University of Kentucky
UKnowledge

Theses and Dissertations--Electrical and Computer Engineering

Electrical and Computer Engineering

2017

Hierarchical Implementation of Aggregate Functions

Pablo Quevedo
University of Kentucky, peqc1300@gmail.com
Digital Object Identifier: https://doi.org/10.13023/ETD.2017.496

Right click to open a feedback form in a new tab to let us know how this document benefits you.

Recommended Citation
Quevedo, Pablo, "Hierarchical Implementation of Aggregate Functions" (2017). *Theses and Dissertations--Electrical and Computer Engineering*. 111.
https://uknowledge.uky.edu/ece_etds/111

This Master's Thesis is brought to you for free and open access by the Electrical and Computer Engineering at UKnowledge. It has been accepted for inclusion in Theses and Dissertations--Electrical and Computer Engineering by an authorized administrator of UKnowledge. For more information, please contact UKnowledge@lsv.uky.edu.

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Pablo Quevedo, Student

Dr. Henry Dietz, Major Professor

Dr. Cai-Cheng Lu, Director of Graduate Studies

Hierarchical Implementation of Aggregate Functions

THESIS

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering in the College of Engineering at the University of Kentucky

By
Pablo Quevedo
Lexington, Kentucky

Director: Dr. Henry Dietz, Professor of Electrical and Computer Engineering
Lexington, Kentucky 2017

Copyright© Pablo Quevedo 2017

ABSTRACT OF THESIS

Hierarchical Implementation of Aggregate Functions

Most systems in HPC make use of hierarchical designs that allow multiple levels of parallelism to be exploited by programmers. The use of multiple multi-core/multi-processor computers to form a computer cluster supports both fine-grain and large-grain parallel computation. Aggregate function communications provide an easy-to-use and efficient set of mechanisms for communicating and coordinating between processing elements, but the model originally targeted only fine-grain parallel hardware. This work shows that a hierarchical implementation of aggregate functions is a viable alternative to MPI (the standard Message Passing Interface library) for programming clusters that provide both fine-grain and large-grain execution. Performance of a prototype implementation is evaluated and compared to that of MPI.

KEYWORDS: AFN, aggregate functions, parallel computation, MPI, OpenMP

Author's signature: Pablo Quevedo

Date: December 8, 2017

Hierarchical Implementation of Aggregate Functions

By
Pablo Quevedo

Director of Thesis: Henry Dietz

Director of Graduate Studies: Cai-Cheng Lu, Ph.D.

Date: December 8, 2017

This work is dedicated to my family. They have always supported me in the good times as well as the bad times.

ACKNOWLEDGMENTS

This work would have not been possible without the help of my professor and advisor Henry Dietz, my collegue Paul Eberhart, my coworkers, Sudhanshu Gaur and Jeremy Osteergard and many other who contributed directly and indirectly in the process of this work.

TABLE OF CONTENTS

Acknowledgments . . . iii

Table of Contents . . . iv

List of Figures . . . v

List of Tables . . . vi

Chapter 1 Introduction . . . 2

Chapter 2 Motivation and Background . . . 5
- 2.1 Pure MPI . . . 5
- 2.2 Pure OpenMP . . . 7
- 2.3 Hybrid: MPI + OpenMP . . . 7
- 2.4 Aggregate Function over the Network . . . 8

Chapter 3 Aggregate Functions Networks . . . 9
- 3.1 STAPERS Role in AFAPI . . . 10
- 3.2 Reduction Operations . . . 11
- 3.3 Scan Operations . . . 11
- 3.4 Communication Operations . . . 13
- 3.5 Control Operations . . . 14

Chapter 4 Implementation details . . . 15
- 4.1 In-node: SHMAPERS . . . 17
- 4.2 Processes Identification . . . 18
- 4.3 Handling Network Communication . . . 20
- 4.4 Inter-node Communication Algorithm . . . 27

Chapter 5 Performance Benchmarks and Comparison . . . 29
- 5.1 Test Construction . . . 29
- 5.2 Testbeds . . . 30
- 5.3 Intra-node Communications Comparison . . . 32
- 5.4 Inter-node Communications Comparison . . . 38

Chapter 6 Conclusions and Future Work . . . 42

Bibliography . . . 44

Vita . . . 46

LIST OF FIGURES

4.1 STAPERS Multi-Level Architecture 15
4.2 MPI Multi-Level Architecture 16
4.3 Process flowchart for CPROC Behavior Type 1 24
4.4 Process flowchart for CPROC Behavior Type 2 25
4.5 Process flowchart for CPROC Behavior Type 3 27

5.1 Latency 34
5.2 Intra-node Latency Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 12 36
5.3 Intra-node Latency Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 6 36
5.4 Intra-node Percent Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 6 37
5.5 Intra-node Percent Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 12 37
5.6 Reduce Operation Results Part 1 40
5.7 Reduce Operation Results Part 2 40

LIST OF TABLES

3.1 Collectives Implemented in STAPERS 11
3.2 Reduce Operation Equivalents 12
3.3 Scan Operation Equivalents 12
3.4 Communication Operation Equivalents 13
3.5 Control Operation Equivalents 14

5.1 Twelve-Core, Two-Socket Shared Memory Computer 30
5.2 Sixteen-Node Cluster Computer Specifications 31
5.3 Average Operation Latency Comparison 34
5.4 Intra-node Latency NPROC = 6 35
5.5 Intra-node Latency NPROC = 12 35
5.6 Reduce Operation Latency NPROC=32, PCNUM=16 39
5.7 Scan Operation Latency NPROC=32, PCNUM =16 41

.

Chapter 1 Introduction

With the Parallel computing is a well-established field in computer engineering that was born from the need for more computing power. Since the creation of computers, engineers always have searched for a way to get the most performance out of computers. At early stages, performance improvement relied on hardware development progress, more significantly by increasing the number of transistors per chip. The rapid increase in the number of transistors lead to the birth of Moore's Law which states that the number of transistors would double approximately every two years, and consequence of performance. In order to keep with the improvement pace, engineers took to parallelization to make better utilization of the higher amount of transistors. In order to make use of parallelization efficiently different schemes came into existence. These schemes range from using multipleprocessors or cores in one computer (SMP machines) to act independently, to using multiple computers connected through a network to behave as one big machine (computer clusters). The field that studies these schemes and paradigms is called high-performance computing (HPC).

Today most systems in HPC make use of hierarchical designs that allow multiple levels of parallelism to be exploited by programmers. The use of multiple multi-core/multi-processor computers to form a computer cluster allows for high granularity computation capabilities, effectively allowing a much bigger processor count that it would be possible with any other architecture. With multi-core computers becoming ubiquitous, commodity hardware clusters have also become quite common for scientific research. Commodity clusters, as their name implies, make use of commodity computer hardware to construct a computer cluster. Their advantage relies not in the raw power attainable but in the high price-performance ratios that can be obtained with relatively easily accessible hardware. The convenience and popularity of commodity clusters made evident the need for software support, specifically the need for a communication layer that allows nodes to communicate, synchronize and share datum.

To meet the need for a communication layer Message Passing Interface(MPI) become the de facto communication scheme for clusters machines since its inception in the 90s[1]. There are many libraries available that implement the MPI standard such

as OpenMPI and MPICH. These libraries are highly optimized for communication and are almost universally used for communication between computer nodes, commonly referred to as inter-node communication. Despite MPI being mainly used for inter-node communication, it also has intra-node communication capabilities that allow multiple processes or processors in the same node to communicate with each other using a common Application Programming Interface (API). Despite this convenience, one of the issues with MPI implementations is the multi-processor and multi-core performance underutilization of resources in shared memory systems. Other libraries like Open Multi-Processing (OpenMP) are regarded better suited for shared memory architectures while also being easier to read and write than MPI, but its performance across nodes lags behind that of MPI. These problems has seen the rise of hybrid programming as a mean to take the best of both worlds and obtain maximum performance. By using MPI for inter-node communication and OpenMP for intra-node execution, it is possible to exploit both paradigms and optimize resource utilization. The third and latest revision of the MPI standard (MPI 3.0) aimed to bridge this gap and defines a shared memory interface to make better use of multi-core systems using the same API. Nevertheless, hybrid computing is still quite common and a preferred alternative.

While hybrid programming allows the use of shared memory and message passing paradigm together in one program, it is hardly the only computing paradigm in existence. In the late 90s, Dr. Henry Dietz and his colleagues at Purdue University introduced a different scheme for parallel execution with the name of Aggregate Function Networks(AFN)[2]. An AFN uses a scheme where synchronization among processes is a byproduct of an aggregate operation, that is, an operation that is requested and performed in unison by all processing elements of the parallel program. Aggregate Function Networks present a different scheme for communication than those mentioned so far and is the main subject of this work. The current work presents a Proof of Concept (PoC) first hierarchical implementation of aggregate functions. This implementation, which is named Shared memory Tcp Adapter for Parallel Execution and Rapid Synchronization (STAPERS), presents an alternative way to attain performance in today's highly hierarchical cluster systems.

With this objective in mind, this work is divided in several chapters. chapter 2 studies standard methods functionality, characteristics and pitfalls that motivates the extension of aggregate functions capabilities. chapter 3 explains the paradigm used

by aggregate functions, history and differences to other methods. chapter 4 explains the implementation of a network layer for aggregate functions. Chapter 5 presents the benchmark results and comparison to more standard and established libraries like OpenMPI. Finally, chapter 6 outlines the conclusions of this work and suggestions for future work.

Chapter 2 Motivation and Background

In order to assess the viability of Aggregate Function Networks(AFN) as a parallel programming model it is necessary to set a reference point for comparison. Studying the characteristics of existing implementations as well as their shortcomings will allow to formalize a set of metrics to compare AFN to existing models. The two paradigms that dominate the HPC community currently are Message Passing and Shared Memory. Both models have implementations that support intra-node and inter-node communication, allowing to take advantage of the highly hierarchical design of today's clusters. Both models use different paradigms for communication and thus perform differently depending on whether communication is intra-node or inter-node. These discrepancies of performance lead to three programming models[3]:

- Pure MPI programming
- Pure shared memory using OpenMP
- Hybrid model: OpenMP + MPI

The following sections study each model separately and establish their advantages and disadvantages and how aggregate functions fit in the overall picture.

2.1 Pure MPI

MPI is a well established API in the HPC world that uses the message passing paradigm to communicate across parallel processes. Message Passing consists, as its name implies, in sending messages among processes to realize communication. Normally this process is done in such a way that the sender doesn't care if the message was received, though they typically are. This message passing tends to resemble function calls as the message buffer is given to a function that sends the message to a destination process. The main difference is that in message passing the buffer is copied by the receiving process, effectively creating two copies of the same information. This is a non-issue for computer networks as the latency to send information across computers is much bigger than the latency created by local memory copies. It is also safe to say that all network communication is some form of message passing.

However, in SMP systems the implied memory copy is the greatest source of latency thus holding much more relevance in operations between processes in the same node. For this reason MPI implements different methods to optimize this communication. These methods are[4]:

- NIC-Based Message Loopback: A Network Interface Card (NIC) can detect whether the destination is on the same physical node or not. This way it is possible to eliminate network overhead. The problem is the message has to be copied to the NIC's memory then back to the receiving process generating another unnecessary copy.
- User-Space Shared Memory: This design alternative involves each MPI process on a local node attaching to a shared memory region. This shared memory region can then be used amongst the local processes to exchange messages and other control information. This is the most efficient of all three methods.
- CPU Based Kernel Modules for Memory Mapping: The Kernel-Based Memory Mapping approach takes help from the operating system kernel to copy messages directly from one user process to another without any additional copy operation. Dependence on the operating system is the mayor drawback as system calls can be expensive.

MPI over the years has added support for hardware and software optimizations that allow the libraries to make decisions based on the underlying system. Based on running hardware and program parameters, such as buffer length, and number of processes MPI chooses which algorithm or message passing model to run the communication process with.

Despite its higly optimized implementation MPI suffer froma couple of issues. The main issue with a pure MPI scheme is that it assumes the message passing is the most effective and efficient model for all levels of parallelism. This is not true for all cases[3]. In some cases, certain MPI implementations allow for the exchange of intra-node messages through shared memory regions but regardless all communication is conducted through IPC mechanisms which can be slow, and therefore incurs in unnecessary high overhead[5].

2.2 Pure OpenMP

OpenMP is an API aimed for portable shared-memory parallel programming and is based on the fork-join programming model [6]. By making use of define directives/pragmas, OpenMP creates regions that fork a number of threads to execute code in parallel. The number of threads is dependent of many factors, from number of processors assigned to the program to environmental variable settings. An OpenMP program begins execution as a single thread called the master thread. When the master thread encounters a parallel region it proceeds to fork, creating multiple execution threads. Each new child then proceeds to do their part of the computation and when the parallel region is finished, all threads join again at an implicit barrier, leaving the master thread to continue serial execution[7]. This model is quite easy to understand and program for. The use of #define directives to enclose regions to be parallelized makes for an easy learning curve compared to MPI. The problem with classic OpenMP is that it only supports SMP systems, and communication across the network is not supported. This led to work that aimed to extend network support [7]. By using a distributed shared memory(DSM) system, it is possible with some restrictions to run OpenMP programs on a cluster. It is, to some extent, a hybrid model, being identical to plain OpenMP inside a shared-memory node, but employing a sophisticated protocol that keeps shared memory pages coherent between nodes at explicit or automatic OpenMP flush points [3]. This approach presents problems, as not all operations are available across nodes in the network, and memory coherency across the network is not a trivial task. Another problem with OpenMP is the constant creation of threads in parallel regions. This overhead can slow performance if such parallel regions are not long enough to compensate for the overhead of creating new threads. This effect can be more significant in the cluster as communications suffer from higher latency issues. Nodes through a network are more likely to execute at different speeds thus breaking synchronization, further increasing latency.

2.3 Hybrid: MPI + OpenMP

Instead of using a specific library or methods, it is possible (and often done) to take the best of both worlds and use a hybrid programming model. This model of programming has multiple variations, but this work concentrates on the more common approach which uses MPI for inter-node communication while using OpenMP for intra-node

computation and communication. By using message passing across the network, it is possible to minimize the amount of time in communication, while using shared memory for intra-node communication to take advantage of memory optimization in multi-core/multi-processor systems. One of the main issues with this model is that it requires MPI to be aware of the multiple threads that can possibly be created by OpenMP regions. Thus, multiple considerations have to be made in order to select the model. Are threads allowed to do MPI calls individually or is just the master process? It is also worth noting that this method also suffers from the same problems of the pure OpenMP method. If not long enough computation regions are used the overhead of forking and joining threads becomes significant thus slowing execution.

2.4 Aggregate Function over the Network

Hybrid programming allows to obtain the most benefit of both worlds but requires knowledge of both interfaces and models which can be a steep learning curve for new programmers. Both models are quite different in implementation and interface, adding to the learning difficulty. MPI uses the well established Message Passing communication model, but this model is not as an effective model for intra-node computing as it is for inter-node. OpenMP suffers from a opposite problem, as it uses a Shared Memory model which is more effective than Message Passing for multi-core systems but it makes inter-node communication difficult. OpenMPI uses a library interface, while OpenMP uses #define directives to obtain parallelization. These differing methods create problems when the two models are combined and add additional layers of complexity. Given the pitfalls from the aforementioned methods, STAPERS aims to establish a different paradigm for programming execution in the form of Aggregate Function Networks, while also providing a more consistent API than a hybrid methodology. STAPERS uses shared memory internally to provide synchronization methods for computations, and uses a TCP interface to provide the same functionality for network communications. STAPERS is an implementation of aggregate functions that provide both fine-grain and large-grain execution. The goal of this work is to establish whether AFN is a viable scheme for parallel computing, and how competitive it is against currently established methods. Furthermore, it aims to provide a more reduced and consistent API than the alternatives. This work tests this hypothesis through the use of commodity built clusters.

Chapter 3 Aggregate Functions Networks

Aggregate Function Networks were introduced loosely back in 1996[8] and more explicitly in March 1998[2]. But the underlying concepts and mechanism were explored and developed much earlier. The earlier work, as far as 1988, started not as a communication model for networked computers but as a general barrier synchronization mechanism for Multiple Input Multiple Data (MIMD) hardware[9]. A Barrier synchronization mechanism is a method for parallel execution to ensure logic and data coherency between the participating processing elements. The barrier synchronization is accomplished by each process individually informing a barrier unit that it has arrived, and then waiting to be notified that all participating processors have arrived. Once the barrier synchronization has completed, execution of each processor is again completely independent. AFN rely on the idea that all participating processes may ,additionally to the barrier, perform a communication operation. That is, communication is a side-effect of all processing elements (PEs) executing a barrier synchronization [9].. Though this model is somewhat similar to synchronous message-passing, it differs in that communication does not need to be point-to-point. This property is consistent across all AFN implementations and have been all condensed in a fully abstract program interface that seeks to provide the most important aggregate communication functions in clean and portable structure. The name of this API is the Aggregate Function Application Program Interface (AFAPI) and it has seen different versions implemented for it in the past[10]:

- TTL_PAPERS: Cluster of Linux or Unix PCs linked by TTL_PAPERS compatible aggregate functions network hardware. Although AFAPI[1] derives from the PAPERs library, which dates back to 1994[9], the AFAPI was first released in August 1996.

- CAPERS: Two Linux PCs (or UNIX workstations) linked by a passive LapLink cable – the Cable Adapter for Parallel Execution and Rapid Synchronization. This library was first released in August 1996.

[1] Aggregate Function Application Programming Interface was initially designed to be a portable high-level interface to the various types of PAPERS cluster hardware. It then became the programming interface for all aggregate function implementing libraries

- SHMAPERS: A uniprocessor or shared memory multiprocessor supporting the UNIX System V IPC shared memory segment calls the SHared Memory Adapter for Parallel Execution and Rapid Synchronization. This library, which is the primary focus of this paper, was rst released in August 1996.
- UDPAPERS: A cluster of Linux PCs (or UNIX workstations) linked by a conventional network, such as Fast Ethernet, which is capable of sending messages using UDP broadcast the User Datagram Protocol Adapter for Parallel Execution and Rapid Synchronization. This library was never released.
- WAPERS: a passive wiring pattern using "wired AND" of open-collector parallel port lines to connect small numbers of IA32 Linux PCs as a cluster.

The AFAPI represents a third, fundamentally different, model for interactions between processors. Each AFAPI operation is an aggregate operation, requested and performed in unison by all processors. Thus, although the AFAPI routines can be called from MIMD or SIMD-style code, all AFAPI operations are based on a model of aggregate interaction that closely resembles SIMD communication. This is not a coincidence; years of research involving Purdue Universitys PASM (PArtitionable Simd Mimd) prototype supercomputer experimentally demonstrated that SIMD-like interactions are more efficient than asynchronous schemes for a wide range of applications [11].

3.1 STAPERS Role in AFAPI

STAPERS differentiates itself to previous AFAPI implementations in being the first one to allow both fine-grain and large-grain parallel execution. By implementing aggregate function capabilities at multiple layers, STAPERS allows make use of AFN in multi-core/multi-process clusters. Before explaining in the implementation details, it is wise to list the collectives [2] that are currently implemented in this work. Table 3.1 shows the operations extended to support network communication.

As listed in the table the functions are divided in 4 types; Reductions, Scans, Communications and Control Operations. To understand the following section explain briefly the purpose of each function and when possible the equivalent in other libraries like MPI and OpenMP.

[2] A collective operation is a concept in parallel computing in which data is simultaneously sent to or received from many nodes

Table 3.1: Collectives Implemented in STAPERS

Reduce Ops	Scan Ops	Communications Ops	Control & Internal Ops
p_reduceAnd	p_scanAnd	p_bcastPut	p_init
p_reduceOr	p_scanOr	p_bcastGet	p_exit
p_reduceXor	p_scanXor	p_bcastPutz	p_confirm
p_reduceMin	p_scanMin	p_bcastGetz	p__wait
p_reduceMax	p_scanMax	p_putGet	p__lwait
p_reduceAdd	p_scanAdd	p_putGetz	p__any
p_reduceMul	p_scanMul	p_gather	p__lany
–	–	p_scatPut	p__all2all
–	–	p_scatGet	–
–	–	p_scatPutz	–
–	–	p_scatGetz	–
–	–	p_all2all	–
–	–	p_all2allz	–

3.2 Reduction Operations

A reduction is an operation that involves reducing a set of numbers into a smaller set via an operation. For example, taking a list of numbers [1, 2, 3, 4, 5, 6] and reducing them to a single number via the sum operation, in this case to the number 21. Same concept applies with a different operation like multiply, in which the result for the previous list would be 120. In the case for parallel computation, each process has a part of the set to be reduced, thus a reduction operation handles the communication between these processes to obtain the desired result. Table 3.2 puts together the equivalent functions in MPI and OpenMP for comparison. Notice that the MPI Equivalent, presents the function name for the operation followed by the operation to be executed. This operation is passed to the function as an argument. In the case for the OpenMP operations column, only the final part of the line was introduced, for a complete command all operation should be preceded by #pragma omp parallel for valid OpenMP syntax.

3.3 Scan Operations

A scan operation is almost exactly as a reduction one. The difference lies in that each process performs only a partial reduction, only taking into account processes of lower rank. For example, taking a list of numbers [1, 2, 3, 4, 5, 6] divided by 6

Table 3.2: Reduce Operation Equivalents

<table>
<tr><th>Operation</th><th>STAPERS</th><th>MPI Equivalent</th><th>OpenMP Equivalent</th></tr>
<tr><td>Bitwise AND</td><td>p_reduceAnd</td><td>MPI_Allreduce, MPI_BAND</td><td>for reduction(&:var)</td></tr>
<tr><td>Bitwise OR</td><td>p_reduceOr</td><td>MPI_Allreduce, MPI_BOR</td><td>for reduction(|:var)</td></tr>
<tr><td>Bitwise XOR</td><td>p_reduceXor</td><td>MPI_Allreduce, MPI_BXOR</td><td>for reduction(ˆ: var)</td></tr>
<tr><td>Minimum</td><td>p_reduceMin</td><td>MPI_Allreduce, MPI_MIN</td><td>for reduction(min:var)</td></tr>
<tr><td>Maximum</td><td>p_reduceMax</td><td>MPI_Allreduce, MPI_MAX</td><td>for reduction(max:var)</td></tr>
<tr><td>Addition</td><td>p_reduceAdd</td><td>MPI_Allreduce, MPI_SUM</td><td>for reduction(+:var)</td></tr>
<tr><td>Multiplication</td><td>p_reduceMul</td><td>MPI_Allreduce, MPI_PROD</td><td>for reduction(*:var)</td></tr>
</table>

different processes, each one of them holding one number based on their ranking, that is, process 1 has datum in location 1, which happens to be number 1, process 2 has the datum in location 2, process 3 has datum in location 3, and so on. Given this distribution, if process 4 wants to perform a sum reduction via a scan, the result would be 10. This is because process 4 only takes the information from processes of lower rank and its own. In our example this mean the summation of numbers 1,2,3 and 4 which results in the number 10. Once again this process repeats for different operation like minimum or maximum. Table 3.3 puts together the equivalent functions in MPI for comparison. Notice that there is no equivalent for OpenMP as it does not support a direct scan operation.

Table 3.3: Scan Operation Equivalents

Operation	STAPERS	MPI Equivalent
Bitwise AND	p_reduceAnd	MPI_Scan, MPI_BAND
Bitwise OR	p_reduceOr	MPI_Scan, MPI_BOR
Bitwise XOR	p_reduceXor	MPI_Scan, MPI_BXOR
Minimum	p_reduceMin	MPI_Scan, MPI_MIN
Maximum	p_reduceMax	MPI_Scan, MPI_MAX
Addition	p_reduceAdd	MPI_Scan, MPI_SUM
Multiplication	p_reduceMul	MPI_Scan, MPI_PROD

3.4 Communication Operations

Communication operations are actually not operation in the strict meaning of the word. These functions do not operate on data, rather these functions share data between processes differently depending on the way the information needs to be distributed. Table 3.4 puts together the equivalent functions in MPI. For communication operation there are not equivalent in OpenMP.

Table 3.4: Communication Operation Equivalents

Operation	STAPERS	MPI Equivalent
Broadcast	p_bcastPut	MPI_Bcast, Caller Process ID != root
Broadcast	p_bcastGet	MPI_Bcast, Caller Process ID = root
Broadcast	p_bcastPutz	MPI_Bcast, Caller Process ID != root
Broadcast	p_bcastGetz	MPI_Bcast, Caller Process ID = root
Gather	p_gather	MPI_Allgather
All-to-All Broadcast	p_all2all	MPI_Alltoall
Info Exchange	p_putGet	MPI_Bcast and MPI_Recv
Info Exchange	p_putGetz	MPI_Bcast and MPI_Recv

The first four operation shown in Table 3.4 are equivalent to broadcasts. The first two are operate on a single datum while the second two act on arrays on data. The letter z following any function signals that such function operates in a array given by the user, and that the user must provide the lenght of such array. Furthermore, these 4 functions highlight the difference in how MPI and AFAPI handles broadcast root selection. The broadcast root is the process name doing the broadcast. MPI allows the user to choose which process does the broadcast by means of an argument passed to the function. AFAPI in the other hand has two functions to obtain the same functionality. The function doing the broadcast will call p_bcastPut while all other processes must make the call to p_bcastGet in order to receive the information needed. The next two functions are very natural for parallel processing and direct match between MPI and AFAPI. The last two functions are more complicated than stated in the table. A putGet operation is byproduct of the early development of AFN. The putget operation encompasses the communication patterns where each processing element supplies one datum to the network and receives one from the pool of data in the network[12]. Given this basic behavior a close MPI equivalent would be a broadcast to all processes followed by a receive call. Once again a putget has a

version which exchanges an array of data rather than a single datum, such version is ended with z like previous functions.

3.5 Control Operations

Control Operation refer to all functions that handle some type of non communication process. Table 3.5 puts together the equivalent functions in MPI. The first two rows are very intuitive as they handle initialization of the libraries. These functions serve the purpose to prepare libraries internals to handle parallel execution. Example of such internal are creation of threads, network sockets and process identification. AFAPI follows with an extra function called p_confirm which is used to check for the status of all processes. This function again is a by product of early implementation, namely PAPERS unit in this case. The last four rows refer to the barrier synchronization methods used to implement a barrier among processes. STAPERS introduces to AFAPI the functions p__lwait and p__lany. Since STAPERS is able to synchronize not only across processes in the same node but also across node in the network, local barrier and global barrier are separated in two function to clarify which type of synchronization is expected. These functions allow STAPERS to specify when the local set of processes actually need to synchronize with the rest of the nodes giving flexibility to the library. This same functionality is obtained in MPI by passing the Communication group id to the MPI_Barrier call. This communication group has to be explicitly set previously in the program but allows to define a arbitrary membership to the group, something STAPERS does not allow.

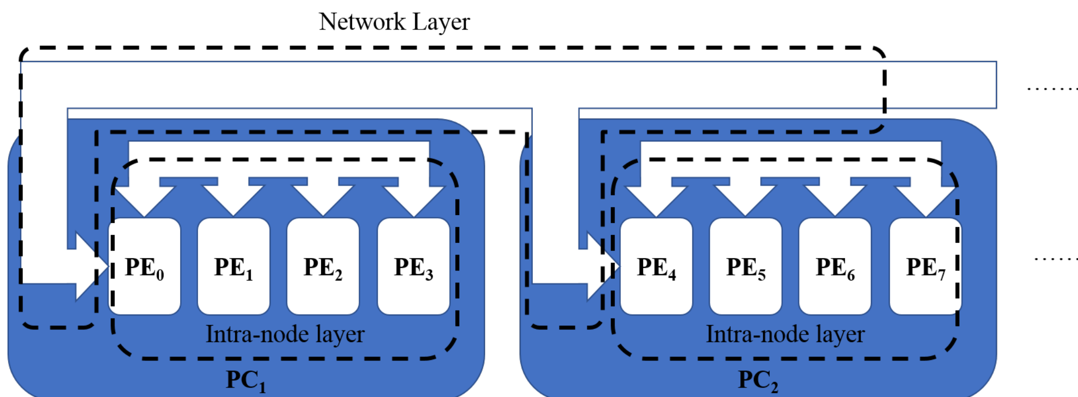
Table 3.5: Control Operation Equivalents

Operation	STAPERS	MPI Equivalent	OpenMP Equivalent
System Initialization	p_init	MPI_Init	#pragma omp parallel
System Termination	p_exit	MPI_Finalize	None
Verification	p_confirm	None	None
Barrier	p__wait	MPI_Barrier	#pragma omp barrier
Local Barrier	p__lwait	MPI_Barrier	#pragma omp barrier
Utility	p__any	MPI_Barrier	#pragma omp barrier
Utility	p__lany	MPI_Barrier	#pragma omp barrier

Chapter 4 Implementation details

In order to accomplish the goal set for this work two levels of communication are necessary. A first layer that allows communication among processes on the same machine and a second one for computers interacting through the network. As a comparison, MPI can be said to use a logical flat network as shown in Figure 4.2. This follows the nature of message passing as every processing element needs to be reachable by any other PE that wants to communicate. However this communication is realized, and however many layers the message has to cross, every PE has the ability to send any other PE a message directly. This constraint allows programmers to not worry about the underlying architecture of the network but in the interaction among processes, but makes internal implementation tricky as different configurations may need to be handled differently at different layers. In contrast Aggregate Functions do not suffer from this constraint as they don't need to make every processing element reachable to every other. Since all communication is done as consequence of a barrier synchronization of all elements, it is possible to separate communication by layer. That is, aggregate functions can be execute either among processes in the same node or elements across nodes separately. In other words, PEs that share a same node can aggregate information among them before proceeding to aggregate them with other nodes. This separation is portrayed in Figure 4.1.

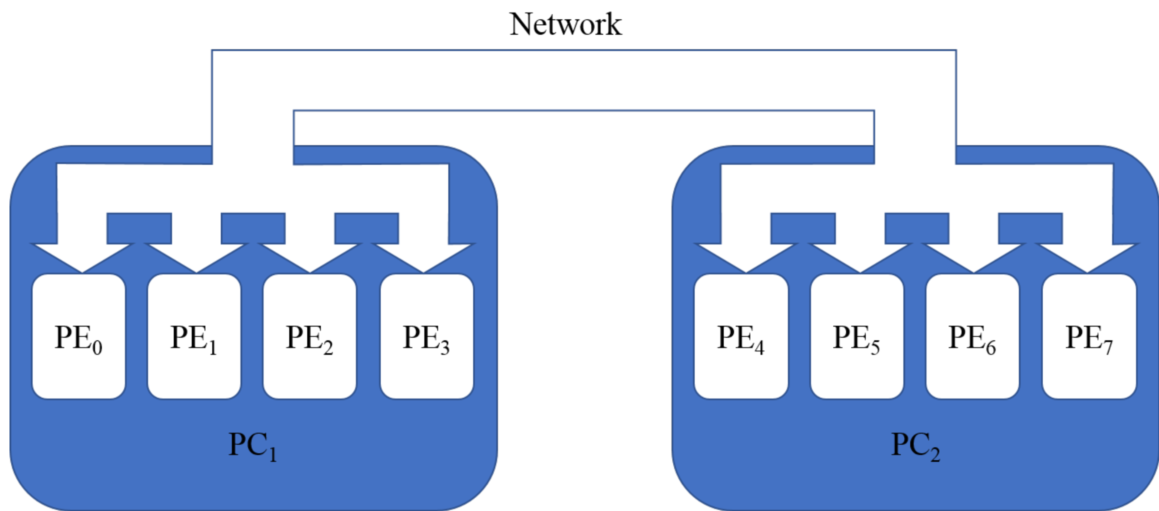
Figure 4.1: STAPERS Multi-Level Architecture



The first layer need is met by using Shared Memory Adapter for Parallel Execution and Rapid Synchronization (SHMAPERS) as a base implementation. SHMAPERS provides the ability to run multiple jobs in the same node and as a well as the

mechanism for these jobs to interact and synchronize between them. Therefore, the code base of SHMAPERS is used for STAPERS in-node communication. The second layer of communication required the major part of development time and in depth implementation will be explained in subsequent sections. It is worth noting that the current work does not implement the AFAPI library exhaustively. For example, the signal subsystem was not extended with network support. Normal and usable parallel computation can be realized without it and thus was deemed not necessary for the purposes of this work. The signal subsystem network support might prove useful in its own right but its left for future work.

Figure 4.2: MPI Multi-Level Architecture



4.1 In-node: SHMAPERS

SHMAPERS is unique in its approach to ensure synchronization among processes. Unlike OpenMP, SHMAPERS does not use any atomic locking mechanism to signal synchronization. Instead, it assigns a barrier counter to each processing element. When all counters have the same value then a barrier is realized. The trick lies in using an specific shared-memory structure that ensures that each counter is contained in a unique cache line. This property allows all barrier counter polling to be satisfied by a cache hit, essentially avoiding touching memory until the corresponding process write to its counter. An update to a barrier counter then triggers a cache miss on other processes, which by using a load, will then obtain the updated barrier counter finalizing the barrier synchronization.In other words, only a processor incrementing its counter will cause spinning processors to experience a cache miss, but this coherence traffic is then carrying the critical information that another processor is at the barrier. The structure used for this purpose is shown in Listing 4.1.

Listing 4.1: SHMAPERS Shared memory Structure

```
typedef struct {
  union p_shm_union datum[2]; /* Double buffered */
  int barno;        /* Barrier number for this PE */
  short pid;        /* UNIX PID of this PE */

  /* Add padding to make a multiple of cache line size */
  char  pad[P_CACHELINE -
        ((sizeof(int) +
          sizeof(short) +
          sizeof(union p_shm_union) +
          sizeof(union p_shm_union)) % P_CACHELINE)];
} p_shm_type;
```

In order to further explain how this works Listing 4.2 is shown. The p__wait function is one of the basic functions in the AFAPI library.Its performs barrier synchronization. More importantly is the contents of the function as it shows the basic synchronization algorithm used by all functions in AFAPI. All collectives use the same logic to realized intra-node synchronization. Before entering the while loop each process updated its barrier counter number, barno. Then a while loop traverse an array of p_shm_type structure. This array contains each process respective counter aligned

to cache line width as explained before. It is possible to just proceed and check if the next PE barrier is equal to our barrier number. The problem with this approach lies on counter wrapping around to smaller values as the counter increases. The solution is simply to compare for equality with either the value of this processor's counter or one added to that value (which may be a smaller numeric value due to wrap-around); if each processor's counter value matches one or the other value, the barrier is complete. Another optimization used is to check if the other process counter is at the next barrier. If such is the case then it follows that such process is waiting at the next barrier thus must have detected that all other processes reached the barrier. This make further polling unnecessary and thus synchronization is complete.

Listing 4.2: SHMAPERS Barrier Sync Example

```
static inline
void
p__wait(void)
{
  register volatile p_shm_type *p = p_shm_all;
  register volatile p_shm_type *q = (p + PCPROC);
  register int barno = ++(p_shm_me->barno);
  register int barnonext = (barno + 1);

  do {
    register int hisbarno = p->barno;
    if (hisbarno == barnonext) return;
    if (hisbarno != barno) {
      do {
        P_SHM_IDLE;
        hisbarno = p->barno;
        if (hisbarno == barnonext) return;
      } while (hisbarno != barno);
    }
  } while (++p < q);
}
```

4.2 Processes Identification

SHMAPERS already implements job creation in each node but it is necessary to expand this initialization process to cover all machines in a network. This means that

the total amount of processes needed for parallel execution will be composed of a multiplication of the number of processing elements on each node and the number of nodes in the network. In other words, the total number of processing elements in the program, lets call it NPROC, should be divided evenly among the number of nodes. This number its named PCNUM. Then, the total amount of processing elements per node is NPROC/PCNUM, or PCPROC. The even separation of processes is natural as this implementation is intended to be run on a homogeneous cluster of computers, which favors node homogeneity across the network to elevate predictability and maintain- ability of the cluster.

The first step to initialize the library's internal components properly is to modify the p_init function. SHMAPERS' initialization system gives each processing element (PE) its own id in the form of the IPROC variable at the moment of job creation. This alone does not work in with STAPERS as each node in a cluster would have a "twin" in other computers making synchronization cumbersome. STAPERS solves this by assigning each node with an id of their own based on a compile time seed. The name of that variable is meid and it is used to establish node ordering on the network. Making use of this network ID in combination with NPROC and PCNUM, it is possible to assign each processing element across the network a unique global id. Listing 4.3 shows the assignment process for this process. Notice that the meid variable is obtained at run-time. The function getmeid() is just a lookup table call that uses the local machine hostname to match it against a precompiled list which is preordered. When getmeid() matches the hostname in the list it returns its rank among nodes in the network. This allows each node to start the id assignment process at PCPROC intervals without any aliasing, making sure all PEs get a unique IPROC. A console process (CPROC) of each node is also assigned to be the first available number in the specific node interval and consequently giving all following processes the next available number in that node's interval.

Listing 4.3: ID assign Code

```
meid = getmeid();
cproc = (NPROC/PCNUM)*(PCNUM- meid - 1);
int proc = (NPROC/PCNUM)*(PCNUM - meid);
for (iproc = proc - 1 ; iproc > proc - (NPROC/PCNUM); --iproc)
    if (fork() == 0)
        break;
```

4.3 Handling Network Communication

With each node able to run multiple processes, the next step is to allow multiple nodes to communicate with each other allowing all processes to interact. This communication is achieved by assigning one process per node to handle the communication on behalf of the other processes in the same node. The assigned node is a console process, CPROC, and every node has it own CPROC. This terminology was introduced in SHMAPERS and is extended upon in STAPERS. Though CPROC was used exclusively for printing information in SHMAPERS, in this implementation it is used to handle the network communication. In STAPERS CPROC is the process that handles the TCP sockets, information printing and network barrier synchronization. CPROC is not a dedicated process for this purpose but a promoted one that serves a representative for network communication across nodes. CPROC will contribute to computations locally and only when it finished its computation it proceeds to synchronize with local node processes and gather their computation results. CPROC then will communicate with other computer nodes in the network to exchange information. When information exchange is done, CPROC will proceed to again synchronize with local processes to let them know the results of all other processes.

Though the process changes depending on the collective called, in general the synchronization process follows a common set of basic steps.

1. Install datum and signal barrier
2. Barrier synchronize
3. Execute local operation
4. If not CPROC barrier syncrhonize, CPROC does network exchange,
5. CPROC uses results to calculate overall result
6. CPROC broadcasts information obtained to local processes
7. Each IPROC return the result back to the caller

Notice that there are multiple barrier synchronization points in the list above. This is due to the multilayer nature of STAPERS. To understand this process lets use a reduction add operation as an example . Listing 4.4 shows a Pi calculation

program using STAPERS. The program receives from the user the amount of intervals to which Pi is to be calculates for. Then p_init() is called to initialize internal components as explained in the previous section. After each process goes through the local computations they will encounter the p_recudeAdd64f(sum) call. This is the call that will go through the process mentioned in previous list. Each IPROC will put their previously calculated sum in the shared memory region used for synchronization and signal the barrier they have done so. When all processes have signaled the barrier CPROC collects the information from local processes and broadcasts the info to all other nodes in the network. Once all nodes have information of every other node, their respective CPROC computes the overall result and proceeds to put the results in each processes shared memory region. Then it synchronizes again with local processes by joining the barrier synchronization. This enables each process to obtain the result of the operation executed. In the case of this example the correct value of Pi.

This process is analogous to a hybrid programming model as the local parallel threads would join together to accumulate information and then proceed with network communication. The differences are that STAPERS creates NPROC processes at initialization much like MPI and unlike OpenMP which creates threads in the parallel regions only.

Listing 4.4: Pi Calculation Program

```
#include <stdlib.h>
#include <stdio.h>
#include <AFAPI/afapi.h>

int
main(int argc,
char **argv)
{
    register double intervals, width, sum;
    register int i;

    /* check-in with AFAPI */
        if (p_init()) exit(1);

    /* get the number of intervals */
    intervals = atoi(argv[1]);
    width = 1.0 / intervals;
```

```
    /* do the local computations */
    sum = 0;
    for (i=IPROC; i<intervals; i+=NPROC) {
        register double x = (i + 0.5) * width;
        sum += 4.0 / (1.0 + x * x);
    }

    /* sum across the local results & scale by width */
    sum = p_reduceAdd64f(sum) * width;// <-- Aggregate Function Call

    /* have only the console PE print the result */
    if (IPROC == CPROC) {
        printf("Estimation of pi is %14.12lf\n", sum);
    }

    /* check-out */
    p_exit();
    exit(0);
}
```

There are three different behavior that follow this skeleton execution path but differ internally due to differences in the collective intended purpose. The following list separates the methods with their corresponding set of collectives

1. Behavior Type 1: Reduce Operations and non-array Broadcasts.
2. Behavior Type 2: Scan Operations
3. Behavior Type 3: Communication Collectives

Behavior Type 1 & 2: Reduce and Scan Operations

The behavior type 1 and 2 flow diagrams are shown in Figures 4.3 and 4.4, respectively. This two behaviors are the most similar of the three types as the differences lie in how the processes handle the data once the communication operation is finished. Both cases go through the original SHMAPERS process of function aggregation. They are in fact identical to the original implementation. Once the local operation is done, both behaviors make use of CPROC and go ahead with the network communication process to obtain all of the other nodes' results. Once the info exchange is made each CPROC updates every other IPROC in the same node with the new

data and all processes re-synchronize. At last, every process updates their return data appropriately. The difference between behavior 1 and 2, is that the first takes the data given to them by CPROC while the second takes the CPROC data update and recalculate the return value. This difference arises by the fundamental difference between a scan operation and a reduce operation. The former one only operates with the set of data from process = 0 to process = IPROC, while the second perform the computation taking into account the results from all processes.

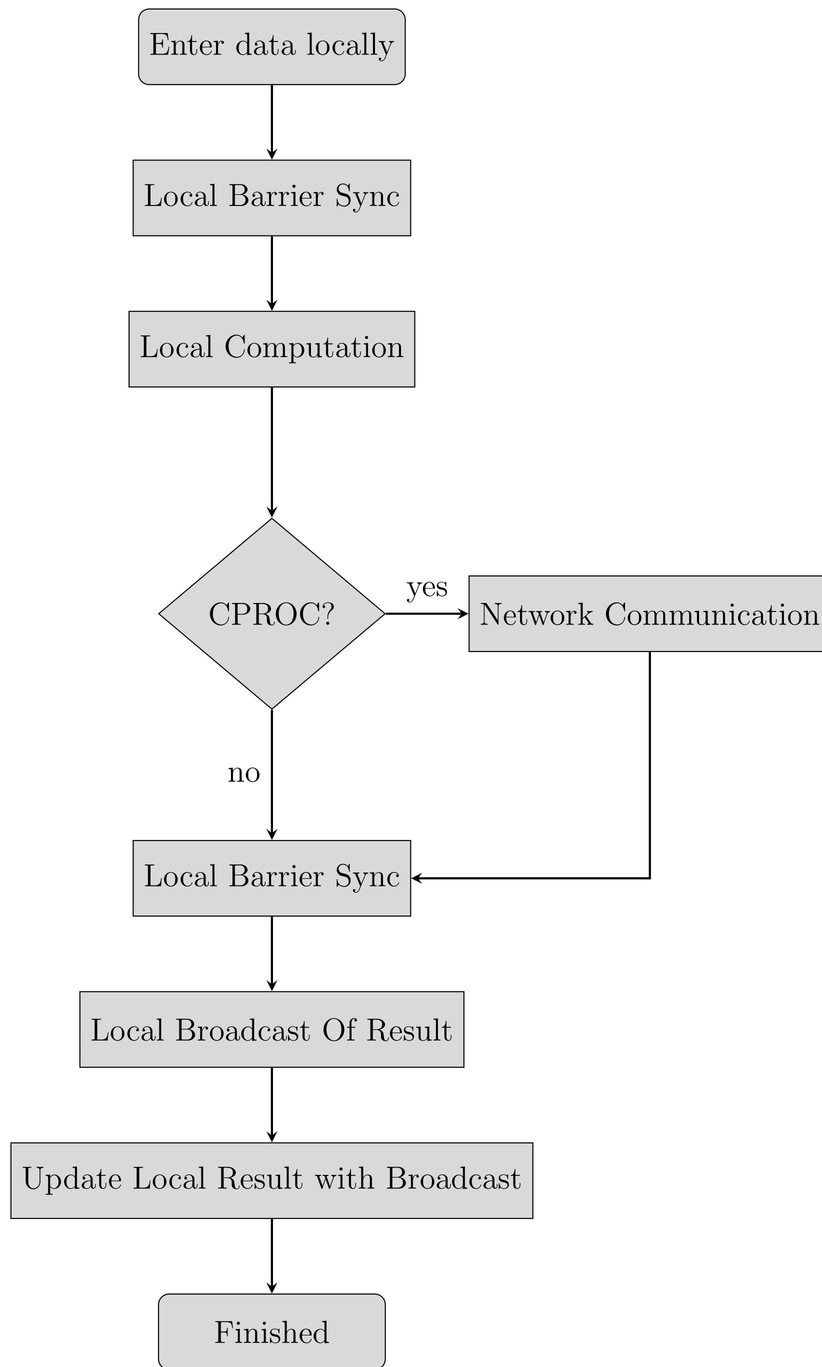


Figure 4.3: Process flowchart for CPROC Behavior Type 1

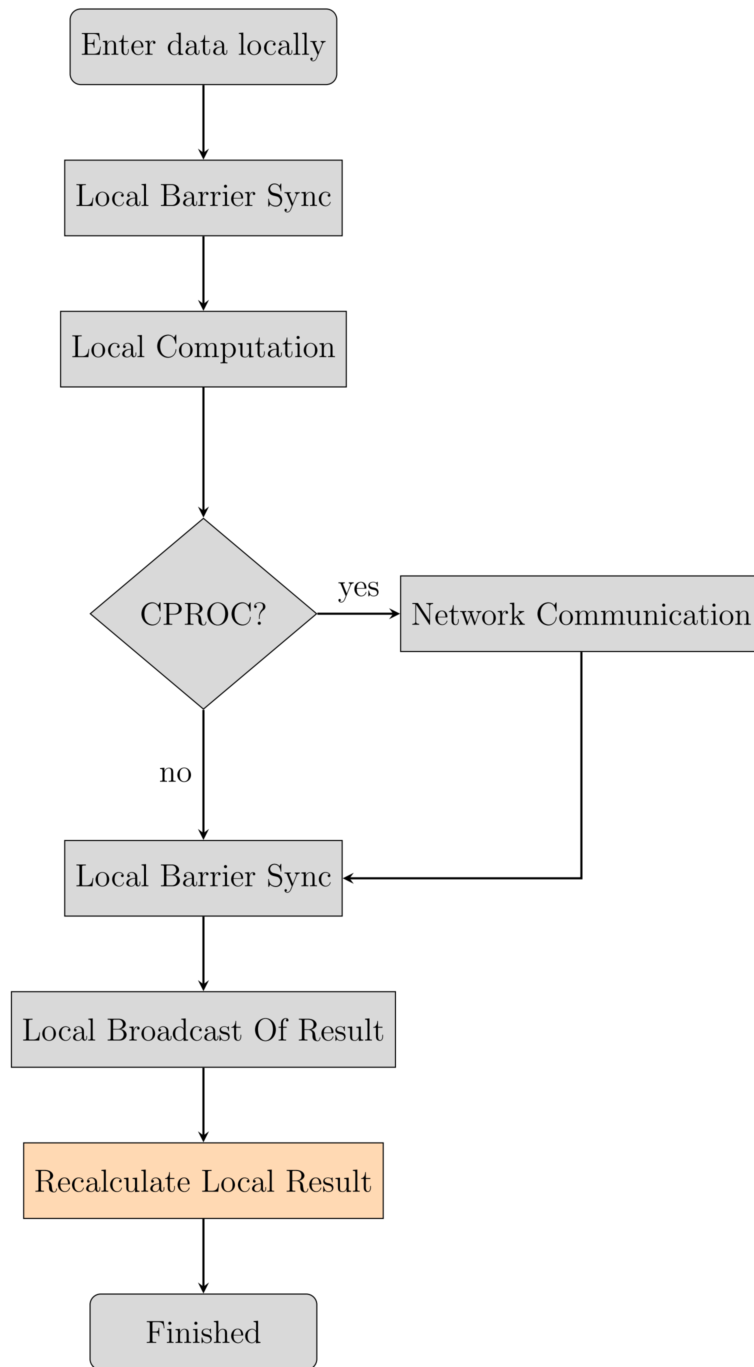


Figure 4.4: Process flowchart for CPROC Behavior Type 2

Behavior Type 3

Behavior three changes the process more deeply that the previous two. The first difference is that communication collectives do not perform operations thus local computations are not executed. Rather, processes within nodes wait for CPROC to finish communications on their behalf and then wait for CPROC to forward them the data obtained. This forwarding process is where this behavior differs from the other two. Instead of calculating results, each process obtains a copy of all processes' information and then operates upon it. We could have optimized this process so that each internal process only obtains the required value desired, but memory operations across processes are cheap in comparison to network latency and thus provide no much extra value for this thesis. These differences are reflected in Figure 4.5. This flowchart gives the impression of being simpler that the previous ones. Certainly logically this is true, but the actual implementation contains more information tracking to enable correct behavior. Contrary to reduce and scan operations, the communication collectives can operate on arrays of data of unknown type. This condition restricts assumptions of the data being transferred and force us to exchange data as blocks of memory across processes within nodes. In Figure 4.5, process 6 can perform multiple barrier synchronization operation as it transfers the information from CPROC to all other nodes. This is due to the fact that SHMAPERS uses shared memory to communicate across processes. This means each local PE has access to a limited amount of memory that they can share with each other, thus making communication buffer limited across them. It is worth restating that compared to network communication local memory access is cheap and this is not a significant factor in the overall system latency.

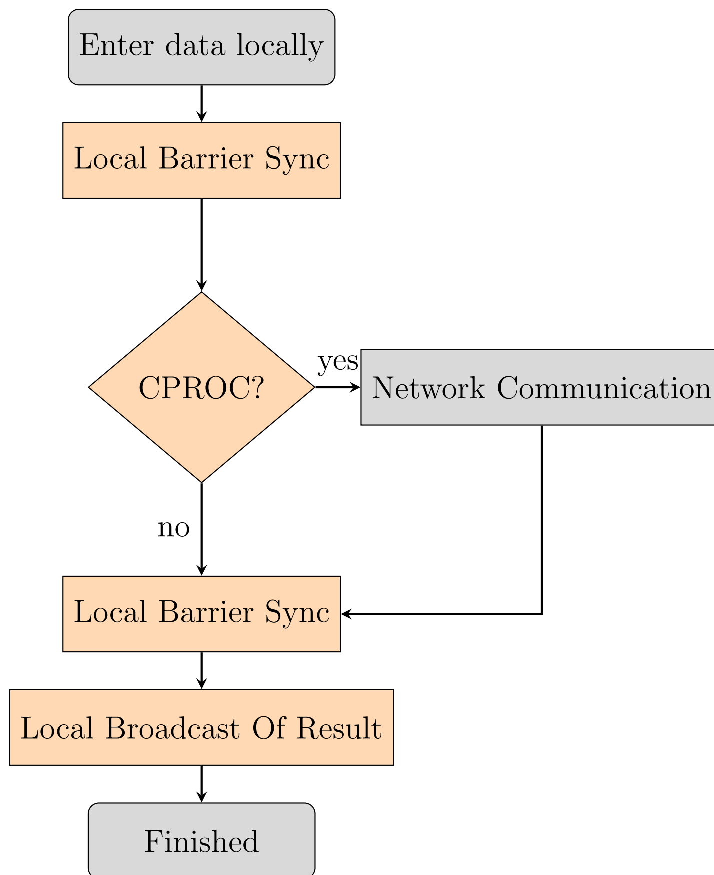


Figure 4.5: Process flowchart for CPROC Behavior Type 3

4.4 Inter-node Communication Algorithm

Following the AFN scheme of All-to-All communication, STAPERS uses one ”broadcast” method for all collectives in the library. Different methods were considered for this purpose.

- Recursive Doubling
- Recursive Halving
- Bruck’s Algorithm
- All-to-All broadcast

Recursive doubling was chosen for this work. Several circumstances contributed to this selection. This work does not concern itself with maximum possible optimization of network latency for the library but to assert, compare and analyze its

competitiveness against more standard approaches. Therefore, for a proof-of-concept implementation, any of the first three algorithms would have sufficed. In order to comply with the aggregate functions methodology, a personalized all-to-all methods is needed for network communication. The most basic method for this "broadcast" necessity is to simply make every node in the network exchange information with every other node. This method is clearly insufficient since it is not scalable. Many more scalable methods have been proposed and used in work like [13], [14] and [15]. These algorithms allow to broadcast information with log(NPROC) passes, making them much more preferable. Given that many use cases for parallel processing use power of two algorithms, a pairwise algorithm was preferred. Bruck's algorithm, recursive doubling and halving are pretty similar and are in use by MPICH[16] and OpenMPI[17] implementations, thus making them the most natural candidates for the personalized all-to-all functionality needed by this work. For most of the collective in AFAPI short messages of less than 256 bytes are used. Since recursive doubling has better performance than Bruck's algorithm with short messages[13], the first one was chosen as the default mechanism for personalized all-to-all. Future work should be concerned on how to make larger messages more efficient as recursive doubling loses its edge against other algorithms as the message size goes up[13].

Chapter 5 Performance Benchmarks and Comparison

To establish the viability of AFNs it is meaningful to concentrate in the simplest performance metric for parallel computing: Latency, specifically the time spent on each collective call not doing computational work. Latency-based comparison is used because the main objective of parallel computation schemes is to minimize the non computational operations. The more time the processor is used for computation the more efficient is the library. In other words, the less time spent on communication the more desirable the scheme is. By comparing latency performance between STAPERS functions and their equivalent in MPI and OpenMP it is possible to establish a frame of reference to compare them by.

5.1 Test Construction

The multi-hierarchy nature of this implementation it is necessary to obtain performance information at all levels of communication. This means, it is necessary to measure inter-node and intra-node latencies separately to obtain meaningful results. This allows a structured method of comparison between STAPERS, OpenMP and OpenMPI. This comparison is not as straightforward as implied since not all functions have a one-to-one match across libraries. For example, while the three of them support reduction operations like OR, AND and XOR, neither STAPERS nor MPI support a subtraction reduction operation like OpenMP. In the other hand, OpenMP does not support scan operations like STAPERS and OpenMPI do which makes one-to-one comparison not trivial. Another more subtle issue is that even when collectives share names among libraries their behavior differ slightly depending on the library. Therefore to accurately compare STAPERS to these libraries functions call are compared depending on the behavior rather than the name of the functions. With this situation in mind the rules of comparison are set as follows:

1. Compare equivalent level communication
2. Compare similar behaved collectives
3. Compare similar data sizes

Table 5.1: Twelve-Core, Two-Socket Shared Memory Computer

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	12
On-line CPU(s) list:	0-11
Thread(s) per core:	1
Core(s) per socket:	6
Socket(s):	2
NUMA node(s):	2
Vendor ID:	GenuineIntel
CPU family:	6
Model:	44
Model name:	Intel(R) Xeon(R) CPU X5650 @ 2.67GHz
Stepping:	2
CPU MHz:	2660.065
BogoMIPS:	5319.78
Virtualization:	VT-x
L1d cache:	32K
L1i cache:	32K
L2 cache:	256K
L3 cache:	12288K
NUMA node0 CPU(s):	0,2,4,6,8,10
NUMA node1 CPU(s):	1,3,5,7,9,11

5.2 Testbeds

Two set of machines were prepared for the experiments. One sixteen-node computer cluster and one twelve-core shared memory system. Tables 5.2 and 5.1 respectively show the specifications of machines. For intra-node communication the machine shown in table 5.1 is used. This machine allows to compare two specific cases for intra-node communication; multi-core and multi-socket communication. For inter-node communication the cluster in table 5.2 was used. Sets of NPROC=4, NPROC=8, NPROC=16 were ran to study the scale issues the current implementation has.

Table 5.2: Sixteen-Node Cluster Computer Specifications

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
CPU(s):	2
On-line CPU(s) list:	0,1
Thread(s) per core:	1
Core(s) per socket:	1
Socket(s):	2
NUMA node(s):	2
Vendor ID:	AuthenticAMD
CPU family:	15
Model:	5
Model name:	AMD Opteron(tm) Processor 248
Stepping:	10
CPU MHz:	2209.974
BogoMIPS:	4419.43
L1d cache:	64K
L1i cache:	64K
L2 cache:	1024K
NUMA node0 CPU(s):	0
NUMA node1 CPU(s):	1

5.3 Intra-node Communications Comparison

For intra-node latency the three libraries were compared as the three libraries have in-node communication capabilities. A section of the test code shown in Listing 5.1 shows the method for comparison . For the case of MPI, we used MPI_Barrier call to make sure all processes were synchronized before timing the operation. This is an important step to obtain the minimum latency across nodes. For OpenMP the thread creation time was estimated in order to create a correction term. Because OpenMP has to create threads every time there is a new parallel block it is necessary to estimate this creation time to adjust the timing of the operations desired. After this correction term is estimated the actual time for the operation is timed once more. Then the correction term is subtracted from the operation time to obtain a more truthful latency of the operation without the thread creation overhead. This process was repeated for the reduction operations OR, AND, XOR, SUM, SUB, MUL, MIN and MAX. The results were then averaged together to created a per-library latency "response". As mentioned before, a reduction substraction operation is not present for MPI nor STAPERS. Since at the processor level a subtraction is just and addition with one of the operators negated the reduction SUM was use in its place.

Table 5.3 shows the results obtained following this method. Figure 5.1 shows the graphic comparison. It can be observed how in all cases up to NPROC=12, STAPERS averages better latency performance than MPI. Against corrected OpenMP values, STAPERS performs equally up to 6 processes. From 7 to 12 processes STAPERS latency term increases more slowly than OpenMP. This graph also highlights the difference between OpenMP with and without the correction term. Without the correction term, OpenMP presents very similar latency to that of STAPERS. This is somewhat expected as OpenMP is highly optimized for this type of operations. This discrepancy can be attributed to the fact that STAPERS initializes processes early in the program while OpenMP does it on demand. STAPERS will have advantage over OpenMP if the OpenMP equivalent program does not take care of thread creation overhead. For the case of MPI things are different. MPI performs worse than both STAPERS and OpenMP(corrected) at all NPROC. MPI does perform better against OpenMP when the correction term is not deducted and the number of processes is a power of two. This indicates that MPI holds the same advantage over OpenMP is the latter does not take care of thread creation overhead. The better performance for power of two number of processes is probably due the internal algorithms being used.

It is safe to assume from this results that the internal algorithms is better suited for power of two number for in node processes.

Listing 5.1: Intra-node Latency Test Code

```
// Code for MPI
MPI_Barrier(MPI_COMM_WORLD);
gettimeofday(&time_start, NULL);
for(i = 0 ; i < TIMES ; i++){
    MPI_Allreduce(&sub_avg, sub_avgs, 1,suffix,
        oper,MPI_COMM_WORLD);
}
gettimeofday(&time_end, NULL);
MPI_Barrier(MPI_COMM_WORLD);


// Code for OpenMP

gettimeofday(&ts, NULL);
for(i = 0; i < TIMES ; i++){
    #pragma omp parallel num_threads(NPROC)
    {
    }
}
gettimeofday(&te, NULL);

gettimeofday(&ts, NULL);
for(i = 0; i < TIMES ; i++){
        #pragma omp parallel for reduction(|:sum) num_threads(NPROC)
        for(j = 0 ; j < NPROC ; j++)
                sum |= a[j];
}
gettimeofday(&te, NULL);
```

Table 5.3: Average Operation Latency Comparison

NPROC	OMP AVG	OMP - Correction	STAPERS AVG	MPI
2	0.8772	0.3508	0.2175	0.4397
3	0.8942	0.3250	0.3610	1.2820
4	1.2141	0.5277	0.5467	1.0228
5	1.3984	0.6410	0.6187	1.6061
6	1.5760	0.6990	0.7387	1.8496
7	1.8053	0.8321	0.7853	2.0753
8	2.2655	0.9783	0.8862	1.5917
9	2.1658	1.0655	0.9304	2.0551
10	2.8509	1.3200	1.0696	2.3641
11	3.0620	1.5184	1.1679	2.5134
12	3.9642	2.0396	1.3079	2.3800

Figure 5.1: Latency

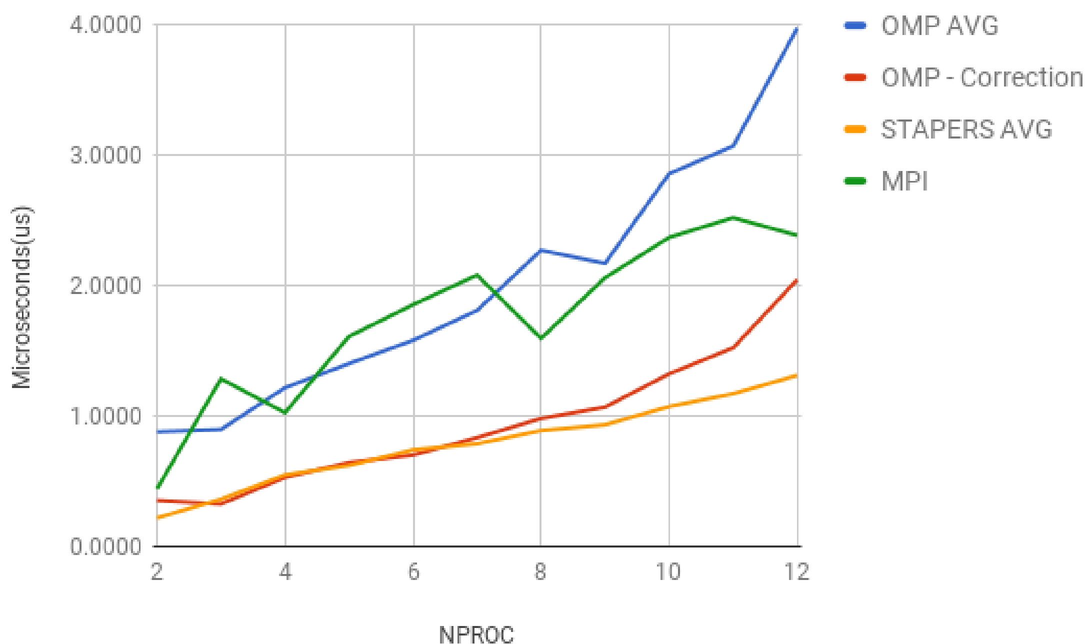


Table 5.4 and Table 5.5 show a split of the differences among reduce operation among the three libraries. NPROC of 6 and 12 are shown. This results lead to think the communication/synchronization is more significant than the operation of itself. With the exception of MIN and MAX all operation can be assumed atomic at the processor level so should not have any difference in latency measurements. Even maximum and minimum operations can be implemented in very few instruction so

Table 5.4: Intra-node Latency NPROC = 6

OP	OMP	MPI	STAPERS
OR	1.04599	1.81717	0.72224
AND	1.02671	1.85684	0.72372
XOR	1.05172	1.83304	0.73252
SUM	1.04306	1.88849	0.72406
SUB	0.99988	1.88849	0.72406
MUL	0.99782	1.85636	0.72221
MIN	0.99534	1.9	0.72881
MAX	1.00251	1.88862	0.72822

the same applies to these operations. This is fact the behavior shown in the results. Figure 5.3 and Figure 5.2 allow a more visually intuitive comparison. Once again the results show clear differences between the three libraries. Finally Figure 5.4 and Figure 5.5 shows a speed up comparison of STAPERS against it counterparts. Both figures show how STAPERS performing much better than MPI. For NPROC=6 Figure 5.4 shows a performance 2.5 faster than that of MPI, while around of 1.8 speed up for 12 processes. For the case OpenMP speedup is rather constant at around 1.5 for NPROC=6 and NPROC=12. This behavior matches the one presented in Figure 5.1. It is worth noting, that when you compare both set of results that at NPROC = 6 one set shows OpenMP actually a little faster than STAPERS. This "anomaly" was actually seen various times across tests. Most of the test showed STAPERS regularly beating OpenMP. Once in a while though, OpenMP would top STAPERS by some margin.

Table 5.5: Intra-node Latency NPROC = 12

OP	OMP	MPI	STAPERS
OR	1.92469	2.39207	1.29096
AND	1.90865	2.44893	1.28751
XOR	1.92014	2.39224	1.29056
SUM	1.88531	2.46541	1.32887
SUB	1.92026	2.46541	1.32887
MUL	1.9205	2.41383	1.24401
MIN	1.97435	2.47827	1.29906
MAX	1.90104	2.47296	1.31272

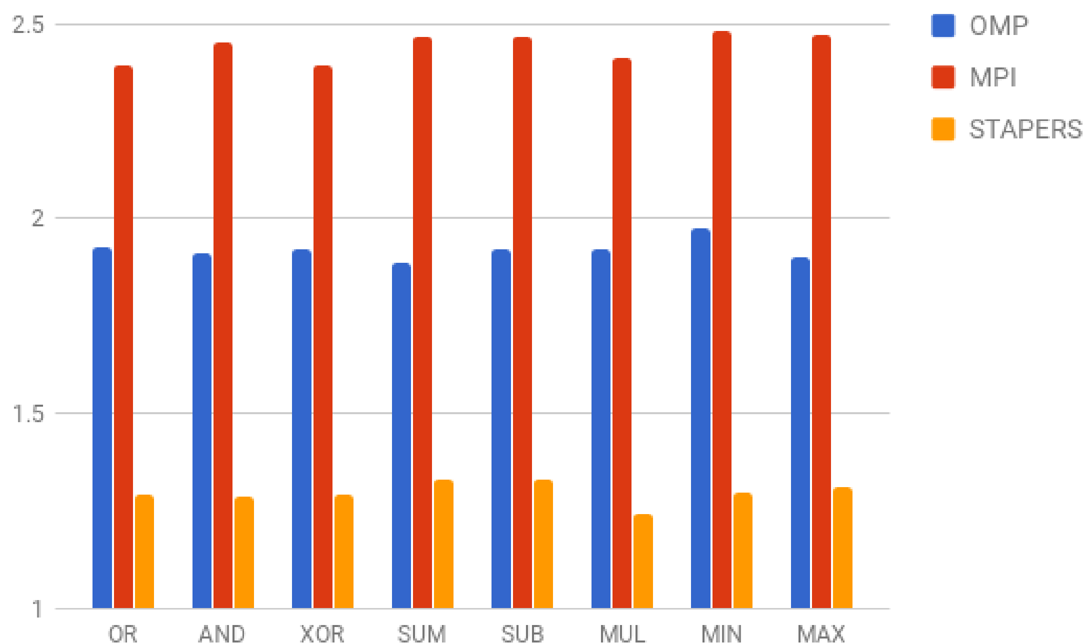


Figure 5.2: Intra-node Latency Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 12

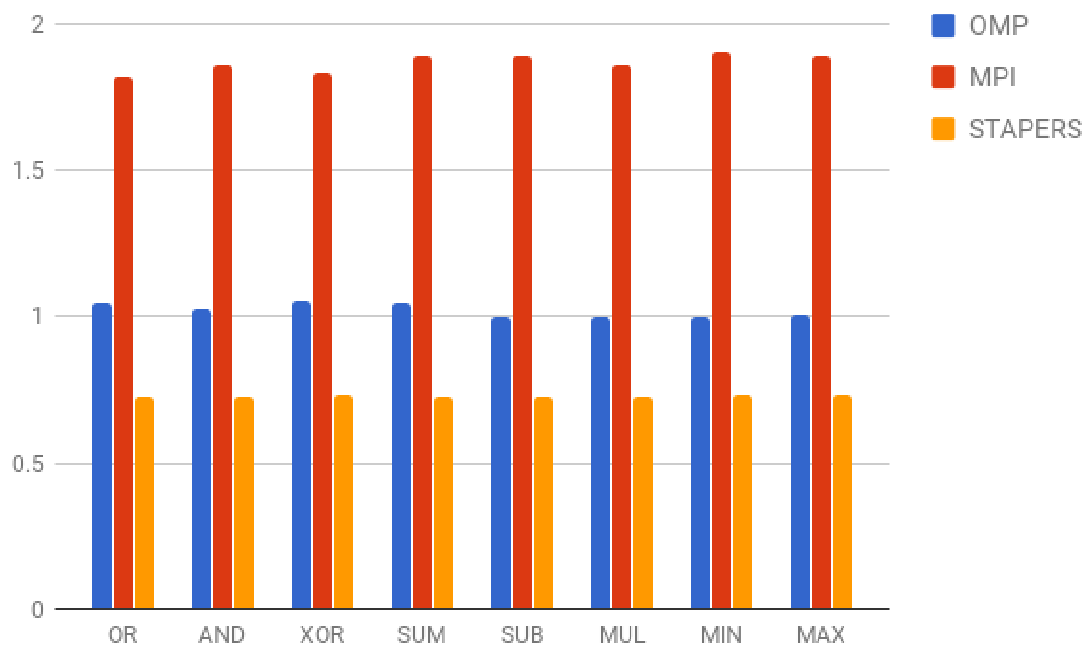


Figure 5.3: Intra-node Latency Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 6

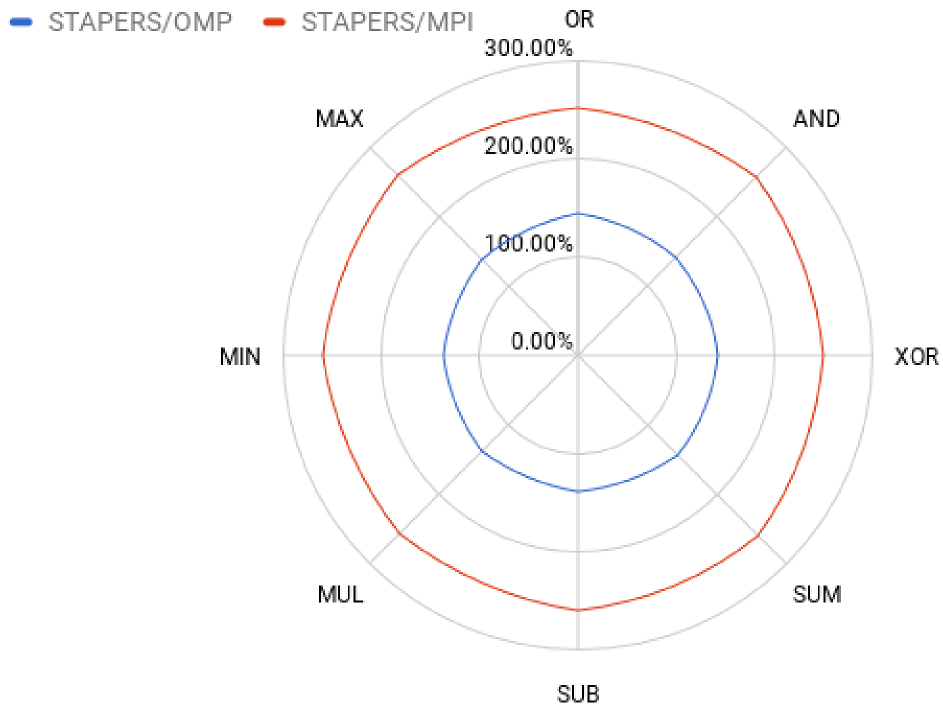


Figure 5.4: Intra-node Percent Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 6

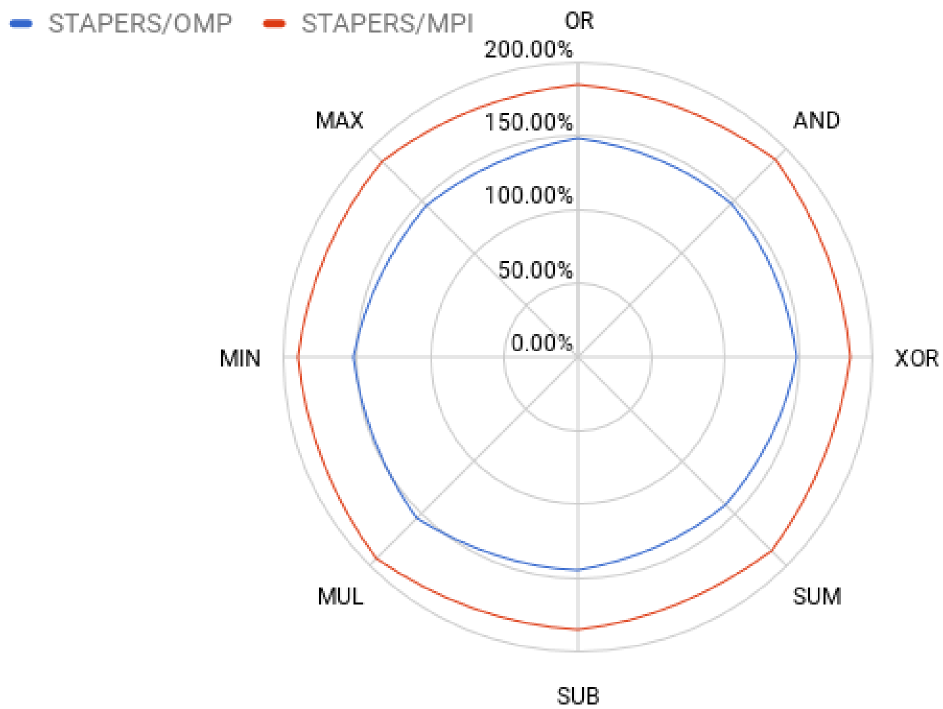


Figure 5.5: Intra-node Percent Comparison STAPERS, OpenMP and OpemMPI For Reduce Operations, NPROC = 12

5.4 Inter-node Communications Comparison

For Inter-node communication STAPERS was compared to OpenMPI only. Results were divided in three different sets of operations. Reduction, scans and communications. The first two are operation as their name implies. That is, reductions and scan operations are compared between MPI and STAPERS. Communication operation are not computational operations but operations with the purpose to share information.

Reduction Operations

One important distinction between STAPERS and its counterparts is the use of a All-to-All based communication scheme. For inter-node communication this distinction is very important. This means that in all collective in STAPERS every nodes involved in the parallel process. Each process will obtain a copy of the information from every other node in the network and act upon it, regardless whether the function returns all the information to the caller or not. For the case of reduction this means that every process ends up with the same result. This makes comparison to regular collectives in MPI meaningless. A Reduction operation on MPI assumes only the root process will have a meaningful result. Every process will call the reduce function but only the root process will have the correct answer. For this reason STAPERS reduce operation is compared to Allreduce MPI operations. Allreduce which exhibits the same behavior as STAPERS. This enables fair comparison between the libraries. Table 5.6 shows the results for reduction operations as stated above. The results shows very similar latency performance between implementations. This behavior is expected as both, OpenMPI and STAPERS uses the same family of algorithms. In general, OpenMPI seems to be slightly slower than our implementation but this is likely due to the larger function overhead of OpenMPI. It is a much more mature library and takes into account many more variables that STAPERS and thus will incur more overhead per call as our more simple implementation. This behavior reflected in the standard deviations of each operation in Table 5.6. OpenMPI is consistently more variable, having standard deviation as high as 73.63 us of and more commonly around 10 us. Compared to this, STAPERS fluctuations are smaller and consistent across test runs. Table 5.4 and Figure 5.4 present more graphical perspective of this effect. Figure 5.4 shows lower how STAPERS is much more consistent in its timing compared to MPI which fluctuates more often.

Table 5.6: Reduce Operation Latency NPROC=32, PCNUM=16

Operation	8-bit	16-bit	32-bit	64-bit	Std.Dev
MPI All Reduce OR	233.67	230.00	352.00	236.00	73.63
STAPERS OR Reduce	233.00	228.00	228.50	224.50	5.01
MPI All Reduce OR	227.00	230.50	225.50	225.00	7.36
STAPERS AND Reduce	223.67	225.50	225.00	225.50	1.42
MPI All Reduce XOR	235.33	235.50	237.00	227.50	8.42
STAPERS Reduce XOR	225.00	225.50	225.50	225.50	0.75
MPI All Reduce MIN	233.67	228.50	223.00	223.50	10.75
STAPERS Reduce MIN	224.00	224.50	225.50	224.00	1.19
MPI All Reduce MAX	221.33	220.00	227.50	218.50	6.45
STAPERS Reduce MAX	224.67	224.50	224.50	223.50	0.78
MPI All Reduce ADD	218.33	229.00	233.00	226.50	11.91
STAPERS Reduce ADD	224.33	224.00	223.50	224.00	1.03
MPI All Reduce MUL	228.00	221.00	234.50	227.50	7.26
STAPERS Reduce MUL	224.00	225.00	224.00	224.00	0.92

Notice that MPI allreduce OR operation is an outlier. This outlier was often seen in test runs. Though the graph shows it to be specific to 32-bit OR operation this behavior was seen randomly across different tests. It is believed though not confirmed that this could be caused by the operative system scheduling out the MPI process. This is more likely to happen in MPI than in STAPERS since MPI library is much more developed and allows for more 'surface area' for which condition could trigger. It is not considered relevant for this works purpose and thus this theory was not pursued nor confirmed. It was left in the results for information purposes.

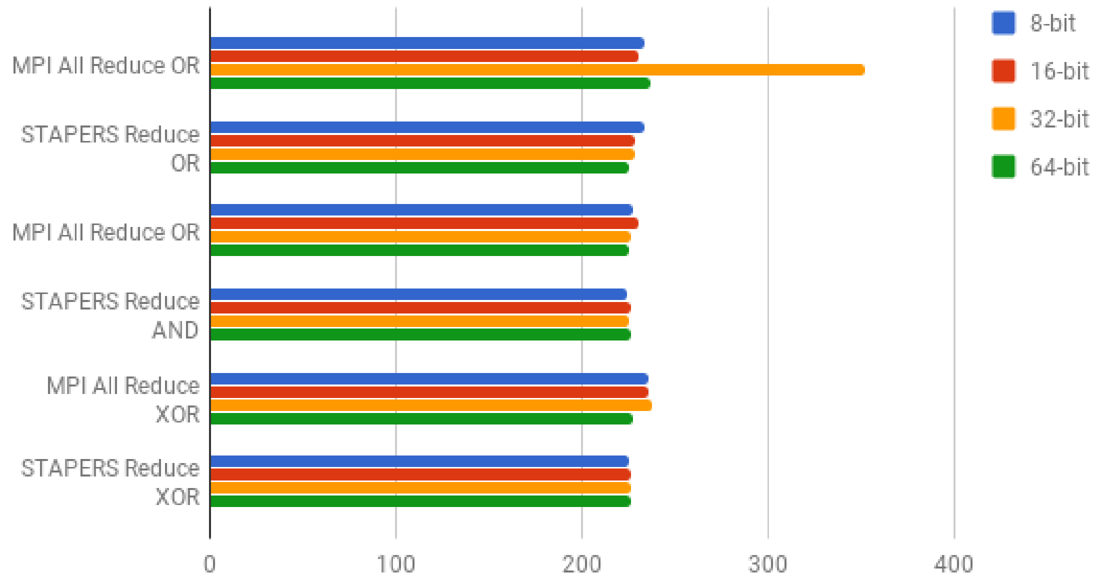


Figure 5.6: Reduce Operation Results Part 1

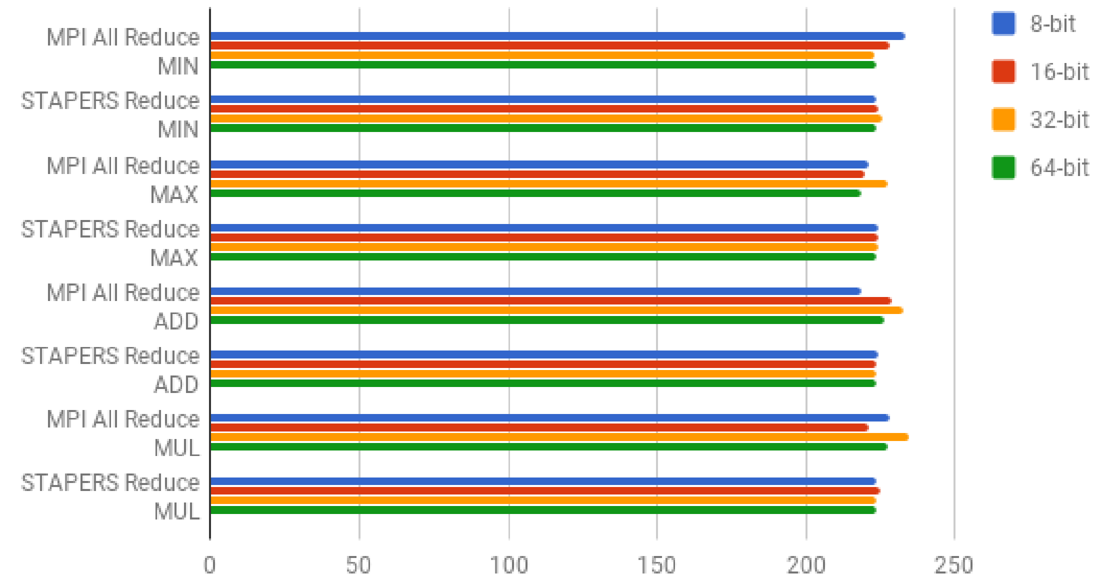


Figure 5.7: Reduce Operation Results Part 2

Table 5.7: Scan Operation Latency NPROC=32, PCNUM =16

Operation	8-bit	16-bit	32-bit	64-bit	Std.Dev
MPI_Allgather	225.86	230.36	224.79	225.71	8.34
ST. Scan OR	224.00	223.50	216.50	216.00	5.87
ST. Scan AND	212.00	219.00	213.50	225.00	6.53
ST. Scan XOR	227.00	226.00	228.50	223.00	3.64
ST. Scan MIN	214.50	215.00	217.67	218.20	2.60
ST. Scan MAX	222.00	222.00	223.00	222.80	1.32
ST. Scan SUM	224.50	223.50	224.33	223.33	0.87
ST. Scan MUL	244.50	249.50	244.33	239.00	7.02

Scan/Gather Operations

Contrary to the case of reduce operations, scan operations may behave differently between STAPERS and OpenMPI, but the result is the same. Every process ends up with a partial reduction result depending on it process number. Scans operation differ no in the end result but the method used to get there. This puts STAPERS in a disadvantage as MPI implementations are much more optimized for this case. Nevertheless Scan operation were compared to two set of functions. Native MPI scan operations which gives the desired results and AllGather funtions which behave similarly to the STAPERS version of the scan collective. Table 5.7 shows the results for 32 processes distributed in 16 computers. This table shows the Allgather vs STAPERS scan case. Once again this is a natural result as the internal algorithms are similar.

Chapter 6 Conclusions and Future Work

The results discussed in the previous chapter indicate that aggregate functions are still a viable method to execute parallel programs and quite competitive as a parallel paradigm. In terms of performance it was shown that the Intra-node implementation to be as competitive as current standards if not potentially better than the top tier libraries. This improvement lies on the fundamental difference in barrier synchronization method and the understating of the common underlying architecture of most modern processors. In the other hand, inter-node performance results tell another story, as latency showed to be similar if not identical to OpenMPI implementation. In hindsight, these results were expected as the underlying broadcast algorithm is an industry standard for personalized all-to-all communication. Regardless, STAPERS shows that from a performance perspective aggregate functions still provide value to HPC community. From a qualitative perspective aggregate functions provide a different thinking model for synchronization. This alternative may very well be biggest value aggregate functions provide the HPC community as more parallel computation move to be more fine/medium-grain parallelism. This makes the case for message passing paradigm less strong and may open the doors for AFNs. With that being said, STAPERS still lack the maturity level than the MPI and OpenMP standards. As a technology, still has much room to improve to be used . Regardless of this reality this work has shown its potential and encourages the continuation of its development. This improvement can take several different forms as the paradigm is not uniquely suited to network computations.

One area future work can take is to improve the current rudimentary TCP communication. Current implementation status makes very little optimization based on compile time information like process number or communication patterns. Many current MPI implementations make use of the compile time information for choosing more efficient aggregation algorithms. In particular all-to-all communication and scan operations can be improved upon to be more efficient in STAPERS. Another potential improvement is to make more run-time optimizations for communication. For example, current state of the library uses recursive doubling for all communication regardless of number of processes and buffer sizes. Using on-time information to change communication to use more efficient algorithms is a path that libraries like

MPI and OpenMP have already taken.

Another direction this work can lead to is the change from TCP-based communication to UDP-based communication. Using true UDP broadcast to serve as the communication mechanism may further improve latency performance. By using UDP broadcast as a barrier signaling mechanism, it is possible to effectively make a constant pass network communication scheme for any number of processes. This is possible despite UDP being intrinsically unreliable because current hardware can compensate for such shortcomings. UDPAPERS had to tackle this challenge before and this work only encourages its continuation.

Another potential use of aggregate functions can be its migration to network equipment. That is, to use the network switch to serve as the barrier unit to which each computer reports to. This effectively moves aggregate computation to the network and it's is actually the original idea for the PAPERS unit back in the 90s. This idea has further developed its validity as companies like Mellanox have been aggressively promoting In-Network computing technology which seems to be similar to this idea. One candidate considered for this work was to obtain a MetaMako's FPGA-based switch. This machine allows to make use of an internal FPGA to control packets at layer 2 and 3 at nanoseconds speed. This hardware makes for an ideal testbed for a aggregate function layer to build upon.

Bibliography

[1] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[2] R. R. Hoare and H. G. Dietz. A case for aggregate networks. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 162–166, Mar 1998.

[3] R. Rabenseifner, G. Hager, and G. Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, pages 427–436, Feb 2009.

[4] Lei Chai. *High performance and scalable MPI Intra-node Communication middleware for multi-core clusters.* PhD thesis, The Ohio State University, 2009.

[5] N. Drosinos and N. Koziris. Performance comparison of pure mpi vs hybrid mpi-openmp parallelization models on smp clusters. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pages 15–, April 2004.

[6] Rudolf Berrendorf and Guido Nieken. Performance characteristics for openmp constructs on different parallel computer architectures. *Concurrency Practice and Experience*, 12(12):1261–1273, 2000.

[7] Jay P Hoeflinger. Extending openmp to clusters. *White Paper, Intel Corporation*, 2006.

[8] H. G. Dietz, R. Hoare, and T. Mattox. A fine-grain parallel architecture based on barrier synchronization. In *Proceedings of the 1996 ICPP Workshop on Challenges for Parallel Processing*, volume 1, pages 247–250 vol.1, Aug 1996.

[9] Henry G Dietz, Tai M Chung, and TI Mattox. A parallel processing support library based on synchronized aggregate communication. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 254–268. Springer, 1995.

[10] Henry G Dietz, TI Mattox, and G Krishnamurthy. The aggregate function api: It's not just for papers anymore. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 277–291. Springer, 1997.

[11] T. B. Berg and H. J. Siegel. Instruction execution trade-offs for simd vs. mimd vs. mixed mode parallelism. In *[1991] Proceedings. The Fifth International Parallel Processing Symposium*, pages 301–308, Apr 1991.

[12] T Mattox. Synchronous aggregate communication architecture for mimd parallel processing. *School of Electrical and Computer Engineering. W. Lafayette, IN: Purdue University*, 1997.

[13] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of collective communication operations in mpich. *The International Journal of High Performance Computing Applications*, 19(1):49–66, 2005.

[14] J. Pjesivac-Grbovic, T. Angskun, G. Bosilca, G. E. Fagg, E. Gabriel, and J. J. Dongarra. Performance analysis of mpi collective operations. In *19th IEEE International Parallel and Distributed Processing Symposium*, pages 8 pp.–, April 2005.

[15] J. Bruck, Ching-Tien Ho, S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, Nov 1997.

[16] Mpich : High-performance portable mpi.

[17] Open mpi: Open source high performance computing.

Vita

Pablo Quevedo was born in Valencia, Venezuela. He obtained his Bachelor Degrees in Electrical Engineering and Computer Engineering from the University of Kentucky. He enrolled in the University of Kentucky's Graduate School in Fall 2013. He worked as a Research Assistant for University of Kentucky in summer 2015. He is currently working for Hitachi America, Ltd as a Research Scientist.