GENERALIZING FINE-GRAINED MESSAGE AGGREGATION

BY

VENKATASUBRAHMANIAN NARAYANAN

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2020

Urbana, Illinois

Adviser:

Professor Emeritus Laxmikant V Kale

ABSTRACT

The divergence of application behavior from optimal network usage leads to performance bottlenecks induced by communication. Communication performances are known to worsen when dealing with large quanities of small messages, due to the overhead of envelopes and going through the communication stack multiple times. Prior work has attempted to mitigate this through the aggregration of small messages[1], but it has only studied the impact for cases where the size of the message is constant and known ahead of time. This thesis explores the applicability of this optimization to variable-sized messages and machines with a large number of cores, analyzing both the theoretical considerations involved and the performance gains achieved in practice. The work is implemented as an update to the Topological Routing and Aggregation Module(TRAM) of the Charm++ parallel programming system.

To my parents, for their love and support.

ACKNOWLEDGMENTS

The author acknowledges the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing High Performance Computing(HPC) resources that have contributed to the research results reported within this paper. URL: http://www.tacc.utexas.edu

This research used resources of the Oak Ridge Leadership Computing Facility, which is a Department Of Energy(DOE) Office of Science User Facility supported under Contract DE-AC05-00OR22725.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

1.1 THE STATUS QUO

The execution of all parallel applications running on clusters and supercomputers consist of 2 parts: computation and communication. Improvements in performance are contingent on optimizing both of them. The evolution of CPU architecture has greatly improved the number of raw arithmetic operations possible per second, but interconnects have not kept pace. Figure 1.1 [2] demonstrates the ratio of network capacity(here, injection bandwidth) to computational capacity in supercomputers, with a visible decline in newer machines.

A contributing factor to this trend is the popularity of benchmarks like LINPACK that primarily stress the CPU, and not so much the network. In real high-performance applications, execution usually must alternate between computation and communication phases. When this is the case, optimizations of the computational stack provide diminishing returns in the face of a network that may be multiple orders of magnitude slower, especially when improperly utilized.

Sending a message on a network requires traversing the network stack once in each direction, and travelling through the entire network. The time required has 2 components to it - latency of communication, and the per-byte transportation cost, which is determined by the maximum point-to-point bandwidth supported by the network. The message to be sent itself must be packed in an envelope to allow the network to handle it correctly, and each message must travel up and down the communication stack. This adds a constant overhead to every messaging event on both the sender and the receiver side.

The root cause of these performance issues is the mismatch between the nature of networks and the nature of the applications' communication patterns. A secondary factor is that networks themselves are not optimized along a single dimension - there are several factors that must be balanced, with message latency and throughput being the two biggest ones. At smaller scales, latency appears to be the bigger problem - it sets an upper bound on the speed of an application(no matter what metric is being used), and in an uncongested network, it is the only factor that is relevant. However, when large amounts of data are being moved across the network, its importance decreases as the capacity of point-to-point links rises in prominence.

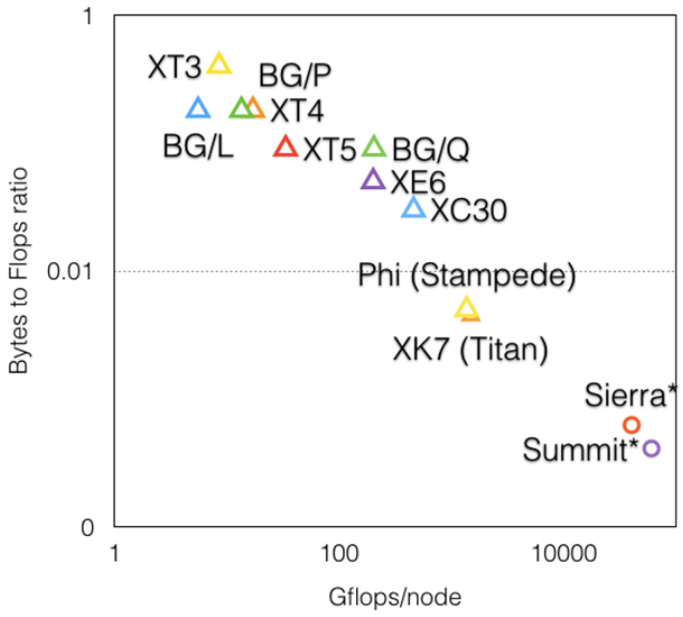

Figure 1.1: Trends in the ratio of injection bandwidth to computational power

1.2 MODELING COMMUNICATION

These facts lend themselves to a linear model of the network, popularly known as the $\alpha - \beta$ cost model. Here, the cost of sending a single message through the network is a linear function of the bytes that must be sent, with a constant term corresponding to the sum of the cost of traversing the network stack at all and the latency bounds of the network.

$$C(n) = \alpha + \beta n \tag{1.1}$$

where $C(n)$ is the cost of the message, and n is the size of the message in bytes

Summing over a number of messages being sent from the source to the destination, the overall cost on the sender side is given by the following equation.

$$C(n_1, n_2, \ldots n_N) = N\alpha + \sum_{i=1}^{N} \beta n_i \tag{1.2}$$

where $C(n_1, \ldots n_N)$ n_i is the size of the i^{th} message in bytes.

In practice, it turns out that the α term, corresponding to the constant per-message overhead tends to be about three orders of magnitude greater than the β term. This means that for "small" messages(here the term refers to messages with sizes significantly smaller than the ratio of α to β), the cost of sending a message is mostly dominated by the α term,

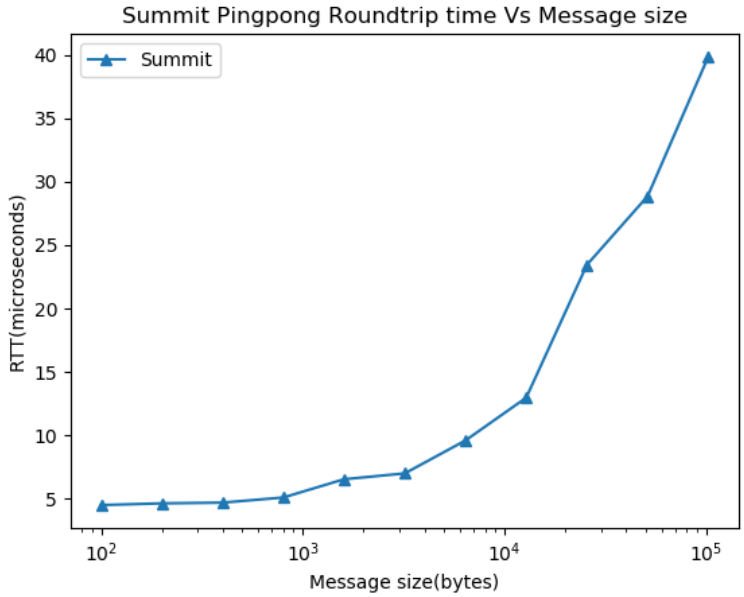

Figure 1.2: Summit Pingpong benchmark (log scale)

Machine	β(ns/byte)
Summit	0.15
Stampede2	100
Frontera	0.17
Cori	0.10

Table 1.1: β estimates from pingpong benchmark

which itself is primarily dominated by software event overhead.

The pingpong benchmark, where messages make a roundtrip from a processor back to itself through another processor, can experimentally verify the validity of this model. The benchmark was run on 4 different supercomputers - Summit, Stampede2, Cori and Frontera. The variation of their pingpong performance(for Charm++ chare arrays) with message size gives an idea of their network characteristics. Figures 1.2, 1.4 and 1.7 clearly demonstrate the relatively constant cost predicted above at smaller message sizes, while a linearly scaled version of the same plot, in figures 1.3, 1.5 and 1.6 show the linear cost behavior at larger message sizes.

From the pingpong benchmark results, we can estimate the one-way β for all of these machine, which are listed in table 1.1.

An analogy can be drawn to the routing problem in real life transportation. As long as roads are not congested, it is optimal to use private transport in order to get from a point A to another point B. However, if the roads are filled to near-capacity, adding more vehicles

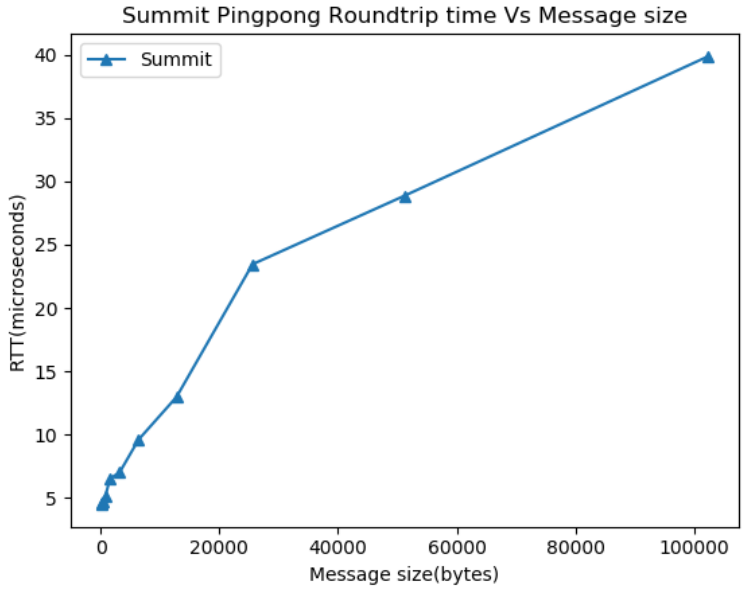

Figure 1.3: Summit Pingpong benchmark (linear scale)

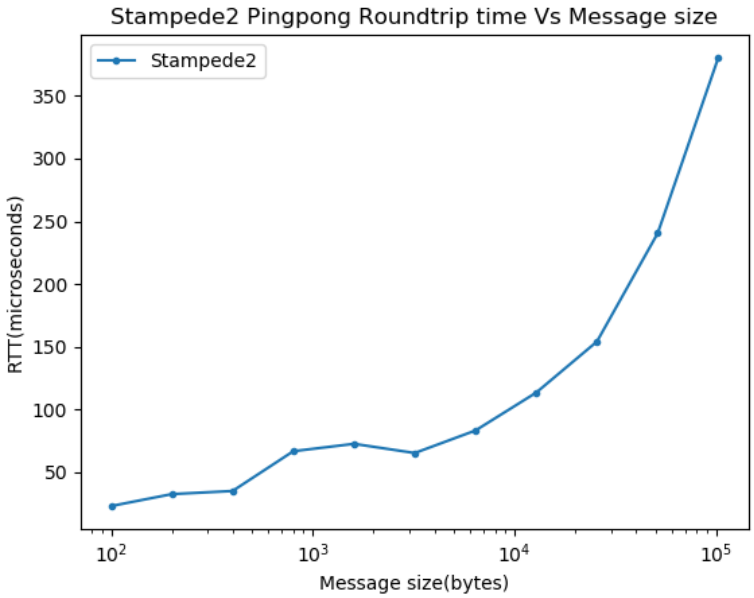

Figure 1.4: Stampede2 pingpong benchmark (log scale)

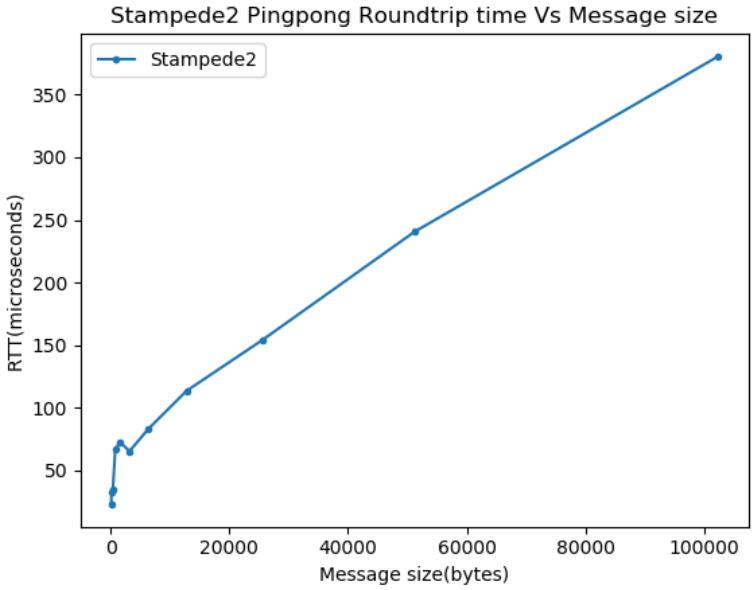

Figure 1.5: Stampede2 pingpong benchmark (linear scale)

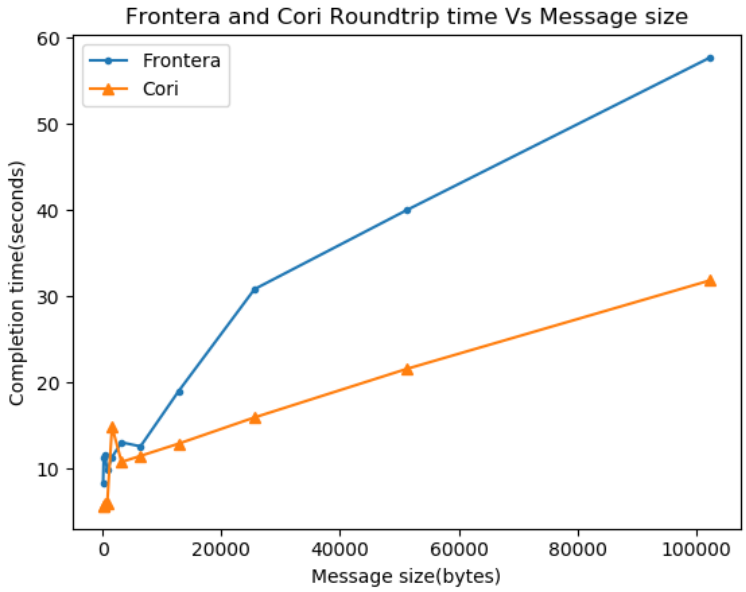

Figure 1.6: Cori and Frontera pingpong benchmark(linear scale)

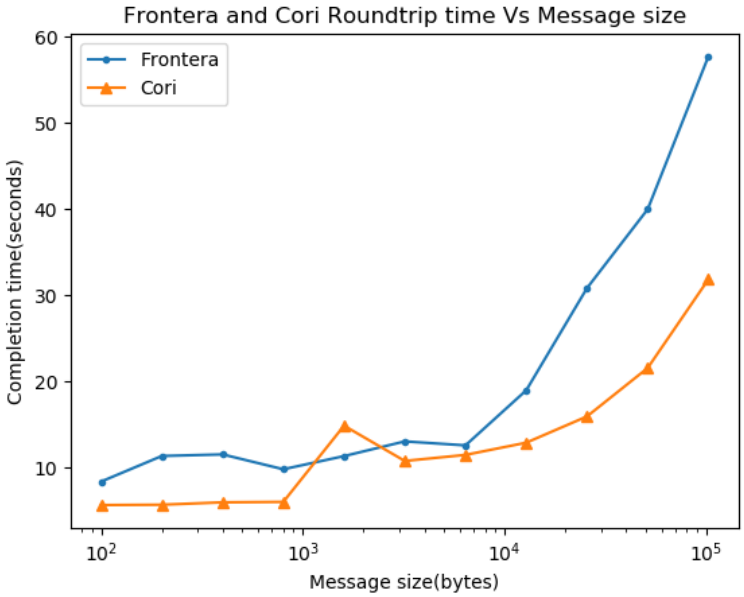

Figure 1.7: Cori and Frontera pingpong benchmark(log scale)

will compound the problem, resulting in severe slowdowns. To make matters worse, in real life there is no centralized entity who can route people to best use the road network. As a result, the average travel time will be much greater than is optimal.

In Figure 1.8, the upper road from source to destination has a fixed cost, while the lower road's cost scales with the fraction of the total people who use it. When entities act individually(and greedily), the average travel cost is 1(with everyone choosing the lower road), whereas a more intelligent, centralized, approach can achieve an optimal average travel cost of $\frac{3}{4}$.

On the other hand, public transportation(specifically, trains) does not particularly suffer from "congestion", barring occasional synchronization delays. Rather than worrying about connecting every single location where people may reside, it chooses a set of important points that a large enough fraction of people can access quickly, and builds a virtual "road"(rail) network connecting those. Trains leave stations at predetermined times, which ensures a near-constant travel time between these major points. If sufficiently many such points exist, then the time taken for people to travel to/from these points can be bounded, resulting in a reliable average travel time from any point A to B, which may even be better than a sufficiently congested private transport network.

Based on this, one method to reduce the cost of communication is to reduce the number of message send events, which requires combining multiple messages. This technique is applicable when applications are sending many small messages that are not individually on

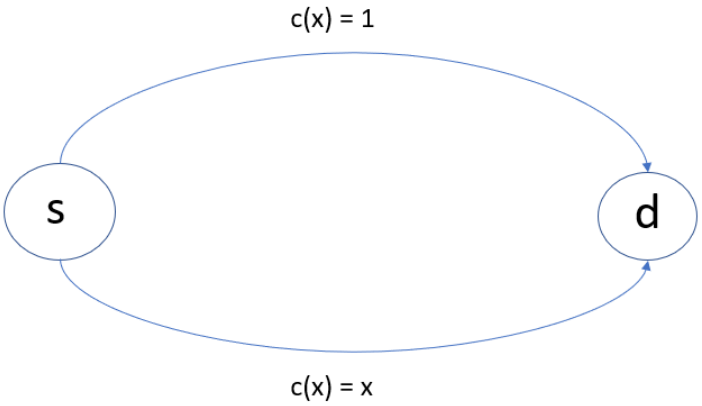

Figure 1.8: Pigou network

the critical path of execution. The idea is for applications to wait until a suitable number of messages are ready to be sent at once, and combine all of them into one large message, saving on having to send all of the individual message envelopes. Since we are dealing with bytes rather than people, it is possible for our virtual "trains"(or trams, as it were) to leave with messages ahead of schedule if they fill up. For liveness purposes(much like with trams maintaining a schedule), this optimization still requires a timeout period to ensure a maximum message latency(and hence, a synchronous system).

If we were to take the real-life analogy to (selfish) routing problems further, performance could theoretically be micro-optimized by choosing to only buffer some fraction of the messages. It turns out that the constraints on dealing with messages in the general case will end up requiring (large enough) messages to be sent directly to the destination without going through aggregation.

1.3 PRIOR WORK

This messaging scheme has been implemented in the Charm++ programming framework already, in the Topological Routing and Aggregation Module [1] (TRAM). TRAM supports this optimization for multiple virtual network topologies, for arbitrary messages of fixed(here meaning "known at compile time") length. The fixed-length restriction on messages permits defining the number of outgoing messages that are aggregated into one as the buffer size required to enable this is easy to compute. The sequence of events for the case of TRAM using a "1D" virtual network topology(discussed in chapter 2) follows:

- User invocation of the entry method - TRAM only supports aggregation of single parameter entry methods(which is sufficient in the general case to support arbitrary

types as long as they can be serialized into a compile-time constant number of bytes).

- Marking message as in-flight for quiescence detection purposes
- Constructing TRAM-level metadata for the message - The destination of the message is used to construct the route, and the first/next hop of the object along the route is used to determine the buffer it is added to
- Storing the data in TRAM's message buffer - Alongside the payload, the source PE, destination PE, and the destination object ID are stored in the TRAM-level message. Finding the destination PE requires a lookup in the Charm++ global location manager. However, in environments with load balancing, it is possible for the global location manager's table to be out of date, and hence TRAM has a facility to forward misdelivered messages to their new destination.
- Full/timed-out buffers are converted into messages, with associated envelopes - it is possible to reduce message size of a fixed-size message further at this stage by trimming unnecessary bytes from the data buffer. Envelopes include information about the source/destination of the buffer as a whole. TRAM's aggregators are implemented as Charm++ groups(one per PE), and hence they do not generally have to worry about outdated locations.
- The message is sent to its destination, via the Charm++ messaging framework - this step involves going through the machine-specific network layer.
- The message is unpacked at the receiving end, and is given to the local instance of the TRAM aggregator.
- The aggregator examines the TRAM-level metadata of the message, and determines the destination object, and marks the message for reforwarding if it has been misdelivered.
- If the message is at the correct PE, the aggregator marks the message as received for quiescence detection purposes.
- The aggregator invokes the entry method marked for aggregation with the payload of the message - this step includes deserialization of the message payload.

1.4 THIS WORK'S CONTRIBUTION

The natural extension to consider is the case of variable-sized messages, which are required by many applications. When an upper bound on the amount of data that needs to be send in

a message is known, this behavior can be approximated by sending fixed-size buffers of this size, but this may end up wasting considerable network resources, and in the cases of certain applications, may even end up overloading the network. A more accurate implementation of this functionality is required to reap the benefits of aggregation.

A wrinkle introduced when dealing with variable-sized messages is that there is no longer a one-to-one correspondence between the number of messages and the size of the buffer required to hold them, because variable-sized messages are unlikely to precisely fill up a buffer in the general case. An extra parameter must be introduced - the threshold fraction of the buffer that once filled, results in the buffer being sent. Selecting these 2 parameters inherently defines the largest possible single message that TRAM can afford to buffer, though performance may be improved by having a lower cutoff size beyond which messages are sent directly instead of waiting in the buffers. These hyperparameters are referred to as the buffer size(b), threshold fraction(f_t) and cutoff fraction(f_c) respectively, and obey the constraint:

$$f_t + f_c \leq 1 \tag{1.3}$$

The message trimming optimization is not as straightforward to fully transfer from the fixed-size to the variable-size case, because there are multiple "variable-length" buffers in the payload(the data and the offsets).

The work covered in this thesis includes the implementation of this fine-grained variable-sized message aggregation scheme, some analysis of how to select good hyperparameters for TRAM, and experiments validating its performance.

CHAPTER 2: TOPOLOGIES, PRACTICAL OPTIMIZATIONS AND IMPLEMENTATION DETAILS

2.1 THE NEED FOR VIRTUAL TOPOLOGIES

The term "message aggregation" has been used several times in this thesis so far, but it hides an entire avenue of optimization. The description given previously, taken at face value, would indicate that every processing element(PE) would have outbound buffers corresponding to every other PE in the system, and separately aggregate messages going to each of them. Such a naïve implementation is referred to as a "1D grid" topology. This works when every pair of PEs communicate regularly with small messages that can be aggregated, and at a roughly uniform rate.

However, this is not the case with most applications. There will be patterns in the destinations of outgoing messages, and not every pair of PEs will be (actively) communicating. The analogous construct in the field of computer networks - the fully connected mesh - is known not to be a one-size-fits-all solution. As such, this message aggregation can be organized in a hierarchical manner to better optimize for reality. One such family of approaches is the "grid" - arranging the PEs into a virtual grid topology, and aggregating messages only between pairs of PEs that are directly connected on this grid. This adds an extra routing step to the message aggregation layer, as messages between 2 PEs that are not directly connected must go through intermediate hops. However, this can be mitigated by including these intermediate messages in the aggregation process.

The reduction in messages traversing the network caused by using a grid topology is quantifiable [1], and in general, for a P-processor system where most pairs of processors communicate, an $n - D$ grid can reduce the asymptotic number of outgoing messages from any given process in a unit interval from $O(P)$ to $O(\sqrt[n]{P})$.

An illustration of this virtual topology and aggregation is shown in figures 2.1 and 2.2.

2.2 IMPLEMENTING AGGREGATION

As the module must convert arbitrary user-level messages into TRAM-compatible data, the Charm++ pack/unpack(PUP) object (de)serialization mechanism is used. This requires users to write PUPers(Charm++ (de)serializers), and to coalesce all the data they wish to send into a single structure/class. In general, this can be done without requiring any extra copies, but may require some changes in the surrounding user code.

Additionally, because variable-sized message handling comes with overhead not required

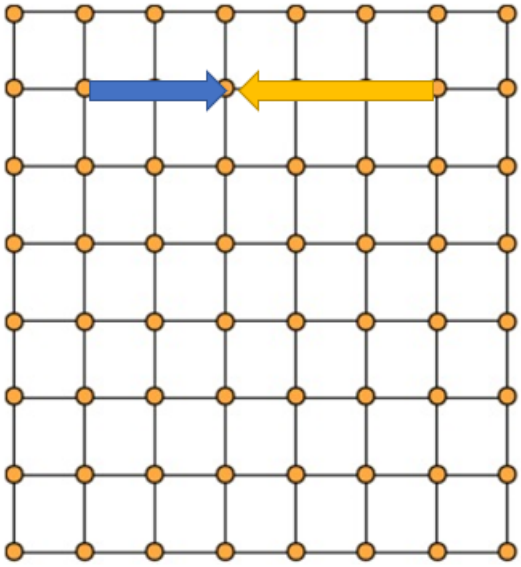

Figure 2.1: TRAM 2D grid first hop

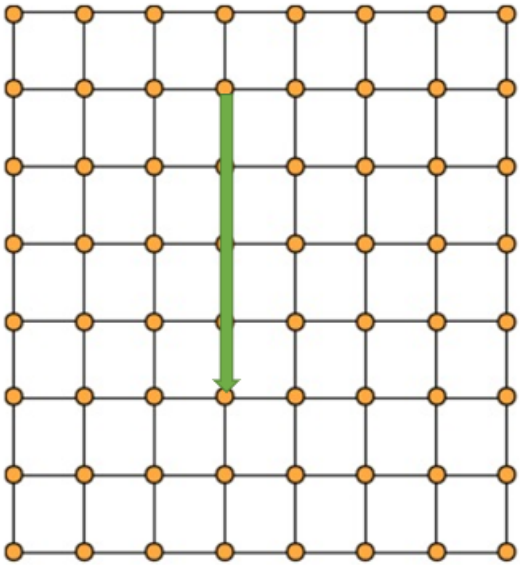

Figure 2.2: TRAM 2D grid second hop

in the fixed-size case(specifically, dealing with the offsets of individual messages in buffers), it is possible for users to opt-in to fixed size message handling.

To be placed into and recovered from a TRAM buffer, each message's source PE, destination PE, destination object ID, and offset within the buffer need to be stored alongside the TRAM data buffer. These are referred to as the TRAM-level metadata.

One implementation concern is minimizing the number of data copies performed. TRAM passes the user data around by reference internally while it generates the metadata and enforces the threshold and cutoff conditions, before performing a single copy into its buffer. This approach also resolves the issue of extraneous memory (de)allocations, as the only ones performed are the buffers themselves, and once (unavoidably) at the receiver in order to reconstruct the message object.

CHAPTER 3: APPLICATIONS AND LIMITATIONS FOR AGGREGATION

Message aggregation is not universally applicable - a system must have "many" "small" messages to benefit from it. Applications that deal with large amounts of data are likely to meet the "many" requirement, since they are more likely to involve heavy communication. However, the definition of "small" depends on the specifics of the machine, the communication stack, and the network underneath.

One of the benchmarks used for validating the work done in this thesis - subsequently referred to as the synthetic benchmark - is helpful in making the definition of "small" concrete. On Summit, the 2-node synthetic benchmark shows performance improvements with TRAM for messages sizes ≤ 1000 bytes for an effective buffer size of $10kB$ bytes.

Stencils are a common pattern in scientific computing. A stencil computation is an iterative procedure on some "shaped" data(here, referring to data in cells that have some spatial relationships with the other cells) where in each iteration, a cell is updated based on its own value and the value of its neighborhood. One example is a five-point Laplacian stencil for data in a 2D grid, where h is the physical distance between adjacent points.

$$\nabla^2 f(x, y, t+1) = \frac{f(x-1, y, t) + f(x+1, y, t) + f(x, y-1, t) + f(x, y+1, t) - 4f(x, y, t)}{h^2} \tag{3.1}$$

where $f(x, y, t)$ represents the value of cell at $[x, y]$ after iteration t, and $\nabla^2 f$ similarly refers to the Laplacian.

Stencil-like applications are a good fit for aggregation, as they must exchange a large number of messages by definition, with the individual messages not being very large, particularly if the cells are small. The LeanMD benchmark, discussed in chapter 5, is stencil-like in the sense that the configurations tested usually involve communication with neighbors or cells 2 away from the source.

Another good candidate for aggregation is a computation reliant on all-to-all communication. Two of the benchmarks discussed in this thesis, the synthetic benchmark and the HPCC Random Access benchmark, both fall under this category. In such code, the requests and responses are prime targets for coalescing, since the bytes reduced by not having to send all of their message envelope headers can add up quickly.

Graph algorithms are another candidate for message aggregation - particularly ones with a lot of per-vertex or per-edge communication. Given that the communication in graph algorithms tend to be heavily shaped by the structure of the graph, they require variable-sized message aggregation for best results.

Parallel discrete event simulations constitute another application domain for message aggregation. Typically, entities in simulation are fine-grained, exchanging tiny messages with other entities(such as a car leaving an intersection in a traffic simulation, or a boolean value flipping in a digital circuit simulation).

One important point to consider when determining the viability of message aggregation is the control flow of the application. Aggregation is fundamentally about the tradeoff between latency and throughput - applications must be capable of paying the price of increased latency. If the critical path consists primarily of sending and receiving messages from a master node, enabling aggregation for such methods will lead to a latency hit from only sending messages at timeouts. This will wipe out any potential throughput gains.

CHAPTER 4: TUNING HYPERPARAMETERS

Table 4.1 defines the variables used to come up with a heuristic to tune TRAM's hyperparameters.

Consider a TRAM aggregator configured with an infinite buffer size i.e. it only flushes on timeouts.

During a period τ, a total of $Q\tau$ messages are generated, and they have a total volume of QS bytes. Plugging these values into the $\alpha - \beta$ model, cost incurred at the sender-side to send all messages generated at this PE during one latency period without TRAM is:

$$Cost = \alpha Q\tau + \beta QS\tau \tag{4.1}$$

In the same period, a TRAM implementation would have sent just one message, carrying all of these individual messages as a payload, and if we assume that messages arrive uniformly during this time, the additional waiting times experienced by messages is also uniformly distributed, with a mean value of $\frac{\tau}{2}$. The cost associated with this extra latency is captured in $c(t)$.

Hence, cost incurred at the sender-side to send all messages generated at this PE during one timeout period with TRAM as per the $\alpha - \beta$ model is:

$$Cost = \alpha + \beta QS\tau + Q\tau c(\frac{\tau}{2}) \tag{4.2}$$

Subtracting 4.2 from 4.1, we can compute the cost reduction caused by TRAM as:

$$Costreduction = Q\tau(\alpha - c(\frac{\tau}{2})) - \alpha \tag{4.3}$$

$\implies$ For TRAM to benefit the application, the cost reduction must be positive. Rear-

Variable	Definition
Q	Average per-PE message generation rate
S	Average message size
c(t)	Cost associated with an added latency of t
κ	linear latency cost factor
b	Size of TRAM buffer
f_t	Threshold fraction
τ	Timeout period

Table 4.1: Variables relevant to tuning hyperparameters

ranging the terms of this inequality, we obtain the constraint in 4.4

$$\alpha(Q\tau - 1) > Q\tau c(\frac{\tau}{2}) \tag{4.4}$$

The above analysis is still applicable for large enough buffer sizes, specifically ones that satisfy the constraint in 4.5

$$b \geq \frac{QS\tau}{f_t} \tag{4.5}$$

If the buffer size is smaller than the above threshold, TRAM will flush as soon as the threshold fraction is reached, instead of waiting for the timeout period, as on average, the number of bytes arriving in a period τ exceed the effective buffer size. Let the mean time to flush be τ'.

$$\tau' = \frac{bf_t}{QS} \tag{4.6}$$

$\implies$ Substituting this duration into equations 4.1 and 4.2, the cost reduction caused by TRAM in time τ can be computed as:

$$Costreduction = Q\tau(\alpha(1 - \frac{S}{bf_t}) - c(\frac{bf_t}{2QS})) \tag{4.7}$$

$\implies$ For TRAM to give an improvement,

$$\alpha(1 - \frac{S}{bf_t}) > c(\frac{bf_t}{2QS}) \tag{4.8}$$

i.e.

$$\implies \alpha > \frac{bf_t c(\frac{bf_t}{2QS})}{bf_t - S} \tag{4.9}$$

If we consider the cost of delay to be a linear function of the delay i.e.

$$c(t) = \kappa t \tag{4.10}$$

$$\implies \alpha > \frac{bf_t \kappa \frac{bf_t}{2QS}}{bf_t - S} \tag{4.11}$$

Rearranging the terms,

$$\frac{2QS\alpha}{\kappa} \geq \frac{(bf_t)^2}{bf_t - S} \tag{4.12}$$

$$\implies \frac{2QS\alpha}{\kappa} > bf_t \tag{4.13}$$

Here, α is a variable specific to the machine, and can be determined ahead of time. Q and S are highly-application specific, and in applications with distinct computation and communication phases, the average over the entire execution time may not be as useful when trying to accurately model behavior. κ (and the validity of the linear model of the cost of latency) depends on both the machine, and the application itself, as the tolerance for delays varies greatly with applications and system configurations.

CHAPTER 5: EXPERIMENTS

As with all optimizations, message aggregation is situational, and hence requires experimental validation of its claimed performance benefits. This has been achieved through separate different benchmarks - a synthetic one designed to simulate applications that send large volumes of messages, the HPCC Random Access benchmark, and the Charm++ LeanMD mini-app..

All experiments in this section were performed on OLCF's Summit. Summit nodes contain 2 IBM Power9 CPUs and 6 NVIDIA V100 GPUs(not relevant to this work). Each CPU has 21 physical cores, with 4 hardware hyperthreads per core. Importantly, the PAMI network layer on Summit does not support a dedicated communication thread. [3].

Most of the benchmarks discussed have also been run on TACC's Stampede2 supercomputer, which is a couple of years older, and has a different architecture. Stampede2 nodes are Knight's Landing nodes, with an Intel Omnipath 100 Gb/sec network structured in a fat tree topology[4].

Unlike on Summit, the network layer on Stampede2 does support dedicated communication threads, which leads to improved performance relative to non-TRAM implementations across the board.

5.1 A BIDIRECTIONAL CONTINUOUS STREAM OF DATA

The simplest case tested was whether aggregation would improve the throughput of a continuous bidirectional stream of data(here, a series of 10^6 messages in both directions) between 2 nodes. Here, latency would be irrelevant since waiting to process a message is not in the middle of a critical path. The time required for the contents of the stream to be received and handled on both nodes is used as the measure of performance in this benchmark. Figure 5.1 summarizes the results from performing this experiment.

We can see that while the behavior of the regular implementation does not change much for the message sizes tested, the TRAM-enabled implementation performs visibly worse in this scenario where messages are being continuously generated. We hypothesize that this is caused by the nature of the benchmark, as due to the absence of a dedicated communication thread on PAMI-based network layers such as that of Summit, a TRAM-enabled implementation still experiences the additional overhead associated with buffering the messages(such as memory costs), while also being incapable of overlapping the generation and sending of messages due to not yielding control to the scheduler during the generation phase. However,

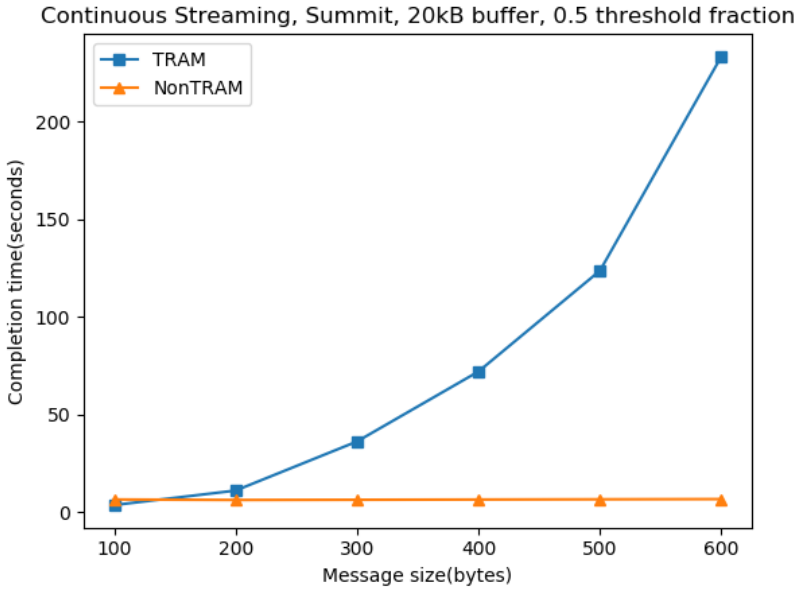

Figure 5.1: Continuous stream of data on Summit

even here, TRAM performs noticeably better on very small messages (size ≤ 100 bytes).

A mitigating factor for the results of this experiment is that it is rare for processors to send arbitrarily large amounts of data to each other without requiring responses/computation of some kind(whether it be direct or indirect). The next benchmark attempts to simulate a slightly more realistic application, where processors must acknowledge(ACK) data they receive, and hence permit the Charm++ scheduler to get involved.

The results of the continuous streaming 2-node benchmark on Stampede2 with $m = 10^5$ messages sent from each node are shown in Figure 5.2. The results are radically different from those achieved on Summit - the TRAM-enabled implementation performs significantly better than the regular version, unlike on Summit where performance was notably worse.

5.2 A BIDIRECTIONAL STAGGERED STREAM OF DATA

Requiring ACKs for every single message(which would make the benchmark very similar to the classic pingpong) will result in failing to meet the requirement that message processing not be a bottleneck in the the critical path. Instead, a variation where cumulative ACKs are sent for batches of 100 messages is more useful, as aggregation can still occur. The variation of the time to completion of the system with the size of packet is shown in Figure 5.3.

Despite the system being quite small(only consisting of 2 processors sending data to each other), we can see that the TRAM implementation performs noticeably better for "small"

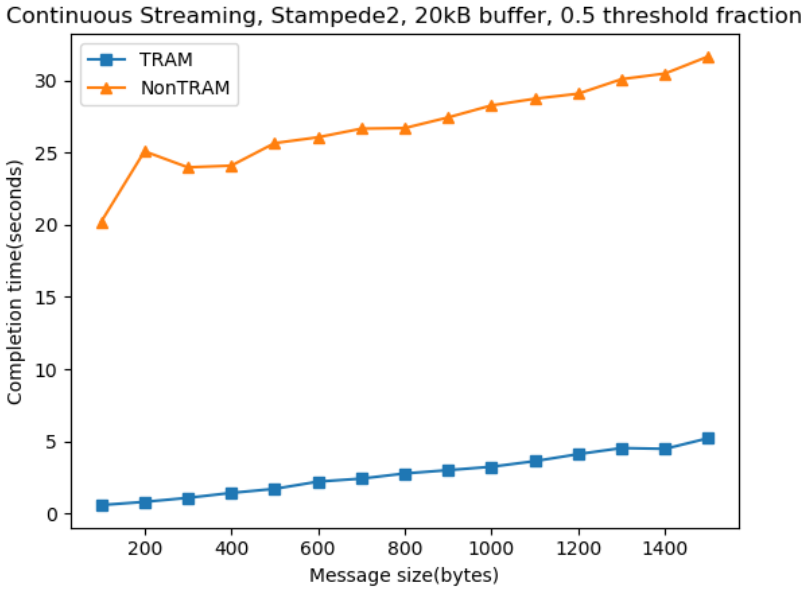

Figure 5.2: Stampede2 Continuous Streaming

messages of upto 1000 bytes, where the performance curves cross over. The effective buffer size of TRAM for this experiment is 10 kB, which would store 10 messages of this size before needing to flush. This validates the idea that TRAM can produce throughput improvements when there are "small" messages being sent.

The results of the 2-way ACKed synthetic benchmark on Stampede2 are shown in Figure 5.4. The performance improvement of TRAM here is unambiguous. Unlike with Summit, performance improvements persist even beyond 1000 bytes, with around a 2x improvement at 1400 bytes.

5.3 THE SYNTHETHIC BENCHMARK

The previous experiment can be easily generalized to an arbitrary number of processors, with each node sending data to all other nodes. To avoid changing the total number of messages for different data points(thanks to divisibility issues), while nodes send acknowledgements for every 100 messages they receive, they only continue sending messages once they receive acknowledgements for a total of $100*(P-1)$ messages(i.e. from every other node in the system). The total number of messages sent during the execution of the benchmark is given by equation 5.1.

$$m_{total} = m * P * (P-1) \tag{5.1}$$

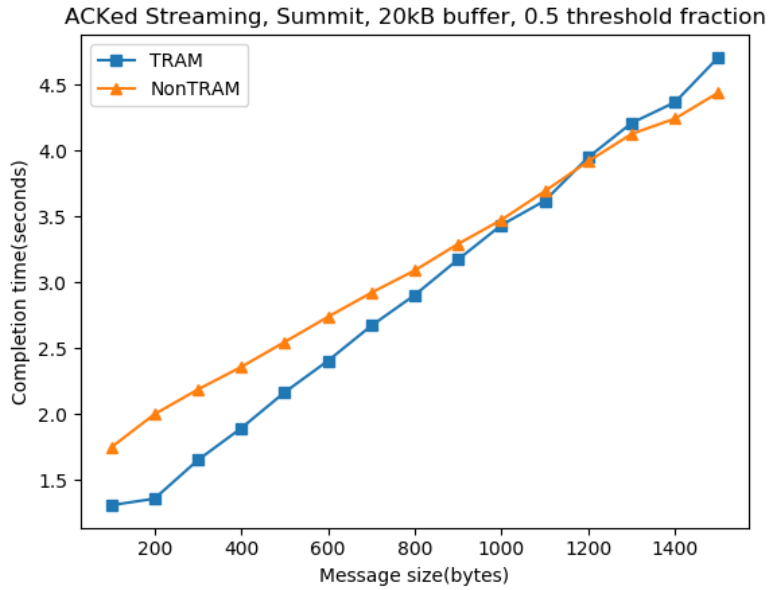

Figure 5.3: Staggered streaming of data on Summit

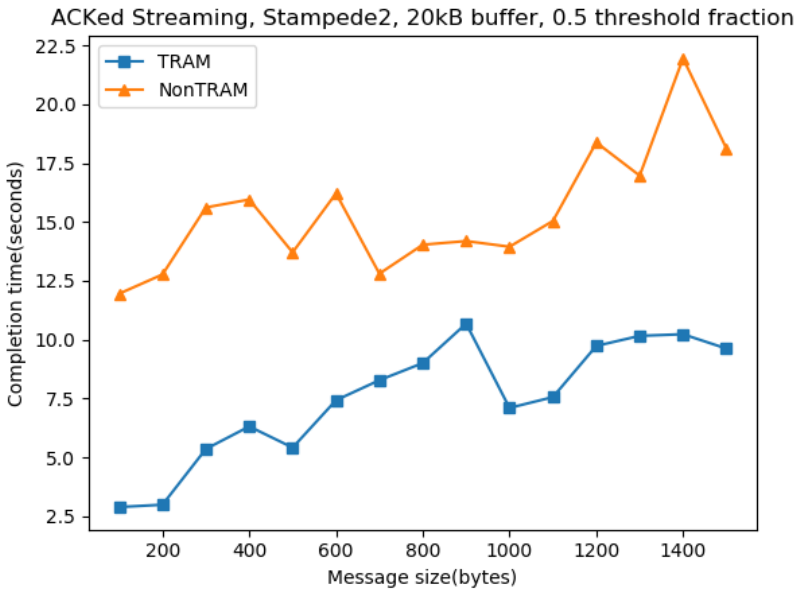

Figure 5.4: Stampede2 ACKed Streaming

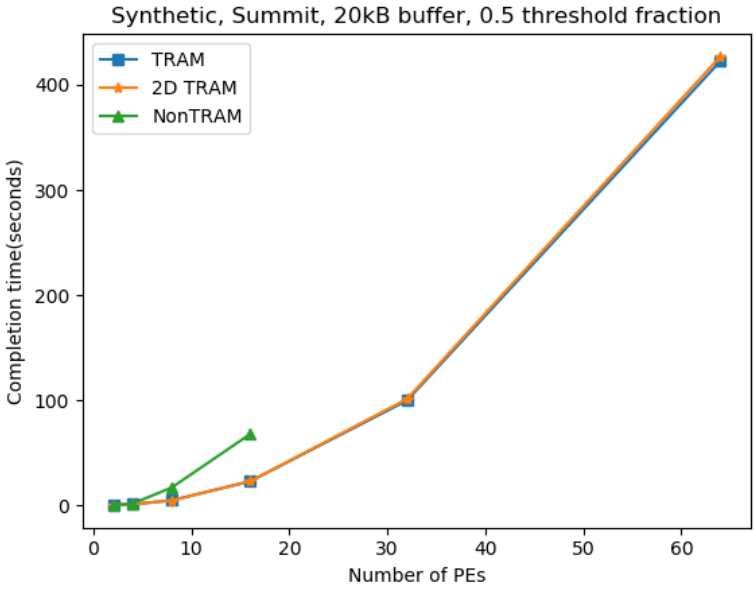

Figure 5.5: Synthetic benchmark on Summit

where m is analogous to the number of messages sent from a processor to another in the previous 2 benchmarks.

The total number of messages sent in the execution of the synthetic benchmark grows quadratically with the total number of processors in the system. This means that as the size of the system increases, we would expect a solution that scales well to have a roughly linear increase in execution time. The results of an experiment with $m = 10^6$, and message size = 100 bytes are given in Figure 5.5 .

We can see that the synthetic benchmark is best suited to a 2D topology. Data for > 16 PEs is not present for the non-TRAM implementation, because the execution time is more than half an hour.

This benchmark clearly demonstrates TRAM's improved scaling on messaging-intense workloads, and hence validates that TRAM can produce performance improvements when dealing with "many" messages.

The results from running the synthetic benchmark, scaling the number of PEs from 2 to 64 on Stampede2 are shown in Figure 5.6. Once again, the performance improvement of TRAM is quite significant, with TRAM-less implementations experiencing drastic slowdowns even at a double digit number of PEs.

When dealing with small messages and a sufficiently large buffer size, the cutoff fraction is mostly irrelevant(since it exists more to enforce correctness than for performance's sake). The variation of performance of the TRAM implementation with different effective buffer

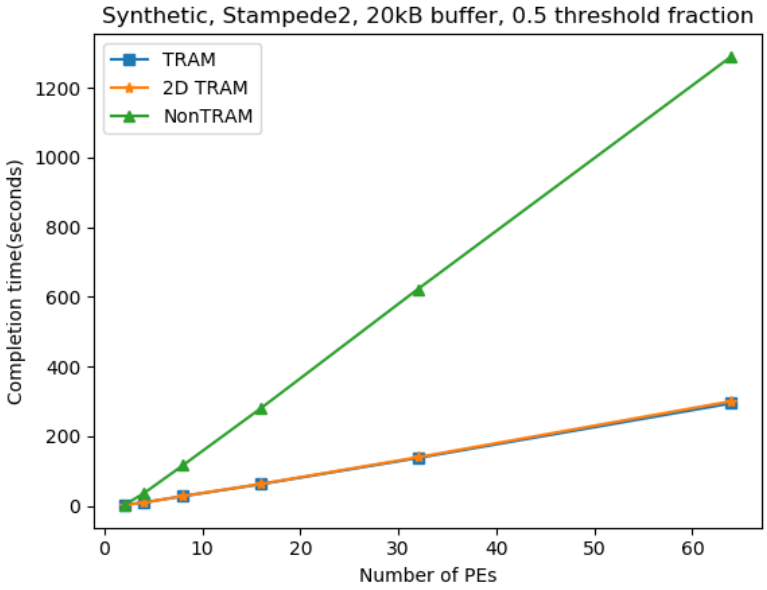

Figure 5.6: Stampede2 Synthetic benchmark

sizes(the product of the true buffer size and the threshold fraction) and virtual topology is shown in Figure 5.7 .

This version of the synthetic benchmark was run with $m = 10^5$, and a message size of 100 bytes. Buffer sizes larger and smaller than shown both resulted in running times of longer than half an hour for both topologies. This experiment once again demonstrates a clear performance gain from using TRAM with a 2D grid topology. It also suggests that a smaller effective buffer size yields better performance(up to a point).

The results from varying the effective buffer size for the synthetic benchmark, with 64 PEs on Stampede2 are shown in Figure 5.8. These results agree with those in Summit, supporting the hypothesis that a lot of the performance benefits of TRAM are achieved even with small buffer sizes.

One hyperparameter that has not been discussed so far is the timeout period of TRAM. This is because in the applications that derive performance improvements from aggregation, the timeout period is only relevant at "boundaries" - near the start or the end of a message exchange phase.

5.4 REAL WORLD BENCHMARK: HPC RANDOM ACCESS

The HPC RandomAccess benchmark [5] was one of the primary real-world tests of the new TRAM implementation. It is designed to measure the throughput of random integer

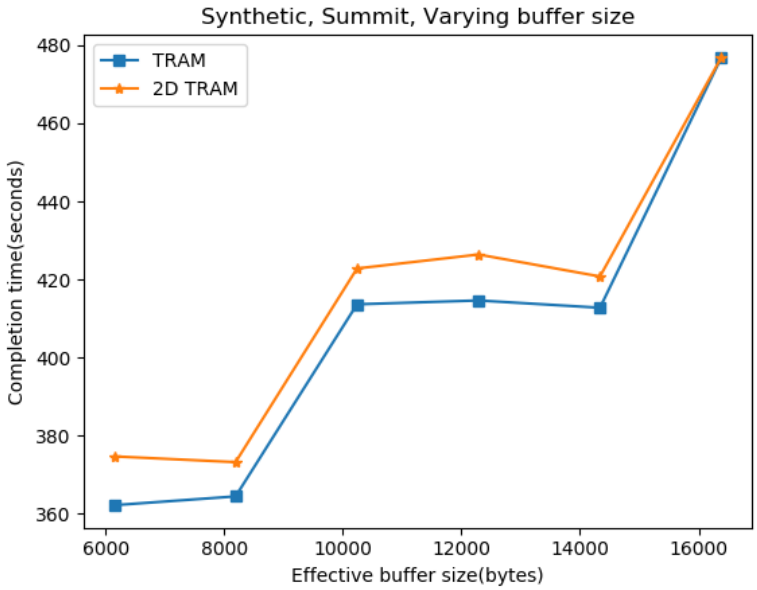

Figure 5.7: Summit buffer size variation on the synthetic benchmark

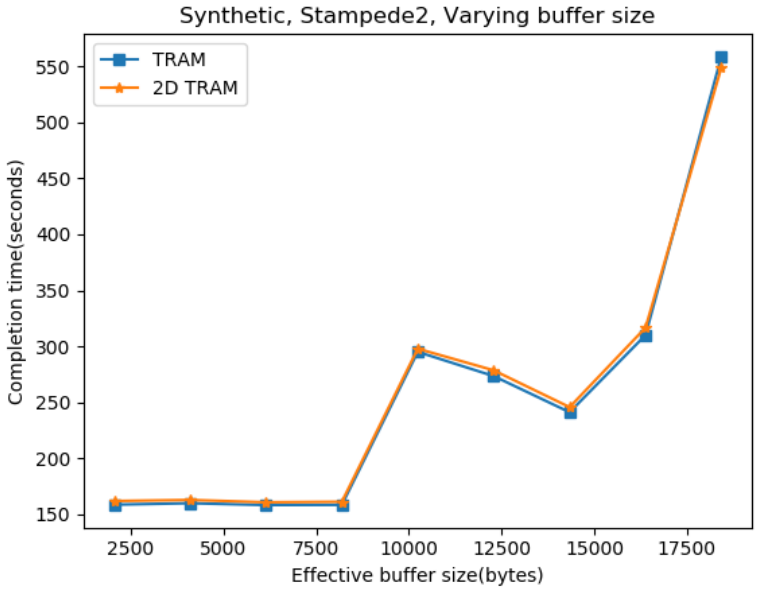

Figure 5.8: Stampede2 Synthetic benchmark, 64 PEs, varying buffer size

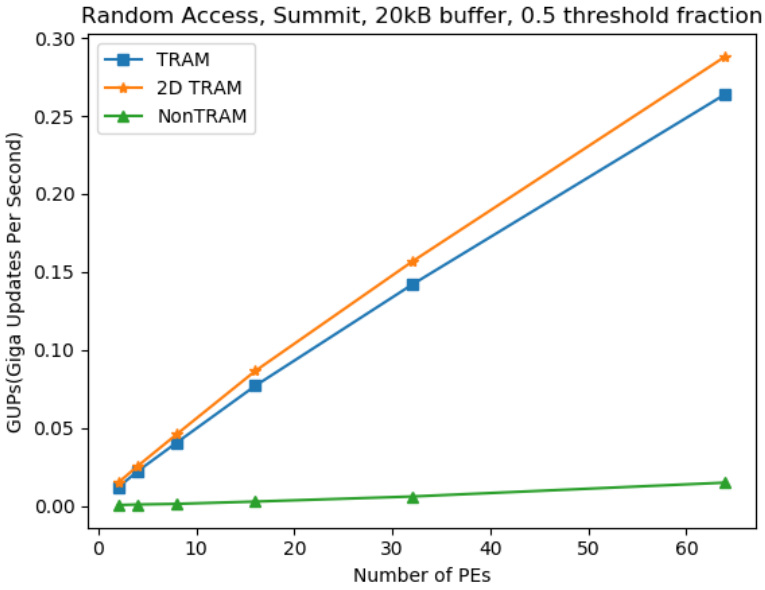

Figure 5.9: RandomAccess benchmark on Summit for table size of 2^{14} words, varying PEs

updates of memory, by issuing randomly targeted update requests from each of the nodes maintaining a distributed table. This all-to-all communication pattern makes it a good fit for TRAM.

5.4.1 Scaling With Number Of Nodes

The scaling of the benchmark with an increase in the number of PEs for various TRAM topologies(configured in variable-sized mode), compared to the vanilla implementation is shown in Figure 5.9 .

The results agree with the previous experiments, with TRAM with a 2D grid virtual topology performing best, and both implementations performing significantly better than the vanilla version.

The performance of the Random Access benchmark on Stampede2, varying the number of PEs from 2 to 64 is shown in Figure 5.10

TRAM performs and scales better than the regular implementation. Both 1D and 2D TRAM perform very similarly, though it is interesting to note that 1D TRAM performs better as the number of PEs goes up.

The relatively small table size (less than twice the effective buffer size) means that the TRAM implementation would likely have faced timeouts during execution as well, suggesting that a repeat of this experiment on a larger table size would demonstrate an even larger

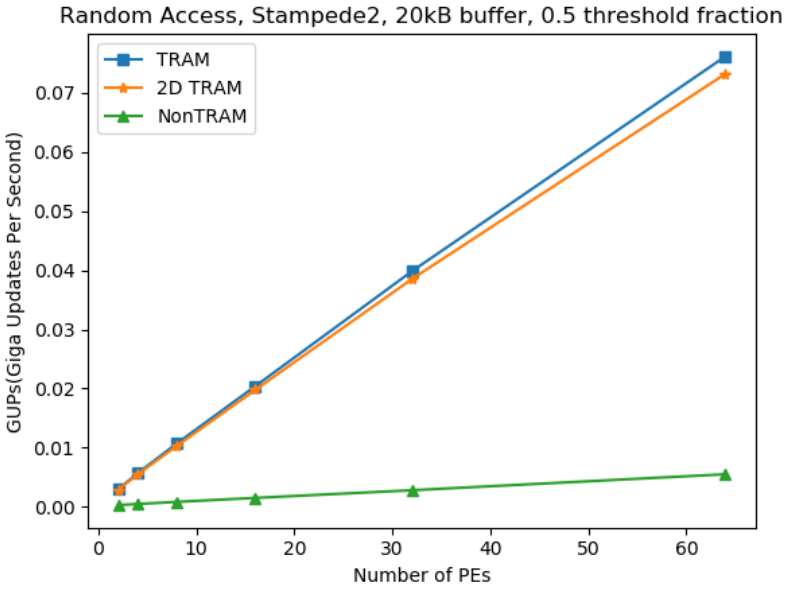

Figure 5.10: Stampede2 Random Access

performance gap on both Summit and Stampede2.

The variation of throughput of the benchmark with the effective buffer size is shown in Figure 5.11.

Similar to the synthetic benchmark, smaller buffer sizes seem to perform best.

5.5 MINI-APP: LEANMD

In chapter 2, stencil-like applications(i.e. applications were communication is mostly between processors that are in each other's virtual neighborhood) were mentioned as potentially benefiting from message aggregation. The Charm++ LeanMD [6] mini-app was chosen to test this idea.

LeanMD is a molecular dynamics simulation based on the Lennard-Jones potential, which is an effective potential between uncharged molecules or atoms. Force calculation in L-J dynamics is done within a cutoff radius, which is accomplished by spatially partitioning atoms in the simulation. LeanMD supports finer-grained partitioning of systems with a "k-away" parameter, that controls the size of the spatial buckets to enforce that cells must communicate only with other cells that are at most k away.

The results from running LeanMD for 100 steps in a 2-way configuration, with particle migration every 5 steps, on systems scaling up at a fixed 16 cells/PE, from 2 PEs to 64 PEs, is shown in Figure 5.12.

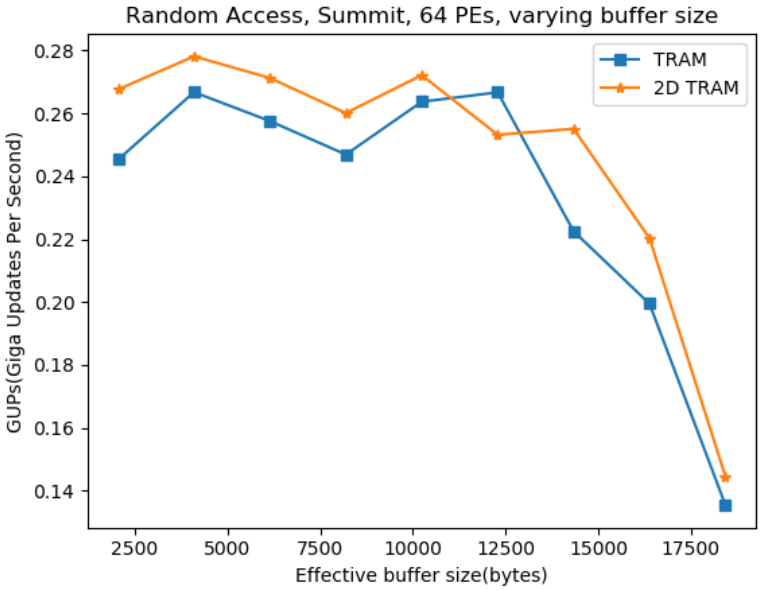

Figure 5.11: RandomAccess benchmark with 64 PEs and table size of 2^{14} words, varying buffer size

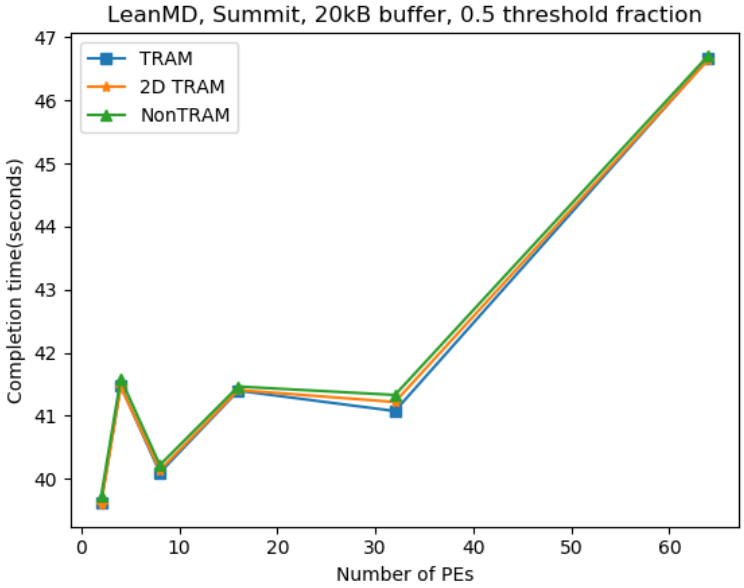

Figure 5.12: LeanMD benchmark, maintaining a ratio of 16 chares per PE from 2 to 64 PEs

The improvements from using TRAM here are modest, as LeanMD does not exchange as many messages as some of the previous benchmarks. Nevertheless, there is an unambiguous performance improvement from using both 1D and 2D TRAM, with 1D TRAM performing slightly better. This can be explained by considering the fact that in a stencil-like application, there are few outgoing message destinations to begin with, and hence the $\sqrt{P}$ factor reduction in the number of outgoing messages is not applicable.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

This work discussed the motivation behind generalizing fine-grained message aggregation to variable-sized messages, and the changes to the theoretical model required to reflect the relaxed assumption. An implementation of the optimization in the Charm++ programming framework has been completed, and the work also demonstrates variable-sized message aggregation's performance characteristics on a number of communication-intensive benchmarks. The implementation of variable-sized message aggregation supports chare arrays in Charm++, with 1D and 2D virtual grid topologies, and user-specified hyperparameters.

Some future work will be to extend the feature set supported by variable-sized TRAM to include supporting other Charm++ collections, more virtual grid topologies, and establishing stronger rules for determining optimal hyperparameters.

There are also some optimizations discussed in the original implementation of TRAM that could yield further performance improvements, such as replacing PE-level aggregation with host-level message aggregation, which could increase the rate of flushing of TRAM buffers by virtue of increasing the number of sources of messages to each buffer, and possibly adopting adaptive hyperparameter tuning strategies, such as dynamically adjusted buffer sizes.

APPENDIX A: THE CHARM++ PROGRAMMING MODEL

Charm++[6] is an object-based parallel programming framework derived from C++. Charm++'s adaptive runtime is responsible for making scheduling and load balancing decisions. This permits separation of application logic from machine-specific parallelization strategies. Charm++ has an asynchronous execution model as opposed to MPI, which permits more parallelism.

Charm++ requires the specification of an "interface" in the form of .ci file in order to encode Charm++-specific information that cannot be expressed in C++ alone, such as information about certain kinds of objects, and annotations of entry methods(examples of which are reductiontarget to label an entry method that can process the result of a reduction operation, and aggregate, which enables TRAM's aggregation for all invocations of a particular entry method). Interface files are compiled into C++ header files (decl.h and def.h) by a Charm++-specific parser that generates all the associated boilerplate code.

To separate concerns of program specification and optimization, Charm++ programs are written in an overdecomposed manner - application developers intentionally separate their logic into smaller pieces of code(and hence, multiple schedulable entities) that are capable of executing independently, without concerning themselves with the attributes of any one machine. The runtime decides where to schedule these overdecomposed entities based on statistics gathered during execution.

Charm++ defines a processing element(PE) as a logical equivalent of a CPU core, and schedules tasks(message handling) onto these PEs. The lower layer of Charm++ maps these PEs onto the physical CPU cores available to use, possibly in a many-to-one manner. Charm++'s runtime is designed to have at most as many PEs as cores available to it, since each PE's scheduler acts as if it is the only entity with access to its assigned core. Contention between PEs for a CPU core can result in large, unpredictable slowdowns.

Programs, ultimately are executed in the form of processes. Charm++ supports 2 primary methods of dividing PEs into processes. One is to put each PE into its own process, and consequently have an instance of the full runtime on every single PE. This can be rather wasteful, and reduces the overlap of computation and communication that can occur on any PE. The alternative is to group a number of PEs into each process(known as symmetric multiprocessing, or SMP), and optionally set aside one additional core to handle communication on behalf of all of the PEs. The exact number of PEs to group into a process is something that must be determined experimentally, based on the amount of communication the application must perform and the characteristics of the machine executing the program.

Under the asynchronous execution model, all such method invocations occur in the form of message-passing, with the methods effectively being syntactic sugar over message handlers. One advantage of this is that it reduces the amount of time that CPUs idle. As long as any of the objects scheduled on a PE have messages awaiting processing, the corresponding physical core has work to do. Additionally, since sending messages is a non-blocking operation(unless the programmer explicitly opts for blocking messages), the scheduler can maximize the overlap between the communication and computation of objects on any PE.

In general, Charm++ does not encourage the use of mutable global variables, to avoid shared memory-related issues and pessimizations, preferring message passing.

Schedulable objects in Charm++ are handled in the pure C++ code in the form of "proxies" to the actual object. Entry method invocations are performed by invoking the corresponding method on the proxy object. In the C++ code, schedulable object classes must inherit from their corresponding proxy class.

The most basic of objects that have a scheduling identity at runtime in Charm++ are referred to as "chares". Chares are the smallest unit capable of sending and receiving messages, and they encapsulate any state that they require. All objects with a scheduling identity must specify an interface that describes the methods of theirs that they expose to other objects.

It is rare for a constant number of unique chares to be all that an application consists of. Chares can be aggregated into collections of objects that share a common interface, and Charm++ automates several collective operations on these collections, such as broadcasts(syntactically represented by invoking an entry method on the proxy representing a collective, rather than on any of the chare elements) and reductions(which are implemented by making all PEs involved "contribute" their value, and indicate the callback that will process the result of the reduction).

The most common types of collections of chares used in Charm++ are arrays and groups. Arrays can include arbitrary numbers of chares, and can be scheduled onto execution units in any way the runtime sees fit. Groups, on the other hand, are restricted to have exactly as many elements as the runtime has PEs, with one element assigned to each PE. A single TRAM-enabled entry method's aggregator is an example of a Charm++ group.

A less frequently-used Charm++ collection is the nodegroup, which as the name suggests, is a "group" for nodes rather than PEs i.e. exactly one chare of the nodegroup is present on any physical node. Nodegroup semantics are different from the rest of Charm++, and are meant to be used as a tool for low-level optimizations rather than for general-purpose code.

Execution of Charm++ code begins with the constructor of one chare specially marked as a mainchare, analogous to the method main() in regular C++ code. At any given time(with

a few exceptions, primarily nodegroups), only one entry method of a chare may execute. This removes the need for expensive locking operations/synchronizations to access the state encapsulated in the object.

APPENDIX B: VARIABLE-SIZED TRAM USER MANUAL

The original fixed-size implementation of TRAM featured support for multiple Charm++ collection types, and had two APIs for marking entry methods for aggregation - a concise way that required only marking an entry method, at the cost of not being able to tune hyperparameters, or a more verbose method involving explicitly instantiating the TRAM aggregators that provided more fine-grained control.

The variable-sized implementation of TRAM only supports Charm++'s chare arrays, but brings together the best of both APIs. Users can mark an entry method for aggregation by using the [aggregate] entry method tag, which can be given any subset of the following parameters:

- bufferSize - As the name suggests, this allows defining the size of a single TRAM buffer in bytes.
- numDimensions - Selects the virtual topology used by TRAM for routing from one of 1D and 2D grids.
- thresholdNum - Numerator of the threshold fraction, which is the maximum portion of the buffer which can be filled before it is flushed to destination.
- thresholdDen - Denominator of the threshold fraction
- cutoffNum - Numerator of the cutoff fraction, which is the maximum fraction of the buffer size an individual message can be, above which the message is not stored in the existing buffer, but is instead sent as part of a separate one.
- cutoffDen - Denominator of the cutoff fraction
- timeout - Timeout period for flushing TRAM buffers to maintain liveness, in microseconds

An example of an annotated entry method is:

```
entry [aggregate(bufferSize: 20480,
numDimensions: 2, thresholdNum: 9,
thresholdDen: 10,
cutoffNum: 1, cutoffDen: 10,
timeout : 10)] void ping(vector<int> msg);
```

Charm++'s Quiescence Detection-based termination detection is the sole supported method to use variable-sized TRAM.

TRAM can be enabled in fixed-size mode by defining the is_PUPbytes type trait for a given data type, and setting the threshold fraction numerator to be equal to the threshold fraction denominator as shown:

```
template <>

struct is_PUPbytes<dtype> {

static const bool value = true;

};

entry [aggregate(bufferSize: 1024,
thresholdNum : 1, thresholdDen : 1)] void ping(dtype v);
```

REFERENCES

[1] L. Wesolowski, "Software topological message aggregation techniques for large-scale parallel systems," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2014.

[2] *Data courtesy Nikhil Jain, presented at Salihshan by L Kale*, 2018.

[3] *OLCF*, 2019. [Online]. Available: https://www.olcf.ornl.gov/summit/

[4] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, and P. Teller, "Stampede 2: The evolution of an xsede supercomputer," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, ser. PEARC17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3093338.3093385

[5] J. Dongarra and P. Luszczek, "Introduction to the HPC Challenge Benchmark Suite,," presented at Supercomputing, Seattle, Washington, Nov. 2005.

[6] 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 for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014. [Online]. Available: https://doi.org/10.1109/SC.2014.58 p. 647–658.