Optimizing Raspberry Pi for Distributed Matrix Multiplication: A Communication and Scheduling Framework

Gerardo Jesus Melesio Sancen

Master of Science

Computer Science

School of Informatics

University of Edinburgh

2024

Abstract

This project develops a communication and job scheduling framework that integrates Raspberry Pi 5 computers into a cluster for distributed matrix multiplication. The framework optimizes the utilization of Raspberry Pi resources, including CPU, RAM, and network capabilities, to enhance overall computational efficiency It also presents mechanisms to reduce the amount of messages required for job execution. By deploying these resources in a producer-consumer model, we demonstrate that the system can scale linearly when integrating additional computing nodes.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics Committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Gerardo Jesus Melesio Sancen)

Acknowledgements

To my family for supporting me in coming back to school after all this time. I owe you everything.

To Natalia, as she has been an amazing partner and supported me more than I can ever repay her.

To Raini and Alec, for rekindling my love for systems and programming through their friendship.

To Enia, Elina, Rohith, Micky, Spyros, Sayon, Jovanna, Amir, and many more friends I had the joy to cross paths during my living time in Edinburgh. It is not about the destination nor the journey, but the people you traverse it with.

To Jeff Schoch, Cesar Jimenez, and Matt Cannon. My coworkers who let me take many pauses at work to accommodate learning time.

Table of Contents

Chapter 1

Introduction

1.1 Introduction

Many algorithms relying on linear algebra use matrix multiplication as a building block. Most notably, machine learning algorithms and graphic processors heavily utilize matrix multiplication. The traditional algorithm, known as the n^3 algorithm, scales poorly on a single CPU due to its $O(n^3)$ complexity. Therefore, several distributed computing algorithms have been proposed to distribute the workload across a grid of processors.

This work proposes a communication and scheduling framework suitable for performing distributed matrix multiplication on a cluster of Raspberry Pis. Furthermore, our communication protocol provides the system with the primitives to reduce the number of messages across the nodes, which is a critical point to provide scalability. We perform this message optimization either by reducing synchronization needs, reusing existing results or inputs from previous jobs, or performing task consolidation.

The proposed architecture for this project is composed of a Primary Node, which is a computer with more resources than the Raspberry Pi. The primary or central node schedules tasks on a cluster of Raspberry Pis. The tasks assigned to each Pi would be fragments of the overall distributed framework. This cluster would work as an extension of a central unit, each node acting as a worker receiving individual jobs from a scheduler. The nodes could be close to each other, or distributed across several points. Our work focuses on providing the most efficient communication to the cluster and scheduling jobs to all the Raspberry nodes in a pipelined fashion to avoid any CPU idle time on them, allowing linear

scalability when adding more worker nodes to the setup.

This paper will follow the following overall structure. This chapter begins by exploring the pivotal background work that has shaped our understanding of distributed matrix multiplication. We will also delve into the specific motivations driving this project, emphasizing the unique capabilities of the Raspberry Pi in our proposed solution. In subsequent chapters, we will describe the required behavior of the communication and scheduling framework according to the requirements of the system and the existing parallelization algorithms. The following chapters will focus on the implementation of our framework, and benchmarking its performance with different job aggregation strategies and scaling out settings. Finally, we will review our results and propose future work for the project.

1.2 Motivation

Given the ubiquitous nature of matrix multiplication in modern computational tasks, particularly in machine learning, a deeper historical and technical perspective is crucial. The following section provides a comprehensive background, laying the groundwork for understanding the evolution of distributed computing techniques that we aim to advance.

With the implosion of machine learning in many fields, users very frequently need to train AI models which involves a series of thousands of matrix multiplications. Deep Neural Networks are required to perform thousands of general matrix multiplications (GEMM). Current options for users training these large models are acquiring costly hardware and training their models locally, or training them using any of the cloud offerings available. Therefore, there is a lot of interest in optimizing the GEMM operations for machine learning training as shown in [27, 31] and many others.

Some computer enthusiasts have proposed building clusters of Raspberry Pi computers to provide users with an efficient and low-cost alternative to cloud computing or server hardware. There are several examples of building low-cost clusters using this mini-computer as an alternative for high-end servers.

Examples of these efforts can be found in their official documentation [10] as well as several academic papers [5]. Most of the articles found in academia talk about their usefulness in education, providing students and schools with a low-cost alternative for high-performance computing. This is supported by the

fact that a Raspberry Pi 5 costs 55 on Amazon as of August 7th, 2024. Building a cluster of tenths of Raspberry Pi nodes would be more affordable than getting a high-end server to be shared by a group of students.

1.3 Background

Since the 1960s, many academics have studied the problem of parallelizing matrix multiplication across a cluster of CPUs. Researchers have proposed different approaches, most of which rely on decomposing a large matrix into sub-matrices of a convenient size and then solving the multiplication with one of the existing algorithms by treating them as individual matrix elements. Examples of this approach for dense matrix multiplication exist for Strassen and approaches based on the classical matrix multiplication algorithm, also referred to as the n^3 algorithm.

In terms of notation throughout this paper, we define matrix multiplication as $C = A \times B$, where the dimensions of A are $m \times k$, B are $k \times n$, and C are $m \times n$. This notation will be consistently used unless otherwise specified.

Cannon's algorithm [4] first demonstrated computational efficiency in performing distributed matrix multiplication. The main innovation proposed by Cannon consisted of the aligned distribution of the sub-matrices across a grid of NxN processors. Each processor could then compute the required sub-products and aggregate them to the final results. This algorithm optimizes data distribution by performing cyclical shifts of rows and columns, ensuring that each processor receives the elements needed for the next phase of calculation without redundant data transmission.

Several investigators built over Cannon's work, proposing innovations in the grid layout and the sub-matrix redistribution. Notably in the 90's PUMMA[24], SUMMA, and DIMMA proposed approaches using broadcast primitives. These are useful for a matrix multiplication problem being solved over a grid, as they allow rows of processors to reutilize a row or column for their next calculation, reducing the need for constant shifting in each iteration and overall the message passing.

The best-performing algorithm from the n^3 family to our knowledge, was developed in 2012 and named CARMA by their authors [6]. CARMA is a BFS/DFS-based algorithm, which operates by recursively dividing the largest dimension of the two matrices to multiply into two to partition the main problem

into two sub-problems. In this sense, it can be cataloged in the family of 3D algorithms, a concept that we will explore in subsequent chapters. During a DFS step, all available processors will work together in solving each one of the sub-problems sequentially, while in a BFS step, all the processors perform the available sub-problems in parallel. CARMA can optimize communication, by taking n DFS steps until the size of each sub-problem fits in the local memory for each processor. When such a threshold is achieved, the algorithm will perform the required BFS steps to solve the problem.

Another branch of research focused on using Strassen-based algorithms to solve the distributed problem. Some algorithms focused on solving the subproblems using Strassen locally, while others chose the approach of generating the subproblems using the Strassen algorithm itself. The most notable algorithm in this family is called CAPS and was proposed by Ballard, Grey, and Demmel[2], who followed a similar approach to CARMA by utilizing a succession of DFS/BFS steps until the Strassen subproblems fit in local memory and then solve them in parallel. The algorithm also requires a special layout for the matrix data in a processor grid, which will be ideally considered base-7 to evenly distribute the 7 multiplications of the Stassen algorithm. This odd number might result in some inefficiencies with most of the CPU architectures.

Some authors Skejllum and Liao [21, 20], have suggested that implementing several algorithms is possible and indeed recommended for different matrix shapes. For example, an algorithm could start dividing the matrices using a 3D approach, and at a certain point switch back to a Strassen-based algorithm if the matrices are square enough. On Skejllum, approach they propose an adaptive algorithm that takes into account CPU grid topologies, memory constraints, and matrix dimensions to propose the optimal strategy for each step of the process.

Most of the algorithms focus on optimizing the communication reducing the amount of messages passed. As modeled by CAPS [2], the communication costs can be split into two: the bandwidth costs and the latency costs. Bandwidth costs can be assumed as the number of words shared, while latency costs are calculated based on the number of messages. In other words, communication costs not only depend on the amount of data a certain protocol passes across the network and reducing the number of times processors need to communicate amongst themselves.

Schatz et al. [29] analyze the existing distributed matrix multiplication al-

gorithms and present a summary of the required communication operations for performing distributed matrix most efficiently, reducing the bandwidth requirements. Their operations can be loosely mapped to MPI collective primitives, the most notable requirements being broadcast, all-gather, reduce-to-one, permute, and gather. Some of these primitives, are key elements to reduce the number of required messages and will be frequently mentioned in this work. The details of these collective operations can be reviewed in Appendix C.

We have now reviewed the most prominent algorithms from the literature, which typically utilize uniform grids of processors. In these setups, all processors equally share the tasks of division, communication, and multiplication within a distributed memory framework. However, only a few models adopt a primary-worker architecture similar to what we propose in this project. Drawing inspiration from the communication optimizations observed in various models, we will adapt these strategies to our context. However, it is important to note that our implementation will not strictly adhere to any single existing model. Instead, we will customize these approaches to better suit our unique system architecture.

In one of the notable examples from the literature that employs a primary-master setup, Pineu [14] introduces a distributed matrix multiplication model under the primary-worker paradigm. In this model, n workers are linked to a single primary node through a star network configuration, where each worker is equipped with a single communication port. Pineu employs two different greedy scheduling strategies: one that assigns jobs to the first available worker capable of executing the task, and another that factors in previous job allocations. While neither method achieves perfect optimization of computation time, they are primarily designed to minimize communication overhead, addressing one of the critical bottlenecks in the algorithm.

Pineu also reuses some of the concepts proposed by other algorithms and comes up with a maximum reuse algorithm for the calculation of a special case of matrix multiplication. Nevertheless, the principles of his algorithms are still useful for the problem at hand. In summary, the strategies followed by Pineu would be:

- Assume a fixed communication cost μ. For any update on C should be done at the worker when possible, since sending the data to the primary and back would incur a penalty of at least 2μ. This is similar to the approach or Reduce-to-one, mentioned by Schatz.

- On systems with limited buffers of memory m, communication efficiency can be achieved by keeping fixed one of the dimensions of the matrix multiplication while periodically sending rotations on the other one.

It is important to remark that the strategies followed by Pineau are not novel, but just a re-factorization of the strategies from other algorithms.

1.4 Raspberry Pi 5

We now turn our focus to the technological cornerstone of our work—the Raspberry Pi. An in-depth technical description of the Raspberry Pi 5 will illustrate how it can be utilized for a distributed framework for matrix multiplication efficiently.

The Raspberry Pi 5 is a small, affordable mini-computer. It's commonly used for enthusiast's projects involving automation, arts, media, and education. Raspberry Pi 5 is powered by a Broadcoam BCM2712 quad-core Arm Cortex A-76 CPU, which runs at a speed of 2.4 GHz. The CPU has 64 KB of L1 Data Cache for each core which is 4-way, set associative, and has 64-byte cache lines. Each core also has 512KB, which is 8-way and set associative, and 2 MB of shared L3 Cache. The CPU possesses instructions for SIMD instruction and floating-point operations, which suggests it might be a good candidate for performing vector multiplication operations.

ARM's memory model facilitates atomic instructions such as load-exclusive and store-exclusive, which work in tandem to achieve atomicity [1]. This implementation is critical for developing robust concurrent algorithms and maintaining system stability in multi-threaded applications. This will be useful in the implementation phase of the project when we discuss subsequent updates to the result matrix.

This mini-computer has a Gigabit Ethernet port, as well as 2xUSB2.0 and 2xUSB3.0 ports that can be used for communication and peripherals. The board also has the ability to support external peripherals through one PCI 2.0 interface, although this option requires additional hardware.

The Raspberry Pi 5 model provides a Micro-SD entry as its primary storage media. According to the datasheet, the Micro SD slot can perform at a maximum transfer speed of 104 MB/s. Considering this, and factoring in the theoretical speed of the Ethernet Connection (Full-duplex at 125 MB/s or 1 Gbps), any operation at the Micro-SD slot should be avoided.

Chapter 2

Proposed Architecture for the Communication Protocol

2.1 Requirements for the proposed system

In the last section, we reviewed many of the parallel matrix multiplication algorithms that have been published by researchers. In most of them, a group of processors work in conjunction to divide the task and then solve all the resulting sub-problems. The approach we will follow is substantially different. In our model, we are proposing to have a central node that will divide the sub-problems into the available Raspberry Nodes.

As mentioned in the Introduction chapter, the goal of this project is to provide the distributed matrix multiplication framework with the most optimized communication protocol possible. Based on the observations of [?, 6, 2] and others, the ideal system should be capable of supporting the bandwidth requirements of the protocol and help reduce the required messages to reduce the latency of the system.

To reduce the number of messages required, we will be taking various ideas from existing algorithms as inspiration. First, we need to provide local in-memory storage, both for input matrices A and B, as well as for C. We are proposing that for individual tasks, the input matrices and output matrices are referenced as pointers to pre-allocated memory. This approach would have several benefits. First, we avoid continuous allocation and freeing of memory. Second, it allows the algorithm to reuse the inputs from previous steps and keep contributing to an output matrix.

Normally, distributed matrix multiplication algorithms assume that the final distribution of C will be load-balanced across the processor's local memory. In our case, the result always needs to be communicated back to the central node upon competition. With the fixed memory approach, we ensure that the output matrix is only communicated once the local node has computed all the steps that would contribute to it.

This approach aligns with the computation flow of CARMA[6], where small dimensions across several DFS steps could be kept in memory at the Raspberry Pi and reused for future individual BFS steps. Furthermore, fixed dimensions in CARMA only need to be sent once to the Raspberry node. Each processor in the node can still take parallel jobs and all reference the same input matrix at the same time, using atomic operations.

Finally, most discussed protocols require broadcast communications, often referred to as all-gather in MPI-related literature [6, 29]. In our model, it is the central node that holds the initial data so it is the only node that needs to broadcast messages. Although TCP protocol fundamentally supports only unicast transmissions [8], we can provide equivalent communication to broadcast by marshaling data once and then using the buffer for multiple writes to all required clients.

2.2 Evaluation of existing communication protocols

In the early stages of our project, we analyzed the performance and capabilities of three options to implement to communicate our nodes. Since part the focus os this work was to optimize the communication decisions, this is a critical point of the process.

When discussing parallel computing, MPI is one of the existing standards for communicating multiple processes in parallel tasks. Most of the literature we reviewed had examples and benchmarks using the MPI protocol to pass information across processors. MPI enables processes to pass messages across pre-established TCP connections by building a full mesh of P2P connections between them [17]. MPI also has multiple communication primitives that align to the communication requirements exposed by [29], with support for scatter, broadcast, all-reduce, and reduce-to-one operations.

However, we encountered significant challenges in aligning MPI structures

with the primary/workers concept of our program. A primary limitation was facilitating MPI communication between the central node and the Raspberry Pi cluster, triggered by differing operating systems and libraries. Therefore, we decided to not use it for our implementation early on in the process.

After realizing that the definition of our problem involved invoking processes to run on a remote machine repeatedly, we considered that deploying an RPC protocol could also serve our purpose. RPC protocols are used by other frameworks that perform distributed computation using a primary/worker architecture. For example, Apache Spark framework uses netty RPC protocol to exchange tasks and information across workers[18, 30].

We started by testing GRPC, an open-source RPC protocol developed by Google. It is easy to compile and integrate into projects and provides strong type management, and streaming capabilities that align with our goal. However, the strong typing enforced by protobufs -gRPC's serialization protocol- presented challenges in handling large datasets used in our implementation. This limitation is acknowledged in the protocol's documentation [23]. Specifically, protobufs lack a native mechanism for transmitting streams of bytes, and the requirement to send streams of integers or other numeric values significantly burdened the CPU. Our initial tests with synchronous communications via GRPC resulted in response times of hundreds of milliseconds for a single matrix multiplication request. Consequently, we determined that GRPC was not ideally suited for our framework.

We then explored the option of designing our own RPC protocol, building over the base of a TCP socket. This option gave us the most flexibility and enabled us to build a protocol that was tailored to our needs.

2.3 Proposed architecture review

In the current section, we will describe the design considerations that we took into account for our solution. We engineered the required communication protocols based on the requirements that distributed algorithms have in the literature, and took into account the capabilities and limitations of the Raspberry Pi 5.

When considering the communication options, the Raspberry Pi has few options. USB ports could be used for communication, but USB communication scales poorly when communicating with different devices due to its device number

limitation (maximum 127 devices) and the strict requirement for it to work in Host/Device mode [16]. Adding an Ethernet over USB adapter provided no further advantage over the existing Ethernet connection, as it would still be bounded by the L2 protocol speeds. We briefly reviewed the options of integrating an external peripheral through Raspberry's capability of using PCI Express. However, there were no affordable options to integrate connectivity.

Therefore, we decided to provide the connectivity through the existing GigabitEthernet connection on the Raspberry Pi. Even if faster connectivity options become available either through a peripheral or a new Raspberry model, communication protocols based on the TCP/IP stack would be portable for other physical interface options available in the future.

For our architecture, we are considering that the system will have one or several primary nodes. These nodes could be a user's personal computer, or a dedicated server orchestrating the cluster. The primary node would have all the data at the beginning of the algorithm execution and would need to receive the results to pass it to the user or to the upper layers of computation that require them. For simplicity on our problem, we are assuming the central node would have greater computing power and more disposable bandwidth than than an individual Raspberry Pi. We are also operating under the assumption that the central node will be in charge of dividing the data into suitable work for the workers as discussed in Chapter 1.

In most algorithms in the literature, it is assumed that all the processors have equal access to the multiplication data and therefore need to collaborate using a full-mesh network connectivity [2, 6]. In our case, the Central Node will have all the information available in its memory or storage, allowing it to be the one that sends the data to every worker to achieve the optimal distribution. So it is sufficient to have a logical star network, where all computing nodes just establish a connection to the central node.

Once jobs are assigned to a Raspberry Pi, we can fully utilize all four cores to execute the tasks, configuring the system as a producer-consumer problem. In this setup, a continuous stream of tasks populates a queue, and each CPU core periodically checks for new jobs, selecting and executing one as soon as it becomes available. This greedy approach, also suggested by [14], ensures minimal idle time by allowing workers to take jobs on demand from the primary node.

Chapter 3

Implementation of the Communication and Scheduling Framework

Since we decided to implement an RPC protocol from the ground up, we had to make several design decisions that influenced the communication protocol. In this section, we will explain the considerations taken for the design and provide comments on the overall impact of the results.

In the first two subsections, we will discuss the strategies taken to make the communication protocol optimal. First, we will discuss the strategies taken to avoid blocking communication for the communication protocol by using an event-based approach instead of continuous polling. After that, we will explore the buffering algorithms and settings we implemented on the TCP socket to make it as frugal as possible. Furthermore, we will also review the marshaling and unmarshalling strategies that the protocol follows.

In the next two subsections, we will provide an overview of the job scheduling system as well as the mechanisms that we used to handle the matrices in memory. We will also review how these parts of the system interact with the communication protocol.

3.1 Optimizations implemented for the TCP Socket

By default, TCP sockets are configured in blocking mode. This forces the user context to continuously poll all the file descriptors involved in a server continuously. Therefore, all read and write operations will block all other operations in their thread until there is an event on the socket's file descriptor. With this approach,

the complexity of polling for network I/O would scale to $O(n)$. Such an approach is sufficient for simple applications, but it would be unsuitable for a real-time producer-consumer system like the one we were trying to build.

Most operating systems provide mechanisms to provide updates on file descriptors. The application context then only needs to scan for ready file descriptors, whether they are ready to write, read, or accept a new connection. On Linux and other Unix-based systems, this is provided by epoll, a Linux system kernel call that only requires monitoring events for network file descriptors when they are ready. The complexity of such an approach is $O(1)$ in the best case.

To simplify the operation of epoll primitives there are several event libraries built around it. The most two famous open-source libraries are libev and libuv. They both provide a wrapper around epoll that allows you to register callbacks for file descriptor events. Furthermore, both libraries have a similar architecture where events are triggered through an event loop that continuously monitors the epoll events.

Libev can be considered a lower-level library since it only provides an abstraction for epoll events and allows you to register callbacks when they originate. Libuv has support for additional higher-level functions. We evaluated implementing both libraries initially; however, we decided to use Libuv as it is easier to implement, has a great support community, and has better documentation [3, 12]. We also participated in discussion forums when we faced issues, with overall good response from the community. [22].

In our RPC project, we configured the system to fire callbacks in response to epoll events such as read, connect, and finish write. Additionally, we made use of asynchronous events, a feature supported by libuv, but also present in libev and other event-driven libraries. These async events allow us to monitor and manage a variety of events not related to I/O, but registering callbacks to them in the same event loop. In our case, we created queues for the messages that needed to be written out and monitored them so that every time something is queued to it, it fires a write action. This ensured continuous, yet thread-safe writing to the network. Further specifics on the use and advantages of async events will be explored in subsequent chapters.

3.2 Marshalling the information for our protocol

To efficiently exchange information between primary and secondary nodes through our RPC protocol, we developed a marshaling strategy centered around reusability, minimal bandwidth usage, and streamlined network operations, as outlined in our requirements section.

We devised three types of messages for our protocol: store messages and job messages sent from the central node to the workers, and result messages sent back from the workers to the central node. Store messages transmit matrix data and provide information to the worker nodes on how to store it in memory and reference the matrix being sent. These messages are designed to carry multiple matrices in one go, with a fixed-length header that records the number of matrices and the total message size. The variable-sized header depends on the number of matrices passed in a single message, as it includes dimensions and an ID for each matrix. See Figure 3.1 for a detailed explanation of the header fields.

The protocol's overhead for matrix communication is minimal, at 24 bytes per matrix. When including TCP and IP header overheads, the total bandwidth overhead remains at a consistent 2.8%, as calculated with a standard MTU of 1500 bytes, which is the standard for LAN networks and the Internet.

The execute-job message will then pass an instruction to be performed on a pair of matrices. Similarly to store messages, multiple instructions can be grouped in a single message. So the job messages would also have fixed and variable length portions. As for any other message in our protocol, each instruction starts with the size of the instruction messages. We have set the individual execute-job messages to be fixed to a length of 24 Bytes. This fixed size facilitates systematic parsing and ensures that instructions are clear and efficiently managed.

The rest of the job message is compromised by the job ID, a number that will also be used to identify the job for future aggregation. It is followed by task instruction, an integer parsed using enumerators on the worker node. This instruction can be any elemental operations on matrices: multiply two matrices, add two matrices, or subtract the two matrices. However, it could be extended to support other matrix operations if required. To determine the A and B input matrices, the message references them by passing the ID used in the store message. The execute-job messages also indicate the number of job aggregations that need to be performed before sending information back to the primary node, through

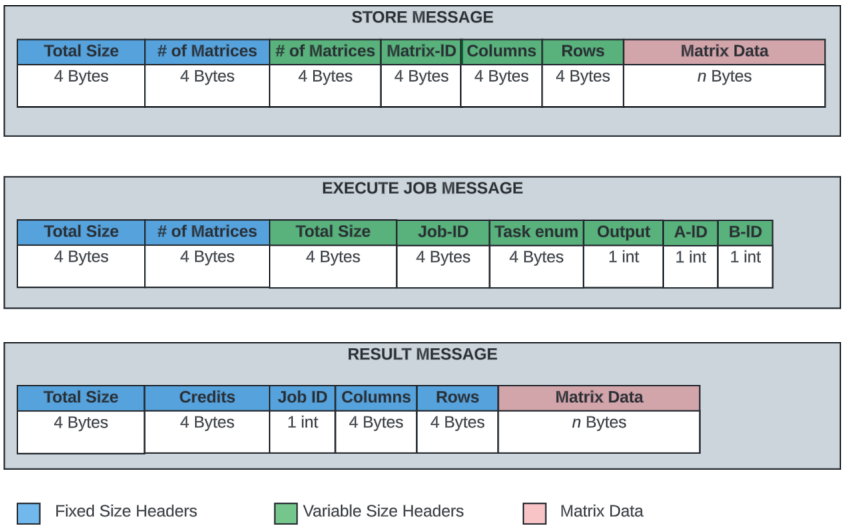

Figure 3.1: Review of the Headers of our protocol

the use of an output field. Details on these headers can also be found in Figure 3.1.

The execute job messages are smaller than the TCP/IP headers and we would need up to 50 instructions to fully utilize the MTU. Therefore, we are aggregating them at the tail of store messages. This way we also maintain the overhead caused by TCP/IP close to 2.8% by utilizing the full MTU every time we can.

In line with the objectives outlined at the beginning of this section, the worker nodes need to communicate only one type of message back to the primary node: the result message. This message is similar to the store message but with two key differences. First, the result message contains only a single matrix, rather than an arbitrary number, to ensure that results are queued immediately as they become available, minimizing idle time on the Raspberry Pi. Consequently, the result message always has a fixed header. Second, the result message also encodes the worker's available credits—a concept that will be discussed in more detail in subsequent sections.

3.3 Memory allocation for jobs and matrices

In the past subsection, we reviewed the messages that each worker node will receive from the primary node. For its subsequent computation, each message is assigned to a data structure contained in memory. We will see that intuitively, the information received for the jobs and results can be structured as queues and

the matrix storage can be structured as an array of matrices.

When a store message is received, the information of the matrix is stored on a pre-allocated memory buffer for its ID. This intuitively conforms to an array. All matrices are assigned the equivalent of 1/12 of the L3 Cache, rounded up to be 167 KB. This is to ensure that, even in the worst case, all cores still have the A, B, and C Matrices in the L3 cache. However, this number would still require more validation.

In our test cases with naive n^3 algorithms, the best performance is achieved when the matrices are constrained to the L1 cache, but the work performed by other students in our research group suggests that we could operate on bigger matrices. This is further discussed in the Future Work section.

To recall the stored matrices for execution, the job structure just needs to use the pointer allocated to each matrix. Such pointers can be easily retrieved using the ID of the matrix as we stored the matrix in the array position of its ID. Result matrices are also stored in this array structure, so they can be referenced as well as inputs for future jobs. This aligns with the concept of job aggregation that we established as one of our requirements. In case the central device needs to reuse one of the IDs, it just needs to schedule a store message before the job message that references the ID.

In our system, the two input matrices A and B for any task are stored in an array, each uniquely identified by an ID that corresponds to its position. To facilitate the management of this array, we introduced divisions we named as blocks. Blocks are essentially designated regions within the array. These blocks serve two key functions in our architecture: First, they act as a reference to the central node that, once the aggregation result is ready and sent back with its corresponding credits replenished, it is safe to write new information to that specific block if needed. This approach ensures efficient reuse of matrix data that has already been processed. Second, the central node's calling API simplifies job creation by only needing to reference the block and the ID. This allows the system to perform the necessary calculations to consistently and accurately refer to the same matrix.

When a job instruction is received, its information is parsed and copied into a structure. This structure will follow the job throughout its lifespan since we get it from the primary, and until it is written back or aggregated. These structures are the same size as a job message, so they just require the allocation and freeing of

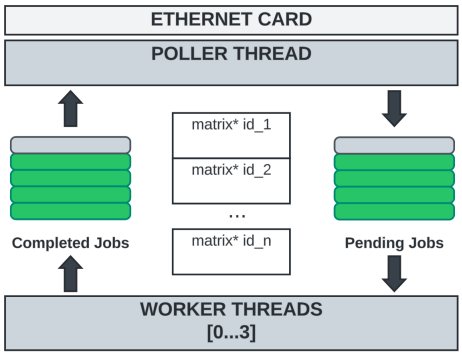

Figure 3.2: Thread interaction on the Raspberry Pi Node

24 bytes at a time. The system will then put the recently created job structure into a queue, which we will refer to as the Pending Jobs queue. This queue will then be processed by the worker thread architecture which will be discussed in the next section.

Once the task is executed, this structure will be freed out of memory if the result needs to be aggregated by a subsequent job. If the result needs to be written back to the primary the same structure will be enqueued to another queue that we will refer to as the Completed Jobs queue. Each time a worker thread puts a job on the Completed Jobs queue, it will also schedule a write operation on the communication thread. Since the job has a pointer to the result, the communication thread only needs to retrieve it from memory and write it to the TCP socket. The Completed Jobs queue is, therefore, a queue of pointers to be written out. As long as there are job structures in the queue, the communication thread will try to write them back.

In summary, the job that the communication engine in our system does then consists in:

- Parsing the jobs received from the central node and queueing them on the Pending Jobs queue for execution.
- Dequeuing the jobs from the Completed Jobs queue and writing the results back to the customer. The results are stored in the array of matrices.
- Storing the matrices received from the central node in an array of matrices.

See the figure below, which illustrates the described architecture for the communication protocol graphically.

3.4 Thread allocation and job scheduling

This subsection will explore the greedy algorithm implemented for our job distribution inside the Raspberry Pi. This algorithm can also be modeled after a producer-consumer problem, where the network layer produces many job instructions that then need to be executed by the Raspberry Pi. Our network interface takes the role of a producer, which then handles tasks to a pool of threads at the Raspberry Pi taking the role of consumers. We explain how we use mutexes and condition variables to synchronize multiple threads and pass messages across threads using queue structures. Such an approach takes inspiration from the buffering and synchronization proposed by Dijkstra [7] and taken further by many examples in academia and industry [13, 28].

We decided that to utilize the available resources in the Raspberry Pi fully, we had to spin multiple threads so that all cores could be used for the distributed algorithm. The Raspberry Pi has 4 available cores. Subsequently, we will refer to them as CPU[0:3]. We used pthread library for all the thread primitives. We pin each thread to a CPU so that we can have more granular control over the system, and split the threads that would be used for communication from the threads that would be used for computation. In subsequent, we will name the communication threads Poller threads (as they are continuously ”polling” the network for more jobs”), and the computation threads Worker threads.

During the initial tests, we verified that our CPU utilization for the Poller threads in the worst case (when the Ethernet channel was fully utilized at 1 Gbps) was around 33%, and dropped below 12% when computation was involved. So, we decided that a poller thread and a worker thread could share a CPU. The resulting worker thread would be slower than the other worker threads, but our greedy algorithm proved to be handy here as it assigned substantially fewer tasks to this straggler thread. These results will be further discussed in the Benchmarking and Performance Testing chapter.

Initially, we spun a worker thread on each CPU and assigned one Poller thread to CPU3. However, we identified that CPU0 was constantly being interrupted by Kernel operations that were generated by the TCP socket. As we didn’t have more available time to identify any optimization opportunities for these interruptions, we assumed them to be part of the penalty imposed by the poller thread on the worker it shared the CPU with. To concentrate all the communication penalties

in one CPU, we moved the poller thread to CPU0. More details on this profiling work can be found in the Appendix.

From the previous section, we can recall that the worker threads receive new jobs in the Pending Jobs queue and schedule the sending of results through the Completed Jobs queue. Each worker thread then needs to periodically read and write to those queues. To provide synchronization and thread safeness to those queues, they both have a mutex and a condition variable.

Each worker thread is put in a while loop after it has been initialized. The first thing it does on the loop is to acquire the mutex on Pending Jobs and try to pop a job. If there are no jobs, it will wait on the condition variable. From the poller thread, an inverse process is followed. Once the job is parsed, the mutex is acquired and the condition variable is signaled. This awakes idle worker threads if there are any. However, the waiting time can be fully avoided if there is a buffering of jobs at the Poller thread. Also, note that the mutex is only kept while the job is dequeued. Therefore, the synchronization time should be minimal.

After a worker node computes the task, it verifies in the job structure if the task requires to be written back to the central node or just stored for future aggregation on another task. If the result is ready and does not require aggregation, the same job structure is enqueued for writing.

Thanks to the queueing strategy that we are following, the central node can still send jobs even if all the worker threads in a Raspberry Pi Node are busy. In our experimentation, we also discovered that the CPU is only fully utilized whenever the jobs are queued asynchronously. If the jobs are sent synchronously, there is inevitably a waiting time due to the effect of network latency and unmarshalling times. In this model, the central node just needs to know that the Raspberry Pi still has space in its allocated memory to accommodate the job information and the matrices.

To signal the number of available slots in the queue buffer, we borrowed the concept of Credit-based flow control. It was first implemented to moderate network congestion on communication networks, with the first examples proposed by Pouzin and Louis [26] and by HT. Kung [19], but it has been implemented widely for other systems that require back pressure on a sending and receiving mechanism. There are even examples of the system being utilized for providing back pressure on the pipelining of deep learning jobs by [25], as well as in other distributed computing frameworks such as Apache Flink [9]. Credit-based flow

control offers significant advantages, such as preventing receiver buffer overflow in theory and dynamically adjusting to keep the receiver consistently busy. However, despite these benefits, there are practical challenges to consider, like the potential issue of lost credits if a job is never executed for an unrelated issue.

In our work, we model a credit to be equivalent to a Matrix Multiplication job. We decided to do this since multiplication tasks would be the core of our application, as well as the most computing-intensive task. As we reviewed in the past section, credits are encoded as part of the result operation. If a single multiplication is performed and the result is immediately communicated, it will advertise 1 credit back to the central node. If several multiplication operations are consolidated by adding them, the result will encode back the number of multiplications that were added.

As we stated in our goals, we intended to utilize all the CPUs available in the Raspberry Pi to their full capacity. We discovered that two parameters were key to achieving complete utilization: the maximum size of the queue and the refresh threshold, which is the minimum credits the central node needs to receive back before attempting to refill the buffer. We determined the optimal values for these settings empirically and found a sweet spot by setting the buffer size to 40 Jobs and refreshing it every time the central node receives at least 4 credits. We will review the empirical testing that we followed to get to these settings in Chapter 4.

3.5 Central server architecture and communication API

While the majority of our job was done on the Raspberry PI side, there are valuable contributions to this project that were made to the design of the central server. During this section, we will describe the architecture of the Central server unit, the API that was developed for communication with Raspberry Pi nodes, and a brief commentary on the matrix subdivision strategies for large matrices.

We initially developed the communication library for the central node using a synchronous approach, which involved individual calls to remote procedures only when a previous task had been completed. However, this method proved challenging to scale, as it did not allow for full CPU utilization across the threads for the Raspberry Pis. In response to these limitations, we transitioned to an asynchronous design. This change significantly enhanced scalability and allowed for more effective utilization of CPU resources.

Load balance was a crucial aspect of our system's design, ensuring that all computing nodes received an equal amount of tasks and that network resources were evenly distributed amongst remote nodes. To achieve this, we adapted the queue principle from our client design to the central node, utilizing a feature provided by our event loop library known as async events.

Async events allow for the introduction of new flags into the event loop, triggering specific actions each time they occur. Similar to other events in our loop, these are executed only once per loop iteration, meaning any calls made between actual executions are coalesced. For our purposes, we established a message queue for each node in the network and tied an async event to every message queue. The queue does not have any data allocated on it, but it is a queue of pointers making it lightweight. This also helps us comply with the requirement of broadcasting as different queues for different working nodes can reference the same pointer in memory, effectively sending the same message.

Each time an async event is triggered, it writes a single message from its assigned queue to the corresponding worker node. If there are remaining messages in the queue, the system queues another async event. This setup ensures that if other nodes have pending messages, their callbacks are processed before a second message from the same node's queue is handled, maintaining fairness and efficiency. This mechanism ensures the communication thread is constantly engaged in sending messages, while simultaneously managing all nodes fairly. By balancing the load in this manner, we optimize both the throughput and responsiveness of the network.

In summary, the API for sending traffic from the Central Node to any Raspberry Pi the process dividing the tasks just needs to enqueue a message to the queue that is tied to each server which fires the async call related to it. This process could tie with any engine doing the division of labor providing an abstraction layer to the communication tasks.

To perform distributed matrix multiplication, the input matrices need to be divided into smaller tasks that usually conform to basic matrix operations such as multiplication, addition, or subtraction, but applied to smaller matrices. Although the optimal division of the input matrices is not the primary focus of our study, we explored this issue to ensure we had appropriate tasks to test our system. There is extensive literature on matrix division algorithms; our approach involved implementing a light version of one such algorithm. We opted for a 3D division of

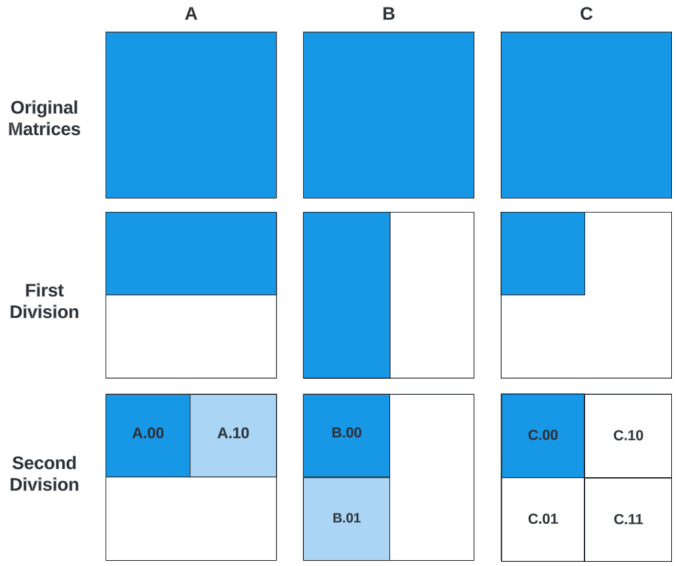

Figure 3.3: Graphical representation of a 3D matrix division algorithm. In the Second Division, additional contributing blocks to the same block are shown in lighter blue, showing their cumulative impact on the final computation.

a matrix to generate the jobs.

When multiplying two matrices, their dimensions must conform to the form $m \times k$ and $k \times n$, where A has dimensions $m \times k$ and B has dimensions $k \times n$. To obtain a pair of submatrices suitable for a job, the system divides one of the dimensions until the resulting submatrices fit within the local memory. This process, known as 3D division, involves recursively dividing the larger of the dimensions—m, n, or k—to produce the required matrices. The jobs resulting from multiplying these submatrices contribute to specific blocks of the result matrix C. However, these are not the only contributions to each block; other jobs also affect the same blocks and must be aggregated to obtain the final result. In Figure 3.3, we graphically illustrate a simplified version of the behavior of a 3D division algorithm applied to a pair of matrices.

3.6 Bandwidth optimization techniques

Previously, we explored strategies to fully utilize computing resources, including maximizing network interface bandwidth and evenly distributing jobs across all CPUs in the Raspberry Pi. This section focuses on optimizing the message requirements of our protocol for distributed matrix multiplication, aiming to

reduce bandwidth consumption while maintaining efficiency.

The naive approach to distributed matrix multiplication would be sending sub-matrices to a Raspberry Pi node, processing them, and returning the results to the central node. This is highly inefficient from a bandwidth perspective. With just three Raspberry Pis connected to a central node with a 1 Gbps interface, which corresponds to our testing setup.

Therefore, the first optimization we propose is related to job aggregation. For all distributed algorithms based on the n^3 algorithm, aggregation occurs via matrix addition of interim results, but not concatenating final results as this does not provide any bandwidth benefits. Additionally, having the central node receive final results as soon as they are available offers significant advantages, particularly in fault tolerance. For instance, if a Raspberry Pi node fails, the central node only needs to request the missing tasks for the incomplete jobs, thereby streamlining recovery and ensuring data integrity.

Consider the scenario depicted in Figure 3.3: instead of separately processing $A.00 \times B.00$ and $A.10 \times B.01$ for $C.00$, these operations can be aggregated at a single worker node. This method, similar to the reduce, gather, and scatter operations discussed in [29], parallels the MapReduce framework [?].

Moreover, since $A.00$ also contributes to $C.10$ through $A.00 \times B.10$, it is efficient to compute this at the same node, utilizing local in-memory storage for input matrices to minimize redundant data transfers. This not only conserves bandwidth but also boosts computational efficiency by reducing data requests.

Managing which worker node retains specific matrices was challenging, so we opted for sending larger data blocks that could be then divided in the Raspberry Pi. We would still send L1-Cache-sized tasks but would send them in a way that they could then be aggregated and their input matrices reused. We crafted two algorithms for this: the Row aggregation option where we sent two vectors of size $nL1$, that when multiplied will produce a single block result. Secondly, we crafted a Square Aggregation approach, which is similar but aggregating and reusing a square matrix of size $nL1 \times nL1$. Both strategies only require the final results to be transmitted, significantly reducing network load.

For our Row Aggregation method, the output would be N:1 of the original input achieving a reduction in the messages that the worker needs to send back to the central location as it provides aggregation. For the Square Aggregation, these same benefits are conserved, but due to the reuse of the pre-stored vectors,

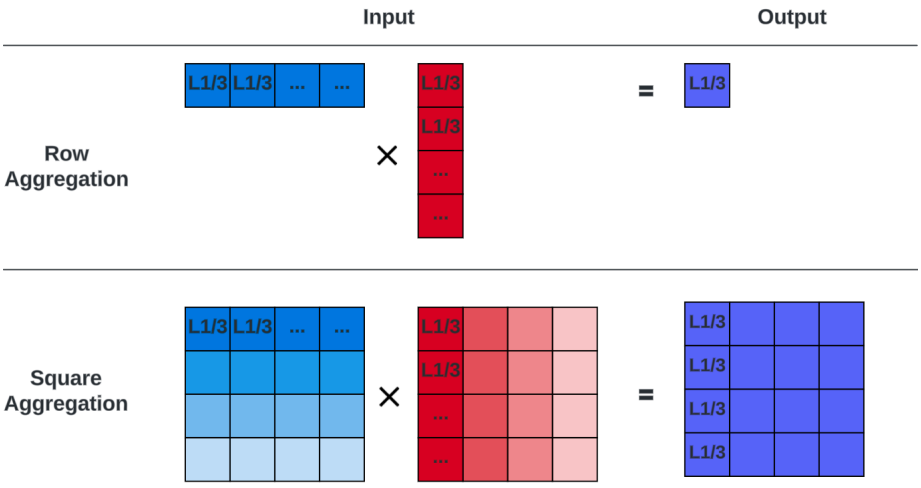

Figure 3.4: Graphical representation of our Row aggregation and Square aggregation approaches for input reuse. Note how all individual tasks are still constrained to fit in L1 Cache

the ratio would now conform always to 2:1. While larger rows or squares enhance aggregation benefits, they also require more processing before a result can be sent, which introduces latency penalties.

These methods draw from established literature and could be viewed as sending a group of DFS steps to a worker node, later decomposed into BFS steps as per [6]. Part of our exploratory work involved performing simulations of sub-matrices generated by this 3D approach. We observed that the succession of DFS jobs often follows the vector-by-vector or square-by-square patterns that we are suggesting here. You can find some examples of this simulation in Appendix G Figure 1.

Another form of viewing this would be considering this as variations of a poly-algorithm [20] where a big job is sent to the Raspberry Pi, and locally decomposed to smaller jobs using another algorithm from the one used by the central node

3.7 Methods employed for result aggregation

We explored two different approaches for result aggregation. The first one was to write the result matrices to the same array we used to store the input matrices and then reference them as inputs for other jobs. We had to implement synchronization mechanisms such as mutexes and condition variables on the output matrices to allow this since the aggregation job can only start adding the interim results after their tasks are finished.

The second one involved enabling atomic operations on the result matrices. Specifically, our algorithm employed the atomic_fetch_add instruction whenever the n^3 algorithm added to a matrix value, using relaxed memory order for further efficiency. This allows multiple tasks to contribute concurrently to the same output matrix without any synchronization primitive. For a comprehensive explanation of the n^3 algorithm's adaptation to include atomic operations, refer to Appendix A.

Each approach to managing concurrent matrix operations has its drawbacks. Using synchronization primitives can lead to inefficient CPU utilization; CPUs performing aggregation often waste time waiting for tasks to complete.

On the other hand, atomic operations increase the time spent on each multiplication task. We evaluated both approaches with a procedure that we will later explain in Section 4.3, which involves multiplying two matrices of size 11520×11520. Our observations indicate that atomic operations perform better than synchronization primitives, with performance gains ranging between 3% and 5%. The details of this comparison will be discussed in section 4.4.

Although this performance advantage might seem modest, atomic operations offer two further benefits: memory efficiency and lock-free logic. For instance, using synchronization mechanisms to add eight matrices requires multiple additions with interim results stored at each step. Specifically, this involves performing seven interim additions and storing these results for later aggregation. An alternative approach involves a function that acquires locks on all eight input matrices to add the results directly into a single matrix. However, this method necessitates waiting on eight locks and generates a substantial amount of boilerplate code to handle multiple aggregation sizes, making both options sub-optimal. Since atomic operations are lock-free, we take away any potential deadlock issues that could appear using mutexes or condition variables.

Chapter 4

Benchmarking and Performance Testing

4.1 Instrumentation techniques

The goal of this subsection is to provide a brief explanation of the measurements used in this chapter and an explanation of the measuring tools we used to obtain them. All the Python and shell scripts used to analyze and obtain our data are included in our software base under the instrumentation/ folder.

- For all measurements on bandwidth, we utilized Wireshark to perform packet captures. Then we used Wireshark's I/O statistic modules to export bytes counters for every 10 ms.
- For all measurements referring to per-thread CPU utilization, we crafted a shell script that retrieved the information from pdistat.
- For the duration of individual and aggregated tasks we measured wall-clock time using <sys/time.h> library.
- Any reference to flame graphs as well as records of CPU cycles spent on specific functions or in the code in general were obtained from perf. We utilized the scripts generated by Brend Gregg at Netflix [11] for all flamegraphs. Their work was fundamental for us to learn Linux profiling.

For the tests outlined in this section, we utilized a 2020 MacBook Air M1 with 8GB of RAM as the central node. In terms of network setup, we connected the cluster of Raspberry Pi nodes using an L2 switch, optimizing the communication infrastructure for reliable performance.

Our primary goal was to assess the efficiency of our framework in processing jobs; therefore, all matrices were pre-divided and task graphs were pre-computed before execution. This setup allows us to focus purely on the performance of the framework without the overhead of real-time job division. For the framework to be scalable in a production environment, the component responsible for feeding jobs must operate efficiently, ensuring a continuous flow of tasks in a producer-consumer manner.

4.2 Benchmarking the communication protocol

The following experiments are designed to measure the efficiency of our protocol's communication capabilities and the overhead the communication causes on compute and latency.

In our initial test, we aimed to evaluate our protocol's ability to saturate the Ethernet connection. To do this, we set up a callback in our RPC to function as an echo server, which replicated and returned a pair of input matrices. We expanded our Matrix Array to hold 1 GB of matrices in RAM and minimized the credit threshold for backpressure to one. We also increased the TCP buffers of both server and node to the maximum set for Unix systems by default, which is 6 MB. This adjustment meant that as soon as the central node received a matrix, it was immediately able to send another one back, allowing us to achieve an average speed of 96.9 MB/s for full-duplex communication.

Despite these results, we did not reach the full potential Ethernet capacity of 125 MB/s, which we attribute to the effects of our backpressure mechanism. Specifically, we noticed that at the beginning and end of the transfer, where backpressure was less influential, the protocol's performance peaked, fully utilizing the available bandwidth.

In appendix D, Figure 1 you can see a bandwidth graph for a test transferring 2 GB of data between the central node and a Raspberry Pi. We also tested extended sessions of up to 5 minutes of continuous traffic to assess stability, which yielded satisfactory outcomes. However, we do not present any graphical data from these experiments, as our capture script was unable to handle the extensive volume of information.

The remaining experiments conducted in this section focused on measuring the latency introduced by the TCP socket and the time required for the marshaling

and unmarshaling processes integral to our protocol. Specifically, these tests aimed to quantify the duration of these activities at the CPU and their overhead to the overall communication cycle and assess the efficiency of our data handling techniques.

To accurately analyze CPU time spent on various tasks we captured the call graph of our framework while it executed different task graphs. This approach helped us bypass the challenges of measuring wall clock time given the asynchronous nature of our framework. We utilized flame graphs to record and classify data, which allowed us to dissect the CPU usage of our algorithm as it performed individual matrix multiplication jobs and sent back results immediately.

During these tests, the four CPUs on the Raspberry Pi collectively used 94.88 seconds of CPU time for a task that appeared to take 23.307 seconds from the central node's perspective, reflecting an efficient 4:1 ratio. Of this time, communication protocols accounted for 3.48%, or roughly 3.30 seconds on a single CPU. The breakdown of activities includes 57.19% of the time spent on write operations (sending information to the central node), 31.12% on read operations (receiving information), and 11.50% on managing event loop overhead, such as running epoll.

Category	Time (seconds)	Percentage of Total Communication Time (%)
Total Communication Time	3.305	100%
Write Operations	1.89	57.19%
Read Operations	1.035	31.32%
Event Loop Operations	0.38	11.50%

Table 4.1: Consolidated Breakdown of Communication Protocol (100%), calculated as a percentage of execution with no interim storage and no job aggregation. A

The complete information on CPU utilization for different process calls on the communication protocol can be found in Appendix F.

4.3 Tests with raw Matrix Multiplication

In these experiments, we model the time required for matrix multiplication on a Raspberry Pi using the n^3 algorithm and utilize these measurements to determine

the optimal task size. Additionally, we assess CPU utilization efficiency and the impact of communication overhead on nodes performing continuous tasks. Notably, our analysis includes the load balancing across worker threads, with a specific focus on the delays introduced by a straggler worker thread arising from shared CPU resources between a Poller thread and a Worker thread.

In Chapter 3, we detailed our methodology for dividing a large matrix into individual jobs, which were then processed by worker nodes. For all experiments in this section, we multiplied two matrices of size 11520×11520 and experimented with various job sizes. Importantly, we opted not to perform any aggregation post-multiplication in this section of our testing but instead sent all partial results immediately back to the central node. That way, we will be able to compare the approaches afterward.

In past chapters we also reviewed existing literature, which suggested that the size of each job should ideally match the local memory capacity of the assigned CPU, targeting utilization of either L1 or L2 Cache—levels of cache memory dedicated to each CPU. Experimental results, presented in Table 4.1, validate our approach: utilizing chunk sizes that fully capitalize on L1 Cache yielded the most efficient outcomes in raw multiplication. Thus, we will continue using this job size moving forward, unless otherwise noted. It is important to note that for square matrix jobs, the size of matrices A, B, and C is $1/3$ of the overall job size. You can see a summary of these experiments in Table 4.1.

Number of Tasks	Job Size (KB)	Cache Level	Average Time (ms)
53,361	32.00	Fits in L1	28,514
23,716	62.45	Fits in L1	23,307
13,456	117.19	Fits in L2	30,861
13,456	263.67	Fits in L2	46,518
3,136	511.89	Fits in L2	66,489

Table 4.2: Summary of Experimental Results for different job sizes on a distributed multiplication (Chunk Size in KB). Average of 50 iterations

We also recorded the total wall-clock time expended on each task by every worker thread to assess the performance under the n^3 algorithm. Worker Threads

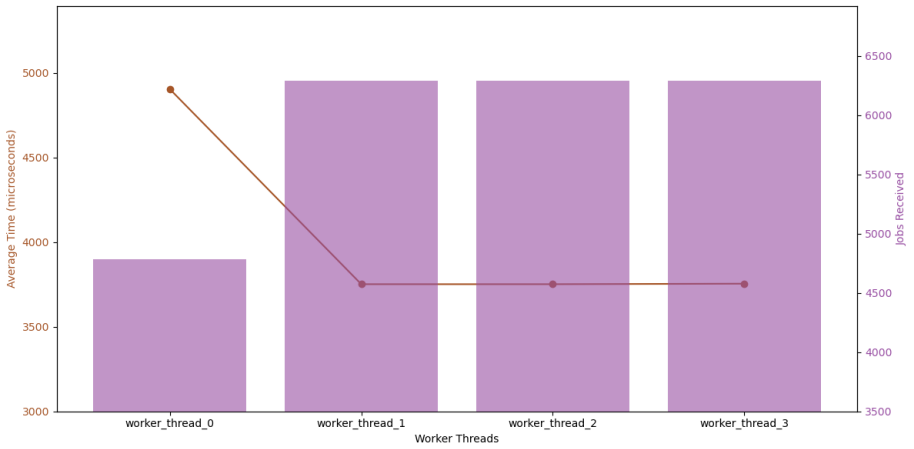

Figure 4.1: Job execution time vs Job allocation per thread. Average of 50 iterations of a 11520×11520 matrix multiplication divided in L1-sized Jobs

1, 2, and 3 demonstrated similar computation times, averaging $3,752\mu$secs for jobs fitting in the L1 cache. In contrast, Worker Thread 0, which shares a CPU with the Poller thread, exhibited longer computation times, averaging $4,905\mu$secs. From the central node's perspective, the average completion time for each job was $4,040\mu$secs.

Extrapolated, means that each CPU running exclusively matrix multiplication tasks could theoretically perform up to 266 operations per second for matrices fitting within the L1 cache. In contrast, the performance of the shared CPU drops to about 203 multiplications per second. This results in a weighted average of 247 matrix multiplications per second across all the CPUs.

Our greedy consumer algorithm effectively adjusted the workload distribution, allocating fewer tasks to the straggler CPU (Worker Thread 0) which handled $4,784$ jobs compared to approximately $6,290$ jobs received by the other threads. This distribution showed a uniform job allocation among the non-straggler threads. You can see these results displayed graphically in Figure 4.2.

As explained in past chapters, we had to find an optimal size for our job queue buffer and the threshold to refresh it with new jobs. We found that for L2-sized jobs, it was enough to queue 12 matrix multiplication jobs to fully utilize CPUs. This could be useful for future work in case other students in the research group find ways of making matrix multiplication more efficient on the Raspberry Pi.

For L1 jobs, which are the primary focus of this experimentation, we determined that an optimal queue length of 40 jobs strikes the best balance, with a refresh threshold set at 4 credits. Increasing the queue size beyond this point does not

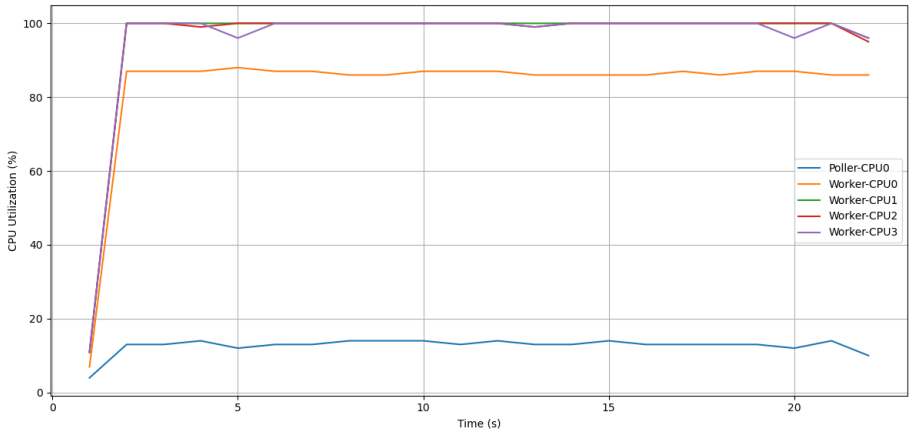

Figure 4.2: CPU utilization with a 40 Job queue. Note: WorkerThread 0 and Poller Thread share resources of the same CPU

enhance performance; instead, it contributes to increased latency as jobs wait longer for queue depletion. Figure 4.3 illustrates CPU utilization when the queue is set to this optimal size. For a comparative analysis, Appendix E includes CPU utilization graphs for queues holding just 8 jobs (Figure E.1) and 36 jobs (Figure E.2), demonstrating the efficiency gains at the optimal queue length.

Moreover, the wall-clock execution time shows significant improvement at the optimal queue size: while a queue of 8 jobs averages an execution time of 27.1 seconds, this is reduced to 23.3 seconds for a queue of 40 jobs, as observed in earlier tests. Additionally, in this configuration, the Poller thread's consumption drops to an average of 13.10% of a single CPU's capacity. Given that our system utilizes four CPUs, we confirm that communication processes account for approximately 3.28% of the total CPU time. This number is consistent with the results obtained from the flame graph profiling of communications exposed in the previous subsection.

Finally, after capturing and analyzing network traffic, we observed that with the current configuration, the central server was sending data at an average rate of 41.18 MB/s and receiving at 20.93 MB/s. Given that our Ethernet connection has a capacity limit of 125 MB/s for either direction, this configuration restricts scalability beyond three (3) Raspberry Pi units. Consequently, we decided not to include scaling-out tests for this configuration. Detailed bandwidth utilization for this specific case is documented in Appendix D, Figure 3.

4.4 Job aggregation and traffic optimization

As discussed previously, the goal of job aggregation is to reduce the bandwidth utilization on the network card. In the design section, we proposed that we could aggregate N number of jobs together before sending them, performing one or several operations of Reduce-to-one across the CPUs, as noted by [29].

We explored the use of atomic operations and synchronization primitives for job aggregation. Our comparative analysis, detailed in Table 4.3, shows atomic operations provide a performance advantage of between 3.35% and 4.71%, depending on the number of jobs aggregated. This modest improvement can be attributed to several factors. Notably, CPU usage significantly decreased when using mutexes, with worker threads operating at only 87.5% of their capacity due to waiting on other threads before performing aggregation. Further details on this are available in Appendix E.3. Although the actual addition operation is relatively inexpensive, taking about $50\mu secs$ for matrices that fit within the L1 cache, the main performance bottleneck was the CPU's idle time waiting on condition variables.

Aggregation	Average compute time Atomic Operations (ms)	Average compute time Synchronization (ms)	Diff %
Aggregate 4 jobs	26638 ms	27530 ms	3.35%
Aggregate 8 jobs	27002 ms	28210 ms	4.48%
Aggr. 16 jobs	27346 ms	28633 ms	4.71%

Table 4.3: Performance metrics for different aggregation methods.

In contrast, when using atomic operations for updating the result block, the performance of the matrix multiplication averaged $4307.0ms$. This included $4055.0ms$ for isolated worker threads and $5063ms$ for a worker thread sharing a CPU with a Poller thread. The global average time for matrix multiplication with atomic operations was $4307.0ms$, which is 6.61% slower compared to using the standard n^3 algorithm $4040.0ms$, presented in the previous section. Additionally, while only $n-1$ additions are needed for the aggregation of n jobs through successive adds, atomic updates impose a performance penalty on all n multiplication operations. The load balancing and the allocation of fewer jobs to the shared worker thread are preserved by the greedy algorithm as shown in Appendix Figure E.4.

Extrapolating as we did with the simple n^3 multiplication, an isolated worker

thread can perform approximately 246 matrix multiplications per second when applying atomic updates, while a worker thread sharing a CPU exhibits a reduced rate of about 197 multiplications per second. Consequently, the weighted average for our setup, when employing atomic updates, stands at 232 matrix multiplications per second.

We tested the two methods proposed in section 3, Row Aggregation and Square Aggregation, each with 4 and 8 L1-Cache-sized jobs. We recorded the traffic consumption for the network card for each one of these cases in 10 different iterations achieving consistent bandwidth ratios with the expected patrons. See in Table 4.3 the Bandwidth utilization and the expected ratio. Note how all our tests get close results to the expected ratio.

Aggregation Type	Central Node to RPI (MB/s)	RPI to Central Node (MB/s)	Expected Traffic Ratio
Row Aggregation, 4 Jobs	24.78	5.20	4:1
Row Aggregation, 8 Jobs	22.97	2.72	8:1
Square Aggregation, 4 Jobs	10.18	5.04	2:1
Square Aggregation, 8 Jobs	5.17	2.52	2:1

Table 4.4: Bandwidth and Expected Traffic Ratios for Different Aggregation Types, from and to the Central Node Server

Please note, that while the bandwidth utilization is less, this does not mean that the algorithm is less efficient or the worker Raspberry Pi performs less computation, but just that it is aggregating several results before sending them back to the central server and reusing existing inputs for calculation.

The analysis of the graphs reveals several observations. Square Aggregation exhibits a square-wave pattern, primarily because at the start of each round of DFS steps, the central node transmits rows and columns from matrices A and B to initiate calculations. Once all rows are received, only columns from B need updating, which halves the bandwidth requirement during this phase. In the scaling-out tests, we will observe this cancels out and the bandwidth utilization is more uniform. This smoother pattern results from the central node's strategy

of sequentially sending tasks to workers, allowing one worker to compute while another is being written to.

Also in the square aggregation scenario, there is a noticeable tail of traffic from workers back to the central node after all tasks have been dispatched and while awaiting results. Ideally, this period could be strategically used by the central node to initiate the next set of tasks, enhancing overall efficiency.

Both scenarios display an initial spike in bandwidth usage at the start of the tests, attributed to the time it takes for backpressure mechanisms to activate. In a continuous operation setting, such spikes would likely be mitigated.

4.5 Scale out testing

Our central server is designed to accommodate several nodes, and, in this way, we could achieve better results by scaling out. We observed that when performing the higher levels of aggregation and storage, the ones we described as block aggregation in previous chapters, we get better results.

We started by adding a second Raspberry, and our results show that it approximately halves the computation time, with the calculation time for the block storage test being around **51%**. This shows the system's potential for near-linear scalability as more worker nodes are added.

To make comparisons more easily, we introduced the performance improvement metric, expressed as a multiple of the time taken by a single server. This metric was calculated by comparing the time taken by multiple servers against the time taken by a single server. Specifically, for each server configuration (2, 3, 4, and 5 servers), we divided the time taken by 1 server by the time taken by the given number of servers. The calculation can be done with the following formula:

$$\text{Performance Improvement} = \frac{\text{Time with 1 Server}}{\text{Time with 2 Servers}} = \frac{25,341\,\text{ms}}{14,691\,\text{ms}} \approx 1.72$$

For example, in the aggregation of 4 jobs, performance improved by a factor of 1.72x when using 2 servers and reached 3.36x with 5 servers. When storage was involved, the improvements were even more substantial, with a 1.95x boost with 2 servers and up to 4.55x with 5 servers.

Our results show that the system scales effectively with additional servers, achieving significant reductions in processing time with up to three servers. However, improvements continue at a linear rate only in square aggregation scenarios.

Initially, we investigated potential network bottlenecks but found no issues via traffic captures.

Profiling the server side on macOS presented challenges due to system restrictions on setting thread affinity and obtaining detailed performance data. macOS employs Quality of Service (QoS) settings that dynamically manage core allocation to optimize efficiency, limiting manual core assignments [15]. That also results in threads being load-balanced across a group of cores according to the process priority.

Nevertheless, we were able to pinpoint the bottleneck on the server side. Using htop we could identify that the central node process was causing saturation in our assigned group of processes (4-7). This was particularly evident in Row aggregation cases, which handle more concurrent traffic as there is less optimization done. The average usage across our group of processors is presented in Table 4.4. The requirement for an optimized design of the central node will be stressed in the Future Work section.

Configuration	1 Server (ms)	2 Servers (ms)	3 Servers (ms)	4 Servers (ms)	5 Servers (ms)
Row aggr: 4L1 Jobs	25,341	14,691	9,772	8,416	7,531
Row aggr: 8L1 Jobs	26,638	14,234	9,628	7,587	7,176
Square aggr: 4x4L1 Jobs	27,908	14,278	9,738	7,517	6,127
Square aggr: 8x8L1 Jobs	27,817	14,234	9,719	7,373	6,094

Table 4.5: Performance of Job Aggregation for a Matrix Multiplication of two matrices with 11520×11520 dimensions with different configurations and aggregation strategies. Measurements are done with wall-clock time on the central node. Average of 50 iterations. Aggr. stands for aggregation

Configuration	2 Servers	3 Servers	4 Servers	5 Servers
Row aggr.: 4L1 Jobs	1.72	2.59	3.01	3.36
Row aggr.: 8L1 Jobs	1.87	2.77	3.51	3.71
Square aggr.: 4x4 L1 Jobs	1.95	2.87	3.71	4.55
Square aggr.: 8x8 L1 Jobs	1.95	2.86	3.77	4.56

Table 4.6: Performance Improvement Relative to 1 Server Configuration

Configuration	1 Server (%)	2 Servers (%)	3 Servers (%)	4 Servers (%)	5 Servers (%)
Row aggr.: 4L1	51.8	72.6	87.9	92.1	97.5
Row aggr.: 8L1	43.0	56.6	61.4	75.4	87.5
Square aggr.: 4x4	47.2	66.8	80.9	89.3	95.7
Square aggr.: 8x8	42.6	55.9	66.7	78.7	87.8

Table 4.7: Average CPU Utilization on Central Server. Average of 5 iterations. Aggr. stands for aggregation

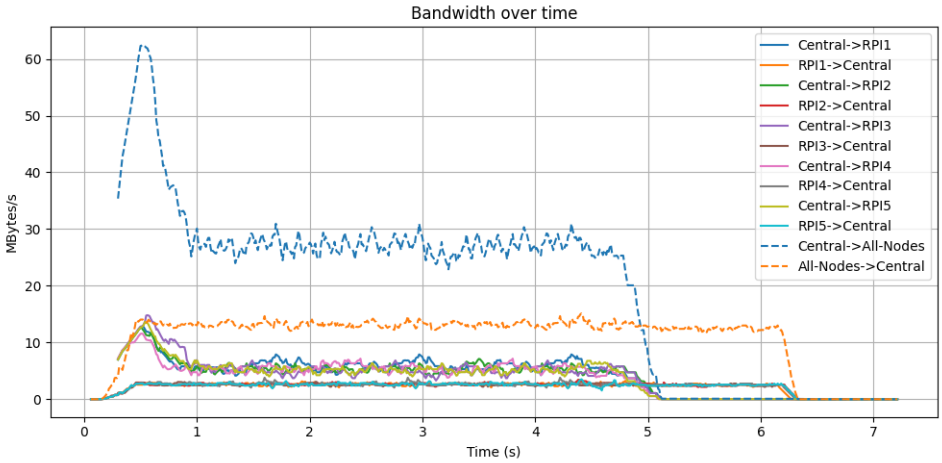

Figure 4.3: Bandwidth utilization for 5 Raspberry Pi Nodes. Square aggregation, 8 L1 jobs

We captured graphs of the bandwidth utilization to show that bandwidth utilization also scales linearly with the addition of nodes. We show an example in Figure 4.3, which is the case with Square aggregation of 8 jobs with 5 servers. In the configuration with a single computing node, this consumed 5.17 MB/s from Central to Worker, and 2.52 in the opposite direction. In this test, we saw average usage of 27.8 and 13.8 MB/s in each direction which shows that the bandwidth was also evenly distributed. Graphics from other scale-out tests can be seen in Appendix D.

Chapter 5

Conclusions

The results of this work, show that the combination of our consumer-producer framework and lightweight RPC protocol achieves a good scaling-out performance in the cases where we provided job aggregation and local storage for input matrices for subsequent re-use. Our best result was achieved when providing input reuse and job aggregation, and running the distributed computation over 5 servers. Under this setup, the distributed computation can perform this in 4.75 times less time, or a 475% increase in performance for the setup with 1 Raspberry Pi node calculating this.

We observe that job aggregation and in-memory storage of pass inputs submatrices allow us to reduce the bandwidth utilization required from the network. This provides us with two key elements for the algorithm. First, they let the bandwidth requirements for our central server scale. The first iterations we made required sending the inputs every time and retrieving all interim results, which consumed unnecessary bandwidth and limited the ability of our framework to scale. Furthermore, they consumed more CPU resources on the central node which caused a performance bottleneck.

The scaling out of our system switches the requirements of intensive computation locally at the central server and switches them to intensive requirements for communication that need to be maintained. It is important to remark that all these tests were performed with computing nodes connected to the same LAN switch, therefore there were no additional constraints on bandwidth or latency. We can argue that such is the adequate setup for a cluster like this to operate, with the central node close to the computing nodes. Having disaggregated nodes would decrease the available bandwidth and increase the latency, which would

make our scaling out less optimal.

The platform we designed is stable and customizable so that the user to control the number of servers and the nature of the task and aggregation for the system from the central node. The worker nodes similarly have configuration options that can be adjusted for further scalability and testing. All of this information can be reviewed on the attached code base, and the details are explained in the README file contained in the attached materials.

Making a critical and personal reflection on the work completed for my dissertation, I will now highlight the substantial challenges encountered in the low-level programming aspects of the project. Although I had experience in this realm, I must admit I was out of shape, and took some time for me to be efficient in this kind of programming again. Navigating the complexities of optimizing communication within a distributed system tested my technical boundaries and required a deep dive into system architecture and performance optimization. While these challenges undoubtedly shaped some aspects of the project's outcomes, they also provided a profound opportunity for professional growth.

5.1 Future Work

Several aspects of this project require further exploration and development. Our current findings are preliminary and represent initial insights into a complex problem tackled over just three months. Moving forward, we highlight the following key areas for future development:

- Optimization of the server-side code. Much effort was devoted to enhancing the scalability of worker nodes, yet the efficiency of the server-side code remains a bottleneck. Future work could involve increasing thread count, conducting advanced profiling to pinpoint performance bottlenecks, and considering alternative hardware for the primary computing node. Despite optimizations, managing coordination and communication with numerous nodes continues to tax the CPU of the central node significantly. A crucial task for future work involves comparing the performance of the central node when performing matrix multiplication locally versus using a distributed approach. This comparison will help determine if the overhead of managing distributed processes outweighs the potential benefits, and whether resources

dedicated to coordination might be more effectively used for computation. Furthermore, the project could reach a hybrid approach where the central node could compute when appropriate and delegate when needed.

- Dynamic Matrix and Job Division: While our system draws on established literature for job and matrix division strategies, it currently lacks the capability for real-time job division. Optimizing this feature is crucial for supporting our system's scalability under a continuous task stream model. Collaboration with other researchers at the University of Edinburgh, who are developing efficient algorithms, could lead to integrated solutions in subsequent phases. Our system pretends to present itself as plug-and-play for them to be integrated in future stages.

- Testing with different Matrix dimensions. Our research primarily focused on a single matrix size. as we consider that the scalability of the system could be addressed with it. However, the matrix dimensions are key for choosing the right job division algorithm. Therefore this work item is heavily tied to the previous point.

- Integrating faster local algorithms for matrix multiplication. Our local algorithm for matrix multiplication, the one that we use to multiply matrices on tasks assigned to the worker nodes, could be improved by integrating faster instruction sets like SIMD, leveraging the GPU, or taking advantage of sparse matrix multiplication algorithms, amongst other strategies. Other students in the research group are also working on this, similarly, our approach is also to design an open framework to make it easy to integrate new algorithms.

Bibliography

[1] ARM Developer. Memory System Implementation, 2023. Accessed: 2024-08-05.

[2] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. Annual ACM Symposium on Parallelism in Algorithms and Architectures, 02 2012.

[3] Andrey Bolonin. Gist: Helpful bash scripts and notes, 2023. Accessed: 2024-08-05.

[4] Lynn Elliot Cannon. A cellular computer to implement the kalman filter algorithm. PhD thesis, USA, 1969. AAI7010025.

[5] Kuan-Lin Chen, Jeremy Feinstein, and Brian Kutsop. Tiny data with raspberry pi's: An exploration of low-cost, low-energy mapreduce clusters.

[6] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. Proceedings - IEEE 27th International Parallel and Distributed Processing Symposium, IPDPS 2013, 10 2012.

[7] Edsger W. Dijkstra. Cooperating sequential processes. Technical Report EWD123, Technological University, Eindhoven, 1965. Section 4.1. Typical Uses of the General Semaphore.

[8] Gorry Fairhurst. Unicast, broadcast, and multicast, 2020. Accessed: 2024-08-05.

[9] Apache Flink. Apache Flink Documentation: Backpressure Monitoring, 2021. Accessed: 2024-08-05.

[10] Raspberry Pi Foundation. How to build a raspberry pi cluster, 2023. Accessed: 2024-08-05.

[11] Brendan Gregg. Linux profiling at netflix - scale 2015, 2015. Accessed: 2024-08-05.

[12] Eddie H. Gist: Xvfb init script, 2021. Accessed: 2024-08-05.

[13] Ralph C Hilzer Jr. Synchronization of the producer/consumer problem using semaphores, monitors, and the ada rendezvous. ACM SIGOPS Operating Systems Review, 26(3):31–39, 1992.

[14] Tehui Huang, Tao Li, Qiankun Dong, Kezhao Zhao, Wenjing Ma, and Yulu Yang. Communication-aware task scheduling algorithm for heterogeneous computing. International Journal of High Performance Computing and Networking, 10:298, 01 2017.

[15] Apple Inc. Dispatch Documentation, 2024. Accessed: 2024-08-05.

[16] Keil. USB Network, 2024. Accessed: 2024-08-05.

[17] Wes Kendall. Mpi tutorial introduction, 2024. Accessed: 2024-08-05.

[18] Bartosz Konieczny. Rpc in apache spark, 2018. Accessed: 2024-08-05.

[19] H. T. Kung. Credit-based flow control for atm networks: A proposal to the atm forum. In Proceedings of the ATM Forum, 1994. Proposal submitted to the ATM Forum.

[20] J. Li, A. Skjellum, and R. D. Falgout. A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies. Concurrency: Practice and Experience, 9(5):345–389, 1997.

[21] Xia Liao, Shengguo Li, Wei Yu, and Yutong Lu. Parallel matrix multiplication algorithms in supercomputing. In 2021 6th International Conference on Intelligent Computing and Signal Processing (ICSP), pages 1–4, 2021.

[22] libuv Contributors. Discussion #4457, 2024. Accessed: 2024-08-05.

[23] Google LLC. Techniques | Protocol Buffers Documentation, 2024. Accessed: 2024-08-05.

[24] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba. Parallel processing of matrix multiplication in a cpu and gpu heterogeneous environment. In Michel J. Daydé, José M. Laginha M. Palma, Alvaro L. G. A. Coutinho, Esther Pacitti, and João Correia Lopes, editors, VECPAR, volume 4395 of Lecture Notes in Computer Science, pages 305–318. Springer, 2006.

[25] OneFlow2020. The history of credit-based flow control (part 1), 2020. Accessed: 2024-08-05.

[26] Louis Pouzin and Louis. Methods, tools, and observations on flow control in packet-switched data networks. In Proceedings of the International Symposium on Flow Control in Computer Networks, 1981.

[27] Cristian Ramírez, Adrián Castelló, Héctor Martínez, and Enrique S. Quintana-Ortí. Performance analysis of matrix multiplication for deep learning on the edge. In Hartwig Anzt, Amanda Bienz, Piotr Luszczek, and Marc Baboulin, editors, High Performance Computing. ISC High Performance 2022 International Workshops, pages 65–76, Cham, 2022. Springer International Publishing.

[28] AM Abhishek Sai, Dinesh Reddy, Perumalla Raghavendra, G Yashwanth Kiran, and VR Rejeenth. Producer-consumer problem using thread pool. In 2022 3rd International Conference for Emerging Technology (INCET), pages 1–5. IEEE, 2022.

[29] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. Parallel matrix multiplication: A systematic journey. SIAM Journal on Scientific Computing, 38(6):C748–C781, 2016.

[30] Apache Spark. Configuration - Spark 1.5.1 Documentation: Networking, 2015. Accessed: 2024-08-05.

[31] Huaqing Zhang, Xiaolin Cheng, Hui Zang, and Dae Hoon Park. Compiler-level matrix multiplication optimization for deep learning, 09 2019.

Appendix A

n^3 algorithms, both simple and including atomic operations

Algorithm 1 Matrix Multiplication Using 1D Array

1: assert dimension and allocation checks passed ▷ Ensure matrices are allocated and dimensions are compatible
2: for $i \leftarrow 0$ to $left.col\_size - 1$ do
3: for $k \leftarrow 0$ to $left.row\_size - 1$ do
4: for $j \leftarrow 0$ to $right.row\_size - 1$ do
5: $result.matrix[i \times left.col\_size + j] \leftarrow result.matrix[i \times left.col\_size + j] + left.matrix[i \times left.col\_size + k] \times right.matrix[k \times left.col\_size + j]$
6: end for
7: end for
8: end for

Commentary:
This implementation of matrix multiplication using a 1D array is designed to maximize cache efficiency. The outermost loop iterates over the columns of the left matrix, while the innermost loop accesses consecutive elements of the result and right matrices.

By iterating over the k index before j, the code accesses the elements of the ‘left‘ matrix in a sequential manner, which is beneficial for cache hits due to spatial locality. This is because accessing elements that are close together in memory (which happens when iterating over k) reduces cache misses.

Additionally, because the innermost loop modifies elements of the ‘result‘ matrix that are also close in memory, the algorithm takes advantage of cache lines efficiently, reducing the frequency of cache evictions and reloads, thus improving overall performance.

Algorithm 2 Matrix Multiplication with Atomic Accumulation

```
1: assert dimension and allocation checks passed ▷ Ensure matrices are allocated and dimensions are compatible
2: for i ← 0 to left.col_size − 1 do
3:     for k ← 0 to left.row_size − 1 do
4:         for j ← 0 to right.row_size − 1 do
5:             mult_result ← left.matrix[i × left.col_size + k] × right.matrix[k × left.col_size + j]
6:             atomic_fetch_add_explicit(&result.matrix[i × left.col_size + j], mult_result, memory_order_relaxed)
7:         end for
8:     end for
9: end for
```

Commentary:
This implementation of matrix multiplication performs atomic increments on the result matrix to ensure thread-safe accumulation. The use of atomic operations, such as atomic_fetch_add_explicit, allows for the safe and concurrent updating of shared data without the need for explicit locks, which can be costly in terms of performance.

The choice of memory_order_relaxed ensures that the atomic operations are performed without enforcing any memory ordering constraints, which can lead to improved performance by allowing more flexible instruction reordering.

Appendix B

Explaining the libuv event loop

Algorithm 3 Event Loop Algorithm in libuv

```
Initialize
Initialize the event loop structure
Set up data structures for events, async events, and I/O
while true do
    Poll for Events
    Check for I/O events using platform-specific mechanisms
    Check for async events (e.g., deferred tasks)
    Handle Events
    for each pending I/O event do
        Execute associated callback function
    end for
    for each async event do
        Execute async callback function
    end for
    Update the Loop
    Update internal state of the event loop
    Manage newly registered or removed events
end while
```

Appendix C

Complete collective communications from Schatzet

Operation	Before				After			
Permute	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x_0	x_1	x_2	x_3	x_1	x_0	x_3	x_2
Broadcast	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x				x	x	x	x
Reduce(-to-one)	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$			
Scatter	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x_0				x_0			
	x_1					x_1		
	x_2						x_2	
	x_3							x_3
Gather	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x_0				x_0			
		x_1			x_1			
			x_2		x_2			
				x_3	x_3			
Allgather	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	x_0				x_0	x_0	x_0	x_0
		x_1			x_1	x_1	x_1	x_1
			x_2		x_2	x_2	x_2	x_2
				x_3	x_3	x_3	x_3	x_3
Reduce-scatter	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(3)}$	$\sum_j x_0^{(j)}$			
	$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(3)}$		$\sum_j x_1^{(j)}$		
	$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(3)}$			$\sum_j x_2^{(j)}$	
	$x_3^{(0)}$	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(3)}$				$\sum_j x_3^{(j)}$
Allreduce	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	$x^{(0)}$	$x^{(1)}$	$x^{(2)}$	$x^{(3)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$	$\sum_j x^{(j)}$
All-to-all	Node 0	Node 1	Node 2	Node 3	Node 0	Node 1	Node 2	Node 3
	$x_0^{(0)}$	$x_0^{(1)}$	$x_0^{(2)}$	$x_0^{(3)}$	$x_0^{(0)}$	$x_1^{(0)}$	$x_2^{(0)}$	$x_3^{(0)}$
	$x_1^{(0)}$	$x_1^{(1)}$	$x_1^{(2)}$	$x_1^{(3)}$	$x_0^{(1)}$	$x_1^{(1)}$	$x_2^{(1)}$	$x_3^{(1)}$
	$x_2^{(0)}$	$x_2^{(1)}$	$x_2^{(2)}$	$x_2^{(3)}$	$x_0^{(2)}$	$x_1^{(2)}$	$x_2^{(2)}$	$x_3^{(2)}$
	$x_3^{(0)}$	$x_3^{(1)}$	$x_3^{(2)}$	$x_3^{(3)}$	$x_0^{(3)}$	$x_1^{(3)}$	$x_2^{(3)}$	$x_3^{(3)}$

Figure C.1: Collective communication operations as proposed by Schatzet et al.[29]

Appendix D

Network Interface Bandwidth for different tests

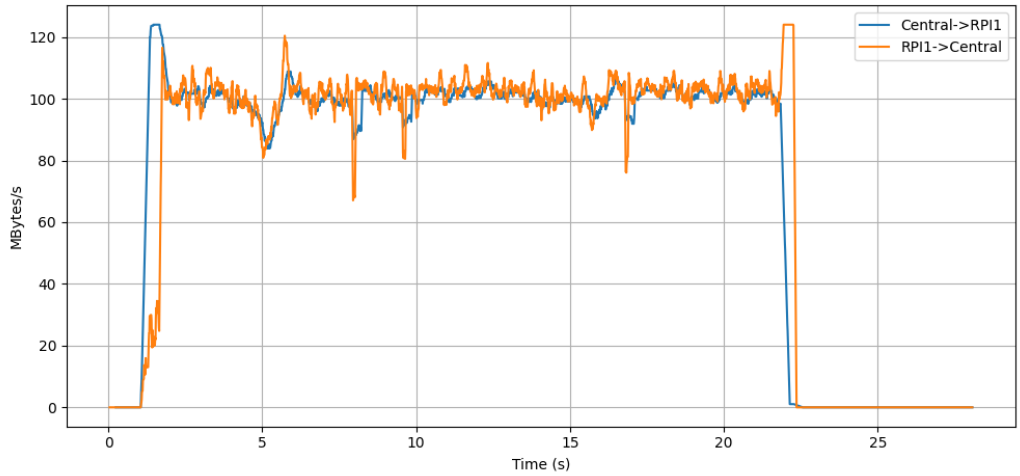

Figure D.1: Bandwidth graph of the protocol performing Echo operations and with communication optimized settings

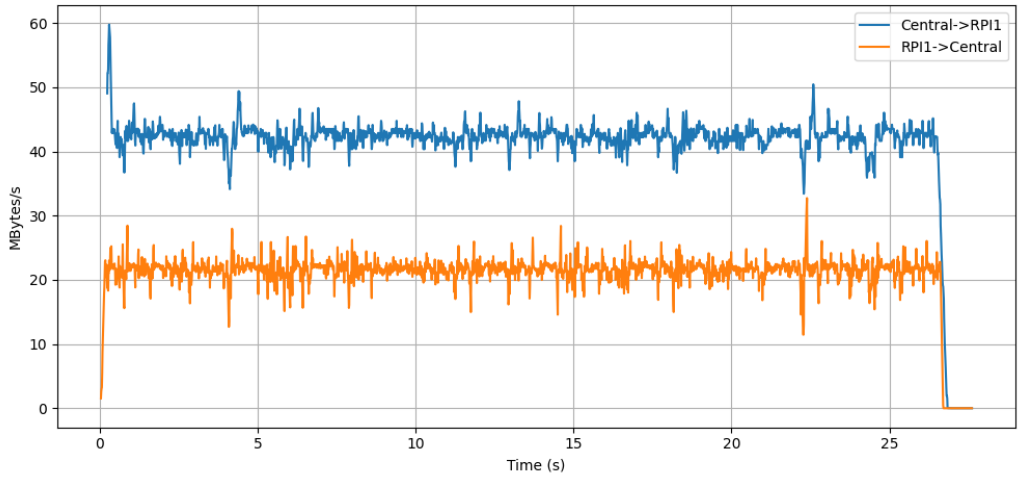

Figure D.2: Bandwidth utilization for 1 Raspberry Pi Nodes. No aggregation, all interim results are sent back to the central node

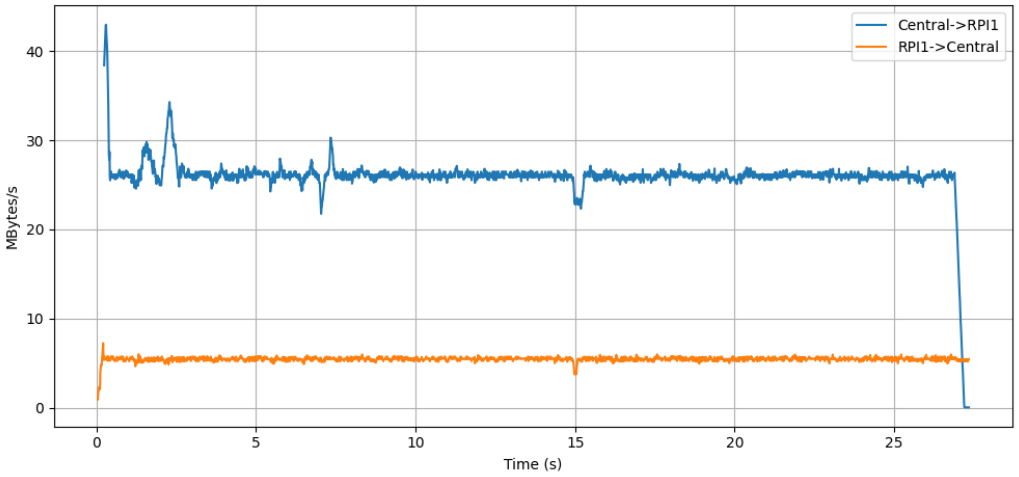

Figure D.3: Bandwidth utilization when aggregating 4 Jobs, in a Row Pattern.

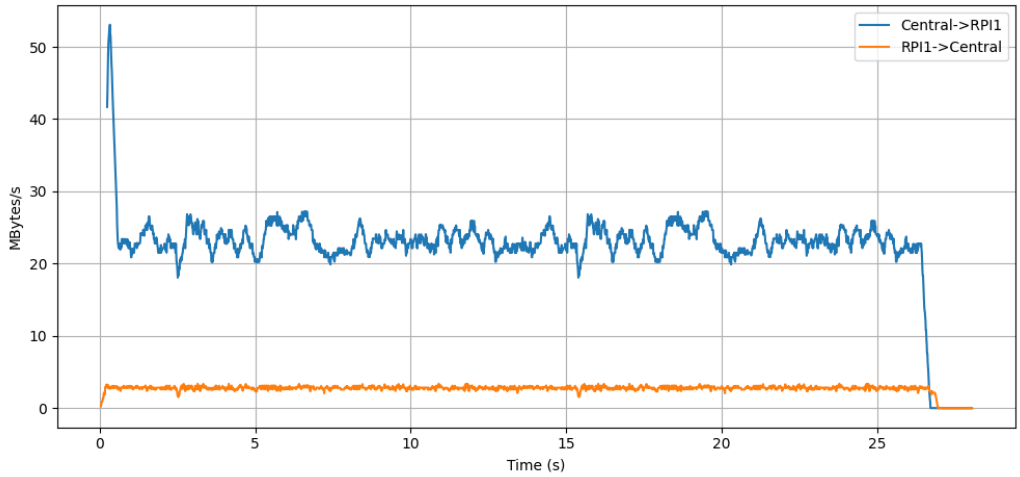

Figure D.4: Bandwidth utilization when aggregating 8 Jobs, in a Row Pattern.

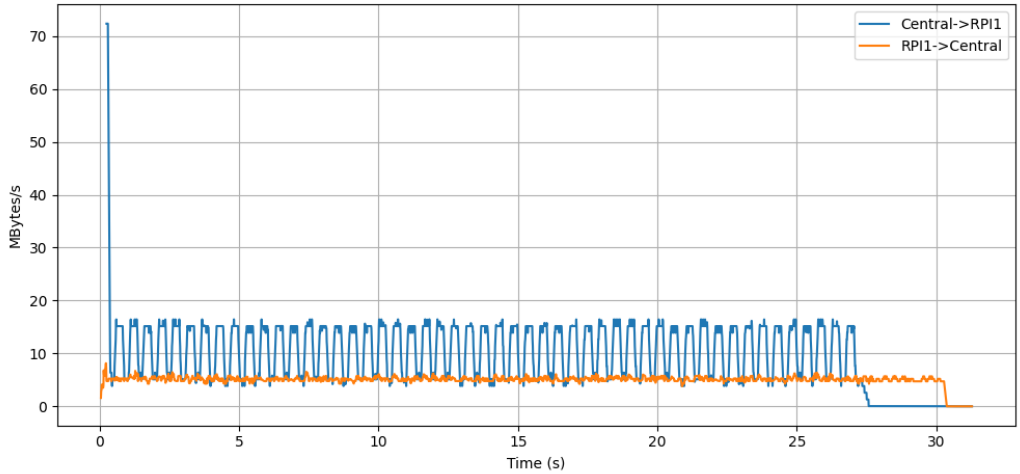

Figure D.5: Bandwidth utilization when aggregating 4 Jobs, in a Square Pattern.

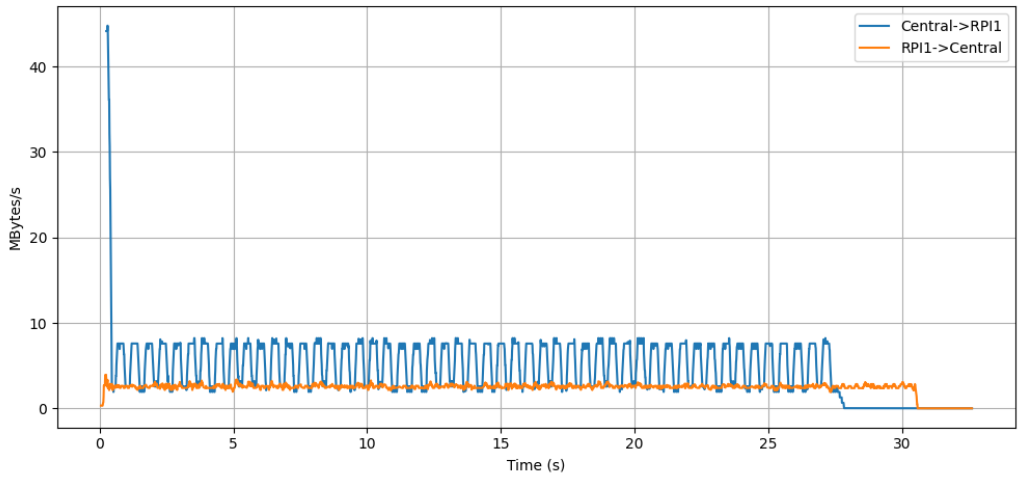

Figure D.6: Bandwidth utilization when aggregating 8 Jobs, in a Square Pattern.

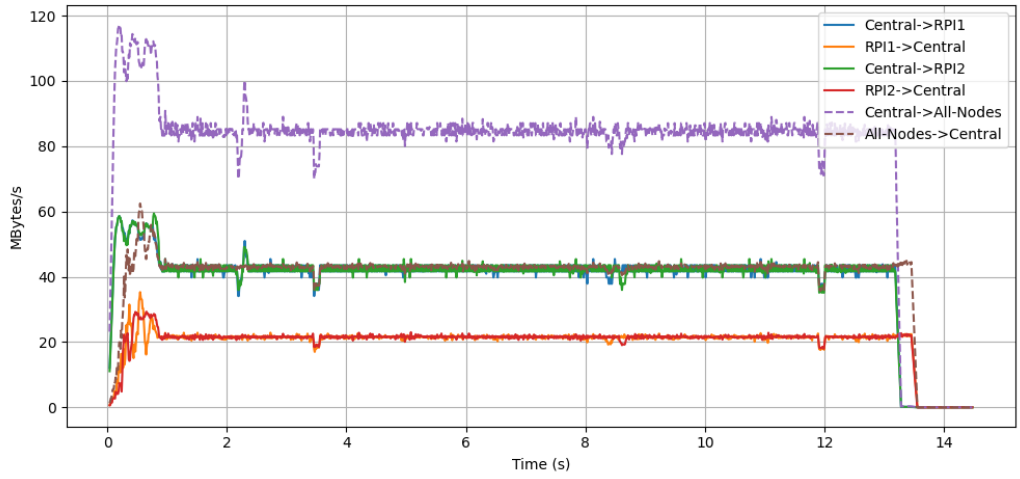

Figure D.7: Bandwidth utilization for 2 Raspberry Pi Nodes. No aggregation, all interim results are sent back to the central node

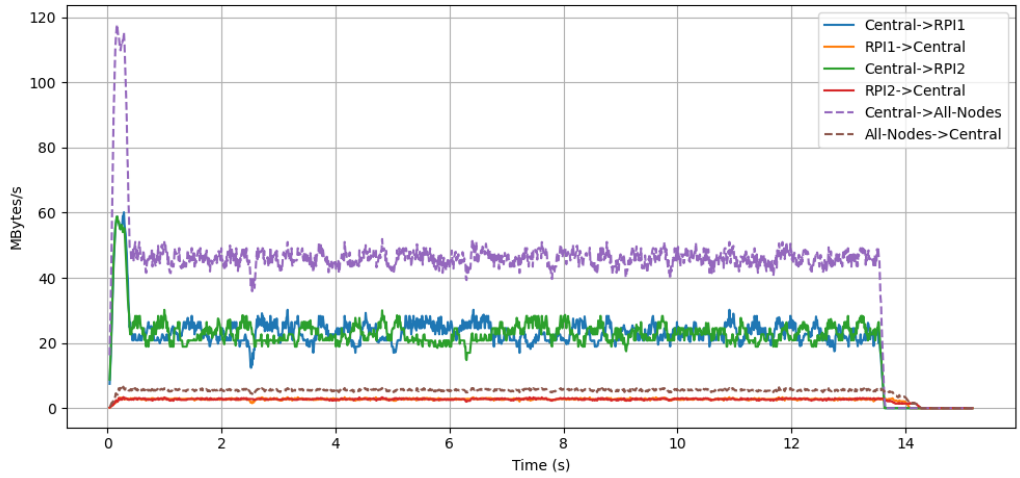

Figure D.8: Bandwidth utilization for 2 Raspberry Pi Nodes. Row aggregation, 8 L1 jobs

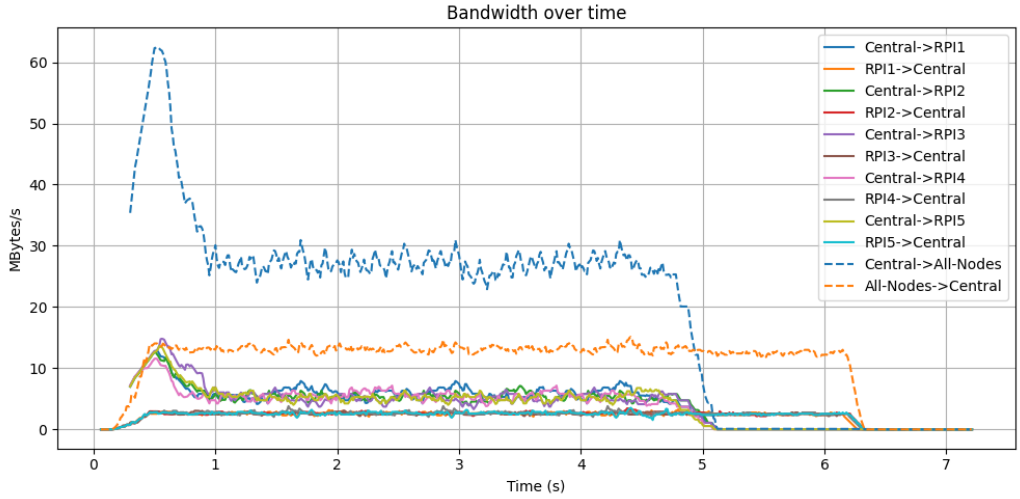

Figure D.9: Bandwidth utilization for 5 Raspberry Pi Nodes. Square aggregation, 8 L1 jobs

Appendix E

CPU utilization of the Worker Nodes for different tests

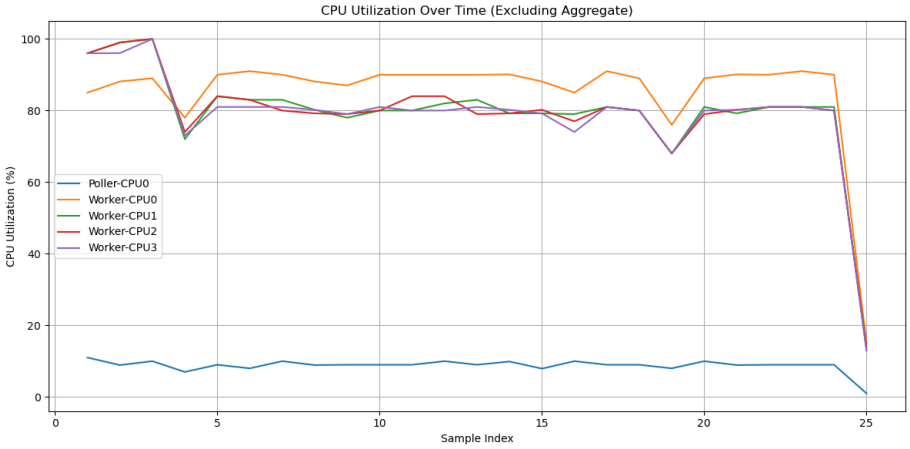

Figure E.1: CPU utilization with an 8 Jobs queue of L1 jobs. Note how the resources are subutilized

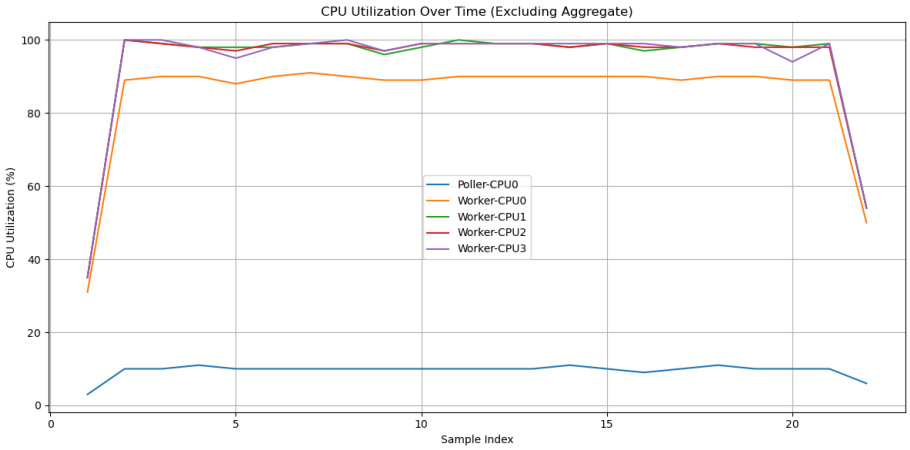

Figure E.2: CPU utilization with a 36 Jobs queue of L1 jobs. This was the previous iteration before reaching full utilization

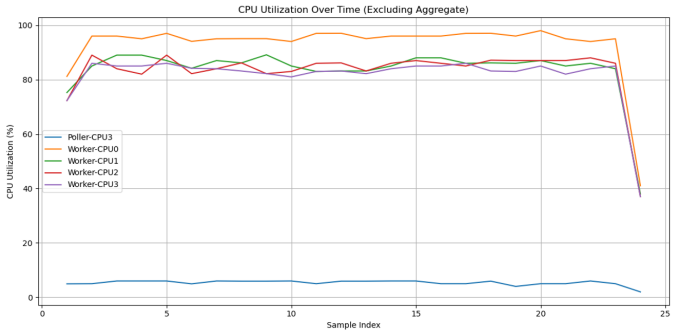

Figure E.3: CPU utilization when performing 8 Job aggregation using condition variables and mutexes to perform synchronization amongst threads that are aggregating to the same block

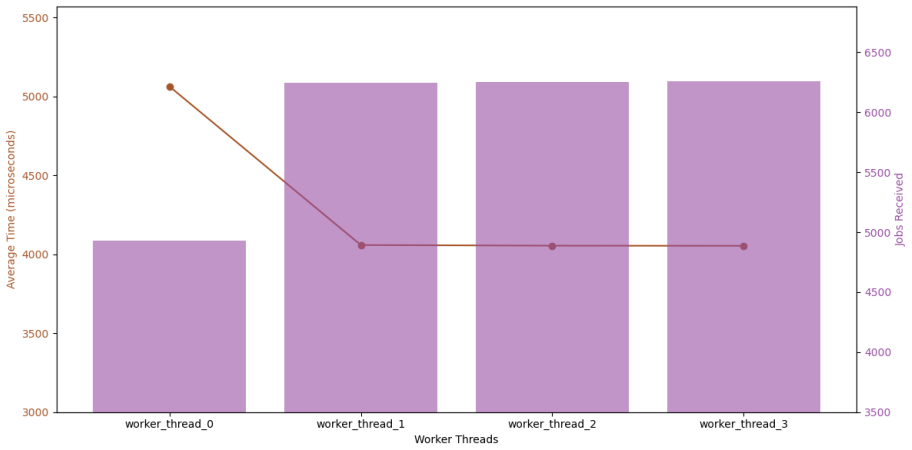

Figure E.4: Job execution time vs Job allocation per thread for multiplications with atomic operations. Average of 10 iterations

Appendix F

Profiling the communication protocol

Category	Time (seconds)	Percentage of Total Communicat
Total Communication Time	3.305	100%
System Context	0.835	25.26%
NIC & Packet Assembly (Read/Write)	0.465	14.07%
TCP ACKs & Checksums (Write)	0.165	4.99%
Synchronization Tasks (Read/Write)	0.09	2.72%
Memory Management (Read/Write)	0.115	3.48%
User Context	2.47	74.74%
Write Operations	1.39	42.06%
Marshalling	0.39	11.80%
TCP Socket	0.99	29.96%
Read Operations	0.70	21.17%
Unmarshalling	0.13	3.93%
TCP Socket	0.46	13.91%
Message Queue Overhead	0.11	3.33%
Event Loop Operations	0.38	11.50%

Table F.1: Adjusted Normalized Breakdown of Communication Protocol (100%) with Annotations. Note: For profiling purposes, tasks labeled as Read/Write were allocated 50/50 between read and write operations. A more detailed investigation would be needed for precise allocation. This results were obtained from Flame Graphs as the ones showed in Figures F.1 and F.2

As demonstrated in earlier sections of our study, we employed pidstat to

Figure F.1: Flame graph of the call graph at one of the Raspberry Pi nodes. Zoom out to show the prevalence of the multiplication task

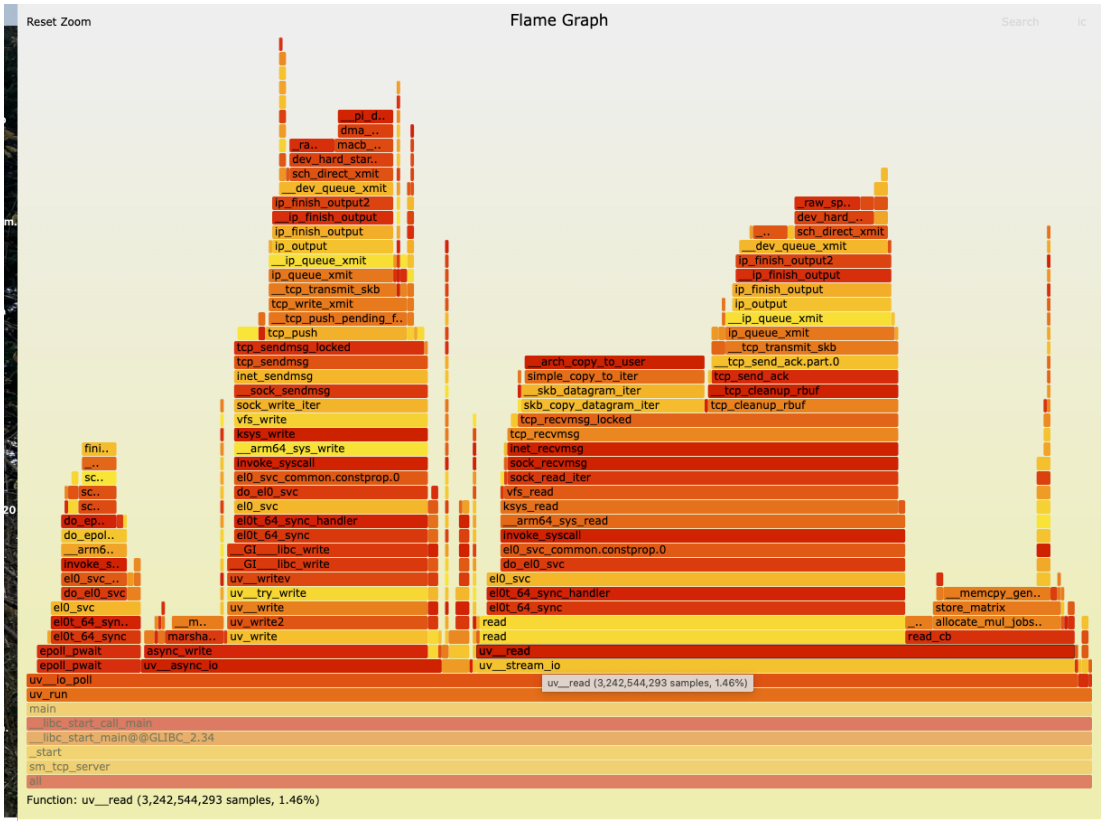

Figure F.2: Flame graph of the call graph at one of the Raspberry Pi nodes. Zoom into the operations of the Poller Thread

01:47:47	UID	PID	%usr	%system	%guest	%wait	%CPU	CPU	Command
01:47:48	0	11593	0.00	4.00	0.00	2.00	4.00	0	kworker/0:1-events
01:47:48	1000	12013	388.00	4.00	0.00	2.00	392.00	3	sm_tcp_server
01:47:48	1000	12802	0.00	1.00	0.00	0.00	1.00	1	pidstat

Figure F.3: System-wide pdistat, showing the interruption being caused by kworker thread

track the performance of both worker and poller threads. In these tests, we observed consistent interruptions on the Worker Thread associated with CPU0. To investigate further, we conducted a system-wide pidstat analysis, which identified a kernel process—a kworker thread—as the source of the disruptions affecting CPU 0. These interruptions were in line with the dips.

To identify the source of these interruptions, we enabled kernel backtraces, pinpointing the cause to socket interruptions. Interestingly, pidstat did not associate these interruptions with the poller thread, possibly because they were executed on a different CPU (CPU0).

Based on these findings, we decided to centralize all our communication processes, including the poller thread, on CPU0 to enhance system efficiency and reduce interruptions.

To further profile the performance of the roller thread, we also obtained flame graphs, using the library developed by Brendan Gregg [11]. Some examples of the graphs obtained in this process are shown in Figures F.1 and F.2.

Appendix G

Simulation of submatrices generated by CARMA

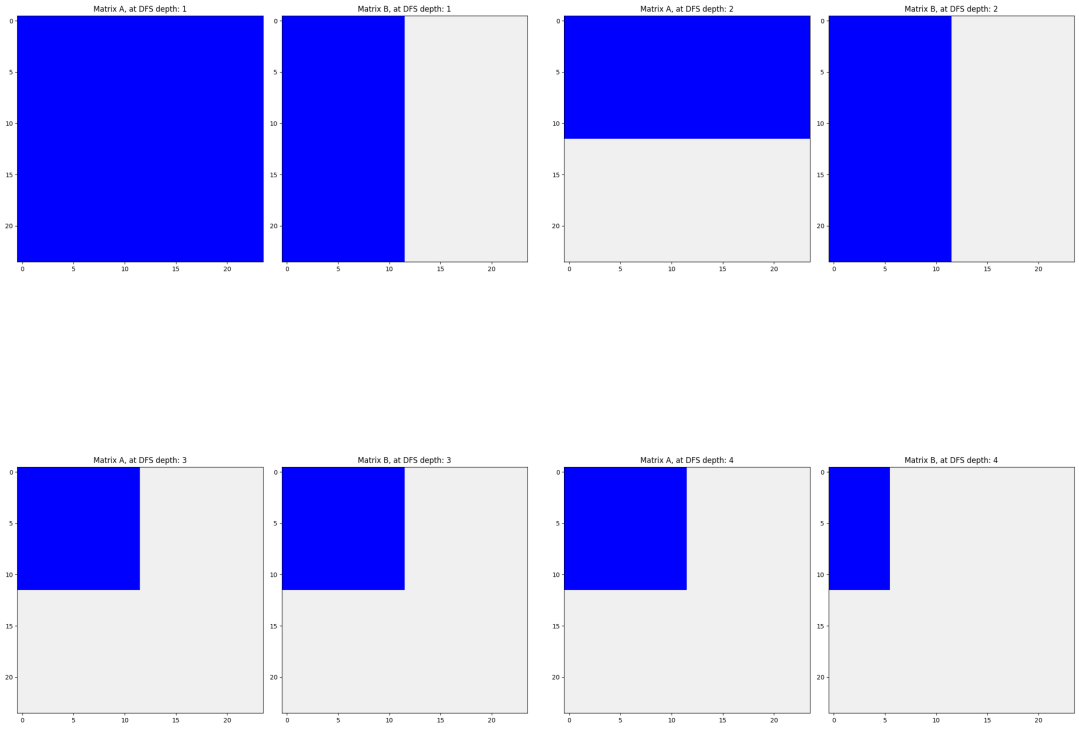

Figure G.1: Dimensions of submatrices generated by a 3D Folding algorithm doing successive DFS steps, like the approach from CARMA [6]. Sub-matrices displayed at various DFS depths