(CM)$^2$PI : Compose-Map-Configure MPI

Design and Implementation of a Scalable Parallel Architecture Specification and its Deployment

by

Ryan Ki Sing Chung

B.Sc., The University of British Columbia, 2012

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in

The Faculty of Graduate and Postdoctoral Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

November 2014

© Ryan Ki Sing Chung 2014
Abstract

In order to manage the complexities of Multiple Program, Multiple Data (MPMD) program deployment to optimize for performance, we propose (CM)$^2$PI as a specification and tool that employs a four stage approach to create a separation of concerns between distinct decisions: architecture interactions, software size, resource constraints, and function. With function level parallelism in mind, to create a scalable architecture specification we use multi-level compositions to improve re-usability and encapsulation. We explore different ways to abstract out communication from the tight coupling of MPI ranks and placement. One of the methods proposed is the flow-controlled channels which also aims at tackling the common issues of buffer limitations and termination. The specification increase compatibility with optimization tools. This enables the automatic optimization of program run time with respect to resource constraints. Together these features simplify the development of MPMD MPI programs.
Preface

This thesis is original work by the author Ryan Ki Sing Chung who discussed with Dr. Alan Wagner, Dr. Humaira Kamal, Sarwar Alam, and Imran Ahmed. The programming library FG-MPI is developed by Dr. Humaira Kamal. This thesis incorporates suggestions and comments from Dr. Ronald Garcia.
Table of Contents

Abstract ................................................................. ii
Preface ................................................................. iii
Table of Contents ....................................................... iv
List of Figures ........................................................ viii
Acknowledgements ...................................................... xiii
Dedication ............................................................... xiv

1 Introduction .......................................................... 1

2 Background .......................................................... 4
  2.1 MPI ................................................................. 4
  2.2 FG-MPI ............................................................ 5
  2.3 MPI Eager Send ................................................... 7
  2.4 Related Work ...................................................... 8
    2.4.1 Component-Based Software Engineering ................. 8
    2.4.2 Channel Communication ................................. 8
    2.4.3 Flow Control ............................................... 9
    2.4.4 Computational Resources and Placement of Work . 10

3 Composition .......................................................... 12
  3.1 Multiple Layered Specification (CPA Architecture) ....... 12
    3.1.1 Specifying the Communication Process Architecture
           (CPA) ...................................................... 12
# Table of Contents

3.1.2 The Assembly Line (The Four Stages) . . . . . . . . . 18

3.2 Communication . . . . . . . . . . . . . . . . . . . . . . . . 23

3.2.1 Process Discovery . . . . . . . . . . . . . . . . . . . . 23

3.2.2 Composition-Based Hierarchical Communicators . . . 25

3.2.3 Channels . . . . . . . . . . . . . . . . . . . . . . . . . 26

3.2.4 Flow-Controlled Channels . . . . . . . . . . . . . . . 30

3.3 Encapsulation of Compositions . . . . . . . . . . . . . . . . 41

3.4 Composition of Programs . . . . . . . . . . . . . . . . . . . 42

4 Placement . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.1 Combination of Concurrency and Parallelism . . . . . . . . 47

4.2 Scaling to Larger or Smaller Resources . . . . . . . . . . . 50

4.3 Heterogeneous Computation Load . . . . . . . . . . . . . . 51

4.4 Glue Functions . . . . . . . . . . . . . . . . . . . . . . . . . 52

4.4.1 A: Main Init . . . . . . . . . . . . . . . . . . . . . . . . 53

4.4.2 B: Map Init . . . . . . . . . . . . . . . . . . . . . . . . 53

4.4.3 C: Map Finalize . . . . . . . . . . . . . . . . . . . . . . 53

4.4.4 D: Get Parameters . . . . . . . . . . . . . . . . . . . . 53

4.4.5 E: Communicator Permutation . . . . . . . . . . . . . . 54

4.4.6 F: Free Parameters . . . . . . . . . . . . . . . . . . . . . 53

4.4.7 G: Main Cleanup . . . . . . . . . . . . . . . . . . . . . . 54

4.4.8 Glue Code Use Cases . . . . . . . . . . . . . . . . . . . 55

4.5 Permutation of Processes . . . . . . . . . . . . . . . . . . . 56

4.5.1 Communicators Permutation Callback Function . . . . 56

5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58

5.1 Integration with Optimization Tools . . . . . . . . . . . . . 58

5.1.1 Proof of Concept . . . . . . . . . . . . . . . . . . . . . 59

5.2 Permutation of Processes . . . . . . . . . . . . . . . . . . . . 60

5.3 Adapting to Resource Size . . . . . . . . . . . . . . . . . . . 60

5.4 Integration of Separately Developed MPI Programs . . . . 61

5.5 Performance Impact Evaluation . . . . . . . . . . . . . . . 61

5.5.1 Process Mapping Storage Requirement . . . . . . . . . 61
## Table of Contents

5.5.2 Process Discovery ........................................... 62  
5.5.3 Hierarchical Communicators ............................... 63  
5.5.4 Channels Information .................................... 63  
5.5.5 CMCMPI_Init ............................................... 63  
5.5.6 Flow-Controlled Channels ................................ 64  

6 Conclusion ....................................................... 66  
6.1 Future Work .................................................. 67  

Bibliography ....................................................... 70  

Appendices  
A Specification Notation ......................................... 74  
A.1 Composition Notation ......................................... 74  
A.1.1 MPI Process Declaration ................................. 74  
A.1.2 Concurrent Composition of MPI Processes .......... 74  
A.1.3 Concurrent Composition of Compositions .......... 75  
A.1.4 Intermediate Composition .............................. 75  
A.1.5 Communication Channels ................................. 75  
A.1.6 Port Hiding ............................................... 76  
A.1.7 Patterns for Communication Port Declaration .... 76  
A.1.8 Forming a Structure ..................................... 77  
A.1.9 Variable Initialization .................................. 77  
A.1.10 Initializing All Variables .............................. 77  
A.2 Placement Notation ........................................... 78  
A.2.1 Container Specification ................................. 78  
A.2.2 Container Allocation to OS Processes ............... 78  
A.2.3 Forming a Container Specification .................. 78  
A.2.4 Mapping Functions to MPI Processes ............... 79  
A.2.5 Glue Specification ...................................... 79  
A.2.6 Environment Variables .................................. 80  
A.2.7 Combining Specifications ............................... 80
Table of Contents

B API ......................................................... 81
  B.1 Process Discovery API ................................. 81
  B.2 Communicators API .................................. 82
  B.3 Channels API ....................................... 83
  B.4 Flow-Controlled Channels API ....................... 84

C Examples .................................................. 86
  C.1 Flow of Execution ................................... 86
     C.1.1 Compiling the Generated Code .................. 87
  C.2 HelloWorld Example ................................. 87
     C.2.1 MPI ............................................... 87
     C.2.2 FG-MPI ............................................ 88
     C.2.3 FG-MPI with (CM)^2PI ............................ 90
  C.3 PrimeSieve Example ................................. 92
  C.4 Farm Example ........................................ 98

D Code sketch - Communicator Permutation ................. 101
List of Figures

2.1 The above launches 16 processes (8 x executable1), (4 x executable2), and (4 x executable3) on a 4 node cluster where each node has 4 cores. Processes are assigned to cores by filling each node in order. ........................................... 5

2.2 FG-MPI adds extra flexibility on placing MPI processes. However, a mapper needs to be written for each change in mapping. In the above, 4 OS processes are launched (1 with 8 MPI processes), (2 with 2 MPI processes), and (1 with 4 MPI processes) on a cluster with 2 nodes each with 2 cores. ........................................... 6

3.1 CPA Example of a prime sieve. The key aspect of this specification include the declaration of the processes and their compositions. This notation is similar to that of FSP (Finite State Process) in Kramer and Magee [28]. ..................................................... 13

3.2 A description of the architecture would indicate a variable sized sieve element section with 1 generator and 1 last element. The generator is PrimeStart in the specification and the last element is PrimeEnd in the specification. .................. 13

3.3 CPAV Example of a prime sieve. Strings can be used as values for function parameters only. ................................................................. 15
List of Figures

3.4 Container Specification Example of a prime sieve. Constraints of what MPI processes can go into each OS process as well as the number of OS processes is defined. In this example, there are 160 OS processes in total. One of them will have a PrimeStart process and as many PrimeElement process as needed. One of them will have a PrimeEnd process and as many PrimeElement processes as needed. Finally, 158 OS processes will each have as many PrimeElement processes as needed. ........................................ 16

3.5 Function Spec Example of a prime sieve ........................................ 17

3.6 The four stages. At each stage of the tool’s execution, an input file is used to combine and further initialize the system. 18

3.7 The MPI ranks are assigned to the processes in the order they are defined in the execution command. A separate rank is given for each nfg value. ................................. 20

3.8 Wrapper functions are provided for each function in the program. The compressed arguments list from the execution command is decoded and then passed on to the user’s function. 21

3.9 The project folder organization. The files generated are contained in a sub-directory of the project. ....................... 22

3.10 Visualizing the CPA using a tree of composition. The naming can be seen clearly here by following from the root to the corresponding MPI process. ................................. 24

3.11 Visualizing the CPA. A name is given to each level of composition as well as each process. ................................. 24

3.12 MPI process information available for packing. The compositions are expanded out and only processes with their full name is kept for the next steps. ................................. 24

3.13 Container Specification defines requirements on the contents of the container. ................................. 25

3.14 OS Processes are formed and placed adhering to container restrictions. ................................. 25
List of Figures

3.15 Channels are specified by connecting port to port. Intermediate ports are used when creating channels through compositions. The grey circles are external ports (intermediate ports visible outside of a composition) while the white circles are internal ports (an MPI process port visible only inside the composition). ................................................... 27

3.16 The types of channels we consider. A) One-to-one channels with a single source and single destination, B) One-to-many channels where there is one source but many destinations, C) Many-to-one channels where there are many sources sending to a single destination ........................................ 34

3.17 Flow-controlled channel communication for a one-to-one channel. ....................................................... 35

3.18 Flow-controlled channel communication for a one-to-many channel. The proxy is co-located with the source to reduce the effect of the communication between the source and the proxy. ........................................ 36

3.19 Flow-controlled channel communication for a many-to-one channel. The proxy is co-located with the destination. Data must travel from the proxy to the singular destination in this channel construct so the goal is to minimize the communication overhead of sending data from the proxy to the destination. 37

3.20 The typical messaging between source, destination and proxy when the sender initiates the termination of the channel. ... 38

3.21 The typical messaging between source, destination, and proxy when the receiver requires the termination of the channel. Notice that the receiver can only request and not demand termination. It is still up to the sender to terminate the channel. ......................................................... 39

3.22 To optimize for performance, communication between co-located processes can be simplified as the communication is always synchronous and flow-control is not an issue. ................. 40
### List of Figures

3.23 Creation of channels through composition. The white circles represent ports while the grey circles represent intermediate ports explicitly exposed by a composition. ........................................... 43

3.24 Assuming there are 3 process types: A, B, and C. If these processes form a chain A) and we want to duplicate it, then the new chain must take a new set of rank numbers where B) and C) are possibilities. Traditional methods of communication using ranks require the developer to consider and handle these possibilities. ................................................................. 44

3.25 A prefix can be used to identify the processes of a particular composition. ................................................................. 46

4.1 The use of wildcard in the architecture initialization gives freedom to place as many processes as need such as the Master process. The use of wildcard in the container specification allows for freedom to place a fixed number of MPI processes (as specified in the architecture) to be placed evenly on the resources available. ................................................................. 48

4.2 Loops can be used to create a simple pattern in the number of processes placed into each container. ................................. 49

4.3 The prefix of a composition can be specified in the container specification in remove ambiguity of process placement. .... 50

4.4 Separation of concerns in the specification allows concise decisions to be changed. The tool automatically fits the program into the resources available and generate the corresponding mapping. For large programs with many processes, changing the mapper and configuration manually is error-prone and grueling. ................................................................. 51
List of Figures

4.5 The permutation of processes using communicators. The dark blue circles represent processes and the white circles represents a rank naming at that communicator. Processes are permuted (green arrows) at the lowest level and the permutation propagates back up. Any channel information (red arrows) is based on the virtual ordering in the new communicators.

C.1 The order in which the tool is run from compilation to execution of the program.

C.2 The generator needs to send out numbers. The sieve elements receive numbers from the front of the sieve and pass numbers on. The last element needs to terminate the program when it receives a prime. The control channel is used for optimization of the termination process.

C.3 A description of the architecture would indicate a variable sized sieve element section with 1 generator and 1 last element.

C.4 Three patterns of composition. A farm can be by itself A), chained B), and nested C).
Acknowledgements

I would like to express my deepest gratitude to my supervisor, Dr. Alan Wagner, for guiding me through my graduate studies. His passion inspired me to explore fascinating topics. His knowledge, resourcefulness, and kindness supported me through countless obstacles.

I would also like to thank Dr. Ronald Garcia for reading my thesis and providing insightful suggestions and comments.

Thank you Dr. Humaira Kamal for her work on Fine-Grain MPI which motivated my research. Her suggestions and support was indispensable.

Thank you Dr. Nelson Souto Rosa for giving me interesting suggestions for improving my research.

To all my friends, I thank you so much for all the memorable moments we’ve shared. Special thanks to my closest lab buddies: Sarwar Alam, and Imran Ahmed.

Finally, thank you so much to my parents, sister, and aunt for their continuing support, love, and encouragement.
Dedicated to my parents, sister, and aunt.
Chapter 1

Introduction

Today’s CPUs are no longer increasing significantly in clock speed but rather in the number of cores available. In order to optimize for performance and to scale to larger problems on a cluster of multicore nodes, it is necessary to use parallelism. A problem that is largely neglected is the placement of processes to reduce communication overhead needed to keep compute cores busy.

Message Passing Interface (MPI) is an example of a library where there is complete freedom in placing processes on compute resources. MPICH from Argonne National Laboratory [26] is middleware that implements the MPI standard library. The term MPI middleware is used to refer to the MPI runtime, tools, and support for communication. As MPI is built for performance, processes run “hot” (constantly polling for messages) in order to minimize overheads. As a result, it is usual that the number of processes needs to match the number of cores. The OS scheduler views every process as independent and [18] found that performance is decreased by 10% when applications are run with two threads per core.

Fine-Grain MPI (FG-MPI) is an extension of the MPICH middleware which supports function-level parallelism where each MPI process is responsible for a function rather than a whole executable. The novelty is in the ability to place more than one MPI process in a single OS process. The MPI processes inside the OS process are scheduled by a lightweight scheduler which is aware of the progress of each MPI process. This added layer of flexibility creates a new dimension for the specification of the placement. However, we do need a way to specify which MPI processes should be co-located and where.

There are frequent computation and communication patterns [29] that
exist in large programs. This is especially true in scalable programs as it is rare for every process to have distinct functions in a program with thousands of processes. When using FG-MPI, these patterns are commonly based on the size of the program and the resources. Yet, it is time-consuming and error-prone to specify these patterns through code. We need a way to externalize aspects of the code such that programs can be reused regardless of their placement.

In Jeff Squyres’ blog [32], he stressed the importance of what goes on to set up the MPI environment before calling MPI_Init. But what’s also important is the configuration that happens before setting up the environment.

This research proposes (CM)^2PI (Compose-Map-Configure MPI), a scalable parallel program specification based on a four-stage approach to decouple the software architecture and deployment decisions. A tool is also proposed to generate code based on this abstraction as wrappers around the users’ functions. The main focus is on being able to configure parallel program placement for performance.

This thesis makes the following contributions:

- We propose a four stage approach to program specification. This method allows decisions on architecture interactions, software size, resource capacity, and function to be separately specified in order to separate concerns from the design to its deployment.

- We propose a component-based architecture to allow parallel design patterns to be reused. A multi-level composition design also allows for encapsulation of a group of functions for further compositions with separately developed applications. This is the cooperation of small parts to perform a bigger role.

- We explore three different ways to specify communication in order to eliminate the reliance on MPI’s rank numbers in order to allow flexibility in placement and the separation of placement and programming. We explore three types of communication specification: a name-based
process discovery library, a hierarchical communicator library, and a library based on unidirectional communication channels.

• We investigate the inclusion of a flow-controlled channel library in order to support communication termination (an agreement problem) as well as reducing the chance of memory exhaustion at the receiver’s buffer.

• We integrate the program specification with an optimization tool to automatically place MPI processes on the resources available.

The rest of the thesis is organized as follows: Chapter 2 explains some of the background information needed to understand the problem and techniques, Chapter 3 explains the programmability aspect through composition, Chapter 4 presents the performance aspect through placement, and Chapter 5 discusses the effectiveness of the proposed solution. In the appendix, we present the detailed specification language, API, and three examples: a hello world program (shows basic usage), a prime sieve program (shows channels usage), and a farm program (shows composition, chaining, and channels usage).
Chapter 2

Background

2.1 MPI

Message Passing Interface (MPI) [3] is a standard API for creating parallel
distributed memory programs that share information through passing mes-
sage between processes. The implementation of MPI used in this research
is an open source library and middleware called MPICH [26].

MPICH programs are launched by the `mpiexec` command, which maps
MPI processes to the compute cores available (see Figure 2.1). A `machinefile`
listing the available processing cores and their locations must be created first.
Then, each different process is created as a separate executable that uses
the MPI library. Finally, a command is executed to specify the number of
each executable to launch as processes and their ordering. Processes are
then created based on the order they are specified in the command and then
mapped to the compute cores in the order specified in the `machinefile`.

All MPI processes have a default communicator called `MPI_COMM_WORLD`.
A communicator is a communication scope which assigns a unique rank
number to identify each member process. This rank number is used as a
target address when sending point to point messages. The default mes-
saging type in MPI is asynchronous. Information can be passed through
end-to-end communication (sending to a specific rank) or through collective
communication (broadcast, scatter, gather) within a communicator.

If more MPI processes are launched than there are cores available, then
these processes are time-shared by the operating system’s scheduler. MPICH
can run on a cluster with multiple compute nodes each with multiple cores
or on a single machine with one or more cores. At launch time, the pro-
cess manager Hydra starts each process on its assigned node while launch
communication with other processes is typically through SSH. This allows
the programmer to statically assign processes in order to optimize for locality when communication latency is important. In the case of heterogenous clusters with expert knowledge and manual adjustment, it is possible to put processes with fewer tasks on slower cores.

2.2 FG-MPI

Fine-Grain MPI (FG-MPI) [23, 25] is an extension of MPI that allows more than one MPI process to share a single processing core. FG-MPI’s integration with the MPICH middleware allows MPI processes to be defined as functions rather than being separate executables. These functions can be packaged into a single executable. The execution command is similar to MPI where the number of OS processes is specified using a -n flag but a new flag -nfg is added to support specifying the number of MPI processes (the number of MPI ranks) to be placed inside each OS process. Since more than one MPI process can be co-located inside a single core, a mapper needs to be written to specify which function should be assigned to each MPI rank within each core (see Figure 2.2).
FG-MPI adds extra flexibility on placing MPI processes. However, a mapper needs to be written for each change in mapping. In the above, 4 OS processes are launched (1 with 8 MPI processes), (2 with 2 MPI processes), and (1 with 4 MPI processes) on a cluster with 2 nodes each with 2 cores.

MPI allows parallelism of processes amongst the available computation cores. However, it is difficult to ensure that cores are never idle or to minimize communication overhead for computation with dependencies (i.e., a process must wait for input before computing). A common technique used to combat this is by programmatically integrating multiple tasks into a single process through the use of conditionals and manually managed time-sharing to switch between tasks. This is cumbersome and is detrimental to the readability and maintainability of the code.
FG-MPI allows the developer to put more than one MPI process on the same physical compute core. This enables processes to run concurrently as well as in parallel. Each MPI process can be programmed separately regardless of the number of physical compute cores available because their concurrent execution can be handled by the FG-MPI scheduler inside the middleware.

With function-level parallelism, code represents the software structure while at the same time giving the flexibility to co-locate processes to minimize communication and the amount of idle cycles of a core. With this new opportunity to customize the placement of processes in parallel and concurrently comes the need for a tool that makes the process of placing MPI processes manageable by breaking away from the need to use MPI rank numbers to identify processes.

### 2.3 MPI Eager Send

MPI has two types of protocol for sending messages: Eager and Rendezvous.

The Rendezvous protocol is used when messages are larger than an implementation-specific eager limit. In this case, an envelope of the data is sent to the receiver first. When the receiver indicates that it has sufficient buffer space to receive the data, it is then sent. This method ensures that buffer space is not exhausted on the receiver side but has extra overhead for the synchronization.

On the other hand, small messages are sent using the Eager protocol. In this case, messages are sent to the receiver assuming there is sufficient buffer space in the middleware to buffer messages that are not yet received by the sender. This method increases performance through uni-directional asynchronous communication but there are potential problems with this method.

One is the assumption that the there is sufficient buffer space on the receiver side may not be true. Buffer space can run out or main memory can be exhausted in the case of resizable buffers. Another issue is that as messages queue up in the unexpected message queue, it adds to queueing
2.4. Related Work

2.4.1 Component-Based Software Engineering

Component-based software engineering [6] suggests that the usage of components to encapsulate related functions improves application, component, and function re-usability as well as maintaining compatibility between different systems through modularity, cohesion, and low coupling. By communicating through precise interfaces, this allows the components to be replaced without the need to know the internals of each component. Other benefits include the ability to test components separately as well as the simplicity of creating precise documentation. This approach is widely used in software development.

This technique can also be applied to parallel programming. Traditional MPI code has a single “program” which when run determines its task based on what rank it is. With FG-MPI, we can define functions as processes. Combining this with component-based software engineering, we can compose a set of functions (or MPI processes) to perform a simple task while encapsulating all the details of its internals. These simple tasks can then be further composed into more complex tasks and so on. Code organization and maintainability are also important factors to the success of a parallel program.

2.4.2 Channel Communication

Channels are a way for communication between two endpoints. They are described in CSP [15] and used in many programming languages like Go [1].

delays and search time as described in [34].

Since eager messages have a lack of flow-control, we propose a compromise that works on top of the eager method by allowing a certain amount of asynchronous messages dependant on the number of tokens provided by the receiving end. This means that once in a while, a message in reverse needs to be sent to recycle tokens.
and Erlang [4]. A synchronous channel is achieved using a rendezvous to exchange data while an asynchronous communication channel is created to be a buffer that processes can send data into and processes can retrieve data from. This is similar to the publish and subscribe pattern where senders are publishing to this buffer while receivers are subscribing to its content. One advantage of using channels is that the sender does not have to worry about where the endpoint of the data should go. Likewise, the receiver does not have to know the exact source of the message. In other words, a channel abstracts away the end-to-end communication by using a named entity for the transport of the message.

### 2.4.3 Flow Control

Flow control is the management of the amount of data that can be sent from source to sink while taking into consideration the speed at which the sink is able to consume this data and the speed at which the source is able to generate this data.

This is important when data is generated quicker than it is consumed. The goal of flow control is to make sure the sink is not overwhelmed with messages while at the same time ensuring that the speed of communication is near its optimal performance. Flow control is commonly used in networking protocols, e.g., TCP.

With MPI it is not possible to drop packets as it would cause the whole program to fail. Furthermore, with the preference for performance, messages sent eagerly assume that there is sufficient buffer space to store the message when the receiver is not ready. Some user level techniques currently used include synchronous sends, polling, and token systems.

Synchronous sends wait for acknowledgement that a message is received before carrying on with computation. Polling and token systems have the sink requesting the source for messages and thus have control over the number of messages forwarded. Obviously, all of these techniques incur extra messages and degrade performance.
2.4. Related Work

2.4.4 Computational Resources and Placement of Work

The end of Dennard’s Scaling [7] showed that manufacturers must look to adding more cores rather than speeding up cores. To exploit this dimensional change in resource growth and to perform well, programs must now be written to use multiple cores and nodes. New issues must be considered as communication incurs larger penalties and synchronization of tasks across multiple cores is required to cooperate.

Many programming languages were developed to aid developers in thinking concurrently and to structure their code for concurrent computation. However, it appears that the support available focused only on the programmability aspect and not the need to efficiently place these computations.

Two major influences on performance are the amount of idle processing overhead, when available processing cores are not doing useful computation due to other dependencies, and communication overhead, time spent waiting due to passing of data and messages in order to cooperate.


These techniques are sufficient for Single Program, Multiple Data (SPMD) programs where most tasks take the same amount of time. However, for Multiple Program, Multiple Data (MPMD) programs, the amount of time a process takes to execute varies greatly, which means some of these processes can benefit from not sharing a core while others prefer the opposite.

Any type of dynamic load balancing technique can introduce unnecessary context switches through transfer of work from one location to another. Also, simply placing work based on a core’s workload neglects how important communication overhead is for some applications and the locality of the processes can play a larger factor than the compute power. Furthermore, the performance of processes may be known ahead of time and thus there is no point in migrating processes during execution. Perhaps dynamic load
balancing may be more adaptive, but optimization tools may be able to find static placement of the processes.

In other words, existing techniques to distribute processes work well for applications such as process farms but are not appropriate as a single global technique especially for MPMD applications where services may stay alive and communication amongst different actors play an important role in the overall performance of the system. To bring performance benefits to concurrent programs, we must also think parallel.
Chapter 3

Composition

3.1 Multiple Layered Specification (CPA Architecture)

With the current situation where the execution command (configfile) and process mapping is so highly coupled, we need to find a way to abstract out the different parts. We use the separation of concerns design principle to design four distinct phases to specifying an FG-MPI program. The aim is to have each of the stages be highly cohesive and loosely coupled.

3.1.1 Specifying the Communication Process Architecture (CPA)

We propose four separate specifications to describe the system.

a) The Communication Process Architecture for specifying the architecture composition and communication

b) The Communication Process Architecture Variables to parametrize the system

c) The Container Specification for defining resource constraints

d) The Function Specification for assigning functions to the entities in the architecture.

For information about the notation, see Appendix A. For examples of how to specify and run (CM)²PI, see Appendix C.
3.1. Multiple Layered Specification (CPA Architecture)

3.1.1.1 Communication Process Architecture (CPA)

The CPA specification is used to describe the overall architecture design of the software system. The intention is that the architecture alone (out of all four specifications) is the only information needed by the developer when programming the functions. An example of a CPA is shown in Figure 3.1 with its corresponding structure diagram shown in Figure 3.2.

Figure 3.1: CPA Example of a prime sieve. The key aspect of this specification include the declaration of the processes and their compositions. This notation is similar to that of FSP (Finite State Process) in Kramer and Magee [28].

```plaintext
structure primesieve (numprimes) {
    PrimeStart = inports <> outports <next, control> ()
    PrimeElement = inports <prev> outports <next> ()
    PrimeEnd = inports <prev, kill> outports <> ()
    R = [0..numprimes-4]

    ||- PrimeSieveChain = ( PrimeElement[numprimes-2] )
    / ( for i in R { next[i] -> prev[i+1],
        receive -> prev[0],
        next[numprimes-3] -> send } )
    @ (receive, send )

    ||- PrimeSieve = ( PrimeStart || PrimeSieveChain || PrimeEnd )
    / ( PrimeStart.next -> PrimeSieveChain.receive,
        PrimeSieveChain.send -> PrimeEnd.prev,
        PrimeStart.control -> PrimeEnd.kill )
}
```

Figure 3.2: A description of the architecture would indicate a variable sized sieve element section with 1 generator and 1 last element. The generator is PrimeStart in the specification and the last element is PrimeEnd in the specification.
3.1. Multiple Layered Specification (CPA Architecture)

The architecture, or structure, of the system consists of the definition of the processes and the composition of those processes. The definition of a process includes assigning a name, defining the input and output ports, as well as any arguments that should be passed in as argv. The definitions of compositions include a list of the processes of compositions which can run concurrently together along with communication information such as channels (see Section 3.2.3) and any exposed ports to be available for further compositions. The component-based design consists of basic components which represent processes at the lowest level and can be composed into larger components representing a group of processes. This is based on the concurrency models from [28]. These components can further be composed with each other to form an even larger software system. For convenience, the multiplicity (number of copies) of each component in a composition is specified using array notation. Expressions in the CPA can include the +, -, *, and / operators along with variables which can be initialized with a value in a later step of the four stages.

A naming system is created during the specification of the CPA. Since each process is given a name and each composition component is also given a name, this creates a tree like structure where the name of each particular process can be identified by following the names of the compositions. (see Section 3.2.1).

To facilitate communication between processes, each process has defined input ports and output ports. During a composition of these processes, channels can be defined to connect an input port of a process to the output port of another process. To facilitate encapsulation, each composition must redefine the exposed ports that can be used when it is further composed with other components. This communication information is made available during program execution allowing processes to identify communication channels by name rather than rank. (see Section 3.2.3).

Furthermore, if a process requires input parameters, the parameter types should also be specified as a list in the CPA. Values of the parameters are to be specified in the next step when their values are defined.
3.1. Multiple Layered Specification (CPA Architecture)

3.1.1.2 Communication Process Architecture Variables (CPA-V)

The CPAV specification is used to initialize a CPA by giving values to any variables used in the CPA specification. This information combined with the CPA creates an instantiation of the system. In Figure 3.3 variables include those used for process multiplicity in a composition and those used for the parameters to each process. Values for the process or composition multiplicity can be specified as an equation. Equations in the CPA can include the +, -, *, and / operators along with other variables defined. Values for process multiplicity can also be a Kleene star (*) indicating a wild card such that any number of processes of this type is acceptable or a plus-sign (+) indicating one or more is acceptable.

In addition to variables being used by the specification for the purpose of multiplicity, parameters to processes defined in the CPA-V can be any string.

Overall, this specification determines how large a system is to be launched.

```
initialize primesieve {
    numprimes = 2000
}
```

Figure 3.3: CPAV Example of a prime sieve. Strings can be used as values for function parameters only.

3.1.1.3 Container Specifications

Once we have the software system, it is then time to match it with the compute resources available. The container specification (see Figure 3.4) provides the restrictions of what (MPI processes) can be put into a container (OS process). This specification consists of two parts: the container specifications, and how many of each container specification to create actual OS processes for. Specifications define how many of each process be can
3.1. Multiple Layered Specification (CPA Architecture)

placed in each container. The syntax used is motivated by regular expressions, however for simplicity, we limit the types of wild card characters. The number may be a positive integer or a Kleene star (*) indicating any number of that process is acceptable or a plus-sign (+) to indicate at least one of that process must be present. The containers can be nested in each other. Finally, the number of each container needs to be specified. The optional number after the slash is a restriction to the number of MPI processes that can be put into this container. When more than one composition uses a particular process, it is possible to create containers that identify the correct instance of the process to place. One way is through the suffix which is specified before the multiplicity (e.g. processname[6]). The other option is through a prefix which can be specified before a colon in front of a container specification name (see Chapter 4).

Figure 3.4: Container Specification Example of a prime sieve. Constraints of what MPI processes can go into each OS process as well as the number of OS processes is defined. In this example, there are 160 OS processes in total. One of them will have a PrimeStart process and as many PrimeElement processes as needed. One of them will have a PrimeEnd process and as many PrimeElement processes as needed. Finally, 158 OS processes will each have as many PrimeElement processes as needed.

3.1.1.4 Function Specification

Processes defined in previous steps are placeholders so we need to bind each of them to a function. The function specification (see Figure 3.5) associates each of the processes specified in the CPA with a “.c” function and an “.h”
3.1. Multiple Layered Specification (CPA Architecture)

file containing the function header. Any additional glue function (see Section [4.4]) that is required to be run by the system needs to be defined in the function specification. This specification basically contains adjustable parameters for user defined functions and user defined variables. The function specification is the choice of which user function to run. There is also a section to specify the user functions that are to be executed as a glue function (see Section [4.4]). And the GENV (Global environment variables) specifications are for exposing user’s internal variables in the external interface.

![Glue Functions](image)

Figure 3.5: Function Spec Example of a prime sieve
3.1.2 The Assembly Line (The Four Stages)

The initialization of the system from the architecture up to creating a runnable program progresses through four stages by reading the defined specifications one at a time (see Figure 3.6). The four stages were chosen in order to split up four decisions: the architecture design, the size of the program, the resource constraints, and the function binding.

Figure 3.6: The four stages. At each stage of the tool’s execution, an input file is used to combine and further initialize the system.

At first, it may seem like the architecture and the container specification seem like redundant specifications of the same thing and that they
could be combined to form a single specification. For simple examples this may look to be the case, however the difference in the two specifications becomes more apparent when some form of communication mechanism is used. When communication is used, it is either based on the naming in the CPA specification or the channels specified there. Then it becomes apparent that function implementations depend on the CPA specification and that changes to the CPA specification may break the program. However, one of our goals is to place the program on different resources. We want to support many different ways to map processes to the resource without having to change the program. Thus, we chose to create two separate specifications.

3.1.2.1 Stage 1 – Design

The CPA is read from a file. Refer to Figure 3.1. The architectural design of the system is now present but uninitialized and so the system size is not yet known.

3.1.2.2 Stage 2 – Instantiation

The CPA-V is read from a file. Refer to Figure 3.3. When combined with the CPA, an instance of the software system is now created. The number of instances of each MPI processes that is packed into the system can be determined now (except in cases where wildcards are specified, the exact number of that process is affected by the container requirements in Stage 3). Compositions that contain a multiplicity adds a multiple of the subgroup of processes.

At this point, we have determined how many of each process is in the system.

3.1.2.3 Stage 3 – Packing

The Container specification is now read in. Refer to Figure 3.4. At this stage, we now have the boxes (OS processes based on container specifications) in which to pack our items (MPI process). So, using the number of containers and the information from stage 2, the processes are packed into
the boxes evenly (with regards to the number of each type of MPI process) while satisfying container restrictions. This packing technique is a policy that can be changed in the tool’s program. Containers specifications need to specify processes that can be packed into each container (this is unrelated to the composition specified in the CPA).

Currently, the packing differentiates between each instance of the same process only when composed separately. All processes that are in the same composition are treated the same. Glue functions may allow reordering of similar processes (see Section 4.4).

At this point, we decide which and how many of each MPI process is in each OS process.

### 3.1.2.4 Stage 4 – Deployment

The Function specification is read from file (see Figure 3.5). Using the file information, we can now create a mapping.c file that imports the appropriate .h file for the functions needed and provide a function mapper that returns the correct function based on the rank of the process. A configfile is also generated. This configfile accompanies the mapping.c file to indicate which contiguous blocks of ranks belong to the same OS process (see Figure 3.7).

![Figure 3.7: The MPI ranks are assigned to the processes in the order they are defined in the execution command. A separate rank is given for each nfg value.](image)

The machinefile can be used to customize which machine and which core each OS process is allocated to. The machinefile is a description of the available compute nodes and the number of cores available on each of them. When mpiexec is called, the OS processes are mapped to their corresponding cores from the start to the end of the machinefile.
The **mapping.c** file creates a wrapper function for each MPI process. The default wrapper selects the correct parameters from the **configfile** and pass on to the original function as regular argc, argv parameters (see Figure 3.8). This wrapper only runs once at the beginning. The wrapper only requires the information of which function to call and the offset for the parameter. That is, the minimum parameter content for each process type is its name (this is to keep the correct offset).

![Figure 3.8: Wrapper functions are provided for each function in the program. The compressed arguments list from the execution command is decoded and then passed on to the user’s function.](image)

The function for determining the parameter input to each MPI process is pluggable meaning the user could provide a separate function to programmatically determine the parameters.

The mapping function is grouped by each line of the **configfile** (each `-nfg -n executable`). Within each group further describing what MPI processes they contain. This way, the size of the mapping function grows proportionally with the **configfile**. The **configfile** already contains the `-n`
3.1. Multiple Layered Specification (CPA Architecture)

and -nfg information in a compacted way and if the user needs to create complicated mappings then the configfile has to be more verbose and thus so does the mapping function. In the worst case, every MPI process calls a different function which forces the mapping function to be completely explicit. In the best case, every MPI process calls the same function and a single line mapping function would suffice.

Figure 3.9: The project folder organization. The files generated are contained in a sub-directory of the project.

The output files are to be in a sub-directory of the project directory (see Figure 3.9). The programmer will create a new target in the project Makefile which will compile its project (leaving out any FG boilerplate) and then invoke make on the Mapping directory giving it the name of the executable make APP=exename. A cmmpi.a is built inside the Mapping directory. This file needs to link with the user’s compiled code into a single executable. A single run script is added to the project directory for execution.
3.2 Communication

of the mapping generated program.

3.2 Communication

One of our main goals is to externalize process names so that re-mapping is possible without the programmer having to do it. We propose three methods of defining communication: Process Discovery, Hierarchical Communicators, and Channels.

3.2.1 Process Discovery

One of the easiest ways to make communication placement-independent is by giving each MPI process an identifier that does not change based on the position of the process\(^1\). Conveniently, our compositions are already named and because it is a hierarchical composition, each process can be uniquely identified by chaining the names of the hierarchy (see Figure 3.10). If a multiple of the same process is in a composition, they are identified by the same name. Therefore, it is possible that more than one process may be identified by the same name. However, this is not an issue because those processes are identical, there is no need to differentiate between them other than having an additional index value for each of them throughout the execution of the program. This means that these processes are uniquely identifiable using a combination of name and index.

Consider this excerpt of the CPA’s composition.

\[
\begin{align*}
| | 3 &= (4 | 5) \\
| | 1 &= (2 | 3 | 6)
\end{align*}
\]

In this example (see Figure 3.11), there is a single process named 1.2 and there is a set of the same processes called 1.3.4 with indexes ranging from 0 to 5 or in other words 1.3.4[0..5]

Only the MPI processes’s information is carried to the packing stage (see Figure 3.12).

\(^1\)See Appendix B.1 for the Process Discovery API.
3.2. Communication

Figure 3.10: Visualizing the CPA using a tree of composition. The naming can be seen clearly here by following from the root to the corresponding MPI process.

Figure 3.11: Visualizing the CPA. A name is given to each level of composition as well as each process.

Figure 3.12: MPI process information available for packing. The compositions are expanded out and only processes with their full name is kept for the next steps.

The MPI processes are put into available containers given they match the container specification (see Figure 3.13).

After the MPI processes are assigned to each container, the container can be seen as an OS process (see Figure 3.14). Now, a process can be looked up using its process ID (pid) and its name given that there is no currently known distinguishing features amongst the MPI processes with the same
3.2. Communication

Figure 3.13: Container Specification defines requirements on the contents of the container.

Figure 3.14: OS Processes are formed and placed adhering to container restrictions.

3.2.2 Composition-Based Hierarchical Communicators

Using a naming system allowed the movement of processes without having to manually adjust the ranks in the programmers point of view. That system is lightweight and efficient for end-to-end communication. However, for collective communication this is not ideal.

In order to take advantage of MPI’s collectives, we need to use communication scopes called communicators. To scope communication, it is natural to have a communicator\(^2\) for each composition. This means that a process that is composed twice would have 3 communicators, one for each level as

\(^2\)See Appendix 3.2 for the Communicators API.
well as an overall world communicator. Since each process is a member of multiple communicators, a proper system to retrieve the intended communicator is required. Conveniently, our compositions were named and the same naming system that is used to identify a process can be used to identify a communicator. For example, a process named ①.③.④ has a communicator called ① and a communicator called ①.③ as well as a world communicator.

The communicators are created at the initialization stage and thus incur no extra overhead during execution of the program. The initialization is done using `MPI_Comm_split` and done starting at the highest level (world) and moving down. This allows us to create as many communicators as possible simultaneously rather than sequentially. Not only are these communicators efficiently created before the rest of the process execution, their smaller scope enable the easier creation of more communicators as there is no need to synchronize `MPI_COMM_WORLD` when creating a new communicator.

### 3.2.3 Channels

The previous two methods abstracted away the need to use MPI process rank numbers. However, there was still a need to know the names of processes in order to discover them and communicate. For example, when using hierarchical communicators, the developer still needs to know the name or level of compositions it is in to be able to get the correct communicator.

In order to make processes completely modular and able to be developed without the knowledge of how they are composed, we need a way to specify only the inputs and outputs of a process and then later connect them together appropriately. This allows functions to be developed to the input and output ports specification which includes a name of the port as well as its direction. The local name of a port is unchanged regardless of where it is attached. So a port called `next` may be connected to a port called `prev` to form a channel and each process refers to that connection by its local name. A tag may also be specified to be used for communication in that channel.

The creation of channels\(^3\) is similar to the creation of connection lines

---

\(^3\)See Appendix B.3 for the Channels API.
3.2. Communication

on labels in [28] where similar labels represent a shared action between the two states. In our case, the label names are used in the program to identify the ports from which communication can be used to send or receive data. So instead of using a renaming technique to make connections, we specify the directional connections between two named ports in order to be distinct about the flow of data.

To facilitate encapsulation, each composition must define what ports are exposed for new channels when further composed. This is inspired by the hiding of actions in [28]. To the exterior of the composition, only these particular ports are visible and their names are as specified by the composition and not by the original process. The exposed port is seen as a double ended port (i.e. an in-port as well as an out-port). To make this clearer, both an inbound and an outbound channel must be connected to an exposed-port. However, to the individual processes, this intermediate port is hidden. The direction of the intermediate port is derived by the directions of the two channels attaching to it (see Figure 3.15).

![Figure 3.15: Channels are specified by connecting port to port. Intermediate ports are used when creating channels through compositions. The grey circles are external ports (intermediate ports visible outside of a composition) while the white circles are internal ports (an MPI process port visible only inside the composition).](image)

Frequently, a set of recurring patterns must be applied to sets of ports when creating channels. Conveniently, we provide a loop option for this purpose. Ranges can be specified such as \( R = [0..\text{numprimes}-4] \) (which...
3.2. Communication

means that R is a value from 0 to numprimes-4 inclusive) and can be used to
generate channels over a pattern. For example, a chain can be specified as
( for i in R next[i] -> prev[i+1] tag 5 ) which creates a channel
from the next port of process index i to the prev port of the process at index
i+1 and to use the tag 5 for any communication across this channel.

Channels can support both one-to-one communication as well as one-to-
many communications as long as all the source processes are of the same
name and all the destination processes are of the same name.

The channel information defined per level in the CPA is translated into
an actual instance. This information is available to the program during
execution. To clearly identify a process’s channels, each type of process
has its own structure to define its channel information. The struct’s name
is generated based on the processes’ base name as defined in the CPA. For
example, if a composed process is called PrimeSieve.PrimeStart, the channel
structure is only based on the process name PrimeStart as they all have the
same channel structure regardless of how they are composed.

Listing 3.1: Channel struct definition

```c
typedef struct chaninfo {
    int* rank;
    int size;
    int tag;
    int direction;
    MPI_Comm communicator;
} chaninfo_t;

typedef struct chan_primeelement
{
    chaninfo_t prev;
    chaninfo_t next;
} chan_primeelement_t;
```

The structure of a processes’ information (see Listing 3.1) is based on
the names of the ports. Intuitively, it is possible to get the information of
the channel through using the process’s port name (as defined in the CPA).
In addition, the tag id to be used for this communication is also available
as well as the direction of the channel. Further, the communicator to be used is also specified. At the moment, the lowest level common hierarchical communicator is used. This opens up for further improvements to how communicators can be leveraged to provide a different layer of scoping for channels. Since there are four pieces of information available for a particular channel, we decided to use an additional C structure to store the four data items.

Given the structure information available to the program during runtime, it is still up to the program whether or not this information is to be used as intended. For example, a channel between two processes can send data in a direction contradicting the original design. In other cases, the specified tag may not be utilised by the program even though it was specified in the CPA. It is important to note that the information specified in the CPA is only made available to the program but is in no way enforced. To enable this requires the creation of wrappers for all MPI send and receive methods.

These channels are assigned according to their process type and index. This makes it possible to rearrange similar processes by plugging a custom permutation as a part of the glue code (see Section 4.4).

One interesting point to consider is what happens when two channels from the same source and destination have the same tag. One way would be to consider each channel as a separate channel and thus we need to create another scope for this channel. In MPI communication, there are 4 pieces of context that scope a communication message: source rank, destination rank, tag, and communicator. Since the first three are unchangeable, the only way we can scope these messages are through the creation of another communicator. However, the creation of a separate communicator for each channel is not scalable.

The other option is to treat the two channels as the same whenever they have the same tag. It is up to the user to create channels in the CPA with separate tags. This is the same responsibilities required in traditional MPI communication so this option is reasonable. In the next section we describe flow-controlled channels which removes the need for a tag (see Section 3.2.4).
3.2.4 Flow-Controlled Channels

Channels allowed a light weight approach to sending messages from source to destination by providing the bare minimal amount of information (ranks, tags, direction, and communicator).

MPI communication is of two types, eager and rendezvous. The eager protocol is such that messages are sent from the sender assuming they are small enough to be buffered by the receiver. Any messages that exceed an implementation-dependant eager limit are sent using the rendezvous protocol which involves sending an envelope data to the receiver and waiting until the receiver indicates that sufficient buffer space is available.

We provide simple channel interface that includes flow control in order to utilize the efficiency of eager sends while limiting the amount of queued unexpected messages.

3.2.4.1 Bounded

The amount of messages unreceived at the receiver must be bounded in order to avoid exhausting the buffers and queuing delays. With this aim, we introduce an intermediate buffer between the source and the destination. The sender sends to the buffer until it is full and then future sends block until space is available. When possible, the buffer sends the messages to the receiver.

In order to not flood the receiver, the receiver and proxy uses a token-based system where one token is consumed for each message sent to the receiver. This is similar to the Guaranteed Envelope Resources (GER) technique \[5\] where an each process-pair has a GER which guarantees against failure if the GER is not exceeded. The GER is estimated based on the number of processes there are per node and the number of possible senders. In our research, the number of tokens is a hard-coded number and is not a guarantee. Further research could gather memory size of the nodes and calculate a guaranteed size.

\[\text{See Appendix B.4 for the Flow-Controlled Channels API.}\]
3.2. Communication

The intermediate buffer allows for an extra buffer to enable more asynchronous messaging while allowing the sender to block when the buffer is full and letting the receiver pass back tokens. When the receiver is out of tokens, the intermediate buffer can still buffer more messages for the sender until itself is full.

The extra buffered message may be sufficient to allow the sender to complete its send and move on to other tasks while the intermediate buffer would send the data to the receiver as tokens are required.

Our technique is similar to TCP’s advertised window mechanism. We do not advertise the proxy size to the sender and instead post MPI receives only if there is sufficient buffer space. A sender using a synchronous send then blocks. In FG-MPI, MPI processes can be co-located to share a single processing core and so cycles can be used by other MPI processes when a sender is blocked.

This also makes the communication architecture simpler as the sender only needs to send and not receive any replies. The intermediate buffer handles the receiver tokens instead. All the receiver API has to do is receive messages and send back to the intermediate buffer some tokens for more messages.

It is more efficient to start updating tokens before the receiver reads all the messages in order to allow the intermediate buffer to send more as soon as possible while keeping the number of token update messages low.

The sender sends messages to the intermediate buffer via two regular sends followed by a synchronous send. The synchronous send allows the sender to block if the intermediate buffer is full because the intermediate buffer does not post further MPIRecv until it has sufficient buffer space. We don’t want to send all messages synchronously because that is a performance hit so we take advantage of the middleware’s receive buffer queue of the intermediate buffer. This set up minimizes the chance of the channel being completely empty in case messages are available to be sent.
3.2. Communication

3.2.4.2 Blocking

The FG-MPI scheduler contains a blocking mode where MPI processes blocked on a send or receive are put into a block queue so as not to waste processing cycles. The sender should then block when the intermediate buffer is full. To ensure that cycles are not wasted, blocking the sender process allows other co-located processes to proceed. Another advantage is that it is much simpler to code and debug a blocking sender as oppose to one where the send may either succeed or fail. The same is true for the receiver.

3.2.4.3 In-Band Control

Channels are considered unidirectional because data is sent in only one direction. However, control data is sent in-band and transparent to the sender and receiver. These control messages include startup, tokens, and termination messages.

With FG-MPI, processes that are created but blocked on a call incur no run-time penalties except a small amount of memory. The FG-MPI middleware uses shared memory access to optimize communication between co-located processes. FG-MPI also provides a zero-copy API that enables the sending of messages through passing pointers rather than copying memory. The communication cost is much less compared to a message through the network. This allowed us to be more flexible when designing the architecture for the intermediate buffer and where messages should go. Instead of time-slicing the sender process to receive token updates and termination messages, we allow FG-MPI to provide concurrency when we use collocated processes to do each of those tasks. The code is more cohesive and thereby more maintainable and easier to understand.

3.2.4.4 Channel Types

We support three types of flow-controlled channels:

a) one-to-one channels with a single source and single destination,
b) one-to-many channels where there is one source but many destinations (data from the source travels to one of the possible destinations), and
c) many-to-one channels where there are many sources sending to a single destination.

### 3.2.4.5 One-to-One

In the one-to-one case, it is a pipe for sending data through (see Figure 3.16 label A).

### 3.2.4.6 One-to-Many

In the one-to-many case, we need to account for the distribution of the incoming data to the destinations available (see Figure 3.16 label B). There is a single intermediate buffer for all the messages that are sent out from the single source. That intermediate buffer is called a proxy process as it distributes its data to the destinations. How it distributes the data is a matter of policy and can be changed as required. In general, there are a few ways to distribute the data: round-robin style, or priority style based on the available tokens.

### 3.2.4.7 Many-to-One

In the many-to-one case, we are gathering up data from many processes and sending them to one process (see Figure 3.16 label C). This case is also simple as the intermediate buffer proxy gathers all data from all sources and forward them on to the destination.

### 3.2.4.8 Implementation Details

Flow-controlled channels are created for each channel specified in the CPA no matter if they are used or not.

Both sides need to open the channel before data can flow through it. We make the assumption that if one side opens a channel, the other side eventually opens the channel also. This is synonymous to MPI’s send and
3.2. Communication

Figure 3.16: The types of channels we consider. A) One-to-one channels with a single source and single destination, B) One-to-many channels where there is one source but many destinations, C) Many-to-one channels where there are many sources sending to a single destination

receive where if one side sends a message, it is assumed that the other side eventually posts a receive. However, with the introduction of the open and close API, it is possible to decide whether to allocate the resources needed to support the channel; such as buffer space.

This also allows many-to-one and one-to-many channels to indicate which sources and destinations are available to send and receive data. Especially in the one-to-many case, it is important to know which receivers are available to send data to and which receivers to not send to because they have not agreed yet to receiving any data. In other words, many-to-one and one-to-many channels have an agreement that once the channel is opened, there must be at least one sender and one receiver at each end at any time. Senders and receivers can unilaterally terminate communication by notifying the channel that it will not send or receive data until further notice as long as it is not the last active sender or receiver. Likewise, senders and receivers may join the channel at anytime as long as it remains open under the terms above. When the last sender closes the channel, the channel is shutdown and
only active receivers are notified. The channel is then permanently closed and can not be opened again. When the last receiver wants to close the channel, then it must seek agreement from the sender and terminate after the sender shuts down the channel.

To keep this efficient, there exists a proxy process (see Figure 3.17) that acts as the intermediate buffer. This intermediate buffer listens for the open and close messages of the sender and receiver as well as receive and buffer incoming messages and sending them to receivers when tokens are available.

![Flow-controlled channel communication for a one-to-one channel.](image)

Tokens provided by receivers are also organized by this proxy process. This proxy is also required in order to support the blocking nature of the send request so that when a synchronous send is sent to the proxy and the proxy buffer is full, then it blocks until the proxy accepts the data. The buffer in the proxy allows the sender to do other work while messages are sent to the receiver when possible. Without this proxy, it is difficult for the sender to decide when to receive token messages while carrying on its other tasks.

If we instead bypassed the buffer proxy and block on token updates when the tokens are depleted, then the only buffer available to us is the non-guaranteed buffering provided by the middleware. This proxy process is generated for each OS-process whenever channels are created in the architecture specification. As a result, every MPI process sharing a single process core, there exists only one shared proxy process. The intermediate
3.2. Communication

proxy process is co-located with at least one side of the channel in order to minimize the extra communication due to having an intermediate process. The most efficient placement of the proxy would be on the single process end of a one-to-many or many-to-one channel in order to reduce communication overhead by leveraging FG-MPI's use of shared memory for co-located processes (see Figure 3.18 and Figure 3.19).

Figure 3.18: Flow-controlled channel communication for a one-to-many channel. The proxy is co-located with the source to reduce the effect of the communication between the source and the proxy.

3.2.4.9 Channel Termination

Channel termination requires both sides to agree. For correctness, channel must be emptied before termination. This implies that at some point the sender stops sending to give the receiver a chance to empty the channel, after which the channel can be closed. There are two cases to the termination problem.

a) If the sender terminates the channel first (see Figure 3.20), then the termination message is sent with the same tag, source, and destination by MPI's message ordering semantics the termination message reaches the receiver after all the previous data has been received. Once the sender sends
3.2. Communication

Figure 3.19: Flow-controlled channel communication for a many-to-one channel. The proxy is co-located with the destination. Data must travel from the proxy to the singular destination in this channel construct so the goal is to minimize the communication overhead of sending data from the proxy to the destination.

After the termination, the channel is closed from the perspective of the sender as the receiver can not demand more data. When the receiver receives the termination message, it is also closed because the sender promises not to send data.

b) The second case is if the receiver closes the connection (see Figure 3.21). It is a matter of policy whether the receiver can force the source to stop sending or merely request the source to stop sending. In either case, a termination request message is sent to the sender and depending on policy, the sender carries out the same procedure when a sender terminates the connection. As a result, when the receiver sends the termination request, the channel continues to receive data from the sender as long as the sender’s termination message has not reached the proxy. To avoid non-blocking senders, termination requests should not be sent directly to the
3.2. Communication

source. With FG-MPI, the issue is solved by having yet another co-located process for handling the termination request. This process is co-located with the sender and can take advantage of the shared address space. Then when this process receives the termination request, it only needs to set a flag in the local memory and the sender can check this flag before sending any messages. This extra process incurs almost no extra resources as it is always blocked on receiving the termination request.

Figure 3.20: The typical messaging between source, destination and proxy when the sender initiates the termination of the channel.
3.2. Communication

Figure 3.21: The typical messaging between source, destination, and proxy when the receiver requires the termination of the channel. Notice that the receiver can only request and not demand termination. It is still up to the sender to terminate the channel.

3.2.4.10 Other Considerations

There is an opportunity for optimization of this protocol when sending messages between co-located processes. In FG-MPI, all message sends between co-located processes are synchronous anyways so there is no need to buffer any of the messages. In other words, flow control is not needed for mes-
3.2. Communication

sages between co-located processes. So, to optimize this, we have messages sent directly from sender to receiver bypassing the proxy (see Figure 3.22). Sender’s termination messages also travel directly to the receiver. As well, the receiver’s termination request is sent directly to the sender’s control proxy rather than through the intermediate proxy. This only works in the one-to-one case because the intermediate proxy in one-to-many and many-to-one also acts to load balance messages in addition to it’s buffering task. Furthermore, we are able to utilize FG-MPI’s zero copy API in order to eliminate an extra buffer copy when sending the data.

Figure 3.22: To optimize for performance, communication between co-located processes can be simplified as the communication is always synchronous and flow-control is not an issue.

We did not implement many-to-many FC channels. MPI collectives using communicators are more suitable for this type of communication. We do not want to re-create collectives since MPI provides them with optimization for communication and message buffering.

The design above describes the messaging protocol to support flow-controlled channels. Alternative designs include changing the policy depending on the needs of the program. We have implemented one policy to this problem but there are other that could be created to change the behavior of: how messages are load balanced, whether the sender can keep sending messages once a termination request is received, whether channels can be reopened after it is closed, whether individual senders or receivers may terminate without the consensus of the others when it is not the last sender/receiver in the channel.
3.3. Encapsulation of Compositions

The API does not completely solve the receiver buffer exhaustion problem but rather mitigates it because the flow control is only present within each channel. If a receiver has too many incoming channels, its buffers could still be exhausted even if each channel sends a single message. With FG-MPI, this issue is exacerbated as more co-located MPI processes means there is less memory available to each MPI process. Changes to the MPI middleware are necessary to make this solution completely safe.

Overall, the design of the proxy above includes the following points:

a) To limit the number of buffered messages in a single channel to the sum of the size of the proxy’s buffer and the number of tokens provided by the receiver.

b) To disallow messages from traveling through the channel when its state is CHANNEL_CLOSED. The sender can demand the closing of a channel while the receiver can only suggest the closing of a channel.

c) To be a reusable design for all three cases, one-to-one, one-to-many, and many-to-one.

d) To solve the issue of scheduling when to receive control messages. Merging the tasks of the proxy with the sender creates a scheduling issue for the developer. The only time the library’s code is run in the user’s code is when it invokes one of the library’s API calls. But it is not efficient to receive all control messages only when a library’s API is invoked.

e) To ensure that no other non-channel related messages with the same tag, source, and destination interferes with the operations of the channel because messages are addressed to the proxy.

3.3 Encapsulation of Compositions

Good software design calls for high cohesion and low coupling of pieces of code. Encapsulation of code normally suggests hiding the internal workings of a black box. In other words, how functions interact with each other and how things are represented inside the black box is unknown to the outside. The only way the external can invoke the black box is through a series of interfaces provided by the black box. So in a parallel program, the internals
are MPI processes (in FG-MPI, these are the functions). Encapsulation in this case would be to hide what these processes are as well as how they communicate. To achieve this we use ports and channels.

Ports are declared for each MPI process so distinct communication end points are known. To the process itself, it only needs to know about the port’s name regardless of how the process is composed or what its name is because channels created between ports provides the communication medium. When composed, all internal ports are hidden.

If any ports need to be available external to the composition, an exposed port must be declared and the internal port attached to it. To the external environment of the composition, this composition has the same parts as a process: a name, and a set of ports. This encapsulation can even be multi-layered. For example, a system using a data structure is a composition of the system and the data structure. But then this composition can be further composed with an application which treats the previous composition as a single entity.

How the channels are created through these compositions may be thought of as threading a string through a series of holes (or ports) and at the end points we tie them to complete the channel (see Figure 3.23). To the process itself, it has no interest in the intermediate holes (or ports) it has to go through but rather the end points only. Implementation wise, the intermediate ports no longer exists after a channel is defined. Overall, this defines how we address messages but as any other MPI programs the type of the data itself must be agreed on separate to this specification.

3.4 Composition of Programs

The ability to easily compose programs together to create a functioning system is important in exploiting the modularity of our programs. In a sequential program, one can include a header file of another program and start using it by communicating through an API. For a parallel program, it is not as simple as we need to account for the communication by identifying the correct targets as well as placements of the corresponding processes.
3.4. Composition of Programs

Figure 3.23: Creation of channels through composition. The white circles represent ports while the grey circles represent intermediate ports explicitly exposed by a composition.

Creating two instances of a data structure is no longer simply having two variables but rather we need to duplicate the processes that provide those functions. Traditionally, one would have to handle the different communication using complex and unintuitive ways of embedding conditionals in the code in order for the process to communicate to the correct instance of its neighbour (see Figure 3.24). One of our goals is to simplify this process by handling the correct communication by providing an API for identifying the correct process for communication.

To support modularity, each structure (or CPA) is given a name. Each can take a set of unbound variables that need to be initialized. In this case, the name of the structure creates a scope for the variables such that variable declarations for each structure do not conflict when structures are composed together. Maintaining modularity, it is possible to include structures from other CPA specifications (see Listing 3.2).
3.4. Composition of Programs

Figure 3.24: Assuming there are 3 process types: A, B, and C. If these processes form a chain A) and we want to duplicate it, then the new chain must take a new set of rank numbers where B) and C) are possibilities. Traditional methods of communication using ranks require the developer to consider and handle these possibilities.

Listing 3.2: Including another structure

```plaintext
# Composition of helloWorld and primesieve
include prime.composition
include hello.composition
```

Each “include” from each file is read by (CM)²PI. Each structure is initialized by name. Once the structure has been initialized, its information (compositions and process declarations) is combined with the other structures that are input into (CM)²PI. Currently, the tool looks at the directory the specification was in for the include files as well as look into other directories where it has already read a specification file. If this is insufficient, there is an option to use a -i flag to specify locations to look for these include files.

Each process is given a name and each further composition is given a name. This creates a tree of names from which we can trace out the full name of an particular process. For example, `PrimeSieve.PrimeStart` means there is a process called `PrimeStart` inside the composition `PrimeSieve`. To support composition of components, different structures can reuse any compo-
sitions created by another structure which it includes. For example, if we were to compose two functioning parallel programs into one single parallel program. We would simply have to create a new CPA which includes the two CPAs of the respective parallel programs. This example does not include any communication between the two components. However, it does show that we would not need to modify any code or rank numbers to achieve this because the tool would do it.

However, if we were to make the two components communicate, there are two ways to achieve this. One is through using the discovery API (see Section 3.2.1) and the second is through channels (see Section 3.2.3). If the communication was to be through the discovery API, then the corresponding code simply has to make sure the communication calls target the correct processes by discovering them through their name. If the communication were to be through the use of channels, then channels can be created between the ports of two compositions while being oblivious about which process inside the compositions uses the port.

In some cases, it may be required to create two copies of the same structure. In this case, to preserve modularity, one can create this structure with all compositions hidden. Then in a separate structure create a composition that uses those compositions in the other structure (see Figure 3.25). In this case, in order to pack each program together, it is possible to identify which instance of that structure we wish to pack into a container by adding the prefix of the composition (see Listing 3.3). For example if there was a hidden composition called LinkedList and we create two instances of it by making a composition List1.LinkedList and List2.LinkedList. We can reuse the original containers for the LinkedList as well as identifying which instances is to be mapped to which container. If the original container was called LinkedListContainer, we can achieve our goal by specifying a prefix List1:LinkedListContainer and List2:LinkedListContainer.
3.4. Composition of Programs

Figure 3.25: A prefix can be used to identify the processes of a particular composition.

Listing 3.3: Duplicating a structure

```c
# Assume that the LinkedList composition is defined in a
# structure defined in linkedlist.cpa
include linkedlist.cpa
structure linkedlistx2 () {
    || List1 = ( LinkedList )
    || List2 = ( LinkedList )
}

initialize linkedlistx2 {}  
containers {
}
```
Chapter 4

Placement

4.1 Combination of Concurrency and Parallelism

The composition specifies the system and its correctness with respect to process communication and composition. There still remains the placement of the computation on the machine. Satisfying the resource constraints we can also optimize for performance. The container specification specifies the available resources that the tool places on the previously initialized MPI processes. The container specification defines two parts: the container specifications, and the initialization of those specifications into boxes (OS processes).

The container specification define how many of a particular process can be placed into an OS process made with that container specification. These specifications can also be composed into other container specifications. Each of these specifications can also specify an \( \text{nfg} \) limit in order to limit the amount of collocated processes that are allowed in the container in order to have fewer MPI processes contending for the same processing power. A number of OS processes (or boxes) are specified using the container specifications defined. Any number of these boxes can be created and each one in the end translates to being an OS process.

When the number of each process to be placed in an OS process is specifically defined, it is easy to allocate. However, when the number of processes to allocate is a wild card or that the number of a particular process that can be placed into an OS process is a wild card, there is more flexibility (see Figure 4.1).

However, with this flexibility comes a problem to allocate the MPI processes to the OS processes intelligently. What is considered a smart tech-
4.1. Combination of Concurrency and Parallelism

Figure 4.1: The use of wildcard in the architecture initialization gives freedom to place as many processes as need such as the Master process. The use of wildcard in the container specification allows for freedom to place a fixed number of MPI processes (as specified in the architecture) to be placed evenly on the resources available.

The technique is subjective and dependant on the problem. However, without any domain knowledge, the reasonable option is to balance out the number of MPI processes on each OS process. Guidance can be given by limiting the number of MPI processes in a container specification. Hence in the end, we have a set of constraints: the number of each type of MPI processes in the system, the number and types of MPI processes that can appear in an OS process, and the maximum number of MPI processes.
4.1. Combination of Concurrency and Parallelism

To quickly calculate a placement, we used a linear programming approach as the constraints fits it well. Lpsolve [2] is the linear programming library used. Dependant on the linear programming library used, when given just the constraints the allocation of MPI processes to OS processes may not be even (i.e. the first OS process is stacked with MPI processes while the rest are empty). To mitigate this problem, a target number of MPI processes is calculated and the linear optimization is to minimize the difference from the target numbers.

Often times there are simple yet interesting patterns of MPI process allocation that yield better performance. For example, in a prime sieve problem where each process is a sieve then it makes sense to co-locate fewer MPI processes at the beginning of the sieve because more numbers are forwarded through the start of the sieve than the rest. A simple loop mechanism is available to support simple patterns like these (see Figure 4.2). A range can be specified for a loop when creating boxes of a container specification. Inside the container specification, it can use in an equation the variable being looped.

![Figure 4.2: Loops can be used to create a simple pattern in the number of processes placed into each container.](image)

Usually, when assigning MPI processes to container specifications the MPI process is named by the CPA’s process name. However, there are times where the same process is used in multiple compositions but when allocating them to container specifications we need to clearly define the composition from which the process belong. In this case, a prefix on the process name can be specified before a colon and only processes that match the composition prefix are allocated in that specification (see Figure 4.3). The prefix is
4.2 Scaling to Larger or Smaller Resources

chosen because it is common that a whole composition is duplicated but with a different higher level composition name. In this case, just specifying the higher level composition name is sufficient as opposed to using the entire name.

```plaintext
structure twodatastructures () {
    DataElement = inports <...> outports <...> ()
    ||- DataChain = ( DataElement[numelements] || ...)
    || Application1 = ( DataChain || ... )
    || Application2 = ( DataChain || ... )
}
containers {
    containerspec datachainspec = < DataChain[*] >
    boxes ( Application1:datachainspec || Application2:datachainspec )
}
```

Figure 4.3: The prefix of a composition can be specified in the container specification in remove ambiguity of process placement.

4.2 Scaling to Larger or Smaller Resources

Software is developed to be runnable on systems of variable size and performance. For SPMD programs, it is easy to scale up the number of OS processes by simply adjusting the \texttt{nfg} and \texttt{n} values on mpiexec. Instead of a single point of extension as in in SPMD, MPMD has many. However, for MPMD programs, it is much more difficult because a mapping must specify the MPI rank for each process. For example, a program requiring that every OS process contain a master process. In this case, changing the number of co-located MPI processes in an OS process or changing the number of OS processes causes a change in this mapping. It is not reasonable to expect the mapping to be rewritten every time the machine or mapping changes. Writing it manually is error-prone and tedious and writing an automatic mapping generator every time is also not efficient.

\textit{(CM)}^2\textit{PI} provides a generalized solution to this problem by abstracting out the decisions of the program size (the number of MPI process) from the specification of the target system resource size (the number of OS processes
4.3 Heterogeneous Computation Load

to spawn). The program size is decided by changing the variables in the CPA-V while the resource size is specified in the container specification (see Figure 4.4). After any changes to these specification, running the tool automatically generates a new mapping based on the new program size and resource constraints. Through the use of wild cards and + in the CPA-V and container specification, it is possible to make the program fit into the available resources or resize the program to fit in the available resources. If the wild card is used in the CPA-V then the program can resize based on the available resources. If the wild card is used in the container specification then the goal is to accommodate the same program size into smaller or larger amounts of available resources.

![Diagram](image)

Figure 4.4: Separation of concerns in the specification allows concise decisions to be changed. The tool automatically fits the program into the resources available and generate the corresponding mapping. For large programs with many processes, changing the mapper and configuration manually is error-prone and grueling.

4.3 Heterogeneous Computation Load

Today’s clusters may not be homogeneous in performance due to upgrades and and purchases of CPUs with different performance. With MPI, it was difficult to to make sure faster cores do more work than the slower cores.
This resulted in delays and bottlenecks caused by the slower cores. With FG-MPI, it is possible to co-locate many MPI processes together. It is then easy to adapt the number of MPI processes to amount of available compute power, memory, and other resource constraints.

By time slicing between more work, it is possible to keep the faster core busy while messages are blocked waiting to be retrieved from the slower cores. In MPI, if we want more work for a particular OS process, we need to manually time slice the code because we can not create more OS processes than there are cores because that would oversubscribe the cores and it is up to the decisions of the OS scheduler how these processes are scheduled.

However, with FG-MPI, this becomes easier as each MPI process is independent process but many of these processes can be co-located together to share a core. But like the placement of MPI processes into the available resources, it is hard to modify the mapping manually by hand for every different cluster where performance changes. With (CM)$^2$PI, you can simply specify more or less of a particular MPI process to be placed in an OS process. Furthermore, it is possible to set a limitation to the number of MPI processes that can be put in an OS process. Just setting this value to be lower for slower cores and higher for fast cores provides better static allocation of resources.

### 4.4 Glue Functions

We want to open up certain places in the (CM)$^2$PI code to give programmers a chance to insert code or to tailor certain functionality. There are 7 places where code can be inserted: the main function’s initialization, the mapping function’s initialization, the mapping function’s termination, the processing of function parameters, the communicator permutation, the cleanup of parameters, and the cleanup of the main function.

These glue functions can be set to run conditional on certain UNIX environment variables (genv) being set. Genv variables can be set in the function specification or at run time using as flags `-genv FLAG` value. An example of glue function injection is specified as a section in the function
4.4. Glue Functions

specification (see Listing C.16). For detailed notation see Appendix A.2.5.

4.4.1 A: Main Init

The maininit() function has a callback location in the main() function before the call to FGmpiexec(). This function cannot make any calls to the MPI or FG-MPI API.

The following is the function prototype:

```c
void maininit( int argc, char** argv );
```

4.4.2 B: Map Init

The mapinit() function has a callback location in the per OS process mapping function. This function is called before FG-MPI maps functions to ranks and can make calls to FG-MPI’s mapping API.

The following is the function prototype:

```c
void mapinit( int argc, char** argv );
```

4.4.3 C: Map Finalize

The mapfinalize() function has a callback location in the per OS process mapping function. This function is called after FG-MPI maps functions to ranks and can make calls to FG-MPI’s mapping API.

The following is the function prototype:

```c
void mapfinalize( int argc, char** argv );
```

4.4.4 D: Get Parameters

The getparams() function has a callback location right before the execution of the user’s co-routine (non-preemptive threads).

The default function is to get the command-delimited parameters from the command line. The alternate is to programmatically determine the parameters to each function. This can be done in the wrapper function where the argc and argv are determined. The user can provide functions to provide custom parameters to the process. The default parameters is the
function’s name along with an parameters specified in the CPA specification. This information is provided as the argv input to the function. The updated argv is to be pointed to by new_argv.

The following is the function prototype:

```c
void getparams( int argc, char *argv[], int offset, int* new(argc, char*** new_argv );
```

4.4.5 E: Communicator Permutation

The permutecommunicator() function has a callback location inside the CMOMPI_Init() function. This is the code for permuting MPI processes within the given communicator. The new communicator with the processes ordered after permutation is to be pointed to by new. Permuting the processes allow for optimizing the position of processes for performance (see Section 4.5).

The following is the function prototype:

```c
void permutecommunicator(MPI_Comm original, const char* name, MPI_Comm* new)
```

4.4.6 F: Free Parameters

The freeparams() function has a callback location after the execution of the user’s co-routine. It is used to clean up any data structures created by the user in the getparams glue function. The new_argv from get_params is a parameter to this function.

The following is the function prototype:

```c
void freeparams(char *new_argv[]);
```

4.4.7 G: Main Cleanup

The maincleanup() function has a callback location after the execution of Fgmpiexec(). It is used to clean up any data structures created by the user in the maininit() glue function. This function has no access to any MPI or FG-MPI API.

The following is the function prototype:
4.4.8 Glue Code Use Cases

Below are some common use cases supported by the use of glue functions.

1. Multiple mapper customizations (e.g. Prime sieve example). The programmer can create a custom order of how processes are to communicate to each other. A custom order is generated before individual MPI processes start execution. To deliver the information to each process, the programmer can invoke MPI processes with a custom set of parameters with the ordering information.

   Required Input: GEnv, world size, nfg (possibly)
   Uses: mapinit, getparams, freeparams

2. Custom parameters. The programmer can create a function to programmatically provide parameters to each MPI process.

   Uses: getparams, freeparams

3. Global data structure (e.g. Skiplist freenode information). The programmer can initialize some global data structure before the execution of the program. It is then possible to clean up the global data structure at the end of the program’s execution.

   Uses: maininit, maincleanup

4. Comma-delimited parameters from command line. The programmer can get parameters from the command line through the special comma-delimited format (i.e. function_name, arg1, arg2,...). This is provided as the default functions default_get_params which the programmer can wrap around to manipulate the parameters before invoking the MPI processes.

   Uses: default_get_params, default_free_params, getparams, freeparams
4.5 Permutation of Processes

Processes as specified in the final specification can be permuted in order to test the performance of different configurations of the processes and how they communicate. Rather than reconfigure the communications or channels between processes, we permute the positioning of the processes into a virtual order and then apply any channels to the new ordering (see Figure 4.5).

The permutation can be done using the MPI’s communicator translation functions. Each permutation is to be performed on the lowest level communicator for a particular type of process. Any process of the same type based on the hierarchy (not the process type or function name) can be reordered amongst themselves. This is done through a call back function where the programmer provides a function that takes in a communicator and the hierarchical name of the communicator and the function transforms it into a new communicator.

The permutation is then flattened through the different levels of communicators from which that process has access. This ensures the ordering to be consistent no matter if the program is using the communicators as hierarchical communicators or using them with channels. For a code sketch of how this is performed see Appendix D.

4.5.1 Communicators Permutation Callback Function

A simple call back function is to be provided by the user as a glue code in the function specification. The original order is provided in the original MPI communicator and the pointer new must point to the new communicator created by this function.

- void comm_permute(MPI_Comm original, const char* name, MPI_Comm* new);
4.5. Permutation of Processes

Figure 4.5: The permutation of processes using communicators. The dark blue circles represent processes and the white circles represents a rank naming at that communicator. Processes are permuted (green arrows) at the lowest level and the permutation propagates back up. Any channel information (red arrows) is based on the virtual ordering in the new communicators.
Chapter 5

Evaluation

We will look at the different ways to use (CM)$^2$PI to improve performance of a parallel program. Then, we will discuss the different overheads that result from using the tool.

5.1 Integration with Optimization Tools

Having the ability to change the process mapping is great but having the mapping explored and discovered automatically can expose certain unknown mappings or at least find them faster than a human.

General parameter search optimization tools work by adjusting a set of parameters each with a valid range then executing the program with those parameters and checking for a “goodness” value. The common parameters that are adjusted when looking for a good mapping are the number of MPI processes, the $nfg$ parameter, and the number of OS processes to spawn. The number of MPI processes can be modified in the CPAV specification while the other two options are adjusted in the container specification.

Depending on the amount of flexibility to be explored, the container specification needs to be equivalently detailed in order to provide sufficient number of parameters to be adjusted by the optimizer. To create parameters for the optimizer to tweak the number of processes in each core, we need to create separate container specifications. However, if there is a distinct pattern that is known to be efficient then the container specification can be simplified to reduce the number of parameters to search over. As with any other optimization problem, a larger parameter space increases the time required for the optimizer to find an efficient set of parameters.

The other issue is the goodness value needed to evaluate a mapping. In
general the value that matters most is the total time required to run the program from start to end. However, parallel programs’ goodness is more than just the total time required. It also involve the amount of idle processing and the amount of communication overhead. With extra information the optimizer can find the optimal point more quickly. However, most optimizers require a single goodness value so further research is required to determine how to calculate this goodness value based on the evaluation techniques available. There are other software out there, like Tau [31], that will provide more insight into how well a program runs and the different amount of overhead as a result of the current mapping.

5.1.1 Proof of Concept

The optimizer SMAC [16] was used to optimize a prime sieve program specified using (CM)$^2$PI. The OS processes were binned into four types of container specifications. The optimizer was then given a range for the number of sieve elements to put into each of these containers. The optimizer was then run to minimize the total run time of finding 1000 primes\footnote{The number of primes was arbitrarily chosen.}. The result was that it was beneficial to put fewer MPI processes near the front of the prime sieve and more at the end. This is because the front of the prime sieve is more busy as fewer elements get through to the rest of the sieve. It then makes sense to put fewer of these busy processes together and to put more of the less busy processes together.

An important use of an optimization tool is the ability to automatically fit a parallel program into the available resources. This could be thought of being similar to a ./configure command but rather than working on dependencies we adjust the mapping to fit better on the resources available when a program is first run on a new system. This ability to auto scale a parallel program facilities the deployment stage.
5.2 Permutation of Processes

Traditionally, permuting processes requires the handling on all the communication points or what ranks these processes need to communicate to. Most commonly, a look-up table is created in order to determine the communication ranks after the permutation of a process. Often the permutation table and its associated communication translations become confusing.

With the addition of channels it is now possible to specify the communication between processes yet be able to move them around to optimize for performance. Processes can be permuted by creating a new communicator with a different ordering of the member processes. The communicator is part of MPI’s implementation is is easy to understand as a permutation from one communicator to the next. FG-MPI has optimized the storage of communicator information to being stored once per OS process. Permuting the processes allow for a balance of having more processing cores active while trading off for communication overhead.

5.3 Adapting to Resource Size

Most programs don’t only run on the computer or cluster it was developed on. Eventually, it needs to run on larger or smaller system. There are two aspects of a parallel program that can be adjusted to increase performance when running on a new system: the number of MPI processes, and the number of OS processes.

In the original FG-MPI mapper, the mapper maps ranks to functions. For SPMD programs, this was easy to create simple patterns that work regardless of the number of ranks. However, for MPMD programs, there may be requirements to the types of processes that must exist on an OS process and the mapper is not as general. If it had a pattern that could repeat as more OS processes with the same nfg is added, it was still not able to adapt when nfg changes. Changing this mapping manually is error prone and is not viable to be recreated every time a system is redeployed. However, with the CPAV and container specifications in place, it is simple
5.4 Integration of Separately Developed MPI Programs

In software development, it is rare that all the code of a software system is written by a single developer and therefore it is important that integration of code developed by different developers are easy to combine into one usable system. The connection points need to be clearly defined and how the rest of the system works is insignificant. Using our composition method, the separately developed components will have two different sets of specification files for CPA, CPAV, containers, and functions. We then need to create a new CPA that includes all the files from before. We then need to create a new composition that contains the two components. Ideally, the communication entry points between the two compositions can be done through ports and channels. Then create a CPAV that initializes this CPA. This is the minimal amount of work required to get a composition of two separately developed programs to work together.

5.5 Performance Impact Evaluation

We will look at the overhead associated with using \((CM)^2\)PI. It falls into two main categories: start-up overhead, and run-time overhead. Parallelism is used to improve performance so it is reasonable to assume programs that are made parallel will run for a sizable amount of time. The start-up overhead is eclipsed as run-time increases.

5.5.1 Process Mapping Storage Requirement

The FG-MPI mapper requires that a mapper function be provided where given rank numbers, a function pointer is returned such that a co-routine
5.5. Performance Impact Evaluation

can be started to run that function. To minimize the storage requirement, this information is stored on 3 levels.

1) One level is to describe the co-located processes inside an OS process. To store this information, a loop around a statement returning a function pointer is used to indicate the number of that function to spawn as co-located processes.

2) The next level is to indicate the number of these OS-processes to spawn and a loop around the previous level is used to indicate this.

3) The third level is a collection of the previous 2 levels to indicate the variety of OS processes to be spawned.

In other words, the first level indicates the \( n_{fg} \), the second level indicates the \( N \), and the third level is a collection of those to form a complete configuration. The space complexity is at the same order as the MPI config file where each line indicates the \( n \) and \( n_{fg} \) to launch of a particular set of functions.

5.5.2 Process Discovery

Process discovery is finding a mapping between process names, ranks, OS process IDs, and node IDs. The mapping between process names and ranks uses the same information as that required by the FG-MPI mapper to map functions to ranks. The OS process id is also discoverable from the information in the process mapping. The node id is discovered through communication of the OS processes. Other than the node IDs (stored as a single integer per OS process) there is no extra amount of information stored for this purpose. The information is extracted when it is requested through the API. However, this information is only required once at the beginning of the program and so this can be seen as a start-up overhead.

The node id information is obtained by performing a single round of ring communication from which each OS process declares its lowest process id number. Since the process IDs assigned by the tool is a sequential and contiguous range, having the lowest process id number of every node is sufficient to determine which node each of the remaining process IDs belong to.
5.5. Performance Impact Evaluation

The discovery of node information is performed only once at the beginning of execution as part of the call to CMCMPI_Init().

5.5.3 Hierarchical Communicators

The hierarchical communicators are created using the MPI_Comm_split() function. This action is only performed once at the start of execution as part of CMCMPI_Init(). Any communicator permutations from the user’s provided glue code is also run at this point.

5.5.4 Channels Information

Code that is generated for channel information has the same space complexity as defined in the CPA specification. That means when ranges are used in the specification, they are translated to corresponding loops in the generated code. Any intermediate ports used in the CPA specification are factored out. In order for this information to be easily accessible to MPI processes, this information is expanded out into structures for each MPI process. However, only the information relevant to the MPI processes within the OS process is expanded. This expansion is performed as part of the call to (CM)²PI.

5.5.5 CMCMPI_Init

CMCMPI_Init() as described before include the initialization of the node information, communicators, and channels. All of this work is performed only once as at start up. This does not affect the run-time performance of the user’s computation. Given a sufficiently large problem, this start up overhead is minimized in proportion to the overall run-time. The call to CMCMPI_Init() and the corresponding cleanup API CMCMPI_Finalize() is optional. If this function is not executed then the corresponding features it initializes will not be available but the extra start up overhead is eliminated.
5.5. Performance Impact Evaluation

5.5.6 Flow-Controlled Channels

The flow-controlled channel APIs are the only functions that adds an operational overhead when used. The send and receives functions are executed during messaging. There is the overhead of having a proxy in the middle as well as a few extra messages (token updates, synchronizations). From the source to the proxy, there is a synchronization overhead due to occasionally synchronized sends in order to block the sender if the proxy is full. There is also an occasional token update message from the receiver back to the proxy to indicate how many messages can be sent to it.

These two control messages can however be optimized to be sent more or less frequently. In comparison, the traditional send and receive model uses a single direct message from source to destination. However, the traditional send and receive models have a higher potential to exhaust receive buffers due to memory constraints and thus causing a failure. Flow control adds some overhead to avoid this problem and to also avoid messages from flooding a receiver.

An example of this is the prime sieve problem where each MPI process is a sieve element. As long as the generator does not receive a message that the required number of primes is found, it keeps pumping out numbers. This flow if uncontrolled floods the receivers down stream. Furthermore, when the terminate message is finally received by the sender, the termination message needs to wait for all the previous messages to clear before traveling down the sieve to terminate it. That means by the time the last prime is found, there are many useless numbers pumped down the chain using up compute cycles and delaying the movement of the termination message.

On the UBC’s cluster Cyclops, the communication-intensive prime sieve problem was ran using 16 compute nodes each with 8 cores. One version used the library’s communication channels while the other was pure FG-MPI without the use of the proposed library in any way. After 3 runs each calculating 38400 primes, the version using Flow-Controlled channels used on average 109 seconds while the non-flow controlled version used 70s (36%)

\footnote{The number of runs was chosen to be at least 3 in order to smooth out anomalies}
less time when using regular send and receive). However, the actual time spend running the generator was 62 seconds using regular send and receives while the Flow-controlled channels consumed 64 seconds resulting in only a 3% penalty in performance. The extra run time (less than 45 seconds) was spent doing other one-time startup initialization such as starting up proxie processes in addition to 38400 MPI processes and setting up their communication channels. For a program that runs for a long time, the startup cost may not be a significant performance lost.

The OSU latency benchmark [33] was ran and found that for inter-node communication on the Cyclops cluster, 4 byte messages had a latency of 15.7 microseconds for FG-MPI while the channels version had a latency of 26.3 microseconds (68% increase). For 4KB messages, the regular FG-MPI latency is 62.5 microseconds while the channels version had a latency of 62.5 microseconds (a 0% increase).
Chapter 6

Conclusion

In this thesis, we have proposed a four stage approach to specifying parallel software architecture and placement. This allows a separation of concerns with separate specifications for architecture interaction, software size, resource capacity, and function. The architecture specification borrows ideas from component-based software design while the syntax is based on FSP. We’ve also implemented (CM)$^2$PI which converts the specifications into a boilerplate that can be compiled with the user’s code to produce a runnable system.

The four stage approach includes the Communication Process Architecture (CPA) for specifying the architecture design and communication interactions, the Communication Process Architecture Variables (CPAV) for parameterizing the architecture, the Container Specification for defining the available resources, and the Function Specification for assigning functions to the processes defined in the architecture. This decoupling of decisions allows the user to make distinct changes to the system without having to make compatible changes to both the execution configuration and mapper. By externalizing MPI ranks, the developer does not have to edit their program.

We looked at three different types of communication techniques to introduce channel communication to make it easier to compose systems independent of MPI rank. The aim of these techniques were to make the programming code depend solely on the CPA architecture specification and be fully compatible regardless of the decisions made in the other stages (especially with regards to system size and process placement).

The first communication technique we looked at was process discovery using the names of the processes as defined in the architecture. This ser-
vice is synonymous to a “white pages” service. The lookup is done at the beginning of execution and does not incur extra per-message cost as the communication is still determined by the user.

The second communication technique we considered was the use of hierarchical communicators. They are initialized at the start of execution and their naming is also based on the architecture design. These were more suited to collective communication.

The third communication technique we examined were channels. This method was the most concise as the code only needs to know the local port names defined in the architecture for that process. The actual channel is specified between ports in the architecture. Regardless of how a process is composed, the only information it needs in order to communicate is the name of its ports.

To tackle the common problem of flow control and termination we also proposed flow-control channels which builds on the channel communication by providing a communication API with the flow control and termination. This method incurs a per-message overhead but the overhead is mostly in start-up so its effect is amortized over time. Flow-controlled channels were included as a way to simplify the handling of common communication issues in MPI. Even though this method does not guarantee that the receiver will not run out of buffer space (due to there being too many senders), it reduces the chance of that happening especially as a result of a single channel.

With the decoupled specifications, parameters of system size and resource sizes are distinct and is suitable for integration with other optimization tools. By using optimization tools to tune parameter, it is possible to automatically fit parallel systems to its resources wherever it is deployed. This makes parallel programs more versatile and scalable.

6.1 Future Work

The current CPA architecture design allows for compositions and simple repetitions to create commonly used patterns. However, some of the more complex but well studied grid networks like hypercubes and toruses were not
6.1. Future Work

explored in the research. It may be interesting to see what kind of change or support needs to be made in order to specify those topologies. The goal is still to keep the specification as general as possible because there could be endless numbers of architecture designs and complex architectures should intuitively require a more complex specification on the user’s part.

Currently, the integration with optimization tools requires that the search space be recompiled ahead of time. In a way, this is efficient if a search with random starts is used then the search space does not have to be recompiled again. However, in a single search, a lot of the search space might not be used and pre-compiling a large space will create a large executable and takes extra time. A better approach to this is to recreate the tool as a run-time tool developed in C. Then during execution, the specification is read and the appropriate mapping is created dynamically and run.

If the creation of mappings can be created dynamically, it may also be possible to perform dynamic load balancing. This way, it is possible to make fine adjustments as the code runs. This was not implemented in our research because it will then be difficult to verify and inspect resulting mappings if code is not generated. Currently, the code generated is intended to be understandable by users and is to be similar to what a user might have written as a boilerplate. Furthermore, our focus was on finding static placement for system software which will then use all of its run-time compute power on the task and not on the placement. However, more flexibility will result if the tool was run-time based.

The issue with receiver buffer’s finite space is partially mitigated by the flow-controlled channels so that a single channel should not overwhelm a single target. However, if there were many channels sending to one receiver, even receiving just one message from each channel may be sufficient to overwhelm the receiver’s buffer. With FG-MPI, this issue is exacerbated as more co-located MPI processes means there is less memory available to each MPI process. MPI program correctness issues are interesting future research.

While integration with optimizations tools require a single goodness value, the overall run time of a program might not always be the best indication of performance. Considering the underlying issues are idle-processing
6.1. Future Work

and communication overhead, further research could investigate the calculation of a better value to optimize on. This is especially important for programs that run continuously rather than terminate.
Bibliography


Bibliography


Appendix A

Specification Notation

A.1 Composition Notation

A.1.1 MPI Process Declaration

Similar to a function prototype, each MPI process is specified with parameters but they are dynamic input and output parameters (as ports for MPI communication). Each MPI process can also have static parameters (as parameters to function) provided at time of function invocation.

Consider the following: The MPI process P has two input ports IN\_1 and IN\_2 as well as two output ports OUT\_1 and OUT\_2 along with two static parameters of type PARAM\_TYPE\_1 and PARAM\_TYPE\_2:

\[
P = \text{inports}<\text{IN}\_1, \text{IN}\_2> \text{outports}<\text{OUT}\_1, \text{OUT}\_2> (\text{PARAM\_TYPE}\_1, \text{PARAM\_TYPE}\_2)
\]

A.1.2 Concurrent Composition of MPI Processes

To group a set of MPI processes to share a single communicator, a composition is created. The multiplicity of each process can be specified as a equation (which can include variables - to be specified). The default multiplicity is 1.

In the example below, the concurrent execution of MPI process P (NUM\_P instances) and S is represented by the composition Q:

\[
|| Q = (P[\text{NUM}\_P] || S)
\]
A.1.3 Concurrent Composition of Compositions

To group a set of compositions to share a single communicator, a new composition can be created. The multiplicity of each composition is specified as an equation (which can include variables - to be specified). The default multiplicity is 1.

In the example below, the concurrent execution of MPI process P and compositions NUM,Q x Q and R is represented by the new composition T:

\[
|| T = (P || Q[NUM_Q] || R)
\]

A.1.4 Intermediate Composition

Compositions can be regular compositions (used for instantiation) or an intermediate compositions (to be further composed inside another intermediate or regular composition). The minus sign (-) is used to indicated an intermediate composition.

In the example below, the composition Q is an intermediate composition while T is a regular composition.

\[
|| - Q = (P || S)
\]
\[
|| T = (P || Q || R)
\]

A.1.5 Communication Channels

Compositions of MPI processes of other compositions can create named directional communication channels. An optional tag for the channel can be specified. The process name can be prefixed to the port to clarify which process the port belongs to. If only one of the processes possess this port then the name is optional.

In the example below, the composition Q containes a concurrent execution of P and S where a channel is created between port NEXT of process P to port PREV of process S with a tag MSGTAG (a variable to be specified). Also a channel is created between port TERMINATE of process P to the only process that contains the port END. When using flow-controlled
A.1. Composition Notation

channels, a buffersize value can also be set for the buffer proxy’s buffer size. The / symbol represents “rename” in [28]. We’ve adapted that symbol to represent the declaration of channels.

\[
|| Q = ( P || S ) \\
/ ( P.NEXT -> S.PREV tag MSGTAG buffersize 10, P. TERMINATE -> END )
\]

A.1.6 Port Hiding

All unconnected (no channels were created for) ports by default are not exposed beyond the current composition. To expose a port outside of the composition, an intermediate port must be specified. The @ symbol represents “expose”.

In the example below, the port S.NEXT is connected to the intermediate port EXT.NEXT which is available to further compositions of Q:

\[
|| Q = ( P || S ) \\
/ ( S.NEXT -> EXT.NEXT ) \\
@ ( EXT.NEXT )
\]

A.1.7 Patterns for Communication Port Declaration

Loops can be used to assign channels to ports in a simple pattern. An equation can be used to identify the port of the nth process of that type.

In the example below, a range R is created from 0 to END inclusively. This range can be reused throughout the rest of the specification. The composition Q contains NUM_R instances of process R and a channel is created from the port NEXT of the previous process to the port PREV of the next process in a chain-like fashion.

\[
R = [0..END] \\
|| Q = ( R[NUM_R] ) \\
/ (for i in R { NEXT[i] -> PREV[i+1]} )
\]
A.1.8 Forming a Structure

The MPI process declaration and compositions are to be specified as the architecture structure. Any uninitialized variables must be declared after the structure name for further declaration.

A structure of STRUCTURE_NAME with variables NUM_P and PARAM_TYPE_1 is to be wrapped as follows:

```latex
structure STRUCTURE_NAME (NUM_P, PARAM_TYPE_1)
{
  P = inport<> outport<> (PARAM_TYPE_1)
  ...
  || Q = ( P[NUM_P] )
}
```

A.1.9 Variable Initialization

Any uninitialized variables from a structure must be initialized. Variables can be a simple equation of another variable.

In the example below, a variable VAR_1 is initialized to the value 10 and the variable VAR_2 is initialized to the value of VAR_1 + 5.

```latex
VAR_1 = 10
VAR_2 = VAR_1 + 5
```

A.1.10 Initializing All Variables

All variables that are uninitialized in a structure needs to be provided in an initialization. An initialization contains one or more declarations of a variable's values. The structure to be initialized is to be specified in the initialization specification. The declaration of extra variables that are not used by the structure is acceptable.

In the example below, we have an initialization of the structure STRUCTURE_NAME by declaring the values of variable VAR_1 and VAR_2.

```latex
initialize STRUCTURE_NAME
{
```
A.2 Placement Notation

The following is a description of the notation used for specifications related to program placement:

A.2.1 Container Specification

A container specification states how many of each MPI process can be in an OS process. The MULTIPLICITY of each MPI process may be a solvable equation (using operators - + * /) or a wild card (*) indicating zero or more, or plus (+) indicating one or more. The maximum combined multiplicity is indicated by the optional MAX_MULTIPLICITY value. An optional composition prefix COMP_PREFIX can be specified.

```
containerspec CONTAINER_NAME = < COMP_PREFIX . PROC_NAME_1 [MULTIPLICITY_1], PROC_NAME_2[MULTIPLICITY_2], ... > / MAX_MULTIPLICITY
```

A.2.2 Container Allocation to OS Processes

The specification of the number of OS process to allocated to each container specification is of the form:

```
boxes ( CONTAINER_SPEC_1[MULTIPLICITY_1] || CONTAINER_SPEC_2 [MULTIPLICITY_2] || ...)
```

A.2.3 Forming a Container Specification

If an equation is used for the multiplicity, a simple loop pattern can also be used. The above two container information is to be wrapped as follows:
containers {
  R = [0..END]
  containerspec CONTAINER_NAME = < PROC_NAME_1[LOOPVAR* MULTIPICITY_1], PROC_NAME_2[MULTIPLICITY_2], ... > /
    MAX_MULTIPLICITY
  ...
  boxes ( for LOOPVAR in R { CONTAINER_SPEC_1 } ||
    CONTAINER_SPEC_2[MULTIPLICITY_2] || ...)
}

A.2.4 Mapping Functions to MPI Processes

To specify the C function (with standard main function signature int PFUN-
CTION(int argc, char* argv[])) name and corresponding header file, the form
is as follows:

```
functionmapping{
  P = { "PHEADER.h", "PFUNCTION" }
  ...
}
```

A.2.5 Glue Specification

Glue functions (callback functions) can be specified. All parts are individ-
ually optional. The header file that has the function’s prototype is to be
provided as well as the name of the function. See Section 4.4 for function
signature requirements.

```
glue {
  gluemapping {
    envcondition =  { "ENV_CONDITION", "CONDITION_VALUE" }
    mapinit = { "HEADER_1", "function_1" }
    mapfinalize = { "HEADER_2", "function_2" }
    maininit = { "HEADER_3", "function_3" }
    maincleanup = { "HEADER_4", "function_4" }
    getparams = { "HEADER_5", "function_5" }
  }
}
```
A.2. Placement Notation

\[
\begin{align*}
\text{freeparams} &= \{ \text{"HEADER}_6\}, \text{"function}_6\} \\
\text{permutecommunicator} &= \{ \text{"HEADER}_7\}, \text{"function}_7\}
\end{align*}
\]

A.2.6 Environment Variables

Global environments to be passed to the program can be specified as follows:

\[
\text{genv} \{
  \text{GENV}_1 = \text{GENVALVE} \\
  \ldots \\
\}
\]

A.2.7 Combining Specifications

Different specification files can be passed to the tool individually or imported by other specification files.

\[
\begin{align*}
\text{include SPEC}_1.\text{containers} \\
\text{include SPEC}_1.\text{functions} \\
\ldots
\end{align*}
\]
Appendix B

API

B.1 Process Discovery API

The following is the API for discovering MPI processes that a run using the generated mapping. Each API returns 0 if success, -1 if error.

- `int CMCMPI_get_info_by_rank(int rank, int* opid, const char** name);`
  Gets the OS process ID and the name of the rank given.

- `int CMCMPI_get_ranks(int opid, char* name, int** ranks, int* size);`
  Gets the ranks with the given OS process ID and name.

- `int CMCMPI_get_all_ranks(char* name, int** ranks, int* size);`
  Gets the ranks with the given name.

- `int CMCMPI_get_collocated_ranks(char* name, int** ranks, int* size);`
  Gets the ranks with the given name in the same OS process as the caller.

- `int CMCMPI_Get_numnodes(int* size);`
  Gets the total number of nodes the program is executing on.

- `int CMCMPI_Get_nodeid(int* id);`
  Gets the current process's node id. Node ids start from 0 to size-1.
B.2 Communicators API

A simple API is available for the user’s code to obtain the instantiated communicators. Each API call returns 0 if success, -1 if error.

- int CMCMPI_Init();
  Initializes the communicators (creates them) and other process information. To create communicators efficiently, they are created in parallel where possible. Since this is a hierarchical pattern, once a child communicator is created, further splitting into other communicators will not require the cooperation of the sibling communicator. This means that all the siblings can split in parallel. If one process calls this function, all processes must call this function. This creates a structure container information for all the collocated processes in the current OS process. This includes for each process, their communicators, and pointers to their name (const char).

- int CMCMPI_Finalize();
  Frees up any structures created to store communicators and other process information. Also, communicators are freed up.

- int CMCMPI_Comm_get(int levels, MPI_Comm* comm);
  Gets the communicator for the particular level. Level 0 is MPI_COMM_WORLD. Communicators are then created for each level of the naming prefix. For example, a process named App.Data.Info. At level 1, a communicator contains all processes who share the prefix App. At level 2, a communicator contains all processes...
who share the prefix App.Data. At level 3, a communicator contains
all processes who share the prefix App.Data.Info.

- \texttt{int CMCMPI\_Comm\_get\_by\_name(char* name, MPI\_Comm* comm)};
  
  Gets the communicator for the particular naming prefix. A blank
  name gets \texttt{MPI\_COMM\_WORLD}.

- \texttt{int CMCMPI\_name(const char** name)};
  
  Gets the current process’s name based. The name is based on the
  composition specified in the CPA.

- \texttt{int CMCMPI\_ospid(int* ospid)};
  
  Gets the current process’s process ID.

## B.3 Channels API

A simple API is available for the user’s code to obtain the instantiated
channel information which was specified in the CPA.

- \texttt{int CMCMPI\_get\_chan\_procname(int id, chan\_procname\_t** channel)};
  
  Gets the channel information struct. The values inside the struct are
  available to the program. For example, to get the next and previous
  process of a prime sieve process (see Listing B.1).

  \begin{verbatim}
  chan\_primeelement\_t* chan\_info = get\_chan\_primeelement(rank);
  int prevproc = chan\_info->prev.rank;
  int prevtag = chan\_info->prev.tag;
  int nextproc = chan\_info->next.rank;
  int nexttag = chan\_info->next.tag;
  \end{verbatim}

  \textbf{Listing B.1: Example usage of channel structs}
B.4 Flow-Controlled Channels API

In the following, we give the API for the flow-controlled channel library.

- **Channel Status**
  The return code for the API calls below are as follows:

  - **CHANNEL_NEW**: Indicates that the channel hasn’t been opened.
  - **CHANNEL_OPEN**: Indicates that the channel has been opened and is ready for sending (for the outbound end) and receiving (for the inbound end).
  - **CHANNEL_CLOSING**: Indicates that the receiver has requested that the channel be closed. The channel is still effectively open at this status.
  - **CHANNEL_CLOSED**: Indicates that the channel is now closed and that no further messages can be sent (for the outbound end) or received (for the inbound end).

- **int channel_open_send(CMCMPI_Chainfo_t* chan);**
  Opens an outbound channel. Returns channel status. If channel status is **CHANNEL_OPEN** then messages can be sent to the channel. If channel status is **CHANNEL_CLOSED** then the channel is closed and no messages can be sent to the channel.

- **int channel_open_recv(CMCMPI_Chainfo_t* chan, int messagesize);**
  Opens an inbound channel. Return channel status. If channel status is **CHANNEL_OPEN** then messages can be received from the channel.

- **int channel_send(void* buf, int count, MPI_Datatype type, CMCMPI_Chainfo_t* chan);**
  Sends the data in **buf** to the channel. Returns channel status.

- **int channel_recv(void* buf, int count, MPI_Datatype type, CMCMPI_Chainfo_t* chan);**
  Receives data from the channel to **buf**. Returns channel status. If channel status is **CHANNEL_CLOSED** then buf is invalid and the channel
is now closed (all future receives will have status `CHANNEL_CLOSED`). The reason for having the channel status piggybacking with the data is to exploit MPI’s message ordering guarantee as well as to reduce extra messages purely for status information. By using the same source, destination, and tag to send data and status information, we can ensure that the status and data messages will arrive in order. That means if the last message sent is a `CHANNEL_CLOSED` status, then we can ensure all previous data will arrive at the receiver before this status message.

- **int channel_close(CMCMPI_Chainfo_t* chan);**
  If a sender invokes this function, then the channel is closed to this sender and no further messages can be sent. If a receiver invokes this function, then the channel remains open but the channel status changes to `CHANNEL_CLOSING` to the sender. It is then up to the sender to invoke this function to close the channel.

- **int channel_status(CMCMPI_Chainfo_t* chan);**
  Returns the current channel status of the channel.
Appendix C

Examples

In this appendix, we will discuss how to use (CM)$^2$PI and three examples:

- A hello world program shows how to make an MPI program run using FG-MPI through the help of (CM)$^2$PI.
- A prime sieve program shows how to create ports and use channels.
- A farm program shows how to compose farm programs to look at more complex usages of channels.

C.1 Flow of Execution

Figure C.1: The order in which the tool is run from compilation to execution of the program.
C.2. HelloWorld Example

All the specifications specified are input into the \((\text{CM})^2\text{PI}\) tool. The typical execution scenario (see Figure\[\text{C.1}\]) involves compiling the output of the tool into a \texttt{cmmpi.a} object file and configfile. Then the user compiles their own code and links with the \texttt{cmmpi.a} object file to create an executable. A shell script is generated by the mapping’s Makefile containing the \texttt{mpiexec} command.

C.1.1 Compiling the Generated Code

The tool’s output is compiled by calling the generated Makefile and providing it with the final executable name through the \texttt{APP} variable. The following is how the generated Makefile should be called:

\[
\text{make APP=execname}
\]

C.2 HelloWorld Example

The first example shows the minimal basics required to go from specifying a program in MPI to FG-MPI to using the \((\text{CM})^2\text{PI}\) tool. This basic example does not expose much of the advantages of using the \((\text{CM})^2\text{PI}\) tool.

C.2.1 MPI

The MPI version of the program is shown in Listing\[\text{C.1}\]

Listing C.1: helloworld.c

```c
#include <stdio.h>
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello I am MPI process %d of %d\n", rank, size);
}
```

87
C.2. HelloWorld Example

```c
MPI_Finalize();
return 0;
}
```

Compiled using:
```bash
mpicc helloworld.c -o hello
```

Executed to run (for example, on 4 cores) using:
```bash
mpiexec -n 4 ./hello
```

C.2.2 FG-MPI

To make the program work with FG-MPI, we need to basically add a mapping boilerplate (see Listing C.2) as well as include fgmpi.h and modify the name of the main function (see Listing C.3). The main function’s name needs to be changed because the main function in the boilerplate needs to start up FG-MPI.

Listing C.2: boilerplate.c

```c
#include <mpi.h>
#include <stdlib.h>
#include <fgmpi.h>

/* Forward declarations */
int mymain(int argc, char **argv);

/* mapping function */
FG_ProcessPtr_t my_map_function(int argc, char **argv, int rank)
{
    if ( (rank == MAP_INIT_ACTION) || (rank ==
        MAP_FINALIZE_ACTION) ) return (NULL);
    else return (&mymain);
}

FG_MapPtr_t lookupfunc( int argc, char** argv, char* str)
{
    return (&my_map_function);
}
```

88
C.2. HelloWorld Example

/* MAIN FUNCTION */
int main ( int argc , char * argv[] )
{
    FGmpiexec(&argc, &argv, &lookupfunc);
    return 0;
}

Listing C.3: helloworld.c

#include <stdio.h>
#include <mpi.h>
#include <fgmpi.h>

int mymain(int argc, char *argv[])
{
    int rank, size;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    printf("Hello I am MPI process %d of %d\n", rank, size);

    MPI_Finalize();
    return 0;
}

Compiled using:
mpicc -c boilerplate.c -o boilerplate.o
mpicc -c helloworld.c -o helloworld.o
mpicc boilerplate.o helloworld.o -o helloworld

Executed to run (for example, on 8 cores with 80 MPI processes in total) using:
mpiexec -n 8 -nfg 10 ./helloworld
C.2. **HelloWorld Example**

C.2.3 **FG-MPI with (CM)**^{2}PI

With (CM)**^{2}PI, the boilerplace is no longer needed. Instead, we split up the specification into 4 separate files: cpa (see Listing C.4), cpav (see Listing C.5), containers (see Listing C.6), functions (see Listing C.7). The programming (see Listing C.8 and Listing C.9) need to important cmmpi.h and optionally execute cmmpi_Init() and cmmpi_Finalize() if their features is required.

Listing C.4: cpa/hello.cpa

```c
#include hello.cpav
#include hello.container
#include hello.functions

structure helloworld (numHello) {
    HelloWorld = inports <> outports <> ()
    || HW = ( HelloWorld[numHello] )
}
```

Listing C.5: cpa/hello.cpav

```c
initialize helloworld {
    numHello = 10
}
```

Listing C.6: cpa/hello.container

```c
containers {
    containerspec helloWorldContainer = < HelloWorld[*] >
    boxes ( helloWorldContainer[8] )
}
```

Listing C.7: cpa/hello.functions

```c
functionmapping {
    HelloWorld = { "hello.h", "mymain" }
}
```
C.2. HelloWorld Example

Listing C.8: helloworld.c

```c
#include <stdio.h>
#include <mpi.h>
#include <fgmpi.h>
#include "cmmpi.h"
#include "hello.h"

int mymain ( int argc , char * argv[])
{
    int rank , size;
    MPI_Init(& argc , & argv);
    CMCMPI_Init(); // OPTIONAL: If communicators/channels/node information not used
    MPI_Comm_rank ( MPI_COMM_WORLD , & rank );
    MPI_Comm_size ( MPI_COMM_WORLD , & size );
    printf ( "Hello I am MPI process %d of %d\n" , rank , size );
    CMCMPI_Finalize (); // OPTIONAL: If communicators/channels/node information not used
    MPI_Finalize ();
    return 0;
}
```

Listing C.9: hello.h

```c
/* Processes */
int mymain(int argc, char **argv);
```

Compiled using:
```
../release/fgmapper.sh cpa/hello.cpa
(cd mapping; make APP=helloWorld)
mpicc -c -Imapping helloworld.c -o helloworld.o
mpicc helloworld.o mapping/cmmpi.a -o helloWorld
```

Executed to run (for example, on 8 cores with 80 MPI processes in total) using:
The primesieve consists of 3 types of processes: the generator, the sieve elements, and the last element (see Figure C.2).

The generator’s job is to basically pump out incrementing numbers in to the sieve. It has one output port that it is to send numbers out of.

The sieve elements takes the first number it receives as a prime number and pass on any future numbers it receives that its number does not divide. This MPI process needs an input port as well as an output port.

The last element is basically the last prime number and has the job of terminating the program. This MPI process needs an input port for receiving numbers only.

The basic ports described above are the bare minimal required for this program to work. Termination request of the channels starts from the last element and propagate to the start which then terminates channels from start to end. When all channels are closed, the program is terminated. A more efficient way is to provide a direct shortcut channel (a control channel) from the generator to the last element. This way, termination request from the last element can be sent directly to the generator and the resulting terminations only need to propagate from start to end.

Figure C.2: The generator needs to send out numbers. The sieve elements receive numbers from the front of the sieve and pass numbers on. The last element needs to terminate the program when it receives a prime. The control channel is used for optimization of the termination process.
C.3. PrimeSieve Example

Once these MPI processes are defined, it is time to compose them together to make them interact (see Listing C.10). In any prime sieve, a single generator is needed and a single last element is required. The sieve should however be resizable and be parametrized by a variable size. This can be done through specifying a multiplicity for the number of sieve elements. But to hide this information from the outside of the sieve, we make an inner composition of the sieve element so that to the big overview, we see 1 generator, a sieve section, and a last element. In order to allow these to communicate, channels need to be created. In order for the sieve section to communicate to the outside, intermediate ports are created as “holes” for channels to attach through without exposing the details of the internal communication (see Figure C.3).

Figure C.3: A description of the architecture would indicate a variable sized sieve element section with 1 generator and 1 last element.

Listing C.10: prime.cpa

```c
structure primesieve (numsieve) {
    PrimeStart = inports <>
        outports <next, control> ()
    PrimeElement = inports <prev>
        outports <next>
    PrimeEnd = inports <prev, kill>
        outports <> ()

    R = [0..numsieve-2]
    ||- PrimeSieveChain = ( PrimeElement[numsieve] )
        / ( for i in R { next[i] -> prev[i +1]},
            receive -> prev[0],
            next[numsieve-1] -> send)
```
C.3. PrimeSieve Example

```plaintext
@ (receive, send)
|| PrimeSieve = ( PrimeStart || PrimeSieveChain ||
PrimeEnd )
/ ( PrimeStart.next ->
PrimeSieveChain.receive,
PrimeSieveChain.send -> PrimeEnd
.prev,
PrimeStart.control -> PrimeEnd.kill)
}

The number of primes is parameterized by a variable which is placed inside the CPAV specification (see Listing C.11).

Listing C.11: prime.cpav

```plaintext
initialize primesieve {
    numprimes = 5000
    numsieve = numprimes - 2
}
```

Then the placement of the processes is specified. There is a lot of flexibility here. For example, it is possible to place the generator on the first core and the last element on the last core (see Listing C.12).

Listing C.12: prime.container

```plaintext
containers {
    containerspec primeSieveStart = < PrimeStart[1],
        primeSieve >
    containerspec primeSieve = < PrimeElement[*] >
    containerspec primeSieveEnd = < primeSieve, PrimeEnd
        [1] >

        primeSieveEnd[1] )
}
```

It is also possible to run everything on a single core (see Listing C.13). This is good for checking for errors as everything is run concurrently instead of in parallel.
C.3. PrimeSieve Example

Listing C.13: prime.container

```
containers {
    containerspec primespecial = < PrimeStart[1],
    PrimeElement[*], PrimeEnd[1] >

    boxes ( primespecial[1] )
}
```

It is also possible to let the system decide completely and say there is 8 cores only (see Listing C.14).

Listing C.14: prime.container

```
containers {
    containerspec primespecial = < PrimeStart[*],
    PrimeElement[*], PrimeEnd[*] >

    boxes ( primespecial[8] )
}
```

Or even limit the nfg on the earlier part of the program (see Listing C.15). This has proven to make a big performance impact on the prime sieve problem when the number of primes is large.

Listing C.15: prime.container

```
containers {
    containerspec primeSieveStart = < PrimeStart[1],
    primeSieve > / 10
    containerspec primeSieve = < PrimeElement[*] >
    containerspec primeSieveEnd = < primeSieve, PrimeEnd
    [1] >

    primeSieveEnd[1] )
}
```

And finally, the functions that needs to be attached to each MPI process is assigned in the function specification (see Listing C.16). Glue code is assigned depending on the value of the GENV variable mapping.
C.3. PrimeSieve Example

Listing C.16: prime.functions

```c
functionmapping {
    PrimeStart = { "primeSieve.h", "generator" }
    PrimeElement = { "primeSieve.h", "sieveElement" }
    PrimeEnd = { "primeSieve.h", "lastElement" }
}
glue {
    gluemapping {
        genvcondition = { "MAPPING", "sequential" }
        mapinit = { "mapping-custom.h", "init_seq" }
    }
    gluemapping {
        genvcondition = { "MAPPING", "strides" }
        mapinit = { "mapping-custom.h", "init_strides" }
    }
    gluemapping {
        genvcondition = { "MAPPING", "random" }
        mapinit = { "mapping-custom.h", "init_random" }
    }
    gluemapping {
        maincleanup = { "mapping-custom.h", "my_free_params" }
        permutecommunicators = {"mapping-custom.h", "custom_comm_permute"}
    }
}
genv {
    GROUP = 25
    SEED = 6
    CUTS = 20
}
```

The channels can be used in two ways, one is through the information based option, the other is through flow-controlled channels (see Listing
Listing C.17: primesieve.c Section showing the usage of communication channels for the sieve element.

```c
int sieveElement(int argc, char **argv)
{
    int rank, size;

    MPI_Init(&argc,&argv);
    CMCMPI_Init();

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);

    // Channel information
    CMCMPI_Chan_primeelement_t* chan_info;
    CMCMPI_Get_chan_primeelement(rank, &chan_info);

    // Opening the channels
    channel_open_recv(&(chan_info->prev), sizeof(int));
    channel_open_send(&(chan_info->next));

    uint32_t num, num2, myprime = 0;
    while (TRUE)
    {
        if (channel_recv(&num, 1, MPI_INT, &(chan_info->prev)) == CHANNEL_CLOSED)
            {
                // No more incoming data, close outbound channel
                channel_close(&(chan_info->next));
                break;
            }
        if (myprime == 0)
            {
                myprime = num;
                printf("%u, ", myprime);
            }
        else if (num % myprime)
            {
                /* Not divisible by this prime */
            }
    }
}
```
C.4 Farm Example

The farm example involves a source and a sink while there are workers that perform the tasks. We will discuss the three architecture composition possible for a farm.

A single farm where jobs go from source to workers to sink is shown in Figure C.4 as A) and is equivalent to that specified in Listing C.18 as FarmAppA.

A more complex example involves chaining the farms together such that the output of one travels to the next. This is shown in Figure C.4 as B) and Listing C.18 as FarmAppB. This type of farm might be used to create a pipeline-like situation where processed data is reprocessed in another method.

A third example involves splitting up tasks to workers which then further splits up the tasks into more workers. Effectively, farms are nested together. This is shown in Figure C.4 as C) and Listing C.18 as FarmAppC. This type of problem resembles a multiple level decomposition problem.
Figure C.4: Three patterns of composition. A farm can be by itself A), chained B), and nested C).
C.4. Farm Example

Listing C.18: farm.cpa

structure farm (numworkers, nestnum) {
  Source = inports <input> outports <sendjobs> ()
  Worker = inports <jobs> outports <result> ()
  Sink = inports <getresults> outports <output> ()

  R = [0..numworkers-1]
  K = [0..nestnum-1]

  || Workers = ( Worker[numworkers] )
    /
    / ( for i in R { alljobs -> jobs[i]},
      for i in R { result[i] -> allresults}
    )
    @ (alljobs, allresults)

  || FarmP = ( Source || Workers || Sink )
    /
    allinput -> Source.input,
    Source.sendjobs -> alljobs,
    allresults -> getresults,
    Source.sendjobs -> getresults,
    output -> alloutput

    @ (allinput, alloutput)

  || FarmP2 = ( Source || FarmP[nestnum] || Sink )
    /
    for i in K { Source.sendjobs -> FarmP.
      allinput[i] },
    for i in K { FarmP.alloutput[i] -> Sink.
      getresults }

    )

  || FarmAppA = ( FarmP )
  || FarmAppB = ( FarmP[2] )
    /
    ( FarmP.alloutput[0] -> FarmP.allinput[1] )

  || FarmAppC = ( FarmP2 )
}

100
Appendix D

Code sketch - Communicator
Permutation

Listing D.1 shows the implementation of how the permutation callback function is called and how the permutation is propagated.

Listing D.1: Permutation of communicators

```c
MPI_Comm original_communicator = communicators[max_index-1]
MPI_Comm new_communicator
permute_comm_function(original_communicator, &
    new_communicator) // Call-back function
communicators[max_index-1] = new_communicator

MPI_Comm prev_old_comm = original_communicator;
for (current_level = lowest_level - 1; level >= 0; --level)
{
    MPI_Comm prev_comm = communicators[level+1]
    MPI_Comm current_comm = communicators[level]
    MPI_Comm new_comm;

    int newcommrank, prevnewrank
    MPI_Comm_rank(prev_comm, prevnewrank);
    MPIX_Comm_translate_ranks(prev_old_comm, 1, &
        prev_new_rank, current_comm, &newcommrank)
    MPI_Comm_split(current_comm, 0, newcommrank, &new_comm);
    communicators[level] = new_comm

    MPI_Comm_free(prev_old_comm);
    prev_old_comm = current_comm;
}
```