SAPIENZA
UNIVERSITÀ DI ROMA

Implementation of Allreduce algorithms on the Tenstorrent Wormhole n150

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica
Applied Computer Science and Artificial Intelligence

Charlie
ID number

Advisor

Academic Year 2024/2025

Implementation of Allreduce algorithms on the Tenstorrent Wormhole n150
Sapienza University of Rome

This thesis has been typeset by LaTeX and the Sapthesis class.

Author's email:

Coming here and doing this was neither the obvious nor the easy choice. Thanks to my partner, friends, family (old and new), supervisor, and the eternal city, I would do it again in a heartbeat.

Abstract

To address increasing demands for parallel compute, new hardware, and new algorithms that effectively utilize that hardware, are being developed. The Tenstorrent n150d, a new 72 core accelerator board specifically designed for vectorized computation, has been utilized for the comparative assessment of multiple collective communication procedure implementations.

The communication procedure being assessed is allreduce, and five different algorithms (Latency Optimal Swing, Latency Optimal Recursive Doubling, Bandwidth Optimal Swing, Bandwidth Optimal Recursive Doubling, and Shared-Memory) were implemented using TT-Metalium. The algorithms were compared, with a clear result that both Bandwidth Optimal implementations performed significantly better, with the more specific result that generally the Recursive Doubling implementation performed slightly better than the Swing algorithm. The results approach the theoretical optimal performance possible on the n150d.

The reasons for the resultant performance are explored, and given the specific hardware topology, the results are in line with expectations. The details of implementation using this hardware and software are analyzed, and challenges and shortcomings discussed.

Contents

Chapter 1

Introduction

1.1 Background and motivation

As computer science has grown and matured as a technology, one thing has been consistent. Computational power and computational demands have continuously increased. For a long time, Moore's law, which states that processing power approximately doubles every two years, meant standard Von Neumann architecture was enough to keep up with demand. It was only the large super computers that required the use of parallel computing. However, as we reach limitations imposed upon us by the laws of physics, we can no longer pack more power into chips [1]. Instead, we must parallelize.

This has been taken to another level with the massive increases in demand for computational power in the last five years to run large AI models. Power consumption, which provides a reasonable gauge for computational power, has more than doubled from 2023 to 2024 to around 6000 MW [2]. To improve performance/power ratio, computation had to move beyond traditional Von Neumann architecture.

With the continually increasing demand for compute the opportunity for innovation has arisen. New architectures are being proposed that can use parallelism to scale to meet demand. The focus of this work is in the optimization of one of these new architectures, the Tenstorrent Wormhole n150d.

Fully utilizing parallel computation requires the use of a range of *collective communication* procedures. These are the different methods that individual nodes, that do not share memory, use to share data. Because optimization of these procedures, which are used in most parallel workloads, can provide significant efficiency gains across many different programs, it is a key area of research.

Each collective communication operation can be achieved using numerous different algorithms. With each new architecture, there are new opportunities to utilize different communication channels and modes, and new opportunities to optimize with new algorithms.

1.1.1 Swing allreduce

The allreduce is one of the fundamental collective communication procedures. Put simply, it allows for simultaneous distribution and processing of data across multiple nodes with non-shared memory. Studies show that allreduce uses between 19-30%

of the total core hours of MPI jobs running on production supercomputers [3], and up to 40% of the total training time of deep learning models [4], [5], [6]. As such, even minor improvements in the efficiency of this algorithm can provide significant reductions in energy use and computation time for large computational jobs.

The Swing algorithm has been designed specifically for **toroidal** network topologies. Classical algorithms often do not make use of the full set of nodal connections, specifically the "wrap-around" connection that directly links the first and last nodes, available to a torus network. This can introduce excessive congestion on bandwidth-limited large vector operations, or utilize longer pathways when there are more direct routes available on latency-limited small vector operations.

In both situations, better utilization of the available connections can increase performance and efficiency.

1.1.2 Tenstorrent hardware

Tenstorrent was founded with AI hardware in mind. The various accelerator boards they offer are generally marketed with the core message that they can be used for AI workloads where computational power and energy efficiency are key metrics. However, the secondary marketing message is that the TT-Metalium low level hardware SDK is both easy to use and versatile, making Tenstorrent hardware an excellent choice for general parallel computing workloads.

1.2 Objectives

The goals of this work are three-fold. Firstly, the Swing algorithm is still novel, and has not reached widespread acceptance. It's performance is theoretically good on many types of network topology, however, theory and practice often diverge, so it was desirable to compare it with the widely used Recursive Doubling implementation, and a simpler Shared-Memory based implementation, on a new hardware architecture.

Secondly, the TT-Metalium SDK has not implemented any state-of-the-art allreduce operation. This work intends to provide a valuable data point for future work towards optimizing the kernel for the hardware Tenstorrent is developing.

Finally, developing efficient allreduce implementations requires careful use of both memory and communication channels. This makes it an excellent test of the viability of using the TT-Metalium SDK for general parallel computing tasks.

Chapter 2

Background

2.1 High performance computing

High Performance Computing (HPC) is the use of computer clusters, or super computers, to solve computationally challenging problems. The same principles used in large computer clusters are also often used at a smaller level, for parallel processing on multi-core CPUs, GPUs and accelerator boards.

The architecture in HPC can vary almost infinitely. Nodes are connected together in different topologies, such as grids, trees or dragonflies. Each of these has many variants within it, for example grids can be 1D (a single line of nodes), 2D (a rectangular lattice), or more. There can also be further dimensions within each node, for example, each node in a grid may have a 2D grid of shared memory processors within it.

Each "dimension" may have different levels of communication and shared resources available, which all require different techniques to achieve speed-up. Dimension's can also be "toroidal". This is when nodes are directly connected to their farthest neighbor in a given dimension. The simplest example is a 1D toroidal structure shown in Figure **??**.

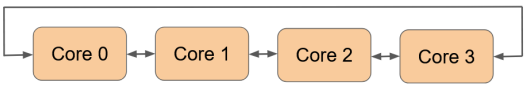


Figure 2.1. A 4 node 1-dimensional toroidal architecture

For parallel computing, communication between nodes can be limited in many ways. For example, bandwidth on a connection limits the rate at which data can be transferred, latency is the delay between a message being sent and it being received and processed, connections may only be possible through one of multiple links at any given time, or there may be limitations on the direction data can travel (i.e. separate sending and receiving links).

In HPC one of the key ways efficiency is measured is by "core utilization", which is the idea that as many cores should be (effectively) used as much of the time possible, meaning that there is not expensive hardware sitting idle. To achieve this, it is vital to avoid all bottlenecks in the computing system. To this end, many

different ways to share data have been developed, and for each of them there are a myriad of different algorithms.

2.2 Overview of collective communication in HPC

Often one of the main bottlenecks in HPC is memory. This memory can often only be accessed, for read or write, by a few workers at a time, and can often not be accessed by all cores at the same time, so numerous routines have been developed to allow efficient node-to-node communication [7]. On the Tenstorrent n150d, nodes, or in Tenstorrent parlance "Tensix cores" (see Section **??**), are able to share and gather data without using shared memory, and instead read and write directly from/to each nodes local SRAM, which means it is suitable for a wide range of collective communication algorithms.

Due to the nature of Deep Learning workloads, these collective communication algorithms are critical to increase efficiency, speed, and core utilization. Given the size and energy of the largest models, small increases in efficiency can have huge benefits in terms of hardware costs, energy usage, and training times [4].

2.3 Collective communication routines

There are four fundamental collective communication routines, several of which can be combined, modified, or simultaneously performed by multiple nodes to produce more key routines. They all have the goal of moving and combining data.

- Broadcast - One node copies data of length l to p nodes, leaving p identical copies of the full original array of length l.
- Reduce - Data of length l from p nodes is sent to one node and operated on element-by-element (e.g. addition, multiplication), leaving one node with a single data array of length l, which is a combination of all the vectors. Note, any algorithm that has "reduce" in it's name, performs some sort of mathematical operation on the data while it is being moved.
- Gather - Data of length l from p other nodes is copied verbatim to one node, resulting in one node with a single array of length $l * p$.
- Scatter - Different parts of data of length l from one node are spread across other p nodes, so each node contains a unique array of length l/p.

By combining these procedures in different ways, or performing them with some subtle changes on multiple nodes simultaneously, many more routines become available to produce all possible varieties of different distributions of the original data.

The allreduce algorithm is where the focus of this project lies, but to understand it, it is also useful to understand reducescatter and allgather, see Figure **??**.

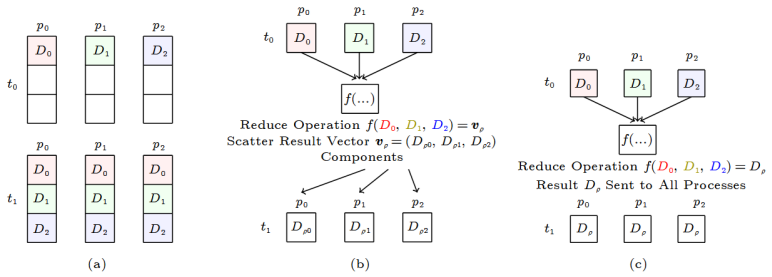


Figure 2.2. Visual representation of *(a)* allgather, *(b)* reducescatter, and *(c)* allreduce, diagram from [7]

2.3.1 Reducescatter

Reducescatter, as the name implies, is a reduce combined with a scatter. To be more specific, p nodes split their data, D_n, into p pieces. Each node will receive a different slice of data from every other node, and do some operation on the received data, leaving each node with an array of length $1/p$. See Figure **??**b.

For example, given 4 nodes:

1. Each node a to d has a different array D_a to D_d respectively.
2. Each node splits it's data into 4 pieces, for example node a will create D_{a1} to D_{a4}.
3. Each node sends these arrays to each respective node, for example node a sends D_{a2} to D_{a4} to nodes b to d respectively.
4. Each node performs some operation on it's data, say addition. Node a receives D_{b1} to D_{d1}, and adds each array element-wise, giving it the output $D_{a1} + D_{b1} + D_{c1} + D_{d1} = D_1$, while the other nodes have simultaneously calculated D_2 to D_4.
5. Each node a to d then has a unique block of data D_1 to D_4, that is a unique combination of data from all of the nodes.

2.3.2 Allgather

Allgather, as the name suggests, is when all nodes simultaneously perform a gather operation, where p nodes with p different sets of data each send their unique data to all other nodes. The end result is all p nodes with exactly the same data, data which is a concatenation of the data from every other node. See Figure **??**a.

For example, given 4 nodes:

1. Each node a to d will have a different array D_1 to D_4 respectively.
2. Node a sends D_1 to nodes b to d, and receives D_2 to D_4 from each node respectively.
3. Each node then combines the data to get D, which is the concatenation of D_1 to D_4.

2.3.3 Allreduce

Allreduce is one of the most important collective communication algorithms for AI workloads [4], [5], [6]. It can be viewed as a combination of reducescatter and allgather. In fact several algorithms achieve allreduce exactly that way, by performing a reducescatter immediately followed by an allgather.

Given P nodes all with different data, the data is combined using some operation, and redistributed so, at the end, every node has an identical array. See Figure **??**c. The combination of examples in Sections **??** and **??** provides an accurate picture of how 4 nodes, a through d, can perform an allreduce on four separate blocks of data D_a to D_d to end up with one common block of data D.

Many algorithms have been developed for this operation, with benefits and drawbacks to each one, mostly depending on vector size, network shape/dimensionality, number of nodes, channel bandwidth, and node-to-node latency. Many of them do not follow the steps from Sections **??** and **??**.

Three algorithms for performing allreduce are the following:

2.3.3.1 Shared-Memory

This is the simplest implementation possible. The Shared-Memory (SM) algorithm, as the name suggests, uses shared-memory for nodes to communicate their vectors. Given p nodes, each with data D, each node writes its full vector to shared memory, in total requiring a $p * D$ chunk of memory.

Each node r proceeds to read data from position r of each of the p chunks. A math operation is performed to join it to that node's chunk of data. Finally, a nodes data is written back to position r of a chunk of memory of size D. Every node then reads the full final array of size D from the same chunk of memory.

This is not an optimal algorithm in any respect, but it is a reasonably straightforward way to perform an allreduce on some data.

2.3.3.2 Hamiltonian rings

The Hamiltonian ring algorithm is an allreduce algorithm where each node-to-node connection in the p-dimensional grid is always one way. Each node passes data to its right, and receives data from its left, and at each step some of the data received is used for computation on the local portion of the vector. It is an algorithm that was developed specifically for toroidal network topologies.

This algorithm uses the reducescatter followed by allgather procedure, performing the reducescatter over p steps, followed by p steps to perform the allgather.

Since at every step the algorithm sends the minimum amount of data possible to complete the algorithm along the maximum possible number of ports, the algorithm is said to be Bandwidth Optimal (BO), meaning it uses the available bandwidth in the most efficient way possible [8]. Or, in other words, the absolute minimum quantity of data has been transferred, at the cost of more individual steps. Specifically, given n bytes of data, the total amount of data transmitted by each node is $2n$, which is the mathematical minimum possible [9].

However, regardless of how many nodes there are, the algorithm always takes p steps, so it is not said to be Latency Optimal (LO). Several other algorithms

exist that can complete the operation in far fewer steps, which in a latency limited network will reduce runtimes.

2.3.3.3 Recursive doubling

The Recursive Doubling (RD) algorithm aims to minimize the number of steps for the algorithm to complete. Each node first swaps data with a neighboring node at distance $d = 1$, and then, at each subsequent step, with a node at distance $d = 2d_{prev}$. At each stage, the distance doubles, the data is swapped, and a mathematical operation is performed. Due to the exponential nature of this algorithm, given p nodes the algorithm requires $log_2(p)$ steps [10].

Communication is done in pairs, so, for example on step one, node *(0,0)* will exchange data with node *(0,1)*, and vice versa. On the following step, on a 1D network, node *(0,0)* will exchange data with *(0,2)*, see Figure **??**. On 2D networks, the dimension of communication also swaps on each step, although the specific direction is completely dependent on the node in question.

Recursive doubling also comes in a second variant. The more basic version, Latency Optimal is as described above. This version is latency optimal as it minimizes the number of steps required for the algorithm to complete, but each node performs more calculations, and transmits the full n bytes of data on every step, for a total of $n * log_2(p)$ bytes transmitted across $log_2(p)$ steps (note, that $log_2(p)$ is the absolute minimum amount of steps necessary for an allreduce to complete [9]).

The second Bandwidth Optimal (BO) variant sends the minimum possible ($2n$) bytes of data. At each step $n/2^{step\ number}$ bytes are sent. I.e. on the first step half the data is sent, on the second a quarter, etc. However, the trade-off comes after this, because all data must then be shared between all nodes using an allgather operation, which again requires $log_2(p)$ steps, bringing the total number of steps to $2 * log_2(p)$ steps.

Chapter 3

The Swing Allreduce Algorithm

While the Recursive Doubling algorithm discussed in Section **??** is considered Latency Optimal (by minimizing steps to complete), it is not optimal in terms of the number of nodes passed through to transmit a message, or optimal in reducing the number of simultaneous messages being passed along a specific link. The Swing algorithm aims to address both of these deficiencies to pass messages faster and using less energy.

3.1 Design rationale

The idea of the Swing algorithm is to take advantage of the toroidal shape to reduce congestion on the most congested links, and reduce the number of nodes a message has to pass through. It is the natural adaptation of RD to the toroidal topology. This can be easily understood by looking at the first steps of the Swing algorithm in comparison to the RD algorithm.

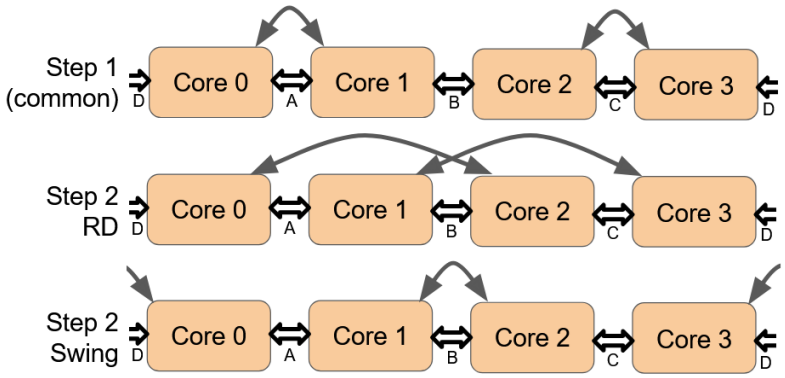


Figure 3.1. First two steps of recursive doubling and Swing algorithms

For a p-dimensional array, the first p steps of each algorithm will be the same. Each node swaps data with one of their direct neighbors in each dimension. On a 1D array, the differences can be seen from step 2 onward.

For recursive doubling, the maximum congestion on a link in a 1D array is always equal to the greatest distance a message has to travel, which is $2^{step\ number-1}$ [11].

E.g. at step three on a 1D torus, the highest congestion will be $2^2 = 4$ messages being passed on each of the most congested links.

To give an explicit example, with Recursive Doubling on step 2, link B between cores 1 and 2 has to carry data between core 0 and core 2 and also between core 1 and core 3, so the congestion is $2^1 = 2$ messages. Each message additionally has to pass through 2 links, which also incurs some overhead, both in time and in energy consumption.

With the Swing algorithm, at step 2 core 0, communicates directly with core 3 over a dedicated and uncongested link, and the same is true for cores 1 and 2. Here, the maximum congestion is only $2^{\lfloor (step\ number-1)/2 \rfloor}$. Although the physical distance traveled may increase for a given message at a communication step (for example in Figure **??**, core 3 is physically further from core 0 than core 2), the reduction in nodes crossed and contested bandwidth can more than compensate for this shortcoming.

3.2 Algorithm description

The Swing algorithm is, at a fundamental level, very similar to the RD algorithm. In fact, each node also uses an algorithm similar to recursive doubling to calculate which node it will be exchanging data with.

Like recursive doubling, the communication partner changes every step, however the calculation is slightly more nuanced. Every step, for every node, the direction of communication changes. On a 1D torus, the first node will pass messages to the right on step 1, to the left (around the toroidal link) on step 2, and back to the right on step 3. On a 2D torus, the direction of communication instead "rotates". For example, the top left node will initially pass messages to the right on step 1, then down on step 2, then left (around the toroidal link) on step 3, then up (again, around the toroidal link) on step 4, etc. The starting communication direction varies based on the nodes position, and the direction of rotation can be either clockwise or counter-clockwise, however the general pattern will always be the same.

For a 1D torus, a more precise definition for the calculation of the communication partner for node r at step s is [11]:

$$\pi(r,s) = \begin{cases} (r + \rho(s)) \bmod p, & \text{if } r \text{ is even,} \\ (r - \rho(s)) \bmod p, & \text{if } r \text{ is odd.} \end{cases} \tag{3.1}$$

Where $|\rho| \mod p$ is the distance and the sign of ρ indicates the direction [11]:

$$\rho(s) = \sum_{i=0}^{s} (-2)^i = \frac{1 - (-2)^{s+1}}{3} \tag{3.2}$$

On a 2D torus, the calculation partner calculation is almost the same, however each dimension is treated independently. In other words, step one will be horizontal, and step two will be vertical, however step two vertical will be calculated as if it is step one. More precisely:

$$\pi^{2D}(r_{xy}, s) = \begin{cases} (r_x, \pi(r_x, \lceil s/2 \rceil)), & \text{if } s \text{ is odd (a horizontal step),} \\ (\pi(r_y, \lceil s/2 \rceil), r_y), & \text{if } s \text{ is even (a vertical step).} \end{cases} \tag{3.3}$$

3.2.1 Bandwidth or latency optimal

Like recursive doubling, there are two versions of the Swing algorithm. Latency Optimal (LO) and Bandwidth Optimal (BO). As the names suggest, these are optimized for limited node-to-node communication latency or bandwidth respectively.

Latency optimal means the nodes send and compute more data than is strictly necessary, the algorithm has been optimized to minimize the amount of time lost to communication latency with other nodes. This is the optimal algorithm when latency is the limiting factor, typically on small arrays.

With LO, for a 1D torus of p nodes, each node r with data D_r at each step s sends the full block of data D_r and receives the entire block of data D_{recv} with another node defined by $\pi(r, s)$. It then performs some element-wise math operation to merge D_{recv} into D_r . This process repeats for $log_2 p$ steps.

Bandwidth optimal means the amount of data that is computed and sent is reduced, thereby freeing up more bandwidth in the inter-node communications channels. This is optimal when bandwidth is the limiting factor and latency is less of an issue.

The BO version of the algorithm performs a reducescatter operation followed by an allgather operation. For a p node 1D array, the calculation of communication partner is performed, as before, using Equation **??**. Then, at each step, rather than sending the entire array of data D, only specific parts of the array are sent.

Intuitively, if node 1 is communicating with node 3, and in the next step node 3 communicates with node 5, then node 1 will need to send D_3 and D_5. In the next step node 1 will communicate with node 8, so node 3 will send D_1 and D_8. With Swing, this data will not be contiguous. With RD, the data is only contiguous with the distance halving implementation. More explicitly, the BO algorithm performs the `reduce_scatter` followed by the `all_gather` as described in the following pseudo-code [11]:

Listing 3.1. Swing reducescatter

```
def get_send_recv_idxs(this_node, step, num_nodes, blocks):
    if step >= log2(num_nodes): return
    for step in range(step, int(log2(num_nodes))):
        peer = pi(this_node, step, num_nodes)
        # Set to 1 the node I directly reach
        blocks[peer] = 1
        # and those that it will reach
        get_rs_idxs(peer, step+1, num_nodes, blocks)

def reduce_scatter(this_node, num_nodes, data):
    for step in range(0, int(log2(num_nodes))):
        blocks_s = blocks_r = [0]*num_nodes
        dest = pi(this_node, step, num_nodes)
```

```
        get_send_recv_idxs(this_node, step, num_nodes,
            blocks_s)
        get_send_recv_idxs(peer, step, num_nodes, blocks_r)
        # Send blocks where blocks_s[i]=1, and recv where
            blocks_r[i]=1
        sendrecv_operate(dest, data, blocks_s, block_r)

def all_gather(this_node, num_nodes, data):
    for step in range(ceil(log2(num_nodes)), 0):
        blocks_s = blocks_r = [0]*num_nodes
        dest = pi(this_node, step, num_nodes)
        get_send_recv_idxs(this_node, step, num_nodes,
            blocks_s)
        get_send_recv_idxs(peer, step, num_nodes, blocks_r)
        # Send blocks where blocks_s[i]=1, and recv where
            blocks_r[i]=1
        sendrecv(dest, data, blocks_s, block_r)
```

After the reducescatter is performed, each node will have $1/p$ of the correct processed data, which is then shared with every other node using the allgather. The allgather works very similarly to the reducescatter, however the steps are performed in reverse order, and no math operations are performed.

3.3 Expected benefits and trade-offs

As mentioned previously, Swing is optimized to minimize link congestion, and minimize the number of links used to communicate. Minimizing link congestion could present significant performance gains over RD on architectures where bandwidth is limited, particularly when compute is idle while waiting for data to arrive on congested links.

Minimizing the number of links used, by taking shorter routes, can improve both speed and energy usage. Every time a message passes through a node, that node must perform some computation to either read the message, or route the message towards its destination. Each of these operations use clock cycles, so reducing them can speed things up.

Due to the logarithmic nature of the algorithm, it is also particularly suited to power-of-2 arrays. Although it can be adapted to any rectangular array, there are significant increases in algorithmic (though, not computational) complexity. A non-power-of-2 array will also mean there won't be full core utilization at every step, meaning it may be better suited to another allreduce algorithm.

Chapter 4

Tenstorrent Wormhole n150

4.1 Overview of the Wormhole n150 board

The wormhole n150 range of boards are a new range of parallel computing AI accelerators designed to be a cost-effective way for training and use of AI by entry-level business's and consumers. Some key specifications of the accelerator board are listed in Table **??**.

Specification	n150
Wormhole™	1
Tensix Cores	72
Core Layout	8×9 Torus
AI Clock	1 GHz
SRAM	108 MB
Memory	12 GB GDDR6
Memory Speed	12 GT/sec
Memory Bandwidth	288 GB/sec
TeraFLOPS (FP8)	262
TeraFLOPS (FP16)	74

Table 4.1. Specifications for Tenstorrent n150 [12]

4.2 Topology and hardware details

Although the n150 is sold as an 8×9 toroidal grid of cores, that is a slight simplification of the reality [13]. In fact, the board is a 10×12 grid. The grid is logically laid out as shown in Figure **??**. Each node is linked with all four neighbors, including the toroidal link between the nodes on the edges. The different types of tile are listed in Table **??**. The only nodes that are visible to and usable by the end user are 72 of the Tensix T nodes. Although there are 80 T nodes, the additional 8 nodes are there only to allow for manufacturing errors. The other node types are used for accessing memory, Ethernet and PCIe.

Each T node contains five "Baby RISC-V" cores. These are the programmable cores that pass instructions to the other units. (See Figure **??** for a simplified

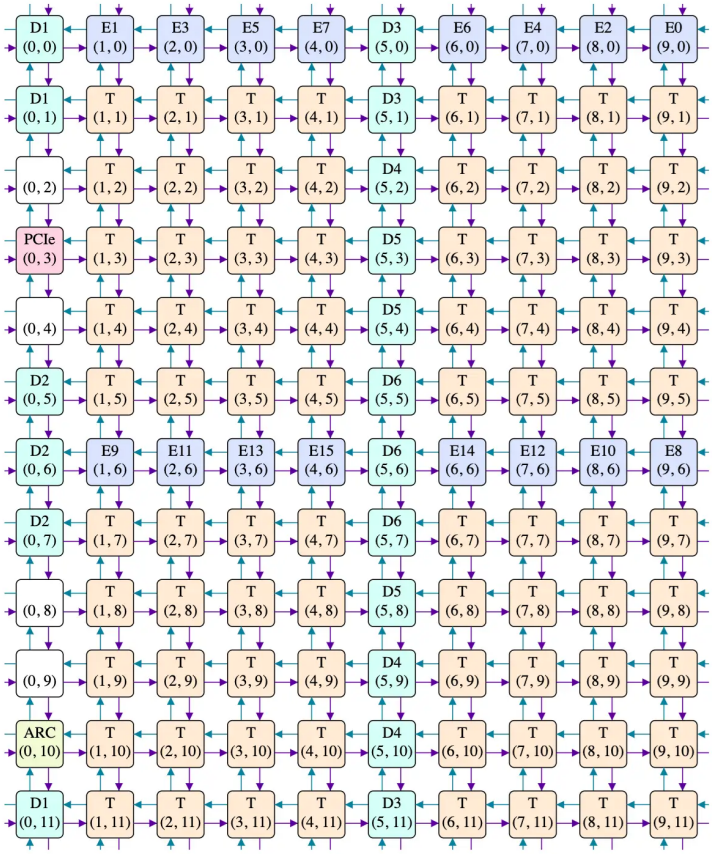


Figure 4.1. The layout of the Tenstorrent Wormhole n150, from [13]

Kind	Count	Contents (per tile)
ARC	1	1x Argonaut RISC Core (also connected to PCIe)
D	6 x 3	Bridge to 2x 1GB GDDR6 (shared by three tiles)
E	16	1x Baby RISC-V CPU (E variant)
		256K SRAM
		Bridge to 100Gb ethernet (if connected)
PCIe	1	Bridge to host over PCIe
T	80 (not all usable)	5x Baby RISC-V CPU (B/T/T/T/NC variants)
		1.5MB SRAM
		1x Matrix unit (2048 multipliers, each 5b x 7b)
		1x Vector/SIMD unit (32 lanes, each 32b wide)

Table 4.2. Specifications for tile types from Figure **??** [13]

interpretation). Although each RISC-V core is technically identical, the five cores are broken up into two Network on Chip (NoC) cores, and three compute cores.

The NoC cores are used to control the NoC interfaces, which connect the node to its neighbors. Interface 0 is generally used to control transmission of data and interface 1 reception, however this is a Tenstorrent convention as opposed to a

hardware limitation. Each NoC interface 0 is connected to the NoC interface 1 of the nodes to its North and West, and vice versa for each NoC interface 1 [14].

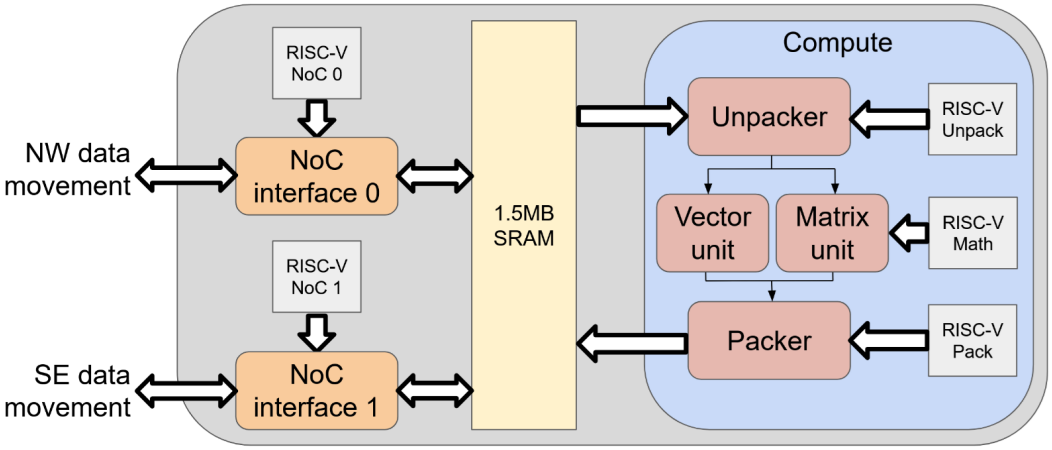


Figure 4.2. Tensix core architecture

Because each NoC interface is also connected to the Ethernet nodes, this allows (in theory) easy scaling between boards. The compiler sees a single large array of Tensix cores, rather than a set of devices each with its own array of cores [15]. This, in theory, allows easier and more efficient scaling, though this is beyond the scope of this project.

Each node in the array has a neighbor to neighbor "hop" communication latency of around nine clock cycles [16]. This excludes the time required to actually create or receive the message. The hop latency was found to scale linearly, and applies to both the hidden and visible nodes equally. It also applies equally to the toroidal link, due to the interlaced physical layout that has been adopted, both horizontally and vertically. If this wasn't performed, the physical distance between the first and last node would be much greater (see Figure **??** for an example), which means the latency along that link would be significantly greater. With this layout, the last core in a row is in fact right next to the first core in the row, as can be seen in Figure **??**.

Additionally, each node has a communication bandwidth of up to 32 B/cycle with its neighbors. At 1 GHz clock speed, this amounts to a bandwidth of 32 GB/s if the maximum message size is passed per cycle [17], after headers and overhead has been included, this upper limit is closer to 29 GB/s for a one to one operation. However, given most communications are not able to fully saturate the bandwidth (e.g. because serial data transfers are performed), the actual communication bandwidth is often substantially lower [18].

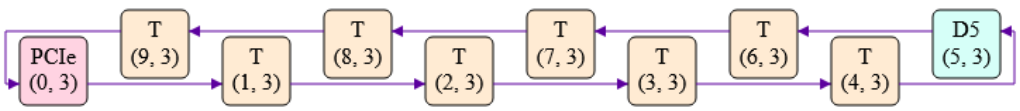


Figure 4.3. The physical layout of a row of Tensix cores [13]

Alongside the two NoC cores on each Tensix node there are three compute

cores. The three compute cores are used by the TT-Metalium SDK to control the matrix and vector units. The cores are broken down into *UNPACK*, which controls the "unpack unit" that moves data from SRAM into 32kB L1 shared memory and performs any operations necessary to pre-process the data (for example, changing the type), *MATH* which controls the mathematical vector/matrix operations unit, and *PACK* which controls the movement of computed data to SRAM, as well as any post-processing [19].

The RISC-V cores use standard RISC-V instructions. The specialized pack/math /unpack units in each Tensix core utilize a specialized internal set of instructions developed by Tenstorrent. As such, the RISC-V units take RISC-V instructions as input, but give Tensix instructions as output.

4.3 Programming model and toolchain

Tenstorrent provides various ways to interact with the hardware, depending on the level of interaction and granularity required.

The highest and simplest level of interaction developed by Tenstorrent is TT-NN, which is essentially a package designed to replicate the capabilities of PyTorch on Tenstorrent hardware. This is designed so that AI workloads can be easily built on, or ported to, Tenstorrent hardware.

The next level down is TT-Metalium. TT-NN utilizes this layer [20]. Although not at the kernel level, it provides very granular control of individual Tensix cores. Each Tensix core can be given its own unique kernel, or kernel parameters. On each Tensix core, the two RISC-V NoCs can be given different kernels or kernel parameters, but the three compute cores are abstracted away by the compiler, so the developer only has to handle one kernel for all three compute cores. This report focuses on TT-Metalium.

The lowest levels of software-level programming for the Tenstorrent Wormhole is TT-LLK (Low Level Kernel). This is a header only library that provides almost complete control over each RISC-V core within each Tensix core, allowing extremely high levels of optimization and customization [21].

4.3.1 TT-Metalium

For the scope of this project, TT-Metalium seemed an appropriate choice. It provides sufficient levels of control to choose the specific channels each inter-core message was being passed along, while also providing abstraction that is in theory ready to use out of the box to optimize the use of the compute units and parallelism.

TT-Metalium's programming model can be broken down into three separate areas. While each area uses C++, they all use different headers and a different syntax that has been designed around each of their specific tasks [22].

Firstly, there is the host layer. This manages the overall orchestration between the host and the device. The key tasks it performs are sending data to and from the device, creating the memory buffers that the device uses, specifying which cores will run which kernels, passing parameters into the kernels and finally launching the kernels before copying data back to host.

Secondly, there is the NoC layer. These kernels handle all movement of data on each Tensix core. This includes inter-core communication, as well as reading from DRAM and writing to SRAM and vice versa. The NoC cores are the only ones able to utilize semaphores, which are generally used for inter-node communication. Although it is not the intended use, each NoC core can, in principle, use semaphores to communicate with the other NoC core located on the same node, rather than with the cores on remote nodes. However, communication between the RISC-V cores of a single node is generally handled with Circular Buffers (CBs).

Finally, there is the compute layer. These kernels handle all mathematical computation on input Tensors that are stored in the memory buffers. The three compute cores have been optimized to maximize levels of utilization, by using an *UNPACK* unit to ensure that the required data is already in the registers at the moment the *MATH* unit is ready, and the moment the *MATH* is finished the *PACK* unit moves data from the registers to L1 memory. The compute layer is optimized around the use of CBs.

The CBs work using tiles. All data is managed in chunks with length defined by the host, and always operated on in these units. So, the NoC cores will (generally) push and pull data that is at least the size of a single tile around the Tensix core. The compute units will always perform math operations on data as a tile. Using TT-Metalium, the compute cores can only use these tiles. This means the only way to interact with data via the matrix and vector units is in tile size chunks. There is never the need to loop through each individual element of an array, it is all abstracted away, although it is necessary to loop through the tiles themselves.

Chapter 5

Implementation Details

Five versions of the procedure were developed and implemented. Bandwidth Optimal and Latency optimal each combined with the Recursive Doubling and Swing ordering, as well as a fifth "Shared-Memory" (SM) implementation. All of the implementations required a deep understanding of the workings of the TT-Metalium SDK. While many of the fundamental tasks were fully catered for in the SDK, some of the key synchronization and computation ordering requirements also required subtle workarounds.

All the implementations have been built around the use of 64 Tensix cores. This is due to the significant additional algorithmic complexity required to implement the algorithms on non power-of-two architectures.

Each version of the allreduce uses many common parameters, and requires much of the same setup, so much of the code was implemented using DRY (Do not Repeat Yourself) techniques.

While the order of communication between Recursive Doubling and Swing is completely different, the actual implementation is very similar. The only differences are the binary sequence representing which NoC core is to be used, and the array of communication partner coordinates, which are both calculated by the host. The BO version also requires a binary array of which data chunks need to be sent, which is also different between RD and Swing versions as it is completely derived from the communication partner calculations. The kernel level implementation of each algorithm is identical.

Between LO and BO, the initial part of the BO kernel is also similar to the LO kernel, however it also contains the allgather procedure that is not performed by the LO implementation. The allgather implementation can be thought of as an "inverted" reducescatter. The Shared-Memory allreduce on the other hand has a completely different implementation at the kernel level.

5.1 Host setup

The fundamental setup was largely common between the different algorithms, and as such was implemented in a common helper function. The only difference between the initial setup of the algorithms is the amount of memory that has to be allocated.

The general setup follows the following steps:

1. Firstly, the input parameters are taken and used to calculate some of the variables necessary for the algorithms. The key variants are the size of the input vectors, the routing algorithm being used, and the BO/LO algorithm, however there are numerous others that are used for debugging.
2. Next SRAM is allocated on each Tensix core for the Circular Buffers. All algorithms use 4 CBs. These are:
   (a) `cb_local` - The local copy of the array. The "master copy", that is never modified by a remote node and is where the computed results are written back to.
   (b) `cb_recv` - The receive copy of the array. This is the "input buffer" where fresh data from other cores is written to before it is used for the reductions, (on each step, `cb_local` += `cb_recv`).
   (c) `cb_SE` and `cb_NW` - These are very small CBs that, as a workaround, are used like semaphores for synchronization of NoC cores, see Section **??** and Appendix **??**.
3. Interleaved buffers are allocated to device DRAM, for the inputs and output.
4. Random input vectors are computed and written to the input interleaved buffers already allocated on device memory.
5. Semaphores are allocated for each node. The maximum number available is 8, and all 8 are utilized by the kernels.

After this point, LO/BO and SM all have different, but similar implementations.

5.1.1 Swing and Recursive Doubling

Both the LO and BO variants use the same host program. First, numerous arguments which are common between all nodes are written to the kernel input arguments (such as size of data, number of steps, memory addresses). Then a loop is initiated that iterates through every node's linear index (between 0 and 63) and computes all of the specific parameters necessary to complete the algorithm.

1. The basic unique input parameters for node `r` are assigned, which are the core's index, and the piece of data this core should read from DRAM.
2. Then, the communication partners, and blocks being transmitted at each step `s` are calculated in a for loop.

   Swing:

   (a) First, Equation **??** is used to compute the communication partner index `comm_partner_idx` for node `r` at step `s`. Equation **??** has been modified to work in a linear coordinate space, as in Appendix **??**. This maps the 2D coordinates to a 1D rank.

(b) Then, the physical core (see Appendix **??**) for the communication partner (computed using Equation **??**) is added to the input arguments for that step.

From this point, the following parameters only need to be computed for the BO algorithm.

(c) The `blocks_to_send` and `blocks_to_receive` arrays are zeroed out, then the blocks for the `comm_partner_idx` and `this_core_idx` are set to `1` in each respective binary sequence (meaning, those blocks are to be transmitted).

(d) Then the remaining `blocks_to_send` and `blocks_to_receive` are computed, these are the blocks that will need to be sent in all remaining steps by the `comm_partner` and `this_core` respectively. If, for example, from a total of 8 nodes, the blocks *1, 3, 4* and *6*, need to be sent in the next step, then the output will be `01011010`. See Appendix **??**.

(e) `blocks_to_send` is then included as an input argument for the dataflow kernel, and for both compute and dataflow kernels `blocks_to_receive` is included as an input argument.

The node is then assigned one of 4 possible communication direction binary sequences according to if its vertical and horizontal coordinates are even or odd respectively.

Recursive Doubling uses a similar process, with the only differences being:

- At step (a), the `comm_partner_idx` assignment is computed by the function in Appendix **??**, and, simultaneously, the direction of communication is computed.
- At step (d), `blocks_to_send` and `blocks_to_receive` are computed using a slightly different version of the function in Appendix **??**.

3. Finally, the `step_directions`, which define which NoC interface (NW or SE) will perform the communication step, are entered into the input parameters, and the kernels are compiled for the given node.

5.1.2 Shared-memory allreduce

The Shared-Memory version also has a similar set up, but allocates a much larger chunk of memory of size $n * d$ for n nodes performing allreduce on data of size d bytes. It uses the same input parameters as the LO version because it uses a modified version of the Swing algorithm as a barrier to synchronize all nodes at various steps during the algorithm.

5.2 Dataflow setup

On both implementations there's around 100 lines of boilerplate code to do the initial setup. This involves reading all of the variables from the input arguments, setting up the allocated interleaved and Circular Buffers, setting up the allocated

semaphores, and reading the data assigned to the node from DRAM. Once the input variables have been read, and the initial setup has been completed, the loop which actually performs the allreduce operation begins.

The details of the loop on each algorithm vary. Once the loops have been completed, the NoC cores synchronize and the data is then written back to DRAM so it can be read and checked by host.

5.2.1 Bandwidth and Latency Optimal allreduce

Most of the initial setup is fairly trivial, however there are two notable parts.

The 8 semaphores available to each Tensix core are divided up into 2 separate categories in a specific way to avoid deadlocks. 6 semaphores, contained in the array `semaphore_0`, are reserved for each of the 6 steps (recall, since there are $p = 64$ nodes, there are $log_2 p = 6$ steps). This means that at the start of each step, there is only one node that ever increments that specific semaphore. This leaves 2 other semaphores, held in the array `semaphore_1`. These are used for mid communication synchronization to allow for parallelization of compute and communication.

As `semaphore_1` is shared (it gets incremented by all 6 different communication partners throughout the program), improper usage could introduce read after write errors and deadlocks. However because access is gated by `semaphore_0`, it is impossible that more than one node tries to increment `semaphore_0` or write to the same CB at the same time.

The other notable part, which can be tuned to optimize performance, is the `num_syncs`, which is the number of synchronizations that occur between one node's communication core (the core being utilized for the writing to the remote `cb_recv`) and the other node's listening core (the core monitoring the status of its local `cb_recv`). More synchronizations means the compute core can begin computation sooner, but increases the amount of extra data that needs to be transmitted, and requires extra synchronization as the communicating core has to use a write barrier.

For clarity, the difference between tiles and blocks must be explained. Tiles are the chunks of data defined by TT-Metalium that can be passed to the computation cores for processing. A block is a series of at least one contiguous tile(s), and the unit used by the allreduce algorithm to split up data. For example, with 64 nodes performing allreduce on 192 tiles, the block size would be 3.

After the initial setup, both of the dataflow cores simultaneously start a loop of $log_2 p$ iterations. For Latency Optimal, this first loop is a full allreduce operation, but for Bandwidth Optimal, the first loop is actually a reducescatter. The steps for each iteration are lain out below, with a corresponding diagram showing the communication patterns in Figure **??**, and the simplified code for the kernel is shown in Appendix **??**.

1. At the start of each iteration, NW and SE cores synchronize by performing a handshaking operation with the CBs `cb_SE` and `cb_NW`. (For the NW core, this is a case of reserving and then pushing one tile to `cb_SE`, then waiting and popping one tile from `cb_NW`, and vice versa for the SE core), see Appendix **??**.

2. Following synchronization, the cores enter one of two options on an if statement depending on if they are the communicating core (e.g. NW core during a

NW communication step) or the listening core (e.g. SE core during a NW communication step).

Communicating core:

This core handles all of the data transmission to the remote node.

(a) First, the core waits until the entire `cb_local` is free, which indicates that the computation units have finished modifying both `cb_local` and `cb_recv`.

(b) The core then increments the remote `semaphore_0` of its communication partner, and awaits the local `semaphore_0` to be incremented, indicating that the remote receive array is free to be written to.

(c) Then, once it has passed the semaphore, it increments through the blocks of data and writes any which are marked in `blocks_to_send` to the partner node. Note that for Latency Optimal, *every* block is marked as true, so every block is written to the other node.

(d) Every 2-10 blocks (depending on the size of the original input array), `semaphore_1` is incremented and the tiles that have already been written from `cb_local` are pushed back to the CB. This informs the compute core that the tiles pushed to `cb_local` can be modified.

This repeats until all of the tiles have been written.

Listening core:

This core waits on the local `semaphore_1` that is incremented by the remote node, so it can immediately pass the data to the compute core.

(a) The listening core first waits until the entirety of `cb_recv` is free, indicating the computation cores have finished modifying both `cb_local` and `cb_recv` in the last step.

(b) The core then waits for `semaphore_1` to be incremented. When it is, the listening core pushes tiles of `cb_recv` back so that computation can begin.

This repeats until the communication partner has written all of the tiles and incremented the semaphore `num_syncs` times, allowing the listening core to push the full set of tiles back.

3. Both cores then exit the `if` statement and continue to the next iteration of the loop.

The LO algorithm terminates here as every node now has the full set of data. The BO algorithm has only completed the first reducescatter operation, and so must continue to complete the allgather operation.

5.2.1.1 Bandwidth Optimal allgather

The second part of the implementation requires that every node passes its data to every other node, doubling the amount of data written in each step, so in total each

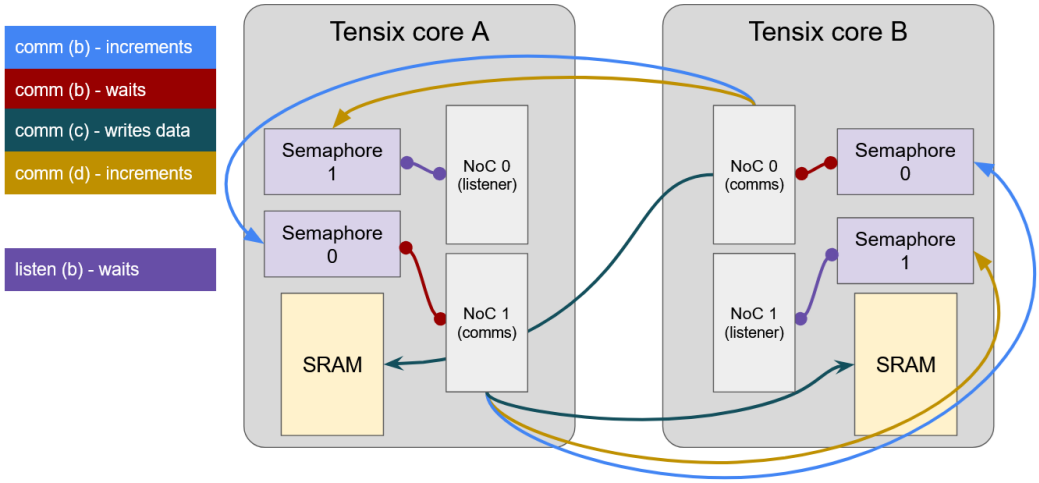


Figure 5.1. Diagram showing pattern of communication between two Tensix cores and the roles of the listening and communicating cores

node sends and receives 63 blocks of data, to join its 1 block of already computed data. The operation is similar to the reducescatter, except backwards. There is less opportunity for parallelism, as there is no computation.

1. Firstly, one of the NoC cores awaits the full `cb_recv`, ensuring that compute has finished its math operations on the array, before the allgather loop starts:
   (a) The NoC cores are synced (again, using the procedure in Appendix **??**).
   (b) The communicating core increments the remote `semaphore_0` and waits upon the local `semaphore_0`. This allows the two communication partners to start only once both `cb_local` buffers have finished being computed. The other NoC core remains idle.
   (c) Then all of the blocks defined in `blocks_to_send` for the given step are written to the remote node.
   (d) There is a write barrier to ensure the data has finished sending, before `semaphore_1` is incremented to inform the remote node that the write phase has finished, before finally waiting on `semaphore_1` to ensure the local data has been written to.

Once the allgather has completed, the algorithm has finished. Despite the fact two NoCs are reading from and writing to the same array (`cb_local`), on any given step they are always accessing different portions, meaning there is no chance of a write after write.

5.2.2 Shared-memory allreduce

As mentioned previously, the shared-memory implementation uses the Swing algorithm to synchronize the nodes just before the allreduce begins.

1. First, each node r of R nodes writes its full vector of size d to DRAM at position $r * d$.
2. Then, all the nodes perform a synchronization step.
3. Next one of the cores reads from DRAM and pushes tiles back to `cb_recv` as they're read, allowing simultaneous computation and data sharing.
4. Following this, each node writes the new reduced portion of the vector to DRAM at position $r * d/R$.
5. Finally, each node reads the vector back from DRAM.

5.3 Computation setup

Like the dataflow kernel, the initial setup of the compute kernel involves reading variables from the input arguments and setting up the CBs in SRAM. The initial setup of the compute kernel is much simpler.

5.3.1 Bandwidth and Latency Optimal allreduce

After the input arguments have been read, the kernel math operations are initialized which define which CBs will be used, and how they will be used. In this thesis, the focus has been addition as it is the most used operator. Following initial setup, the loop begins, iterating through each step of the algorithm. The LO, BO, Swing and RD variants all use identical code for this kernel. The only differences are the input parameters. The procedure is outlined below, with a corresponding diagram in Figure **??**.

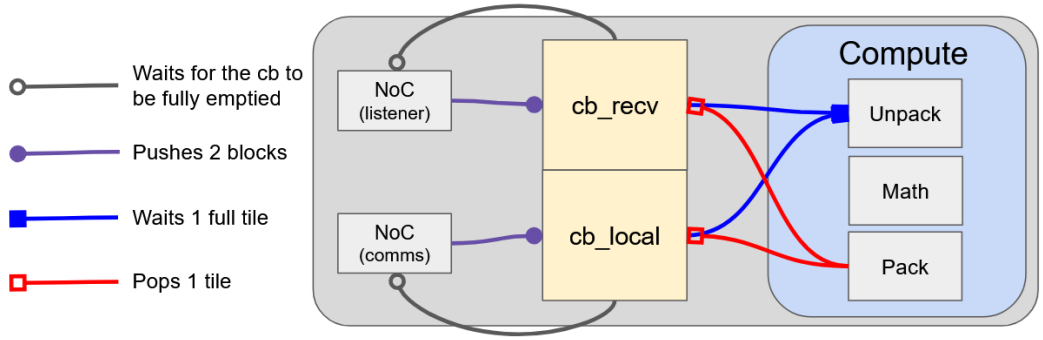


Figure 5.2. Diagram showing pattern of CB actions

1. First the outer loop defines each of the $log_2 p$ steps of the algorithm. Then the algorithm loops through the p blocks.
   (a) The boolean `recv_block` value is defined, which tells the compute kernel if the next tile(s) will be computed, or just immediately popped. For the LO version, every block is marked true. The algorithm then loops through each tile in the block.

i. Next, the compute kernels await (at least) one tile to be pushed to both `cb_local` and `cb_recv`.
Once both tiles have been pushed in, this signifies the local array has been read from, and the receive array has been written to.

ii. Then, if `recv_block` is true, the computation is performed using the TT-Metalium compute kernel abstraction, see Appendix **??**.

iii. Finally, one tile is popped from both `cb_local` and `cb_recv`.

At the end of each of the $log_2 p$ steps of the loop, the entire CB will be empty. This is used as a signal to the NoC cores that computation has finished and they can proceed to the next step of the algorithm.

5.3.2 Shared-memory allreduce

Following the initial setup, the Shared-Memory allreduce uses a slightly different method of performing the reduction to the other methods.

1. The registers are specifically prepared so that the portion of the vector d that node r is processing never leaves the registers. This is done to avoid race conditions, where the register's contents are moved in to and out of SRAM simultaneously by the *UNPACK* and *PACK* cores of the compute kernel. This requires distinct setup to the normal use of the compute core's registers.

2. Then, a loop begins for each of the p blocks of data.

   (a) From here, the compute kernel simply awaits tiles to arrive into `cb_recv`, at which point they are added to node r's portion of vector d.

   (b) Once the computation has been completed, the *PACK* core copies the register's contents to SRAM.

3. Following that, the buffers are freed up and control is handed back to the *dataflow* kernel.

5.4 Limitations and engineering challenges

During the implementation of the Swing algorithm on the Tenstorrent hardware, numerous challenges were encountered.

Tenstorrent is young, and the hardware is relatively new. As such, the tools are also not as mature as other tools like MPI, CUDA or HIP. This was very noticeable in the documentation, which frequently did not fully correspond with the source code. Function parameters were missing, deprecated functions were listed and new functions were omitted.

The specific details of how several of the key features of TT-Metalium functioned were often not clearly explained, leaving it up to the user to understand them. This required careful analysis of the source code, or trial and error.

5.4.1 Compute core synchronization and other difficulties

TT-Metalium is designed to simplify parallel programming by abstracting how instructions reach individual cores and how those cores synchronize. For the simpler LO implementation, this made the initial version of the code relatively straightforward. Core synchronization occurs behind the scenes. Although different compute cores handle different tasks, a single compute kernel controls them all, and the compiler distributes its instructions. Writing this kernel is effectively writing three kernels simultaneously.

However, for the BO and SM implementations, this abstraction made synchronization harder, and, for many of the initial versions, resulted in a lot of redundant computation. The three compute cores offer no explicit synchronization control short of kernel-level adjustments.

5.4.2 Circular Buffers

Circular Buffers are a fundamental part of how parallel processing is achieved on a Tensix core. The dataflow and compute cores all, on the surface, have the same levels and modes of access to the CBs. These are the corresponding pairs, `wait`/`pop` and `reserve`/`push`. This is how synchronization between compute and the NoCs is meant to be achieved within a node. As such, that is what was used.

In practice, when a dataflow kernel calls `cb_wait`, the core truly waits; when a compute kernel calls it, only the *UNPACK* core waits. Because the compiler generates three kernel versions—*UNPACK*, *MATH*, and *PACK*—this undocumented behavior led to a flawed approach that assigned NoC steps through the compute kernel, often deadlocking when the *PACK* core idled before the *MATH* core, resulting in CBs becoming under/overfilled.

Additionally, there is a minimum tile size for compute that is not defined. Because the NoC cores had to be carefully synchronized for allreduce, a tiny tile size was initially used for the "semaphore" Circular Buffers `cb_SE` and `cb_NW`. Eventually, through trial and error, it was discovered this was causing deadlocks and tiles must be at least 2048 bytes.

It was also not clear how the buffers can be safely used. Because the role of consumer and producer on a CB changes during the program, different calling pairs were used. However, it was later discovered this was a source of deadlocks and must not be done, and a kernel should always use either one pair of commands or the other on any given buffer.

Additionally, the CBs, and their abstracted pointers, are vitally important to how the compute cores access data. The documentation implied you can perform computation on data at any point within the CB, however experimentation, GitHub and the source code showed that the only tile accessible to the compute buffer is the tile at the front. As it turns out, every call to `cb_wait`, `cb_pop` and `cb_push` moves the read and write pointers of the tiles, so rather than putting that logic in the `add_tiles` function, the pointer must be separately controlled through calls to `cb_wait`.

All of these difficulties led to inaccurate results, and also very frequently to deadlocks.

5.4.3 Hardware stability

Every deadlock reached during the development of this program caused delays. In typical parallel computing development, a deadlock can be resolved by simply terminating the program. This is not the case with the n150d.

Deadlocks frequently required a hardware reboot. The interface provides a way to quickly reset the accelerator board, which should only take a few seconds. However, after several crashes of the entire server that hosts the Tenstorrent n150d, this command was found to be heavily correlated with the issue. The crashes required admin level resets, so could only be done during working hours. The only other way to "quickly" reset the board is to `sudo reboot` the entire server, causing all other connected users to be kicked out for several minutes while the server rebooted. In the worst cases when even `sudo reboot` wasn't working, the integrated Dell Remote Access Controller had to be used.

This meant that development was slow, and identifying the causes of deadlocks was particularly difficult and time consuming. Additionally, it meant that work on the project had to be conducted during unconventional hours to avoid negatively impacting other users.

Chapter 6

Performance Evaluation

6.1 Experimental setup

To understand the differences in performance between the different implementations, a range of tests were conducted on different vector sizes. Due to the nature of the problem, smaller and larger vectors are inherently better suited to different algorithms. As such, a range of vector lengths from 2kB up to 640kB were tested. Due to the implementation of the BO and SM algorithms, these were only tested from 128kB (1 tile per core) to 640kB (5 tiles per core).

6.1.1 Hardware and software configuration

No special configurations were used on the hardware. The Wormhole accelerator was tested in such a way that any external hardware changes should have no effect.

To eliminate effects from external hardware, the core loop of the algorithm was the only part that was timed. This means that any setup required, such as calculating communication partners, reading from shared memory, or setting up buffers has no effect on the timings. To ensure these had no effect, each algorithm was run 5 times consecutively without stopping to access DRAM. Because each allreduce loop works to synchronize the nodes (every node has a completion dependency on every other node, no node can finish until all have, at least, started), doing multiple runs of the algorithm provides a more realistic picture of how long the algorithm actually takes to run.

It was found that after three runs the nodes became fully synced and the timings plateaued so, to be certain, five consecutive runs were conducted for each configuration, and the timings from the final run was taken for analysis.

Each configuration was repeated a total of 20 times to understand the consistency and distributions of each configuration. Repeats were not consecutive to help ensure a fairer test.

6.1.2 Benchmark suite

Benchmarking was conducted using Tenstorrent's modified version of Tracy [23], a profiling tool. This allows timings to be taken for any code block in the kernel with minimal impact on the runtime of the code. The impact is kept to a minimum

by simply not processing the data until the kernel has finished running. After the kernel has finished, data is written to shared memory, then the host reads the raw data from the device, and saves it to a CSV file.

Processing of the data was performed manually from the extracted CSV. The timings for each node were calculated using the following methodology:

- The starting time of each node was the earliest start time between the SE and NW cores.
- The ending time of each node was the latest end time between the SE and NW cores.
- The timings were normalized based on the start time of the first node to start.

These calculations and further analysis of the data were performed using a combination of Python and R.

6.1.3 Benchmarking methodology

Effective benchmarking of collective communication algorithms is an area of some debate. Different sources suggest different methods of taking timings.

For example, NVIDIA by default reports the average across all ranks [24], Keysight recommends showing the minimum, maximum, 50th and 95th percentiles [25] and Hoefler and Belli report the maximum for the performance evaluation of a reduce operation [26]. Because following an allreduce computation generally cannot proceed until all nodes have finished, the maximum (or slowest) run-time will be used at many points during the analysis.

To provide the most relevant figures of algorithm run time, the context in which they are used should also be considered. Allreduce is used during parallel computing applications where multiple nodes are likely to reach the same point at around the same time, and are unable to continue until the operation has been completed. Once that is complete, they are again likely to be blocked in a similar manner. From this point of view, the run time of any specific node should be:

$$t_{node\ run\ time} = t_{node\ finish\ time} - t_{first\ node\ start\ time} \tag{6.1}$$

However, when considering algorithm run time, one must also take into account the non-determinism of multicore systems. Taking the run time from Equation **??** can result in skewed and unrealistic results if one node happens to start particularly early. To prevent this skew, a slightly different equation can be used which negates this effect:

$$t_{node\ run\ time} = t_{node\ finish\ time} - t_{node\ start\ time} \tag{6.2}$$

As mentioned in Section **??**, the algorithm was run multiple times sequentially to smooth out the effect of an early starting node skewing the results, however in the interests of completeness both of these equations will be used during the analysis.

In terms of assessing how many runs to take, this is again a source of debate. When comparing MPI collective operations, Pješivac-Grbović et al took at least 10

runs, removed the slowest, and then reported the results based on if the remaining results are within 5% standard deviation [27]. As it turns out, this methodology is unnecessary as the results from the n150d were extremely consistent, as we will show in Section **??**.

6.2 Results

The timing results of the tests are shown in Figure **??**. As expected, the Bandwidth Optimal algorithm was faster in all tested cases, however this is partly because only the larger data sizes can be run with the BO implementation. The differences between Swing and Recursive Doubling are negligible, the choice of LO/BO had a far larger effect on runtimes. Other than one exception (see Figure **??**b) Swing was found to always perform slightly worse. In all cases, both algorithms outperformed the simple Shared-Memory implementation by more than an order of magnitude.

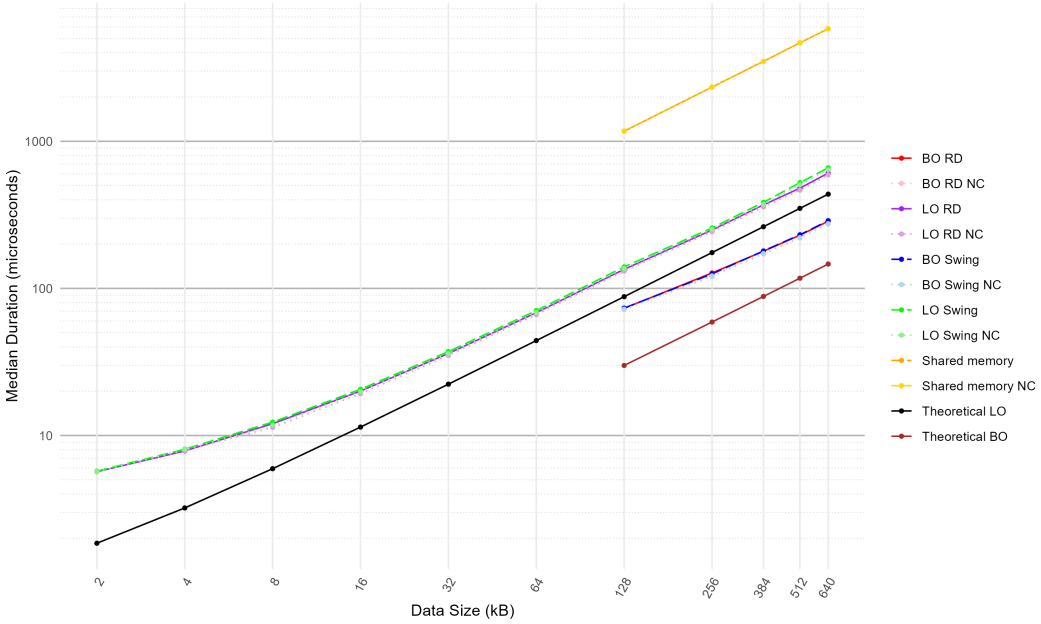


Figure 6.1. Median runtime of slowest core for all algorithms on all data sizes, using Equation **??**.

To assess how computation contributed to runtime, Figures are also included for a "No Compute" (NC) run, where the algorithm is run in full but the actual mathematical operation on the array is not performed. As is to be expected, these were always faster than the normal run, however the speed increase is negligible, and is in fact barely visible on the plot.

The theoretical runtimes that are achievable given maximum hardware utilization are also included in the plot. These numbers were calculated using the methodology in Appendix **??**. These calculated numbers take account of maximum FLOPs, maximum communication bandwidth (using approximated values based on [18]), and communication latency. The factors that have not been considered at all are

communication congestion (that Swing was specifically designed to reduce), communication overhead, other hidden overheads, and the fact that 100% parallelization is not achievable given the tools available in TT-Metalium. At larger array sizes there was at most a 2× divergence in predicted versus actual performance.

6.2.1 Variations in runtime

To assess runtime consistency, box plots were generated for different algorithms and data sizes (Figures **??**, **??**, and **??**). Each figure presents results under two configurations: (i) including every node from every run as an independent data point ("all nodes"), and (ii) considering only the slowest node from each run ("slowest node").

The variability in individual node run times is low for all combinations of the Swing, RD, LO and BO algorithms, see Figures **??** and **??**. Across the board, the inter-quartile ranges are up to around 1% of median value on larger data sizes, and up to around 5% on smaller data sizes. When only considering the slowest nodes, the range of IQR drops well below 1% of total runtime on even the smallest data sizes.

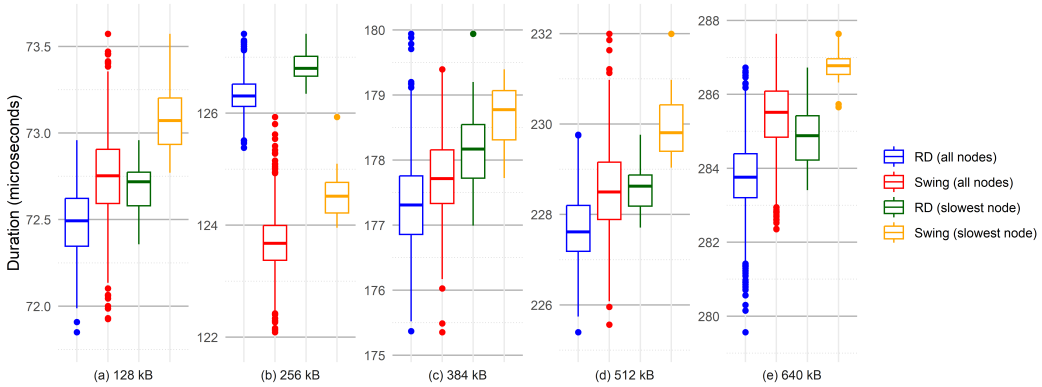


Figure 6.2. Individual node run times for different data sizes using Bandwidth Optimal algorithm

This runtime consistency can be attributed to little to no OS interference. The OS does not use the n150d, and tests were conducted when no other users were connected, meaning there is no context switching. Additionally, the n150d does not actually support multitasking, only one program can run at a time.

The actual median values of all nodes and slowest nodes are also comparable. In the worst case, when comparing the median of all run times with the median of the slowest node, the values are within 5% even on the smallest datasets, and well below 1% on the larger datasets.

These are good results, and are in large contrast with the SM implementation in Figure **??**. Even on the largest data size, the IQR of the SM implementation is close to 20% of the median value. When comparing only the slowest nodes, the variability in runtime of the SM implementation is very low, however the slowest nodes are incomparable to the full set of node results, and in all cases they sit

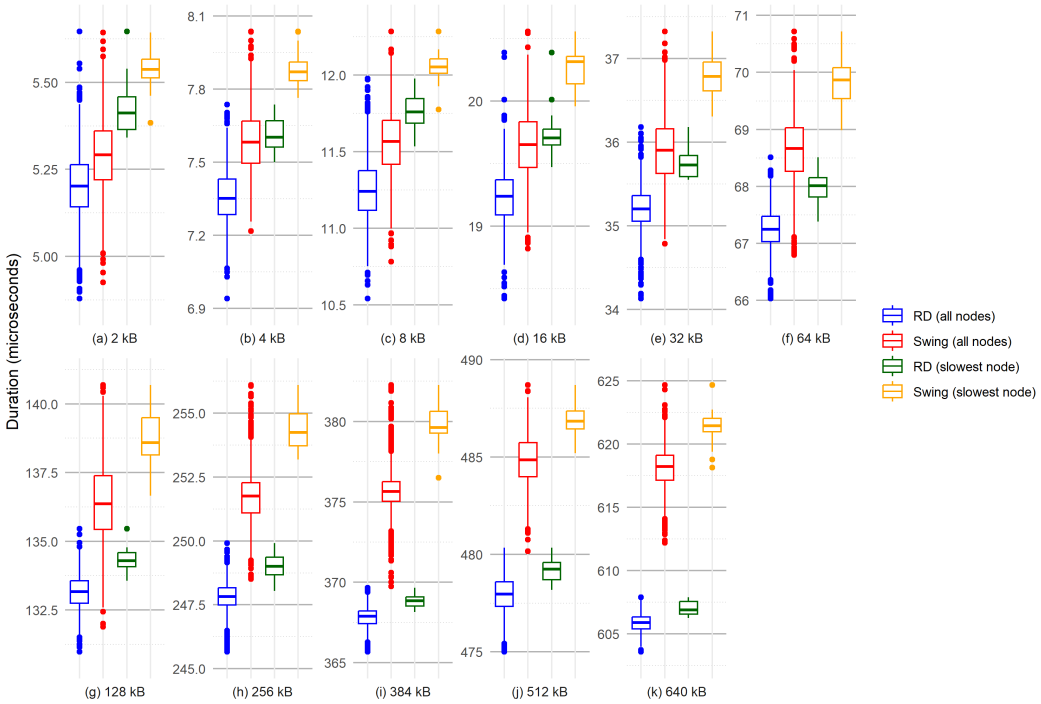


Figure 6.3. Individual node run times for different data sizes using Latency Optimal algorithm

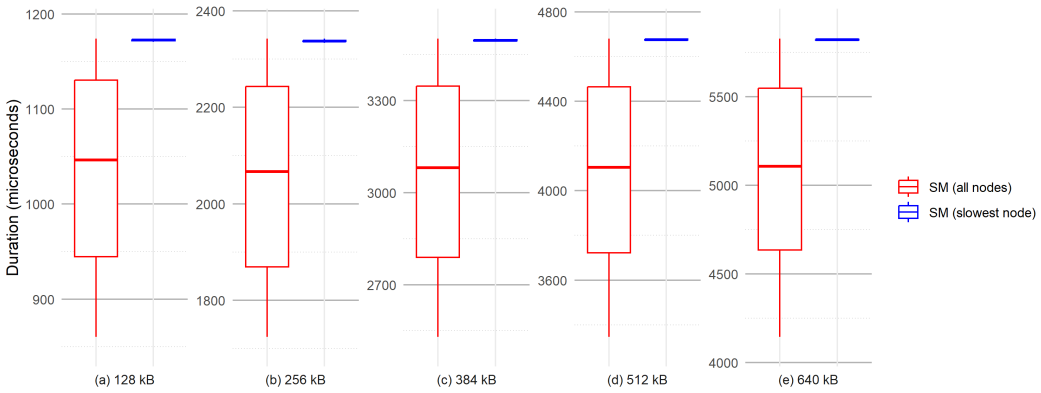


Figure 6.4. Individual node run times for different data sizes using the Shared Memory algorithm

completely outside of the all node IQR. This is almost certainly due to memory access limitations. The nodes are unable to concurrently access the data they require.

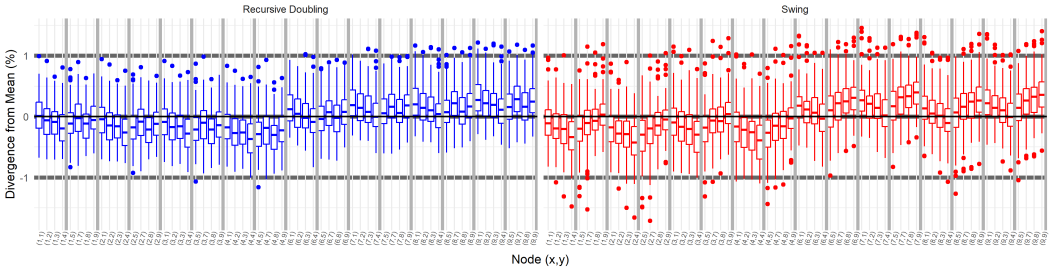
6.2.2 Effects of node on completion times

To help understand the algorithm, hardware performance, and potential deficiencies or anomalies, the run times of individual nodes have been analyzed. This is to see if any nodes do not perform as well, or if any algorithms consistently underperform

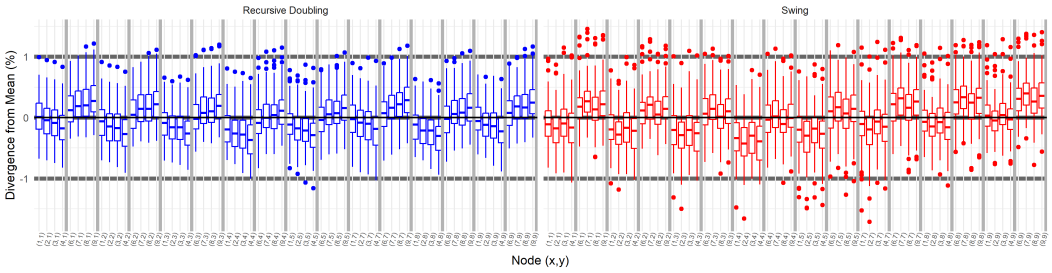
on some nodes. To generate the data, individual node run times were normalized by computing their percent deviation relative to the median completion time of the run. This provided a figure for variation for every node on every run. This data was then combined to produce box plots for every algorithm.

Because the grid is in 2D, each box plot was generated using the same data but from both column major, and row major perspectives. This is to help understand the effect of a node's row position and column position on the completion time. The plots show vertical dividers to help understand the "block" of the node grid, shown in Figure **??**, that any particular node belongs to. Note, the node x and y values correspond to those depicted in Figure **??**. The results of this are shown in Figures **??** to **??**.

In each of these plots, when multiple nodes in a given block tend to finish later, or have more or less variation, this can indicate there are inherent biases in the hardware or algorithm. Depending on if the plot is row major or column major tells us if the bias is related to the row or the column of the node, and also could give hints as to what the cause is.



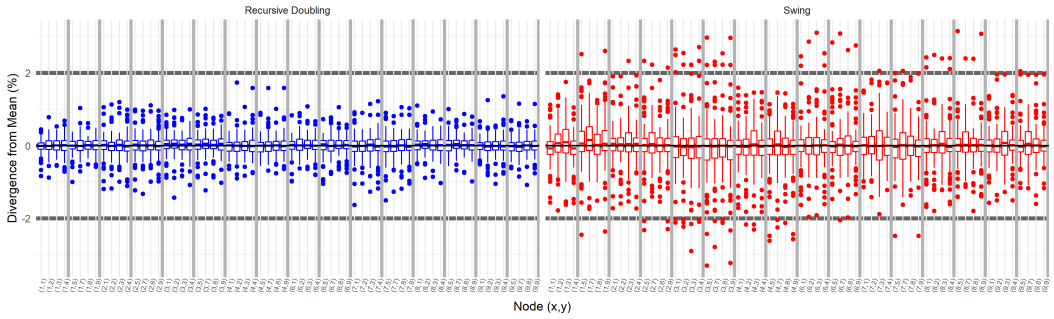
(a) Column major



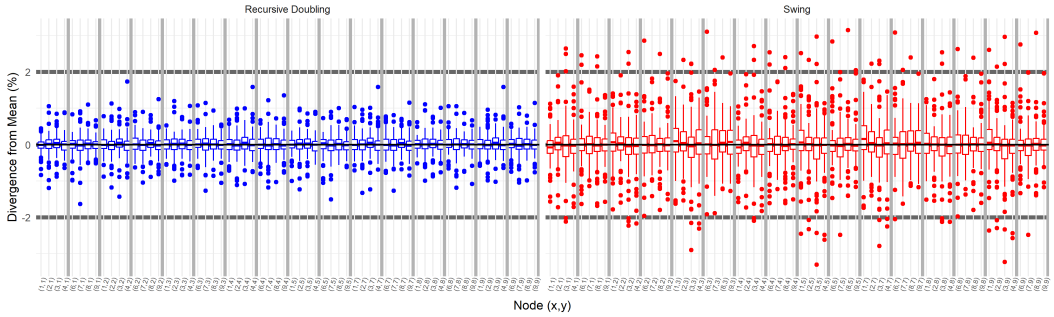
(b) Row major

Figure 6.5. Node divergence, Bandwidth Optimal

These graphs show some interesting correlations. Firstly, the effect of row and column position on both variance and relative finish time is inconsistent between algorithms. On Bandwidth Optimal (Figure **??**) both Swing and RD show a correlation between column and runtime. For example, column 7, and to some extent column 9, show strong correlation, consistently completing earlier than the average for both Swing and RD, although almost always within 1%. For Swing, columns 1 through 4 all tend to finish slightly earlier. However, from a row major perspective, only Swing shows correlation. The first 4 nodes of rows 5 to 9 consistently finish slightly later, whereas on RD there is little to no correlation with row and run time.

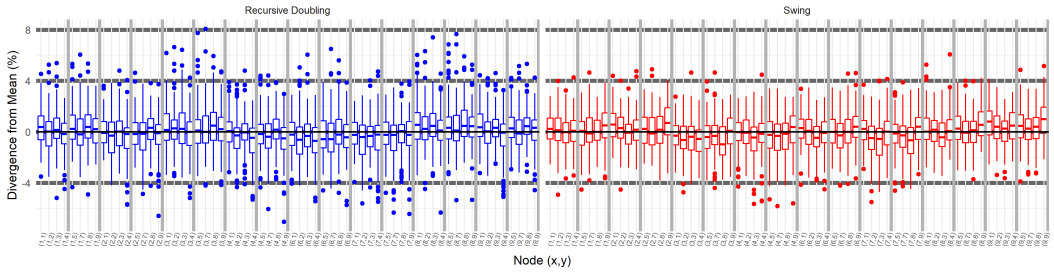


(a) Column major

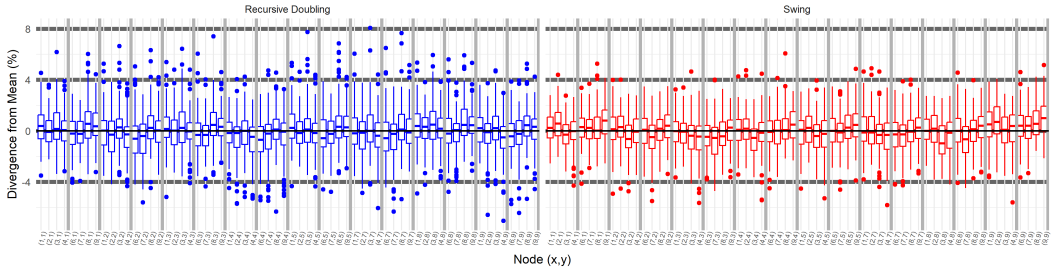


(b) Row major

Figure 6.6. Node divergence, Latency Optimal, data size $\geq$ 128 kB



(a) Column major



(b) Row major

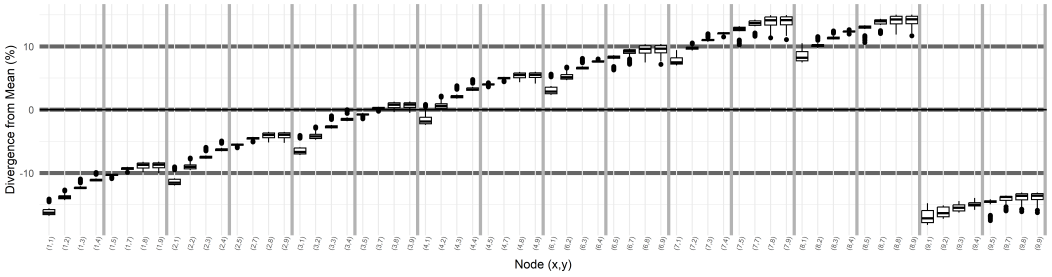
Figure 6.7. Node divergence, Latency Optimal, data size $<$ 128 kB

This is in contrast with the large LO results (Figures **??** and **??**). Here, there

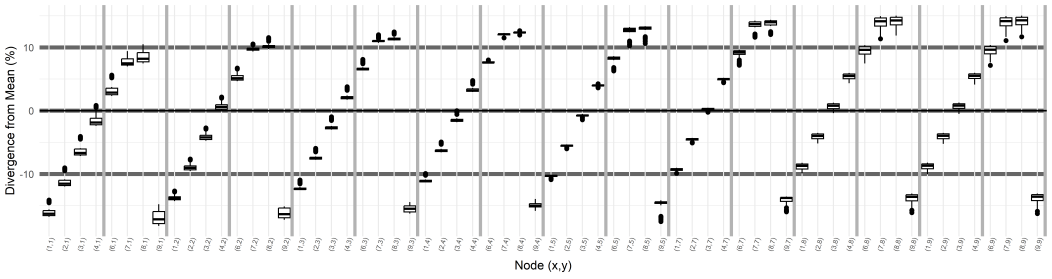
aren't significant correlations between row or column on early or late finishing, however there is correlation between these and the variability. For example, using RD column 7 has more variability, and column 1 has quite a lot less. Using Swing, columns 3, 6 and 8 have significantly more variation, while rows 1 and 9 have less. From a row perspective, with RD, there is very limited correlation, whereas with Swing there is some, row 3 for example has more variability. Generally, the Swing algorithm has significantly worse consistency performance here, with much more variability than RD.

Between large LO (vector size $\geq$ 128 kB, Figures **??** and **??**) and small LO (vector size $<$ 128 kB, Figures **??** and **??**) there are also differences. With smaller vectors, there is very comparable variation in run times between RD and Swing (which, as is to be expected with the shorter run times from smaller data sizes, is generally greater than on larger vectors). At the same time, the effect of node location is much lower, with no obvious or strong correlations between location and run time.

Although there are clearly some correlations between node and completion time/variability, the lack of consistency between the affected nodes and the run configuration suggest that this is not simply a question of hardware performance. The different results between configurations suggest there are some biases in the algorithms, that potentially interact with some of the nuances of the hardware. Sections **??** and **??** explore this further.



(a) Column major



(b) Row major

Figure 6.8. Node divergence, Shared Memory

The most significant correlations between node location and run time are seen on the Shared Memory algorithm, see Figures **??** and **??**. Here, there are very strong correlations between the column and completion time, with some additional

correlation for the row, as shown when comparing Figures ?? and ??. Starting from column 9 and wrapping around to the right, the completion times steadily increase until column 8. Additionally, for each column, from the top row to the bottom row times also increase, however this effect is secondary to the column. These extremely strong correlations are likely due to the communication directions and the mode in which nodes access DRAM, see Section ?? for further analysis.

6.2.3 Effect of run time calculation

To better understand the timing distributions, the results were recomputed using Equation ?? from Section ??. The full analysis will not be repeated as it is not necessary; the general results were relatively consistent with the methodology previously shown. The overall results of the recomputed analysis can be seen in Figure ??.

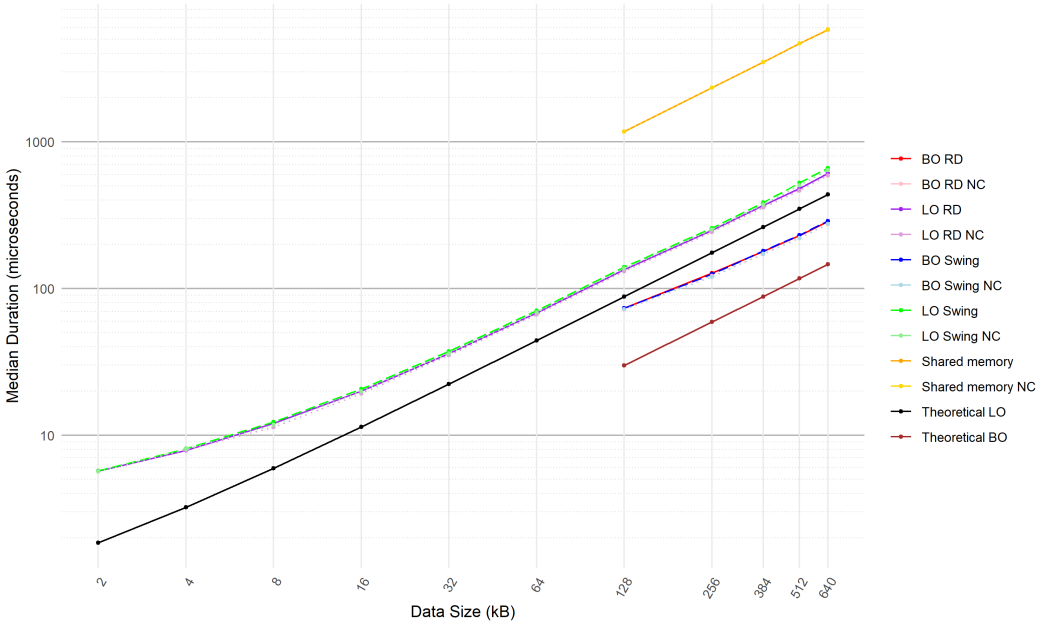


Figure 6.9. Median runtime of slowest core for all algorithms on all data sizes, using Equation ??

As can be seen, the overall ordering does not change at all. BO with RD is consistently fastest, followed by BO/Swing, LO/RD, LO/Swing and finally SM. Comparing Figures ?? and ??, the largest change in results is with the LO Swing algorithm, particularly at larger sizes. To better understand why this may be, the node divergence plot was repeated for LO $\geq$ 128 kB in Figure ??.

The reason that LO Swing changed significantly and LO RD did not is clear. Using Swing, several nodes ((3,3), (3,7), (7,3) and (3,7)) consistently finished early (up to 6%), and all other nodes tended to finish either slightly early or slightly late. The particularly early nodes skew the results seen in Figure ??. When comparing with the results using Equation ?? (Figure ??), the extremely early nodes do not show any tendency to be strong outliers. This signifies that the nodes are both

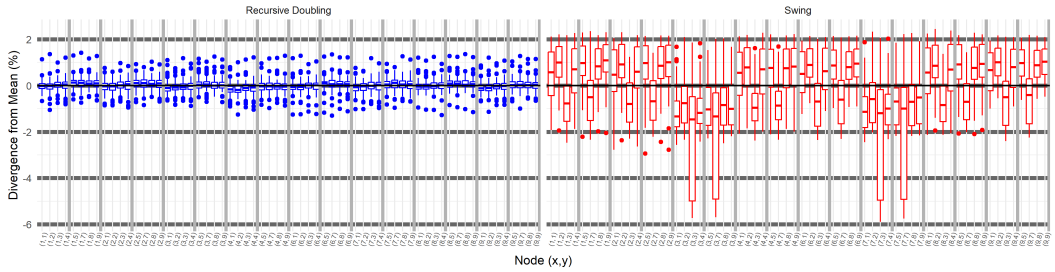


Figure 6.10. Column major node divergence, Latency Optimal, size $\geq$ 128 kB, using Equation **??**

starting and finishing early. Since the loops are repeated, if the node has a tendency to finish early, it will also have a tendency to start the next iteration early. This is notably in contrast with the Recursive doubling algorithm, where each core has a tendency to start and end at the same time.

6.3 Analysis

Generally, the results were strong for the LO and BO implementations. There is a high degree of repeatability between runs, and there is also relatively good consistency between individual nodes on a given run. Both of these are useful characteristics for an effective allreduce algorithm. Additionally, both show a massive improvement in performance over a simple memory based algorithm.

In general, there was the slightly surprising result that the Swing algorithm often underperformed in comparison to the RD algorithm. This result was unexpected, and potential reasons will be discussed in the following sections.

6.3.1 Bandwidth and FLOPs

Given the n150d performance numbers, the theoretical runtime of the algorithm was calculated using the equation in Appendix **??** and plotted in Figure **??**. Further analysis of the theoretical data can be seen in Figure **??**. These calculations use numbers provided by Tenstorrent.

In all cases, the limiting factor in performance was always the node to node communication bandwidth, this dominated the total runtime in all but the very smallest array sizes. The theoretical total computational time was always around 3 orders of magnitude faster than the theoretical total communication time, regardless of algorithm or data size (see Figure **??**c). This is supported by the No Compute testing that was conducted. This testing resulted in run times that were almost the same as those without computation.

The last block of tiles to be transferred cannot be parallelized. Computation cannot start until the dataflow core has completely finished the transfer for a block of tiles, and the dataflow core requires that the full computation must be complete before the next step can be started. This unparallelizable part is not factored into the theoretical analysis, but may be a contributing factor between the differences from the NC run and the normal run.

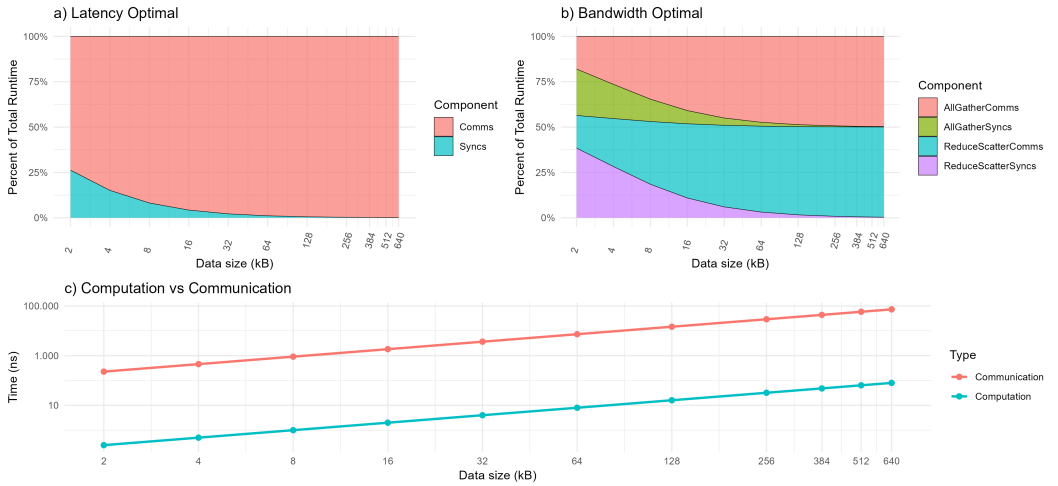


Figure 6.11. Theoretical proportion of total runtime of different operations for a) Latency Optimal and b) Bandwidth Optimal. c) Comparison of total time to Compute/Write different vector sizes.

Figures **??**a and **??**b also show the distribution between the total time spent synchronizing (which is a latency limited operation) and data transfer (a bandwidth limited operation). As can be seen, at the smallest data size the synchronization process takes the majority of runtime, however with larger data the runtime is dictated by bandwidth.

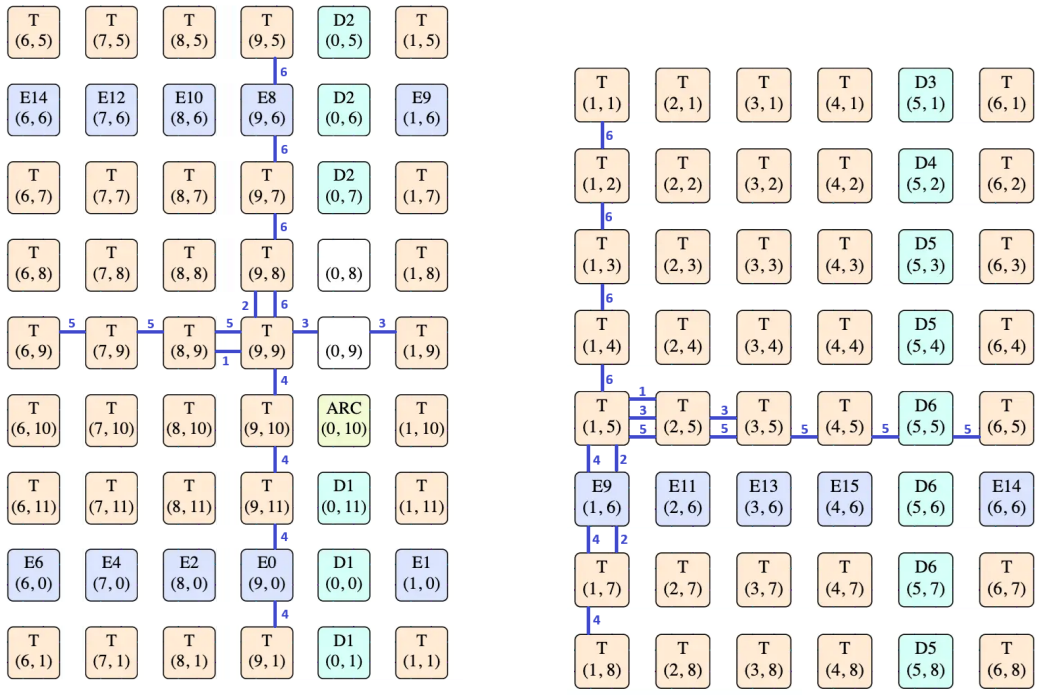
Despite the experimental results being lower than the theoretical optimal, the theoretical optimal is a roofline that allows a reasonable estimation of the upper limit of performance, rather than a specific target that must be achieved. The overall performance of the algorithm is in line with the expectations that the theoretical evaluation should set. The runtimes seen are relatively close to the theoretical maximum possible performance, highlighting the quality of the implementation.

6.3.2 Hidden nodes

One of the key properties that Swing aims to take advantage of is the "short" jump of the toroidal links. However, on the n150d this is a simplification. The more complex reality of the situation is shown Figure **??**, there are numerous nodes on the grid that are not usable for computation, but still have an effect on message passing latency.

In theory, the Swing algorithm should result in around 33% fewer hops in total over the RD algorithm. Because the 8×8 grid used in the algorithm is in fact 10×12, the assumptions that underpin the algorithm fail to hold and, as a result, these hidden nodes may be having a real effect on performance. T node (9,9) on a true 8×8 grid would make a total of 10 hops over the course of the algorithm. When accounting for the hidden nodes of the n150d, the total number of hops becomes 15 (see Figure **??**). Node (9,9) is one of the worst affected nodes, but because every node has a dependency on every other node, the worst affected nodes have a direct impact on total run time of all nodes.

For RD, one of the worst affected nodes is node (1,5). In the ideal case (no



(a) Swing communication pattern from point of view of node (9,9)

(b) RD communication pattern from point of view of node (1,5)

Figure 6.12. Comparison of communication patterns from different node perspectives

hidden nodes) RD would require 14 hops from node (1,5), however using the Tenstorrent hardware, including hidden nodes, the total hops reaches 17 (see Figure **??**). Although RD still does require more total hops than RD, the proportional impact of the hidden nodes is notably worse for Swing.

The timing and extra distance of the long latency hops is likely also having an effect. On every step from step 2 through to 6 of the Swing algorithm, some (but not all) of the nodes are affected by hidden nodes, and on step 5 the affected nodes have to make 3 extra hops. In contrast, RD is not at all affected by hidden nodes on step 3, and on the affected steps (2, 4, 5 and 6) it never uses the long top to bottom toroidal link, and instead communications have at most 1 extra hop, meaning the difference between affected and unaffected nodes is less.

When comparing the worst cases (the most hops made by any node on each step), for Swing, for steps 1 through 6 respectively, the situation is $1+2+2+4+4+4 = 17$ whereas for RD its $1+2+2+3+5+5 = 18$. The extra hops come slightly earlier with the Swing algorithm, allowing more time for any delays to propagate further through the network.

The hidden node situation is a complex one. There are multiple factors simultaneously impacting communication times, all with knock on, and potentially compounding, affects. As such, the only conclusion that can be drawn with confidence, is there is definitely some effect from this, particularly at smaller sizes where latency is more of an issue.

6.3.3 Communication direction

The results of the Shared Memory algorithm show some clear patterns regarding completion time and node position. The Shared-Memory implementation utilizes the Tenstorrent convention of using a single NoC interface (SE) for reading and the other (NW) for writing. The final step of the SM implementation is a read operation, as such it is conducted by every SE NoC interface in the grid after a full grid synchronization.

The results are somewhat unexpected. The earliest nodes to finish aren't necessarily closest in the SE direction to the D nodes (see Figure **??**) that connect with DRAM. The full array that is being reduced would likely be held on just one of the DRAM banks, and as such would be accessed through only one of the six triplets of D nodes (e.g. it could be held on the DRAM bank connected to the D1 nodes). There is no set of DRAM nodes who's distance from each Tensix node in the SE direction matches the timings seen in Figure **??**.

However, this mindset may be an oversimplification. The node synchronization step merely guarantees all nodes have reached the barrier, it does not guarantee that all nodes exit the barrier at the same moment. Other contributing factors could be that there is some inherent skew on barrier exit time, a hidden DRAM access ordering preference, unusual DRAM allocation patterns, or the n150d node position diagram may be outdated. This can explain the pattern visible in memory access ordering.

6.3.4 Variability of start times

The variation of start and end times seen in Figure **??** with the LO with large vector size is notable. The fact is, the LO algorithm is less relevant on these larger sizes, as it significantly under performs the BO variant. However, there are several possible reasons for the discrepancy.

The variation in start times, as mentioned previously, is a direct result of variations in finish times. While the allreduce algorithm does have some degree of "synchronizing" properties (no node can finish until all nodes have started) it does not provide any guarantee that all nodes will finish at the same time. In fact, in the last 2 steps of the LO allreduce, the early finishing nodes (3,3), (3,7), (7,3) and (3,7) form a clique (they only communicate within their group), which explains why they consistently finish together.

A reason they likely finish early is that for all nodes in the clique they have around the minimum hidden-node extra-hops possible from the Swing implementation, a total of 12 or 13 hops for all nodes in the clique. Because the LO algorithm ends on the "long hop" step (unlike the BO algorithm, which ends on a neighbor communication step), this bias can show on LO while not showing on BO. Finally, these effects likely compound as the run is repeated, with the clique always finishing early, allowing them to start early, and solidify their "lead".

However, clearly this isn't seen on the RD algorithm. This is likely because Swing tries to keep communication "tight", where nodes only communicate with other nearby nodes. RD on the other hand "spreads out" more, more distant nodes communicate, meaning these "cliques" don't form.

Chapter 7

Conclusion and Future Work

To conclude, the allreduce procedure was successfully implemented and compared using five different algorithms. Performance of these algorithms has been assessed, and is as close as can be reasonably expected to reaching the theoretical optimal result possible. The performance of the Bandwidth Optimal allreduce algorithm was significantly better than the other algorithms tested, and the Recursive Doubling implementation was found to perform slightly better than Swing, but with minimal difference in performance. The reason for the nonoptimality of the Swing algorithm is adequately explained by the presence of "hidden" nodes, which cause significantly longer communication distances than the algorithm is optimized for.

Overall, to conclude, there is no question that the implementation of an optimized core-to-core allreduce algorithm is much better than any implementation that requires accessing the shared DRAM. While Swing is not optimal in this specific case, the result is still useful to understand the limitations of the algorithm.

7.1 Suitability of Tenstorrent hardware for parallel computing

This body of work required a lot of low-level programming on the Tenstorrent n150d. Despite many ups and downs, in the end it was possible to implement all algorithms in an optimized and parallelized way that utilizes compute and NoC units simultaneously, which shows that TT-Metalium is a versatile tool.

However, the difficulty in programming on the hardware must not be overlooked. The lack of clear documentation is a serious stumbling block, however it is one that is being improved and worked on. In fact, over the course of this investigation, numerous documents were updated that provided further clarity on the details of the n150d operation, and this should for the most part only improve, although with the rate of development of the library this is a slightly Sisyphean task.

But the documentation was only one aspect. The real challenge was the hardware instability. When developing parallel processing applications, deadlocks are a fact of life. The frustration of having to reset the entire server on each deadlock, the lost time and the annoyed colleagues, cannot be understated. This was the challenge that made the task truly hard.

7.2 Potential Optimizations

Following the development and analysis of the implementation, several potential optimizations were identified that could result in significant performance gains.

The first potential improvement is relatively simple. Extensive testing should be conducted on the number of synchronization steps to understand the optimal for different data sizes. As it is, a series of relatively quick and surface level tests were conducted that resulted in an optimal figure of 32 synchronizations. It is likely that this optimal number is not fixed, but should be tuned according to data size.

A second potential improvement that was identified, but cannot be implemented given the API available, is that multiple steps could be performed synchronously. Once compute has finished adding the first tile of `cb_recv` to `cb_local`, both of these tiles are theoretically free to be written to/read from respectively. Currently the method of synchronization requires that the entire buffer is fully emptied before the next step is started. If another method of synchronization could be used, potentially two steps of the algorithm could be conducted simultaneously by NW and SE cores respectively.

7.3 Directions for Future Research

Besides optimizations of the algorithm, or multiport versions of it, an allreduce algorithm that is likely to be very strong is the Hamiltonian Ring. The Tenstorrent layout is well suited to this because of its Toroidal shape, but also because it is very much bandwidth limited. Hamiltonian rings are bandwidth optimized, and it is only on the very smallest vectors that communication latency begins to take up a large part of the runtime.

Additionally, this work has been purely about understanding the *performance* of allreduce algorithms. A suitable next step is to pursue a working, generalized, implementation that can be deployed to the TT-Metalium kernel, so it can be used for general programming applications.

Appendix A

Appendices

A.1 Clarifications

A.1.1 Physical and logical core coordinates

Note, the logical coordinates of a node don't correspond with the real physical coordinates of a node. The logical position is the position of the Tensix core for the purposes of algorithms, so the top left core would logically be (0,0). The real coordinate is the node's actual physical position in the full 10×12 array, which for logical (0,0) could actually be (1,1). This is not fixed, so is computed at run time and cannot be hard coded.

A.2 Source Code Snippets

A.2.1 Swing communication partner

Implementation in `C++` of Equation **??** that returns the linear index of the communication partner of `node` at `step`:

Listing A.1. Swing communication partner

```
int get_comm_partner_swing_2D(int node, int step, bool
    horizontal_step, int SIDE_LENGTH, int TOTAL_NODES) {
    int row = node / SIDE_LENGTH;
    int col = node % SIDE_LENGTH;
    step = step / 2;

    // straight line distnce
    int dist = (int)((1 - (int)pow(-2, step + 1)) / 3);

    int comm_partner;
    if (horizontal_step) {
        comm_partner = (node % 2 == 0) ? (node + dist) : (node
            - dist);
        if (comm_partner / SIDE_LENGTH < row || comm_partner <
            0) {
            comm_partner += SIDE_LENGTH;
        } else if (comm_partner / SIDE_LENGTH > row) {
```

```
            comm_partner -= SIDE_LENGTH;
        }
    } else {
        comm_partner = (row % 2 == 0) ? (node + SIDE_LENGTH *
            dist) : (node - SIDE_LENGTH * dist);
        if (comm_partner < 0) {
            comm_partner += TOTAL_NODES;
        } else if (comm_partner >= TOTAL_NODES) {
            comm_partner -= TOTAL_NODES;
        }
    }
    return comm_partner;  // will  loop round  to  always be
        in  range
}
```

A.2.2 Blocks to send calculations

A 64-bit binary sequence is computed that contains the binary sequence representing which blocks need to be transmitted at any given step.

Listing A.2. Swing blocks to send assignment

```
void get_swing_block_comm_indexes(
    int node, int step, uint32_t* blocks, bool horizontal_step
        , int SIDE_LENGTH, int TOTAL_NODES) {
    int num_steps = (int)log2((double)TOTAL_NODES);
    if (step >= num_steps) {
        return;
    }
    for (int s = step; s < num_steps; s++) {
        int peer = get_comm_partner_swing_2D(node, s,
            horizontal_step, SIDE_LENGTH, TOTAL_NODES);
        if (peer < 32) {
            *blocks = *blocks | (1 << peer);
        } else {
            *(blocks + 1) = *(blocks + 1) | (1 << (peer - 32))
                ;
        }
        horizontal_step = !horizontal_step;
        get_swing_block_comm_indexes(peer, s + 1, blocks,
            horizontal_step, SIDE_LENGTH, TOTAL_NODES);
    }
    return;
}
```

A.2.3 Recursive Doubling communication partner

The RD communication partner calculation, which returns the index of the node being communicated with as well as the direction of communication, required for assigning the step to the NW or SE core.

Listing A.3. Recursive Doubling communication partner

```
int get_comm_partner_recdub_2D(
    int node,
    int recdub_step,
    bool horizontal_step,
    int message_pass_depth,
    uint32_t& step_directions,
    int SIDE_LENGTH) {
    int row = node / SIDE_LENGTH;
    int col = node % SIDE_LENGTH;
    int node_position = horizontal_step ? col : row;
    int node_other_position = !horizontal_step ? col : row;

    bool sending_SE = node_position % (2 * message_pass_depth)
        < message_pass_depth;
    step_directions = sending_SE ? (step_directions | (1 <<
       recdub_step)) : (step_directions & ~(1 << recdub_step))
       ;

    int recv_node = node_position + (sending_SE ?
       message_pass_depth : -message_pass_depth);
    return horizontal_step ? recv_node + node_other_position *
        SIDE_LENGTH
                           : recv_node * SIDE_LENGTH +
                              node_other_position;
}
```

A.2.4 Dataflow loop

Simplified code for the Latency and Bandwidth Optimal dataflow kernels.

Listing A.4. NoC dataflow loop

```
for (uint32_t i = 0; i < algo_steps; i++) {
    direction_SE = (packed_direction_bools >> i) & 1;
    sync_NOC(cb_id_this, cb_id_that);

    uint32_t n_block_sync = sync_stride;
    if (this_core_SE == direction_SE) { // Comm core
        dst_noc_semaphore_0 = get_noc_addr(dst_core_x[i],
           dst_core_y[i], semaphore_0[i \% num_sem_0]);
        dst_noc_semaphore_1 = get_noc_addr(dst_core_x[i],
           dst_core_y[i], semaphore_1[0]);

        cb_reserve_back(cb_id_local, num_tiles);

        noc_semaphore_inc(dst_noc_semaphore_0, 1);
```

```
        noc_semaphore_wait_min(semaphore_0_ptr[i \% num_sem_0
           ], semaphore_wait_count);

        for (uint32_t n_block = 0; n_block < total_nodes; ) {
            bool send_block = shouldSendBlock();
            uint32_t blocks_to_send = 0;
            if (send_block) {
                dst_noc_addr = get_noc_addr(dst_core_x[i],
                   dst_core_y[i], l1_write_addr_recv + offset)
                   ;
                while (send_block && n_block < total_nodes &&
                   n_block < n_block_sync) {
                    blocks_to_send++;
                    n_block++;
                    send_block = shouldSendBlock();
                }
                noc_async_write(l1_write_addr_local + offset,
                   dst_noc_addr, tile_block_size *
                   blocks_to_send);
            } else {
                n_block++;
            }
            if (n_block >= n_block_sync) {
                // Periodically (every num_tiles/num_sync
                   blocks) synchronize the nodes and
                // increment the the circular buffers,
                   allowing computation to proceed
                noc_async_write_barrier();
                noc_semaphore_inc(dst_noc_semaphore_1, 1);
                cb_push_back(cb_id_local, num_tiles /
                   num_syncs);
                n_block_sync = n_block_sync + sync_stride;
                if (n_block > num_tiles) {
                    n_block += total_nodes;
                }
            }
        }
    } else { // Listener core
        cb_reserve_back(cb_id_recv, num_tiles);
        for (uint32_t n_block = 0; n_block < num_syncs;
           n_block++) {
            noc_semaphore_wait_min(semaphore_1_ptr[0], i *
               num_syncs + n_block + 1);
            cb_push_back(cb_id_recv, num_tiles / num_syncs);
        }
    }
}
```

A.2.5 NoC core synchronization

A simple function that uses two one-tile Circular Buffers to synchronize the two NoC cores of a Tensix core.

Listing A.5. NoC core synchronization

```
void sync_NOC(int cb_id_this, int cb_id_that) {
    cb_reserve_back(cb_id_that, 1);
    cb_push_back(cb_id_that, 1);
    cb_wait_front(cb_id_this, 1);
    cb_pop_front(cb_id_this, 1);
}
```

A.2.6 TT-Metalium math abstraction

The abstraction for summing two vectors and storing the result at position `tile_num` in a Circular Buffer.

Listing A.6. Math abstraction

```
tile_regs_acquire();
add_tiles(cb_id_local, cb_id_recv, 0, 0, reg_index);
tile_regs_commit();
tile_regs_wait();
pack_tile<true>(reg_index, cb_id_local, tile_num);
tile_regs_release();
```

A.2.7 Shared memory dataflow implementation

Shared memory implementation of allreduce, utilizing the TT-Metalium standard practice of reading through NoC interface 1 and writing through NoC interface 0.

Listing A.7. SM dataflow implementation pseudocode

```
if (!this_core_SE) { // Write this node's data to DRAM
    noc_async_write(l1_write_addr_local, common_noc_addr,
       total_vector_size);
    noc_async_write_barrier();
}

sync_nodes();

if (this_core_SE) { // Read relevant parts of other vectors
   from DRAM
    cb_push_back(cb_id_local, num_tiles);
    for (uint32_t i = 0; i < total_nodes; i++) {
        read_offset = this_core_i * tile_block_size;
        common_noc_addr = get_noc_addr_from_bank_id<true>(
           common_bank_id, common_addr + read_offset);
        noc_async_read(common_noc_addr, l1_write_addr_recv + i
            * tile_block_size, tile_block_size);
        noc_async_read_barrier();
```

```
        cb_push_back(cb_id_recv, num_tiles_per_node);
    }
}

cb_wait_front(cb_id_this, 1);
cb_pop_front(cb_id_this, 1);

if (!this_core_SE) { // Write computed portion of vector to
    DRAM
    noc_async_write(l1_write_addr_local, dst0_noc_addr +
        offset, tile_block_size);
    noc_async_write_barrier();
}

sync_nodes();

if (this_core_SE) { // Read all other computed portions of
    vector from DRAM
    noc_async_read(dst0_noc_addr, l1_write_addr_local,
        total_vector_size);
    noc_async_read_barrier();
}
```

A.3 Calculations

A.3.1 Theoretical node runtime

The following numbers and equations were used to calculate theoretical runtime of the algorithm:

$$\text{Total TFLOPS (FP8)} = 262, \tag{A.1}$$

$$\text{Nodes} = 64, \tag{A.2}$$

$$\text{TFLOPS per node} = \frac{262}{8} = 4.09. \tag{A.3}$$

$$\text{FP} = \frac{\text{data\_size (bytes)}}{\text{size of FP8 (bytes)}}. \tag{A.4}$$

$$t_{\text{comp}} = \frac{\text{FP}}{\text{TFLOPS/node}}, \tag{A.5}$$

$$B_{\text{comm}} = 9 \text{ GB/s}, \tag{A.6}$$

$$t_{\text{comm}} = \frac{\text{data\_size}}{B_{\text{comm}}}. \tag{A.7}$$

$$t_{\text{sync/hop}} = 9 \times 10^{-9} \text{ s}, \tag{A.8}$$

$$\text{hops} = 18, \tag{A.9}$$

$$t_{\text{sync}} = 18 \times 9 \times 10^{-9} = 1.62 \times 10^{-7} \text{ s}, \tag{A.10}$$

$$t_{\text{sync\_1}} = 3 \times t_{\text{sync}} = 4.86 \times 10^{-7} \text{ s}, \tag{A.11}$$

$$t_{\text{sync\_2}} = 2 \times t_{\text{sync}} = 3.24 \times 10^{-7} \text{ s}. \tag{A.12}$$

$$N_{\text{steps}} = 6, \tag{A.13}$$

$$T_{\text{LatencyOpt}} = N_{\text{steps}} \max\big(t_{\text{comm}}, t_{\text{comp}}\big) + t_{\text{sync\_1}}, \tag{A.14}$$

$$T_{\text{BandwidthOpt}} = \max\big(t_{\text{comm}}, t_{\text{comp}}\big) + t_{\text{comm}} + t_{\text{sync\_1}} + t_{\text{sync\_2}} \tag{A.15}$$

Bibliography

[1] J. Shalf, "The future of computing beyond moore's law," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 378, no. 2166, p. 20 190 061, 2020. DOI: 10.1098/rsta.2019.0061

[2] A. de Vries-Gao, "Artificial intelligence: Supply chain constraints and energy implications," *Joule*, vol. 9, no. 6, pp. 1–5, Jun. 2025. DOI: 10.1016/j.joule.2025.101961

[3] S. Chunduri, S. Parker, P. Balaji, K. Harms, and K. Kumaran, "Characterization of mpi usage on a production supercomputer," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2018, pp. 386–400.

[4] B. Klenk, N. Jiang, G. Thorson, and L. Dennison, "An in-network architecture for accelerating shared-memory multiprocessor collectives," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, IEEE, 2020, pp. 996–1009.

[5] A. Sapio et al., "Scaling distributed machine learning with in-network aggregation," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, USENIX Association, Apr. 2021, pp. 785–808.

[6] W. Wang et al., "Topoopt: Co-optimizing network topology and parallelization strategy for distributed training jobs," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, Boston, MA: USENIX Association, Apr. 2023, pp. 739–767.

[7] A. Weingram, Y. Li, H. Qi, et al., "Xccl: A survey of industry-led collective communication libraries for deep learning," *Journal of Computer Science and Technology*, vol. 38, no. 1, pp. 166–195, Jan. 2023. DOI: 10.1007/s11390-023-2894-6

[8] P. Patarasuk and X. Yuan, "Bandwidth optimal all-reduce algorithms for clusters of workstations," *Journal of Parallel and Distributed Computing*, vol. 69, no. 2, pp. 117–124, 2009, ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2008.09.002 [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0743731508001767

[9] M. Barnett, R. Littlefield, D. Payne, and R. van de Geijn, "Global combine algorithms for 2-d meshes with wormhole routing," *Journal of Parallel and Distributed Computing*, vol. 24, no. 3, pp. 191–201, 1995. DOI: 10.1006/jpdc.1995.1018

[10] R. Thakur, R. Rabenseifner, and W. Gropp, "Optimization of collective communication operations in mpich," *International Journal of High Performance Computing Applications*, vol. 19, no. 1, pp. 49–66, Feb. 2005. DOI: 10.1177/1094342005051521

[11] D. D. Sensi, T. Bonato, D. Saam, and T. Hoefler, "Swing: Short-cutting rings for higher bandwidth allreduce," in *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, Santa Clara, CA: USENIX Association, Apr. 2024, pp. 1445–1462. [Online]. Available: https://www.usenix.org/conference/nsdi24/presentation/de-sensi

[12] Tenstorrent, *Wormhole™: Tensix-core processors (n150 / n300)*, Product page on Tenstorrent website, Details on Wormhole n150d/n300d boards: Tensix core count, SRAM and GDDR6 specs, memory bandwidth, power, and cooling, 2025. [Online]. Available: https://tenstorrent.com/hardware/wormhole

[13] Corsix, *Tenstorrent wormhole series part 1: Physicalities*, Corsix.org blog post, Detailed breakdown of Wormhole n150s/n300s PCIe card hardware layout and tile grid architecture, Sep. 2024. [Online]. Available: https://www.corsix.org/content/tt-wh-part1

[14] Tenstorrent, *Tenstorrent engineers talk open-sourced bare-metal stack*, YouTube video, Feb. 2024. [Online]. Available: https://www.youtube.com/watch?v=oeKRtNkx_Ek

[15] D. Patel, *Tenstorrent wormhole analysis – a scale out architecture for machine learning that could put nvidia on their back foot*, SemiAnalysis (blog post), Originally published on SemiAnalysis, Jun. 2021. [Online]. Available: https://semianalysis.com/2021/06/25/tenstorrent-wormhole-analysis-a-scale/

[16] Corsix, *Tenstorrent wormhole series part 3: Noc propagation delay*, Corsix.org blog post, Analysis of tile-to-tile propagation delay (≈ 9 cycles) on Tenstorrent Wormhole NoC using cycle-counter measurements, Sep. 2024. [Online]. Available: https://www.corsix.org/content/tt-wh-part3

[17] Tenstorrent, *Introduction to data movement (tt-metal / data movement documentation)*, GitHub documentation file, Discusses data movement primitives, memory hierarchy, memcpy versus custom methods, and performance implications within TT-Metal, 2025. [Online]. Available: https://github.com/tenstorrent/tt-metal/blob/ca60f6a466d95ba20a73932186463d5a71b707c2/tests/tt_metal/tt_metal/data_movement/documentation/intro_to_dm.md

[18] Tenstorrent, *One to one packet sizes (wormhole b0)*, GitHub image in TT-Low-Level Kernel Documentation, Performance plot showing data movement for one-to-one packet sizes on Wormhole B0, 2025. [Online]. Available: https://github.com/tenstorrent/tt-low-level-documentation/blob/913e4d16301942e4c13d47a3f31a19a36e52550b/data_movement_doc/perf_plots/wormhole_b0/images/One%20to%20One%20Packet%20Sizes.png

[19] S. Ward-Foxton, "Tenstorrent engineers talk open-sourced bare-metal stack," *EE Times*, Feb. 2024. [Online]. Available: https://www.eetimes.com/tenstorrent-engineers-talk-open-sourced-bare-metal-stack/

[20] Tenstorrent, *Tt-metalium™*, Tenstorrent software page, Open-source low-level AI hardware SDK giving direct access to Tensix Core, RISC-V, NoC, Matrix and Vector engines, 2025. [Online]. Available: https://tenstorrent.com/software/tt-metalium

[21] Tenstorrent, *TT-LLK: L1 (sram) introduction*, GitHub documentation file (docs/llk/l1/intro.md), Introduction to L1 local memory (SRAM) in TT-LLK low-level kernels, 2025. [Online]. Available: https://github.com/tenstorrent/tt-llk/blob/3924087d64f6873f9c69d4b54f41e26e2afccbec/docs/llk/l1/intro.md

[22] Tenstorrent, *Tt-metalium guide: Architecture and programming model*, GitHub documentation file (METALIUM_GUIDE.md), Introduction to TT-Metalium programming model, scalable architecture, memory hierarchy and RISC-V kernel control, 2025. [Online]. Available: https://github.com/tenstorrent/tt-metal/blob/main/METALIUM%5C_GUIDE.md

[23] Tenstorrent, *Tracy: Tt-metalium fork of the wolfpld/tracy profiler*, GitHub repository, Real-time, high-resolution profiler adapted for TT-Metalium (fork of wolfpld/tracy), 2025. [Online]. Available: https://github.com/tenstorrent/tracy

[24] DeepWiki Contributors, *Performance measurement in nccl-tests*, DeepWiki Technical Documentation, Explains timing, bandwidth metrics, and implementation details in NCCL-tests benchmarking framework. Includes source references to 'README.md', 'PERFORMANCE.md', and 'src/common.cu', 2025. [Online]. Available: https://deepwiki.com/NVIDIA/nccl-tests/2.3-performance-measurement

[25] Keysight Technologies, *Benchmarking collective operations: A key to optimizing ai infrastructure performance*, White paper (via Keysight Technologies), Publication No. 7125-1009, explores benchmarking methodology for collective communication in distributed AI systems, Jan. 2025. [Online]. Available: https://www.keysight.com/us/en/assets/7125-1009/white-papers/Benchmarking-Collective-Operations.pdf

[26] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '15)*, New York, NY, USA: ACM, 2015, pp. 1–12. DOI: 10.1145/2807591.2807644

[27] J. Pješivac–Grbović, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra, "Performance analysis of mpi collective operations," *Cluster Computing*, vol. 10, no. 2, pp. 127–143, Mar. 2007. DOI: 10.1007/s10586-007-0012-0