC Challenge Benchmark Suite," University of Tennessee, Dept. of Computer Science, Tech. Rep. UT-CS-05-544, 2005.

[12] L. Kale, A. Arya, A. Bhatele, A. Gupta, N. Jain, P. Jetley, J. Lifflander, P. Miller, Y. Sun, R. Venkataraman, L. Wesolowski, and G. Zheng, "Charm++ for productivity and performance: A submission to the 2011 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 11-49, November 2011.

[13] P. M. Dickens, D. M. Nicol, P. F. Reynolds, Jr., and J. M. Duva, "Analysis of bounded time warp and comparison with yawns," *ACM Transactions on Modeling and Computer Simulation*, vol. 6, no. 4, pp. 297–320, Oct. 1996.

[14] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. New York, NY, USA: ACM, 2014.

[15] C. L. Barrett, K. R. Bisset, S. G. Eubank, X. Feng, and M. 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*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413408 pp. 37:1–37:12.

[16] J.-S. Yeom, A. Bhatele, K. R. Bisset, E. Bohm, A. Gupta, L. V. Kale, M. Marathe, D. S. Nikolopoulos, M. Schulz, and L. Wesolowski, "Overcoming the scalability challenges of epidemic simulations on blue waters," in *Proceedings of the IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '14. IEEE Computer Society, May 2014.

[17] F. Gioachin, A. Sharma, S. Chakravorty, C. Mendes, L. V. Kale, and T. R. Quinn, "Scalable cosmology simulations on parallel machines," in *VECPAR 2006, LNCS 4395, pp. 476-489*, 2007.

[18] P. Jetley, L. Wesolowski, F. Gioachin, L. V. Kalé, and T. 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*, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010.

[19] R. Thakur and A. Choudhary, "All-to-all communication on meshes with wormhole routing," in *Parallel Processing Symposium, 1994. Proceedings., Eighth International*, 1994, pp. 561–565.

[20] L. V. Kale, S. Kumar, and K. Vardarajan, "A Framework for Collective Personalized Communication," in *Proceedings of IPDPS'03*, Nice, France, April 2003.

[21] S. Kumar, "Optimizing communication for massively parallel processing," Ph.D. dissertation, University of Illinois at Urbana-Champaign, May 2005.

[22] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Active pebbles: parallel programming for data-driven applications," in *Proceedings of the international conference on Supercomputing*, ser. ICS '11. New York, NY, USA: ACM, 2011. [Online]. Available: http://doi.acm.org/10.1145/1995896.1995934 pp. 235–244.

[23] A. Morari, A. Tumeo, D. Chavarria-Miranda, O. Villa, and M. Valero, "Scaling irregular applications through data aggregation and software multithreading," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, May 2014, pp. 1126–1135.

[24] R. Garg and Y. 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*, ser. SC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188569

[25] V. Aggarwal, Y. Sabharwal, R. Garg, and P. Heidelberger, "Hpcc randomaccess benchmark for next generation supercomputers," in *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, May 2009, pp. 1–11.

[26] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow, A. Choudhury, Y. Sabharwal, S. Singhal, and J. J. Parker, "Looking under the hood of the ibm blue gene/q network," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389090 pp. 69:1–69:12.

[27] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger, "Optimization of all-to-all communication on the blue gene/l supercomputer," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2008.83 pp. 320–329.

[28] R. Pearce, M. Gokhale, and N. Amato, "Scaling techniques for massive scale-free graphs in distributed (external) memory," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 825–836.

[29] N. Edmonds, J. Willcock, and A. Lumsdaine, "Expressing graph algorithms using generalized active messages," in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465441 pp. 283–292.

[30] W. Yu, V. Tipparaju, X. Que, and J. Vetter, “Virtual topologies for scalable resource management and contention attenuation in a global address space model on the cray xt5,” in *Parallel Processing (ICPP), 2011 International Conference on*, 2011, pp. 235–244.

[31] M. Koop, T. Jones, and D. Panda, “Reducing connection memory requirements of mpi for infiniband clusters: A message coalescing approach,” in *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, 2007, pp. 495–504.

[32] D. Chavarría-Miranda and J. Mellor-Crummey, “Effective communication coalescing for data-parallel applications,” in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP ’05. New York, NY, USA: ACM, 2005. [Online]. Available: http://doi.acm.org/10.1145/1065944.1065948 pp. 14–25.

[33] J. Su and K. Yelick, “Automatic support for irregular computations in a high-level language,” in *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, April 2005, pp. 53b–53b.

[34] M. Alvanos, M. Farreras, E. Tiotto, J. N. Amaral, and X. Martorell, “Improving communication in pgas environments: Static and dynamic coalescing in upc,” in *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ser. ICS ’13. New York, NY, USA: ACM, 2013. [Online]. Available: http://doi.acm.org/10.1145/2464996.2465006 pp. 129–138.

[35] N. Jain and Y. Sabharwal, “Optimal bucket algorithms for large mpi collectives on torus interconnects,” in *Proceedings of the 24th ACM International Conference on Supercomputing*, ser. ICS ’10. New York, NY, USA: ACM, 2010. [Online]. Available: http://doi.acm.org/10.1145/1810085.1810093 pp. 27–36.

[36] P. Sack and W. 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*, ser. PPoPP ’12. New York, NY, USA: ACM, 2012. [Online]. Available: http://doi.acm.org/10.1145/2145816.2145823 pp. 45–54.

[37] P. Balaji, H. Naik, and N. 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*, ser. ICPADS ’09. Washington, DC, USA: IEEE Computer Society, 2009. [Online]. Available: http://dx.doi.org/10.1109/ICPADS.2009.117 pp. 586–593.

[38] C. Iancu, P. Husbands, and W. Chen, “Message strip-mining heuristics for high speed networks,” in *Proceedings of the 6th international conference on High Performance Computing for Computational Science*, ser. VECPAR’04. Berlin, Heidelberg: Springer-Verlag, 2005. [Online]. Available: http://dx.doi.org/10.1007/11403937_33 pp. 424–437.

[39] T. Hoefler and T. Schneider, "Optimization principles for collective neighborhood communications," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389129 pp. 98:1–98:10.

[40] L. Kalé and S. Krishnan, "CHARM++: A Portable Concurrent Object Oriented System Based on C++," in *Proceedings of OOPSLA'93*, A. Paepcke, Ed. ACM Press, September 1993, pp. 91–108.

[41] L. V. Kale and G. Zheng, "Charm++ and AMPI: Adaptive Runtime Strategies via Migratable Objects," in *Advanced Computational Infrastructures for Parallel and Distributed Applications*, M. Parashar, Ed. Wiley-Interscience, 2009, pp. 265–282.

[42] "Top500 supercomputing sites," http://top500.org, 2013.

[43] R. Alverson, D. Roweth, and L. Kaplan, "The Gemini System Interconnect," in *2010 IEEE 18th Annual Symposium on High Performance Interconnects (HOTI)*, 2011.

[44] G. Faanes, A. Bataineh, D. Roweth, T. Court, E. Froese, B. Alverson, T. Johnson, J. Kopnick, M. Higgins, and J. Reinhard, "Cray cascade: A scalable hpc system based on a dragonfly network," in *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, Nov 2012.