Supporting a process-oriented model in MPI through fine-grain mapping

by

Cody Ryan Brown

B.Sc., The University of British Columbia, 2006

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

MASTER OF SCIENCE

in

The Faculty of Graduate Studies

(Computer Science)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2012

© Cody Ryan Brown 2012
Abstract

The need for intuitive parallel programming designs has grown with the rise of modern many-core processors. Process-oriented models promote high scalability by using small isolated components that interact to produce complex applications. Such models are intuitive by forcing scalability to be a design requirement. The popular MPI messaging library has not exploited fine-grain models due to its coarse-grain implementations. The binding of processes often uses a coarse-grain management system, which is done by sequentially assigning ranks to a list of machines. This may be suitable for coarse-grain applications, but inadequate for fine-grain applications with large process grouping demands; a more flexible, manageable and scalability specification is necessary to support a process-oriented model.

The use of FG-MPI exposes additional concurrency through a new layer of mapping by providing smaller units of parallelism: a desirable feature in function-level programming. This collocation layer requires a fine-grain mapping mechanism to optimize communication. A graph specification is proposed that allows communication patterns, collocation of MPI processes, and binding optimizations to be extracted from such a structure.

The work presented extends and evaluates MPI to a fine-grain process-oriented model and provides a graphical mapping and binding specification. Evaluation of function-level applications is done through Pilot, a CSP-like library for MPI. The smallest unit of parallelism in this architecture is evaluated and shows that small communication overheads occur when comparing hundreds of large tasks to hundreds of thousands of fine-grain tasks. The graph representation is based on Kahn Process Networks. This provides a simplistic and intuitive model to represent and organize large function-level applications. A tool is developed that reads in a graph structure and
Abstract

performs operations such as auto-constructing wiring diagram code, determining optimal collocated maps based on communication, and producing a binding specification. This tool is modular and extensible to other graph related operations.
Table of Contents

Abstract .................................................. ii

Table of Contents ........................................ iv

List of Tables ............................................. vi

List of Figures ............................................ vii

List of Code ............................................... viii

Acknowledgements ......................................... xi

Dedication .................................................. xii

1 Introduction ............................................ 1
  1.1 Motivation ........................................... 4
    1.1.1 An extended mapping layer ....................... 5
    1.1.2 Function-level parallelism ....................... 6
    1.1.3 A distributed algorithm approach ............... 9
  1.2 Organization ....................................... 11

2 Background and Related Work ......................... 13
  2.1 MPICH2 ............................................ 13
    2.1.1 Communication .................................. 14
    2.1.2 Process-core binding ............................ 15
  2.2 Fine-grain MPI .................................... 17
  2.3 Pilot ............................................... 18
  2.4 Related work ...................................... 20
List of Tables

5.1 Communication costs for a single one-way MPI\_Send in FG-MPI for a message size of 8192 bytes on the test setup. . . . 55

5.2 A seven module chain with 35 collocated processes within one OS-process. Task size and number of tasks were varied to maintain a constant amount of work. Average time per intra-process communication is shown. . . . . . . . . . . . . . . . . . . . . . . . . 59
List of Figures

1.1 An illustrated example of an \( (a) \) SPMD and an \( (b) \) MPMD program. Solid circles represent processes, and rectangles are the tasks attached to the process. Dotted arrows represent the send-receive mechanism MPI provides. .......................... 2

1.2 A Sieve of Eratosthenes prime number implementation. Each circle is an MPI process and has a possible prime associated with it. Consecutive numbers are passed down until they are dropped or declared as a prime number. .......................... 10

2.1 The FG-MPI architecture. The shaded parts are the MPICH2 pieces that FG-MPI touches. .............................. 17

2.2 The Pilot execution flow. Pilot will run the configuration code and then start a PI.StartAll. This will spawn several process functions and then tell PI.MAIN to continue executing. 19

3.1 The single node case representing the hostfile. The solid rectangle is the physical representation. .......................... 25

3.2 The node and channel case where channels are one-way communication links between two nodes. This list of pairs represents the PI.CreateChannel command. The solid rectangle is the physical representation. .......................... 26

3.3 The node, channel, and hierarchy case. The solid rectangle is the physical representation. This represents a graph and can be illustrated in the same way as the representation. The hierarchical representation shown illustrates groups. .......................... 27
List of Figures

4.1 The shared global variable issue. Dotted circles are OS-processes, solid circles are MPI processes, and squares are shared global variables. (a) is the MPICH2 case, where each MPI process has its own global writable variable, while (b) is the FG-MPI collocated case, where several MPI processes share the same global variable; care must be made when writing to these variables. ................................................. 30

4.2 A simple processor task farm module showing five collocated processes. Tasks flow through the module where the Worker process will execute the task and the remaining four MPI processes buffer tasks. ...................................................... 33

4.3 Eight Pilot farm modules. The large encompassing dotted rectangles represent machines. (a) is stacked to optimize communication; in this case neighbouring modules are collocated together. (b) is distributed round-robin to optimize computation; in this case neighbouring modules are separated to allow for quick distribution of tasks over resources. ........................ .... 35

4.4 A GraphViz graphical representation of a Pilot farm. Circle nodes represent MPI processes, edges represent channels, and rectangles represent modules. ......................................................... 37

4.5 The architecture of the graph_map Python tool. This represents an approach where modules act on the graph information independently. The cmdlist, which contains skeletal information, can be manipulated and shared by modules. In this approach users may add custom modules. ................. 40

4.6 Illustrating the binding of MPI processes to OS-processes. The large dotted circles represent OS-processes, while the small solid circles represent MPI processes. (a) is the normal MPICH2 case, while (b) is the FG-MPI case. .................. 46
List of Figures

5.1 Showing a reducing unit of parallelism from (a) to (b) to (c), for each consecutive OS-process. Large dotted circles represent OS-processes, while small solid circles represent MPI processes. ................................................................. 54

5.2 A module Pilot farm varying task size. The amount of work per OS-process remains constant by changing the number of tasks accordingly. 63 and 126 modules were used in the one module and two modules per OS-process cases respectively. 63 cores were used over eight machines. ................................. 57

5.3 Speed-up for a seven module Pilot farm for a fixed task size and amount of work. Each module is mapped to its own OS-process and core on a distributed machine. .................. 61

5.4 Shows the linear relationship for the increase in communication computed over one node with constant work. The increase in time represents the overhead in the communication cost for the increased number of tasks. ......................... 62
List of Code

3.1 An example hostfile. This example will launch OS-processes in the provided order and wrap this list until all MPI ranks have been assigned. ................................................................. 24

4.2 Snippet definition for a repeated module initialization. .......... 42
4.3 Filled in snippet definition for a repeated module initialization. 42
4.6 The mapping functions FG-MPI uses to assign MPI ranks to functions. The listed example maps all ranks to a single function mainfg providing an SPMD behaviour. ................. 46
4.7 A sequential example of shifting an array in C. ................. 50
4.8 An OpenMP example of shifting an array in C. Care must be made with shared loop variables. .......................... 50
4.9 A function-level MPI example of shifting an array in C MPI. Array elements are MPI processes and the moving of numbers is done through message communication. ................. 50

A.1 The wiring code for the Pilot farm program. .................... 78
A.2 A prime number solver written in Pilot. Only the Worker process is shown. ......................................................... 79
A.3 A GraphViz representation of a farm module. .................. 80
A.4 Portions of a cmdlist structure from a Pilot farm graph. ... 81
Acknowledgements

This work would not be possible without several individuals for whom I am indebted too. First I would like to thank my supervisor, Alan Wagner, who has provided continual support, discussion, guidance, ideas and insight throughout the shaping of this thesis. The tremendous amount of assistance he has provided throughout my time at UBC has been very much appreciated. I would also like to thank Norm Hutchinson for his invaluable comments as the second reader to this thesis.

The work presented relies heavily on FG-MPI and I would like to acknowledge Humaira Kamal for developing and maintaining the software. Pilot has been used throughout this thesis and my thanks go to William Gardner for providing the library. I would personally like to acknowledge the debt I owe to Felix Herrmann and Michael Friedlander, who have provided overwhelming support in pushing me to where I am today; I am very grateful for your efforts.

Further appreciation must be shown to my fellow graduate students and labmates Matt Brehmer, John Chia, Patrick Colp, Sara Dadizadeh, Ben Jones, Andrej Karpathy, Jean-Sébastien Légaré, Mihir Nanavati, Mark Spear, Nathan Taylor, Paul Vanetti, and many others: all of whom have provided much needed sanity control at critical and opportune times.

Finally I am particularly grateful to my personal friends and family for their immense support. All of these people have directly or indirectly shaped this work for the better.
Dedicated to my Mother and Brother,
for all their support.
Chapter 1

Introduction

The organization of data and resources for massively parallel applications can be a challenging task in high performance computing (HPC). When dealing with a large number of parallel components, difficulty arises on how a user can easily and flexibly manage an ordering scheme. This organization can be viewed as a naming problem [68, 69], where units of parallelism must be named and assigned a physical location. When applied to massively parallel applications, the mechanisms for organizing these units of parallelism tend to not scale well. In the Message-Passing Interface (MPI) standard [24, 48, 50], a popular parallel programming model in HPC, the mapping of ranks and the binding of processes to hardware is left to the implementation. Popular implementations such as MPICH2 [5] and Open MPI [77] execute a binding specification based on hostnames with a round-robin algorithm. Implementations based on these, such as MVAPICH [51] and FG-MPI [22, 40] will often mimic the mapping and binding techniques as well: MPI will map the code to OS-processes and bind these OS-processes to machines.

The round-robin binding algorithm provides an adequate interface for Single-Program Multiple-Data (SPMD) applications where the number of parallelized units is small. This means that the unit of parallelism, or the smallest computational unit that can be expressed, in these applications are large to make up for a small number of available processes. In such applications, a binding technique that spreads the OS-processes over as much hardware as available is sufficient. In MPI, programming models with complex communication and process grouping demands are often avoided to maintain simplicity.

Multiple-Program Multiple-Data (MPMD) applications are parallel pro-
grams that often express a complicated communication and mapping pattern: workflows and cacti programming environments are examples. These applications involve multiple programs communicating together as shown in Figure 1.1. In such cases, certain components of the application may experience higher communication demands. The collocation of these components could be beneficial, but are often difficult to control. With current MPICH2 implementations, the user can manually specify the one-to-one binding of processes to hardware for a particular machine. This task becomes particularly difficult depending on the number and complexity of processes in the application, as well as the amount of physical resources available.

![Figure 1.1](image)

**Figure 1.1**: An illustrated example of an (a) SPMD and an (b) MPMD program. Solid circles represent processes, and rectangles are the tasks attached to the process. Dotted arrows represent the send-receive mechanism MPI provides.

Function-level programming, a process-oriented model, is an MPMD type of programming where elements are broken down into smaller units of parallelism. This contrasts the existing MPMD model where large programs communicate together to complete a task. Using these small components, the application is constructed from building blocks into a more complex system. Function-level programming leads to a large number of processes with
high communication demands. Often to deal with such a massive number of processes, programmers must constrain applications to simplify the binding specification.

Pushing MPI into cases where a large number of processes exist \[6, 45\] provides challenges \[71\] pertaining to the scalability of MPI emerge. Fine-grain MPI (FG-MPI) is an implementation which extends MPICH2 to utilize a fine-grain processing model by exposing additional concurrency through its smaller unit of parallelism. For example, MPICH2 uses heavy-weight OS-processes for each of its MPI processes. In FG-MPI, light-weight coroutines \[43, 79\] are used inside one or more OS-processes, making available a smaller unit of parallelism. This type of expression maps well to function-level programming techniques. Building blocks, or modules \[3\] of processes, can be mapped within a single heavy-weight OS-process benefiting from fast communication through collocation. Furthermore, by providing this extra level of process layering, massive numbers of processes may be launched on constrained hardware further contributing to the function-level programming paradigm.

With FG-MPI’s extra level of abstraction, the current round-robin binding algorithm becomes even more cumbersome. Representing groups of collocated processes is difficult and scales poorly in a text-based approach. One natural way to represent a large collection of processes is a graph, where the nodes represent processes and edges represent communication structures. MPI provides group communication capabilities that nicely map to graph structures. Furthermore, function-level programming is group based, where a collection of processes are combined together to function as a single task. A further benefit to using a graph allows for the use of groups to determine possible collocation. The optimization of these groups, together with the hardware specification, can provide an automatic mapping scheme. The size of these graphs may become a concern, however in systems such as Pregel \[46\], it is shown that in some large cases, graphs with billions of vertices and trillions of edges can still allow for efficient graph processing.

The use of a graph format provides an expandable and natural way \[86\] to determine information from a parallel application. Exploiting information
1.1 Motivation

in the graph simplifies programmability by auto-generating wiring code or other process dependant tasks.

Some specific terminology is used throughout this thesis. \textit{OS-processes} will refer to the heavy-weight operating system processes. The terms \textit{MPI process} and \textit{fine-grain process} are used interchangeability, and refer to the light-weight coroutines used within FG-MPI. A collection of MPI processes which use the same address space are referred to as \textit{collocated}. The term \textit{stacking} is used when a group of MPI processes are collocated and repeatedly added to an OS-process. \textit{MPI rank} and \textit{rank} are used to represent the communication identification number that represents a particular MPI process. The term \textit{binding} refers to the assignment of OS-processes or MPI processes to physical hardware, such as machines. The term \textit{mapping} refers to the assignment of code, such as functions, to MPI ranks. The term \textit{process-oriented model} is used when a model of a large number of small concurrent components is being discussed, while the term \textit{function-level programming} specifically relates to the programming methods of a process-oriented model.

1.1 Motivation

It is important for MPI implementations to support a large number of processes in a process-oriented model. Current implementations of MPI use a process binding method that is adequate only for a small number of MPI processes. In adopting a function-level programming paradigm, many small building block components will need to be allocated and managed. This creates a problem with the scalability and manageability of these MPI processes. Furthermore, with a reduced unit of parallelism, a large increase in communication can occur. Further problems with binding strategies will persist as the number of MPI processes increase.

The goal of this thesis is to solve these problems by providing a graph specification which is easily manageable, extensible and portable to many MPI implementations. Providing this graph specification to FG-MPI further extends MPI to a process-oriented model. Within this type of model, a small unit of parallelism is important and this unit of parallelism is evalu-
1.1. Motivation

Discussions on how the current MPICH2 implementations are unable to handle a large number of processes are mentioned below, as well as motivation related to FG-MPI’s extended layer of mapping.

1.1.1 An extended mapping layer

FG-MPI provides an extended layer of abstraction that collocates lightweight processes inside heavy-weight OS-processes: this is further discussed in Section 2.2. This method of collocating processes adds a new mapping layer where collocating bundles of processes together will be important for communication benefits. Currently these processes are mapped sequentially from the MPI rank of the OS-process. With this new layer, care must be taken with the number of MPI processes to launch as well as the location and choice of neighbouring OS-processes.

MPICH2 represents $N$ as the number of MPI processes that are bound to OS-processes. FG-MPI decouples this relationship of binding MPI processes to the hardware; this provides a solution to the function-level parallelism problem, where a large number of MPI processes are needed. By using lightweight coroutines, it further provides a small unit of parallelism. The problem of collocating so many processes still remains a challenge; by providing a flexible and scalable process binding specification, this would minimize the problem.

Decoupling this relationship to hardware and reducing the unit of parallelism allows FG-MPI to open many possibilities to different programming models. A flexible process binding specification that can mimic these unique programming models will closer represent FG-MPI’s design goals. The proposed specification is a graph approach. Because MPI already has a natural process-to-group system, and FG-MPI extends this, graphs provide an easy extension to these established internals. Using groups in the graph can manage a large number of processes. Furthermore many graph formats are highly extensible, where information about the application can be entered into the specification. From a general graph structure, communication patterns can be distinguished and provide further information to the process
1.1. Motivation

A graph specification can also represent many different processing elements. Pipelines are a good example where the output of one operation could go into the input of another. This would be represented as a chain of nodes connected by communication edges. More examples of graph designs are discussed in Chapter 3 and Section 4.2.

FG-MPI extends MPI with a new layer of mapping. This new layer requires a more flexible mapping and binding system to provide easier management of processes and optimized hardware binding.

1.1.2 Function-level parallelism

MPMD programming is a programming style where multiple programs all communicate and run concurrently to complete a common operation. Programs are separate and become unique entities within the application that only interact outside its components through message communication. This differs from data parallel applications that split the data up into small parts and executes the same application over these minimized units of data.

As MPMD applications become more complex with many entities communicating together, the process binding becomes more complicated. Different numbers of each program must be allocated, and often these applications must follow strict mapping requirements. For example, the number of processes to spawn for a certain entity may depend on another entity in the application. This again adds a layer of mapping to the process binding which may complicate the binding specification. Because the binding specification provided by the MPI implementation is limited, the overall mapping scheme becomes complicated. Furthermore, often some of the mapping information is directly related to application-level information, which currently must be known by the user to utilize.

By having an expandable graph mapping and binding specification allows for information about the application to be derived from the graph patterns and physically entered into the graph format. This will give a more flexible
binding process for future applications. Because a graph is used, a lot of information can be extracted. Communication patterns are already illustrated in such a structure. Determining the collocation of MPI processes can be determined by analyzing groups within the graph. The graph information provides a large potential for mapping. Finally, other information could be added to the graph as attributes that the user can further exploit.

Process-oriented models have been a popular parallel programming model in the past. Communicating Sequential Processes (CSP) [32], Actors [31] and Kahn Process Networks [38] are all early examples of the process-oriented approach. More recently, languages such as Scala [72], Go [25] and ERLANG [81] have attracted attention. The appeal of such a model is the ability to build up parallel programs as compositions of simpler processes.

Function-level programming is a form of a process-oriented model. In function-level programming, entities are again separate to perform tasks that only interact with each other through communication. This programming style goes one step further and breaks entities down into simple parts. These simple parts are used as building blocks, which are combined together to perform a more complicated behaviour. Languages such as Occam-π [54] and Haskell [12, 59] follow these principles. Since small entities are independent blocks, the number of processes can become very large and a strong need for FG-MPI’s light-weight processes arises.

The benefits of programming in this style are code simplicity and scalability. Programs can execute seamlessly on one machine or among many machines. The downsides are process organization and communication costs. By using small building blocks, users can often create modules and stack these modules together to create a larger program. Simple languages to teach children programming [66] use similar techniques. By having a module made up of several very simple processes, and having the ability to use this module to create larger modules, gives a structured and understandable code organization. By having processes be the simplest entities in a program, many corner or edge cases in applications do not exist. Furthermore, since each process is a simple entity, often a large number of these entities are active at any given time; a feature in this scenario is increased
scalability since adding more hardware simply distributes the entities over more resources.

In a parallel approach, these entities must communicate to participate with each other. When so many entities exist, communication costs can become costly. By having optimizations, such as in FG-MPI where MPI processes can be collocated within an OS-process, communication between collocated processes is made efficient. A more detailed discussion on FG-MPI is presented in Section 2.2. The challenge now becomes utilizing an effective mapping and binding system that can provide optimized communication to entities with large messaging demands. Using a graph binding specification is one solution for providing an effective and scalable mapping. For example, by using information that is presented in the graph structure, in the communication, and in the edges, optimizations can be determined and processes can be collocated together; this is further explained in Section 4.3.4. By making the specification expandable, the need for finer control over the application can be achieved by adding external information into the graph format.

This programming model can be seen as exposing parallelism to the program. Constructing programs in a building-block design allows the system to be flooded with many small parts. This provides a very scalable approach to parallel programming and also pushes programs to be more parallel efficient: parallel efficient programs are when the parallel time is in the order of the sequential time divided by the number or processors. This further provides beneficial criteria via Brent’s Lemma [8]. After exposing parallelism, the mapping of all the components becomes a challenge.

In summary, a process-oriented model in MPI promotes simpler programmability. A large and well-establish code-base from the MPI community is provided. Using a flexible graph mapping specification will help with the high communication and the large process manageability operations which are prevalent in the function-level programming design.
1.1. Motivation

1.1.3 A distributed algorithm approach

Using a ridged framework in solving parallel problems has been popular as seen with MapReduce [19]. This breaks the data into small chunks and solves the task by going through many stages. This technique is similar to using function-level programming in that both break the problem down into smaller chunks. In the design of parallel programs, the fundamental “scaling starts at 1” [23] rule is used. Breaking programs into a smaller unit of parallelism forces programmers to think about scalability from the beginning.

Using a process-oriented model, simplistic algorithms can be designed through messaging. A prime number solver shown in Code Listing A.2 in the appendix is such an application. Consider a Sieve of Eratosthenes [53], where the task is to find the first \( N \) consecutive prime numbers. Using function-level programming, one can spawn \( N + 1 \) MPI processes, where the first process sends numbers down a chain. An illustration of this is shown in Figure 1.2. An MPI process will accept the number as a prime if it has claimed no number yet. If it has a number, it will see if it divides evenly with its accepted prime number, if so the number is dropped, else the number is sent further down the chain. This continues until all MPI processes contain a prime number. Termination occurs when the last MPI process receives a number and sends the termination signal, which gets passed through all the MPI processes.

This simple program finds the first \( N \) consecutive prime numbers. The benefits of this type of programming are that each MPI process is small and that many of these small MPI processes exist. The scalability potential of such a program will be high minus the communication overhead. Simplicity is another benefit since all elements are small entities. In this example, the communication becomes the bottleneck. An algorithm designed through messaging needs fast communication to be effective. Understanding the communication of the application and having many small MPI processes will allow for a dynamic range of process bindings. For example, in the prime number implementation a chain skeleton is used. MPI processes near
1.1. Motivation

Figure 1.2: A Sieve of Eratosthenes prime number implementation. Each circle is an MPI process and has a possible prime associated with it. Consecutive numbers are passed down until they are dropped or declared as a prime number.

the end of the chain will not be utilized until after the chain reaches steady-state. MPI processes could be stacked, such as in Figure 4.3a, to allow neighbouring processes to be collocated thus optimizing communication and exploiting FG-MPI’s fast intra-process communication. Another possibility, as shown in Figure 4.3b, could be to stack round-robin which would more evenly spread computation intensive MPI processes around.

A distributed skip-list [44] has also been implemented. More discussion on distributed function-level algorithms can be found in Section 4.3.3. With this simple function-level program, complicated mapping and binding questions arise. Having an effective way to express the binding will make it simpler to program applications in this model. Furthermore by using information in the graph, an automatic binding strategy can be determined. The knowledge of how the different binding schemes affect the application can be used to improve performance for general applications sharing the same skeletal structure.

**Summary** It has been argued that parallel applications should follow an ideal model which should be easy to program, should have a software development methodology, should be architecture-independent, should be easy
to understand, should guarantee performance and should provide accurate information about the cost of programs [74]. Many of these features are improved by extending MPI to a simple process-oriented model and using tools to improve the manageability and understandability of code.

The contributions of this thesis include extending MPI to a graphical binding and mapping specification to support a process-oriented model. This new binding technique is flexible, manageable, scalable, and allows the user to add additional information about the program into the mapping specification. This also addresses the demands of the MPI process mapping required by FG-MPI which more closely resembles the programming paradigm that is promoted. It also allows the managing of a large number of MPI processes. Further contributions include exploring advanced mapping techniques, evaluating the smallest representable unit of parallelism and providing an extensible graphical format which allows the programmer to simplify code by representing information within the graphical format. The tools developed in this thesis are portable to other MPI implementations.

1.2 Organization

This thesis explores the representation, implementation, and evaluation of a flexible graph binding specification for MPI. The chapter organization is as follows:

Chapter 2 presents tools used in this thesis. It describes the MPI framework and the basic communication processes that are implemented in MPI. MPICH2’s [5] mapping techniques are discussed. FG-MPI [22] [40] is described along with the differences it provides and how these differences allow further extensibility. Pilot [11] is described as a CSP-like library for MPI. How this extends MPI is explored and basic usage is illustrated. Section 2.4 discusses related work.

Chapter 3 talks about the representation of the graphical specification used. It discusses Kahn Process Networks (KPNs), which the model specification is based on. It constructs the model from the basic nodes up to hierarchal modules and bundles.
1.2. Organization

Chapter 4 goes into the implementation of the tools constructed in this thesis. Section 4.1 begins with a discussion on Pilot. The chapter goes into the experiences and problems encountered with using this library and motivation for why Pilot is necessary. The set of tasks in creating a Pilot farm is given including basic design and implementation. A discussion is presented on the automation of tasks which simplify programmability and other difficulties where a graphical mapping approach would provide a solution.

Section 4.2 proposes a possible graphical representation that would meet the demands for managing a large number of processes and allow extensibility to store information about the code. The design and implementation of the graph_map Python tool is discussed with a brief discussion about its portability. Usage of the tool is provided, as well as visual constructions of the graphs. Section 4.3 continues with mapping techniques for use with the graph representation. More advanced graphical techniques are described including possible optimization benefits. Formalization of the model is shown as well as the function-level architecture.

Chapter 5 evaluates the unit of parallelism, or the smallest computational unit that can be expressed in MPI and FG-MPI. The evaluation shows small communication overheads when comparing hundreds of large tasks to hundreds of thousands of fine-grain tasks. Exploiting FG-MPI’s efficient intra-process communication shows that the increase in communication within an application does not affect performance. Furthermore, by decoupling MPI processes from hardware, a situation where collocated MPI processes become tightly packed is very likely. This implies that it is feasible to achieve a small granularity with acceptable overheads, an ideal property in a process-oriented model.

Chapter 6 concludes with the benefits of extending MPI to allow for a graphical mapping specification. Section 6.1 provides possible extensions to the graph_map tool as well as other future work to the graphical specification.
Chapter 2

Background and Related Work

The three main tools used in this thesis are MPI, FG-MPI, and Pilot. For MPI the current process-core binding process which MPICH2 uses, as well as generic MPI notations are discussed. FG-MPI's differences to this binding process are explored. Finally a basic description of Pilot and the benefits it provides are presented.

2.1 MPICH2

MPI is a language-independent messaging library specification that provides basic communication primitives. It was created by a committee consisting of university, government and industry members. The MPI Forum continues to evolve MPI and there are plenty of implementations that use this standard. MPICH1 [4] is an open-source implementation that conforms to the MPI-1 standard [50]; it currently supports heterogeneous environments. MPICH2 [5], developed at the Argonne National Laboratory, is the successor which conforms to the newer MPI-2 standard [24]. MVAPICH [51] provides support for InfiniBand networks. Open MPI [77] is another popular MPI-2 open-source implementation. The MPI-3 standard [49] is currently in the proposal stage. Popular closed-source implementations include Intel’s MPI Library [37] and Platform MPI [62]. Furthermore, there are implementations which are optimized for certain architectures, such as the MPI on BlueGene/L [2]. From the many flavours of MPI, each follow an MPI Forum standard and thus MPI applications are highly portable. The portability MPI provides has been one of the features that has made it so popular:
when designing a new binding specification it is important to incorporate this feature into the extension.

Basically MPI is commonly used in distributed shared nothing architectures. MPI is executed on a cluster of machines using a send-receive mechanism to communicate between processes. Programmers often use different mechanisms such as OpenMP [17, 55] when the goal is within multi-core shared memory architectures. Recently, MPI has started to push into this space as well and it has been shown that MPICH2 can be as efficient [10, 20] in some cases within a shared memory machine as its competitors.

The tools developed throughout this thesis will depend on the standard and not on a specific implementation of MPI. MPI is language-independent with application programming interfaces (APIs) for many languages such as C, C++, Fortran, and Python. The code presented in this thesis will consist mainly of Python [63] and MPICH2 [5] in C.

### 2.1.1 Communication

There are two main types of communication in MPI: point-to-point communication and group communication. For all communication, processes are split into ranks. Every MPI process will have a unique rank within a communication group in which it can communicate.

```c
1 int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
2 int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status * status)
3 int MPI_Ssend(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
```

Listing 2.1: MPI point-to-point communication in C.

The basic point-to-point send-receive APIs are presented in Listing 2.1. These will take a message and a destination rank, and then transport the message over some communication fabric. The MPI middleware hides the underlying communication mechanisms, so the send-receive can seamlessly
be abstracted over any number of machines. Cases where the message is sufficiently small, the message may be put into a buffer and execution will continue without the send fully executing. This is called an *eager send*. The **MPI_Ssend** is a synchronous send and will confirm that the sending message has been emptied from the buffer so that the buffer may be reused before returning.

Group communication modifiers, such as those shown in Listing 2.2 provide more complicated communication structures. For example, a **Bcast** provides the ability to send a message from one member of the group to the rest of the group. Collective groups can be created, modified, and freed during execution. This form of communication is usually optimized and provides an effective way to communicate to a large number of processes.

```c
1 int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
               int root, MPI_Comm comm)
2 int MPI_Comm_create(MPI_Comm comm, MPI_Group group, MPI_Comm *
                      newcomm)
3 int MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm *
                    newcomm)
```

Listing 2.2: MPI group modification communicators in C.

Using point-to-point and group communication can complicate communication patterns. Since these communicating processes can be spread over many machines, efficient process-core binding is necessary to overlap communication and computation among processes.

### 2.1.2 Process-core binding

The binding of processes to cores can become complicated depending on the applications communication complexity and the number of cores. The default process manager used in MPICH2 is Hydra [35]. Hydra’s job is to start the application by: obtaining the command-line parameters, determining the number of OS-processes to launch, and assigning the OS-processes to specific machines.
2.1. MPICH2

Hydra currently has a basic round-robin allocation strategy [35]. This strategy will take a list of hosts, either through a `hostfile` or from the command-line, and allocates the MPI processes one at a time to the hosts specified; it will continue this and wrap the hosts until all the necessary MPI processes have been mapped. This method is further discussed in Section [3.1].

```
mpiexec -f hostfile -n 8 ./my_mpi_app
```

Listing 2.3: Execution of an MPICH2 application.

This simple strategy is powerful enough to handle every allocation pattern since it can always default to the one-to-one mapping of processes to cores, which can provide a full user-defined mapping. Because MPI deals with heavy-weight OS-processes and mostly SPMD applications, programs tend to have moderately simple allocation demands and the construction of this `hostfile` is often just using all the physical hardware available.

Hydra also provides two topology-aware allocation strategies for the binding of heavy-weight processes; these are CPU-based or cache-based options. Hydra uses information about the system via PLPA [61], or `hwloc` [34] to determine the optimal placement of processes that minimize cache or CPU sharing. These tools work by using operating system support to determine the information.

These allocation strategies tend to be sufficient when dealing with non-complex communication patterns, or a trivial number of MPI processes. Neither of these techniques use specific input provided about the application, nor an ability to cluster large groups of heavily communicating MPI processes together: both characteristics of a function-level program. For these types of applications a more flexible and powerful mapping specification is necessary, which is also portable to other MPI implementations.
2.2 Fine-grain MPI

Fine-grain MPI (FG-MPI) [22, 40] is based on MPICH2 and developed at the University of British Columbia (UBC). A goal of FG-MPI is to decouple MPI processes from OS-processes.

MPI implementations adopt an SPMD programming style, by using a parameter $N$ that indicates the number of MPI ranks. This often maps directly to the number of OS-processes. Because of the strong correlation $N$ has to the hardware, programmers tend to write applications around this number to represent the amount of parallelism in the code. Problems with scalability and programmability can occur with this approach. FG-MPI decouples this relationship by launching many MPI processes in one or more OS-processes. This allows users to relate $N$ to the algorithm used inside the code instead of the hardware outside the application.

![FG-MPI Architecture](image)

Figure 2.1: The FG-MPI architecture. The shaded parts are the MPICH2 pieces that FG-MPI touches.

Decoupling of the MPI processes from the hardware allows FG-MPI to launch massive numbers of light-weight MPI processes within a limited memory environment. FG-MPI uses coroutines [43, 79] to accomplish this, which reduces the unit of parallelism that MPI can express. This ability to ex-
press a small unit of parallelism is a necessary component for function-level programming.

FG-MPI introduces a new parameter, \(-\text{nfg}\), to the command-line arguments. This parameter represents the number of fine-grain processes to launch per OS-process. For example, the command in Listing 2.4 will start up 16 MPI processes total: 8 OS-processes with 2 MPI processes within each. More details about the FG-MPI mapping specification are described in Section 4.3.1.

```
mpiexec -f hostfile -n 8 -nfg 2 ./my_fgmpi_app
```

Listing 2.4: Execution of an FG-MPI application.

The mapping specification of OS-processes to machines in FG-MPI still mimics that of MPICH2, however a new layer of light-weight MPI processes now exists. As with MPICH2, all mapping positions are still possible, since a user can always provide a one-to-one mapping allocation. However, each process is no longer equal, as they can have a different number of fine-grain processes; thus the allocation can mimic an MPMD program and provide a more complicated mapping schematic. Introducing a new graphical specification for mapping and binding allows for external information about the application to determine a possible mapping pattern: a feature the binding system of MPICH2 does not allow.

2.3 Pilot

Pilot [11] is a CSP-like library for MPI, developed at the University of Guelph. The goal of this project is to make HPC programming easy and error free by adopting CSP’s formal language principles while hiding many communication mechanisms under the implementation. CSP describes patterns of communication between interacting agents. Others [27] have also notice that collective communication should replace potentially harmful low-level primitives such as send-receive, further supporting a channel-like communication structure. Pilot uses these formal language principles to provide
2.3. Pilot

Figure 2.2: The Pilot execution flow. Pilot will run the configuration code and then start a `PI_StartAll`. This will spawn several process functions and then tell `PI_MAIN` to continue executing.

a communication pattern between MPI processes, while hiding the MPI API.

The benefit of providing this formal model to MPI is that many complicated parallel errors, such as dead-locks, are avoided by promoting a programming style that intrinsically avoids such problems. Communication patterns are explicitly defined at the beginning of an application. However, this can create new difficulties with the task of creating a CSP-like schematic diagram within the application.

When introduced to function-level programming, communication and common parallel errors can become an increasingly difficult task to tackle, due to the large amount of communication between small entities. At the expense of producing a difficult schematic diagram, Pilot provides many of the benefits of a CSP library. Since many small computational entities do exist in this type of model, an accurate communication diagram may be just as difficult to create. Furthermore, information that is needed by this schematic diagram can also be shared to other components, such as
providing an efficient mapping scheme.

**Summary** Combining the three tools presented in this chapter, MPI, FG-MPI, and Pilot allows for a solid foundation to extend a process-oriented model to MPI. MPI provides a well-established messaging library. FG-MPI adds the ability to provide a small unit of parallelism within a constrained environment. Pilot provides a CSP-like programming model for using a large number of communicating processes.

With the ability to create function-level applications, the problem of dealing with massive numbers of processes still remains. In the current setup, the task of organizing a large number of these MPI processes is cumbersome, both with the communication schematics and the mapping or binding scheme. A simple way to organize and manage a large number of processes would be beneficial, as well as the ability to exploit information to optimize and simplify code generation.

### 2.4 Related work

There is past work on thread-based implementations of MPI that share similar goals to FG-MPI. One such project is Adaptive MPI (AMPI) [33]. However, its implementation is different to FG-MPI in that it implements MPI over Charm++ [39], requiring a custom run-time system to be used. In AMPI, a virtual MPI process is represented as a collection of objects, which pass messages to other MPI processes. FG-MPI instead extends the process-based model of MPICH2. Phoenix [57] is another run-time environment designed for multi-processors that provides a light-weight process abstraction built on top of threads. Again they have chosen to develop a custom run-time system.

Run-times have been developed for light-weight communicating processes in a process-oriented model [67]. This shows that multi-core schedulers can be efficient with many small isolated components. In this example, they develop their own run-time. Building on top of MPI provides a seamless distribution over a shared memory or distributed memory architectures.
2.4. Related work

There was a proposal \cite{75,76} to the MPI-3 standard that also tried to extend MPI by allowing for multiple endpoints to support threads inside a single OS-process. This work only reached the proposal stage and differs from FG-MPI in that the focus was on the existing MPI model.

Process-oriented models have been used in large concurrency systems \cite{70} with pattern languages. MPI has not supported the unit of parallelism necessary to extend to this architecture. Use of parallel models is well known outside MPI such as Tarragon \cite{14,15}, a library which uses an actor-based programming model. Its goals involve implementing parallel applications requiring fine-grain asynchronous communication; Tarragon focuses on scientific applications using the actor model.

The idea to push some of the programming and wiring into a graph format for programmability is not new. Several visual programming techniques exist in this space for many types of applications. Orange \cite{56} is an open-source data visualization tool that allows for data mining through visual programming or Python scripting. This tool’s primary focus is on manipulating and analyzing data. Red-R \cite{64} is such an application that provides a visual programming interface specifically for the statistical software R. The cumbersome approach of creating wiring diagrams is tackled with Wire-It \cite{85}, an open-source JavaScript library to create web wireable interfaces for data-flow applications, visual programming languages, graphical modelling or graph editors. This application provides a nice interface to create complicated wiring, but does not allow for easy extensibility to do other tasks on the graph or output specific code, such as interfacing with Pilot.

As more multi-core machines become available, binding parallel applications to these cores becomes a problem. StreamIt \cite{26,78} is an MIT programming language that defines a new language to make this binding easier. It defines three levels of parallelism, data, task, and pipeline, and uses these to express in the language how the computational units should be bound to hardware. The tools presented in this thesis attempt to solve this using an existing MPI framework and express parallelization via a communication graph. Modules could be added to mimic similar behaviour to
2.4. Related work

that of StreamIt. Furthermore, StreamIt uses its own compiler. In the MPI
space, the use of automating this binding process is also present \cite{42, 80} but
relies on specific small scale setups, and do not allow for a process-oriented
model to be expressed.

Kahn Process Networks (KPNs) \cite{38} are a popular process-oriented mo-
del for concurrent programming and are further discussed in Chapter 3.
Process-oriented approaches to the new multi-core era are common such
as S-Net \cite{29, 30, 58}, FastFlow \cite{1} and YAPI \cite{18}. For example, S-Net is
a coordination language used to identify components in a multi-core setup.
The specific goal is to turn legacy code into asynchronous components which
are run on a stream-processing network. The approach presented in this
thesis extends MPI to use a function-level programming interface and uses
tools to flexibly bind processes to hardware outside the application. Using
MPI provides the ability to use distributed and multi-core architectures.
Most of the tools mentioned focus on shared memory architectures, with the
exception of Distributed S-Net \cite{28} which has only recently began exploring
a many machine framework.
Chapter 3

Representation

Extending the MPI binding interface to include a graphical specification provides many challenges. Flexibility and scalability are both important factors when expressing a process-oriented model. Care must be taken to allow for a representation that is flexible and general enough to handle many possible situations. The creation and naming of these components are also important \[68, 69\]. Scalability concerns in using a graphical interface can also affect the flexibility of the system.

Kahn Process Networks (KPNs) \[38\] are a popular process-oriented model for concurrent programming. As mentioned in Section 1.1.2, many current languages exist for this type of model. More specifically, KPNs are used often because of the simplistic model expressed for parallel computation. Nornir \[82, 83\] is such a run-time system that executes KPNs because they are simplistic, intuitive, graphically representable and deterministic. These features make KPNs very desirable in large parallel programming environments.

KPNs are groups of deterministic sequential processes which communicate through channels. The graphical representation involves easily formulated nodes and edges. Because of this, KPNs are ideal to model the graphical representation in an MPI graph binding specification. Also, KPNs are a model of concurrency that relies on message passing, extending directly to MPI.

All these features make KPNs an ideal fit to adopt into the MPI framework. KPNs provide large parallel strategies with deterministic traits. MPI provides a large parallel deployment mechanism and an active user base. FG-MPI extends the MPICH2 framework to a large number of MPI processes. Managing a large number of processes effectively becomes difficult.
3.1 Nodes

The ease of representing KPNs assist in the management process.

Currently MPICH2 provides a basic hostfile as an interface to the user. This simple hostfile has a collection of machine names which are given sequentially. Subsequently, MPI ranks are mapped sequentially down this file and are bound in this order to the hardware. If more MPI ranks need to be mapped than machine names given, the file is then wrapped round-robin and repeated until all MPI ranks have been assigned. An example hostfile is shown in Code Listing 3.1.

```
1 # This is a sample hostfile
2 node04  # The first 2 procs are launched on node04
3 node04
4 node05  # The next 3 procs are launched on node05
5 node05
6 node05
7 node06  # The last proc is launched on node06
```

Code Listing 3.1: An example hostfile. This example will launch OS-processes in the provided order and wrap this list until all MPI ranks have been assigned.

This simple process works well for MPICH2. It is easy to use and flexible for any situation of bindings the user may wish to do. The problem with such a format is scalability and the lack of specification for communication. With FG-MPI, the number of MPI processes can become large, and as this number becomes large, the flexibility this system provides fades. The manageability of so many hosts can become cumbersome and often programmability is compromised to simplify this task.

Pilot’s mapping of functions to MPI ranks also uses this simplistic method. Currently Pilot will directly map the processes it creates, on a first-come first-serve basis, to the MPI ranks sequentially. Because Pilot is built on top of MPI, the large scaling issues are equally affected by Pilot.
3.2 Nodes and an edge

FG-MPI further complicates this by collocating MPI processes in OS-processes. The mechanism provided to accomplish this is done through the `mpiexec` command. This collocation of processes can only sequentially map MPI ranks within an OS-process, so no further representation is needed except for MPICH2’s mapping of MPI ranks to OS-processes.

This list of machine names can be thought of as a collection of single unattached nodes as shown in Figure 3.1. Because of scalability concerns, a move from this hostfile approach is necessary.

### 3.2 Nodes and an edge

Pilot can be thought of as an unbounded KPN with non-determinism, similar to the envisioned model that is being constructed. Following Pilot, another step is to express communication patterns in a CSP-like pattern. This can be thought of as mapping communication to MPI processes. Channels are the resulting structure, which represent a single exchange of messages between two MPI processes.

This is still not a graphical format, instead Pilot represents this channel wiring diagram as pairs of nodes as shown in Figure 3.2. As stated, because Pilot is bounded by the number of processes MPI can provide, thus these wiring diagrams are often limited in size. Nonetheless, the addition of channels is still a powerful expression and gives the user a simple communication
3.3 Nodes, edges, and hierarchy

By extending this model to a graph structure, nodes can be represented as MPI processes with multiple channels. This becomes a two-dimensional structure that extends KPNs with non-determinism. The non-determinism comes from Pilot’s bundle abstraction.

Pilot has an added representation of bundles. These bundles are a collection of channels with a shared endpoint. In the representation formalized in Section 4.2, bundles can be expressed as special nodes that are between collections of MPI processes. This can be thought of as a group of processes communicating to a single process.

To be able to easily manage the scalability concerns introduced by FG-MPI, a process group must be formulated: the final addition to this model is a hierarchal structure. A module is a collection of nodes bundled together as shown in Figure 3.3. Allowing a module to exist helps with the scalabil-
3.3. Nodes, edges, and hierarchy

Figure 3.3: The node, channel, and hierarchy case. The solid rectangle is the physical representation. This represents a graph and can be illustrated in the same way as the representation. The hierarchical representation shown illustrates groups.

This model can be thought of as a KPN with non-deterministic bundles. The bundles can be either a collection of communication, such as a Pilot bundle, or a collection of nodes, such as a module. Intuitive and simplistic are the core design concepts of this model. Also like KPNs, message passing is relied on exclusively for the channel structures. Further mapping formalizations based on these three representations are defined in Section 4.3.2.
Chapter 4

Implementation

4.1 A Pilot example

Pilot [1] is an MPI library that provides a CSP-like process and channel model for parallel computation. This provides a function-level style of programming where multiple PI-processes communicate amongst themselves using a simplified messaging API over predefined communication channels. In Pilot, these processes are defined through a communication diagram. This communication diagram represents channels where two processes will communicate together. Communication can only occur through these channels and is implemented as a layer on top of MPI.

Because Pilot is built on top of MPI, Pilot processes are directly mapped to MPI processes and therefore are coarse-grain components. Providing a fine-grain processing model to Pilot allows for better expressiveness of the CSP fundamentals that Pilot emulates. A potential disadvantage of this type of model is the cost of all the expensive communication.

The difficulties of porting the Pilot library to FG-MPI are presented. Pilot is then used to construct a processor farm consisting of a set of modules. The farm is an illustration of a function-level type of programming construction.

Evaluation is shown using the Pilot farm to measure the task size and determine the minimum unit of parallelism in Chapter 5. Furthermore, the farm is used to measure small-sized messaging costs to determine the overhead for collocated MPI process messaging. Other overheads, such as the cost of inter-process communication, are further explored.
4.1. A Pilot example

4.1.1 FG-MPI porting

Pilot was ported to FG-MPI to provide a fine-grain processing model. This allowed Pilot to run a much larger number of processes compared to standard MPI implementations. FG-MPI applications are backwards compatible to existing MPI code; this means that most FG-MPI applications can execute without the FG-MPI flag. Furthermore, true MPI code can also run within FG-MPI without using the FG-MPI flag. The main steps to porting applications and libraries to FG-MPI are:

1. add in the FG-MPI broiler-plate function and headers,
2. create a mapping function for the collocated MPI processes,
3. remove non-static global variables,
4. fix up application level quirks such as `sleep` and `yield`.

Pilot attempts to hide the MPI syntax from the user and executes all the MPI calls from within the library. This is done so that Pilot can provide a simpler messaging interface. Since all the MPI calls were contained within the library, the necessary FG-MPI broiler plate code could be easily inserted; the files that require specific FG-MPI code could be edited without the user’s knowledge. With the broiler plate code in place, the specific FG-MPI function calls could be put into the library.

The most difficult part of the porting was with the global variables. FG-MPI uses coroutines [43, 79] that have private `stacks` but share a `heap`. Since MPI applications are often scattered over many machines which do not share memory, global variables can cause confusion and are often avoided. Within MPI, static global and dynamic global variables can still be used, but the term `global` changes. In FG-MPI, static global variables which do not change are possible since within a single OS-process the variable is only ever read. In MPI, global variables are sometimes used to share information over functions within the same OS-process; however the global variable can have different values over other OS-processes in the application. This is shown in Figure 4.1. The Pilot library stores its Pilot process structure, which contains specific Pilot information, in a global variable. Because of
4.1. A Pilot example

Figure 4.1: The shared global variable issue. Dotted circles are OS-processes, solid circles are MPI processes, and squares are shared global variables. (a) is the MPICH2 case, where each MPI process has its own global writable variable, while (b) is the FG-MPI collocated case, where several MPI processes share the same global variable; care must be made when writing to these variables.

The shared state between coroutines, it was necessary to change the dynamic global variables within the code.

There are many ways to remove global variables; some methods involve attaching the global structure to the MPI communicator, which is often passed around the MPI application anyway: however Pilot does not do this. Tools could be developed to automatically attempt to remove global variables, such as Photon [52] for Fortran code. Furthermore, the code could be rewritten to avoid such structures. The approach taken in this instance was to make the structure private and pass the structure through each function call, where functions would read the information from the parameter. Every Pilot function call must be changed to accept the new Pilot global parameter and edited to read from this structure. Furthermore, this also causes users to change the Pilot application by initializing a Pilot structure within the application code, breaking backwards compatibility to existing Pilot programs. A more elegant approach to maintain compatibility is left for future work.
4.1. A Pilot example

With each Pilot function and macro altered, the next process was to handle special functions such as `sleep` and `yield` correctly. `Sleep` is a problem; for example since `sleep` acts on the OS-process, and with FG-MPI many MPI processes may be in one OS-process, allowing each process to call `sleep` does not achieve the desired effect. The `yield` is further detailed in Section 4.1.2.

When collocating many MPI processes within OS-processes, memory may become a concern. Pushing static shared variables into the heap would allow Pilot processes to reduce the memory footprint by making coroutines share these variables and thus increasing the number of MPI processes that could be collocated. Three large structures exist within each of the Pilot processes: a process structure, a channel structure, and a bundle structure. Each of these structures contains all the respective elements. For example, in the MPICH2 case launching $N$ OS-processes would result in each of the Pilot process structures, which every OS-process stores, to have a space complexity of $O(N)$; there are $N$ of these OS-processes so the overall space complexity would be $O(N^2)$.

This becomes worse in the FG-MPI case because of the collocation. Assume a uniform setup where every OS-process $n$, has the same number of collocated MPI processes $c$, then the total number of MPI processes would be $N = nc$. Calculating the space complexity would be

$$S_{\text{perproc}}(N) = O(cN) \quad \text{where} \quad c = \frac{N}{n}$$

$$= O\left(\frac{N^2}{n}\right)$$

$$S_{\text{overall}}(N) = O(cnN) \quad \text{where} \quad n = \frac{N}{c}$$

$$= O(N^2)$$

where the number of OS-processes $n$, is limited because of the hardware. To scale to a large number of MPI processes, $n$ will be much smaller than $N$, and for each OS-process the space complexity would be $O\left(\frac{N^2}{n}\right)$: this is not a good situation. Memory optimizations have been implemented into Pilot, namely...
4.1. A Pilot example

with the Pilot process structure, bringing the space complexity back to $O(N)$ for each collocated OS-process. By taking the other global structures out of the Pilot processes and having them as shared global resources over the OS-process, the memory footprint would be further reduced. Optimizing the memory in the Pilot library further is left for future work.

4.1.2 Pilot farm construction

Pilot was used to construct a processor farm consisting of a set of modules which are made up of several smaller processes. The farm is an illustration of a function-level style of programming. More efficient designs of a farm are possible; however to express a clearer evaluation, a structure with high communication was chosen.

Design

The process structure of the Pilot farm is shown in Figure 4.2. The farm is demand-driven with tasks flowing down a chain and results flowing out. Each module is built from several specialized processes. The smaller processes are divided into Worker and non-worker processes. The main purpose of the non-worker processes is to buffer tasks across non-local communication links overlapping computation with communication. A single module farm will contain five MPI processes where ideally each module, or Worker process, is mapped to one OS-process or core.

The Worker process does the majority of the computation. It requests work from the inTask, and sends finished work to the outResult. As work gets passed down the chain, the inTask will receive new work and pass it to its Worker if it is free, else work is given to the outTask to be passed down the chain. The inTask will buffer the work if both the Worker and the outTask currently have work.

The results are handled in a similar fashion; the Worker passes the finished result to the outResult. This will send the finished result back up the chain to the next inResult and the inResult will continue to pass the work to its outResult, which continues until it reaches the top of the chain.
Figure 4.2: A simple processor task farm module showing five collocated processes. Tasks flow through the module where the Worker process will execute the task and the remaining four MPI processes buffer tasks.

Again these processes will buffer the results if the next module already has a result.

A Manager module is also needed to initially pump tasks into the farm and drain off the finished results. The farm module provides simple and extensible building blocks for a processor farm. These modules can be launched along a chain. Modules can be stacked to combine communication or computation. Each module is designed from five simple building blocks, which can further be expanded into more complicated structures. For example, by adding a reduction operation to the outResult process, it is possible to do map-reduce operations; also by adding channels to outTask and inResult results in a tree structure instead of a chain. After the initial generic pieces are in place, these small parts can be reused in other applications. Modules can be assembled and passed as objects to other processes. This architecture is shared nothing and all communication is done using messages, so
scalability is seamless over a single node or multiple machines.

4.1. A Pilot example

Mapping and aggregation

Using a process-oriented model increases the number of MPI processes. Each module contains five MPI processes: a chain of 16 modules on an eight core machine, could result in over 80 processes being launched. Even worst, spreading these 80 processes over multiple machines could have a large impact on communication. Section 5.1.3 demonstrates that using FG-MPI to collocate intra-module processes reduces communication costs and provides effective scalability.

Each set of five processes can be considered part of a single component and it is expected that one or more components are mapped to an OS-processes. This allows for aggregation to occur outside the program. By providing a mechanism to stack multiple modules within an OS-process, the number of modules stacked can be directly related to the amount of work each task possesses. Few modules would be stacked for coarse-grain tasks and many modules could be stacked for fine-grain tasks. All this can be done on the command line as shown in Section 2.1.2.

Other mapping techniques are also possible, such as those shown in Figure 4.3. A clustering of the chain with mappings of neighbouring modules as shown in Figure 4.3a might be advantageous to optimize the communication. However, because a chain is used, the start of the chain receives work earlier and would not distributed computation evenly. On the other hand, mapping neighbouring modules to different machines, as shown in Figure 4.3b, would distribute work quickly, but provide expensive communication. This provides an aggregation of tasks available outside the program and a further control for load balancing.

These novel techniques of modular programming with Pilot allows the use of simpler programming focused on small components, overlapping communication and computation, as well as providing external aggregation outside of the application. Each MPI process is easy to specify in terms of its behaviour. The structure of the farm is simple to compose and illustrates
4.1. A Pilot example

Figure 4.3: Eight Pilot farm modules. The large encompassing dotted rectangles represent machines. (a) is stacked to optimize communication; in this case neighbouring modules are collocated together. (b) is distributed round-robin to optimize computation; in this case neighbouring modules are separated to allow for quick distribution of tasks over resources.

how small components can, with the inclusion of an arbitrary procedure, create complex behaviour.

Pilot implementation

Implementation of the farm was done with the Pilot library. The first step was to construct the wiring diagram needed to represent the communication channels. A form of this diagram is shown in Figure 4.4, with corresponding wiring code in Code Listing A.1 in the appendix. Since the Pilot farm is constructed in such a way to dynamically connect many modules together, a flexible way to wire these modules must be constructed. Parameter struc-
tutes were created so that many channels could be allocated and connected depending on the number of modules used.

The physical wiring for the communication is done through the `PI_CreateChannel` command. These are one-way communication links between different MPI processes. Translating the edges in Figure 4.4 to channels for each module will result in the final wiring diagram. Special attention is needed to handle the first and last module of the chain: the first must be connected to the `Manager` module, while the last must represent a termination signal.

Each MPI process within the module is a function. Most of the non-worker processes read and send tasks to new MPI processes when the channel is free, or it buffers the work if everyone is busy. All the non-worker processes are similar; the code consists mainly of Pilot setup code and a few lines of messaging. A sample Pilot process of a `primesiv` is shown in Code Listing A.2 in the appendix.

The `Worker` contains the function which does the computation. For the purpose of the evaluation, the farm was used to calculate a static portion of the Mandelbrot set [47]. This allows all the work to be equal for each `Worker`. An interesting complication arose due to the non-preemptive threading model of FG-MPI. If the `Worker` process does not `yield` sufficiently often, the process will interfere with the ability to buffer processes and keep the tasks flowing through the chain. Adding an `MPI_Yield` command sufficiently often was enough to avoid this problem. This problem occurs whenever message-driven processes are combined with compute-intense processes. This is not a new problem in MPI since compute processes can cause other processes to idle while they wait for their turn to execute. To operate correctly it was also important to use `synchronous sends` that block and not `asynchronous eager sends` that would flush too many tasks down the chain.

Timing was done within the `Manager` module to measure when the chain started to be populated with work, when it stopped being populated with work, and when the chain was finally emptied. These respectively represent the start-up, steady-state, and wind-down phases of a standard processor
Figure 4.4: A GraphViz graphical representation of a Pilot farm. Circle nodes represent MPI processes, edges represent channels, and rectangles represent modules.

farm [84]. Overall start and stop times of the Pilot program were also measured. These times are discussed in Chapter 5.

Summary This section showed that the design and implementation of certain applications can effectively exploit smaller units of parallelism. As the number of MPI processes within an application increases, a need for a more simple representation is necessary. Representing the modular components within a graph format allows for several automated advantages, such as automatic Pilot wiring code generation, optimizations, and more advanced mapping and binding techniques.

A modular approach using Python is adopted in Section 4.2, which can read in a general graph format, generate schematic code for Pilot, use optional optimization modules to provide the mapping of MPI processes, and record extra information for adaptable module tasks.
4.2 Graph representation of MPI processes

Creating wiring diagrams can be difficult and error-prone. To reduce the complexity of wiring diagrams, such as those common in Occam-π \cite{54} and Pilot, a tool was developed for transferring a graphical representation of a wiring diagram to Pilot source. This tool takes as input a graphical representation in the form of a GraphViz \cite{21} file, and through a number of customizable modules can perform a large range of tasks such as Pilot wiring source generation, automation of mapping information and optimization of MPI process collocation.

The goal of this tool is to be portable and extensible for multiple tasks. To achieve these goals the tool was written in Python and organized via a series of modules. Users can create their own modules that can hook into the system to produce a vast range of operations. The tool can be seen as a graphical representation of a `hostfile`; a `hostfile` is a large text file that maps OS-processes to cores. More information on mapping can be found in Section 4.3.

Using a graphical mapping specification allows further information to be extracted and shared from this graph. For example, the Pilot wiring diagram can use information about the communication. Because of the modular approach used by the tool multiple graph formats are possible. For example, part of a GraphViz \cite{21} graph is shown in Code Listing A.3 in the appendix. Other formats, such as Kroket \cite{7}, are supported and future formats can be added by creating a parser module. For most of the work presented the GraphViz format is used. Currently these graph files can be created by hand, but because the format is open, graphical tools could generate these files.

4.2.1 Visual representation

A GraphViz graph file is used to represent a graph of MPI processes and communication channels. This can be viewed as a visual programming environment by adding extra attributes to the graph. A sample GraphViz file is shown in Code Listing A.3 in the appendix. A visual representation is
shown in Figure 4.4.

The file contains nodes and edges, each with attributes. Each node represents an MPI process. Special nodes such as NULL are distinguished by specific attributes. For example, the NULL node is a node with the special name null and is not interpreted as a process. bundles are a special Pilot operator which is a collection of channels with a shared endpoint. These bundles are special nodes which sit between many MPI processes. Currently they are represented using channels, and converted to the channel equivalent of the structure.

Edges contain attributes which represent the communication input and output. These edges are defaulted to in and out channel names unless specified by the PIin or PIout attribute.

A collection of edges and nodes are clustered together with a GraphViz subgraph element. This provides a representation of a module which can contain processes and communication elements. Clusters are given the required attribute name PName which must match the module the cluster represents.

### 4.2.2 Source-to-source translation

The tool first parses the arguments and determines the user’s request. The general architecture is listed in Figure 4.5. To generate Pilot code it will parse the GraphViz graph file into several specific structures. This is a custom format containing a list of attributes for each element. The information parsed is the moduledict, modulenumdict, nodenamelist, nnidx, and edgenamedict. These structures are simple adjacency lists representing certain components of the graph. This is done to maintain flexibility with future graph formats where adding a new format will only contain the transformation code to the custom structures. The important information in these structures is the edge and node trees that contain the names and attributes of each element.

Next a cmdlist is produced. A cmdlist is a long Python list of transformations which are necessary to construct the resulting Pilot wiring code.
4.2. Graph representation of MPI processes

Figure 4.5: The architecture of the graph_map Python tool. This represents an approach where modules act on the graph information independently. The cmdlist, which contains skeletal information, can be manipulated and shared by modules. In this approach users may add custom modules.

A sample cmdlist is shown in Code Listing A.4 in the appendix.

Depending on the application, the cmdlist may represent many different source-to-source translations. In the Pilot module case, this cmdlist is a list of macros which are replaced by information that is generated from the graph files. The list is in sequential order and thus has a set order that each command can be added and executed. The generation of the cmdlist is also done as a separate Python module. This part of the code contains the decision framework on how each macro should be ordered and how to generate the resulting Pilot program. The overall result is the ability to create wiring diagrams and Pilot schematic code from the graph files. For this section, the tool can be viewed as a simple source-to-source translator which changes specific macros in the Pilot code to the information presented in the graph file.
4.2. Graph representation of MPI processes

The graph extraction can be difficult and many optimizations or machine learning techniques could be used for more complicated tasks. However in the Pilot case, the overall structure of all Pilot programs is static. Thus a simple macro substitution, with special parsing from the graph file, is sufficient.

The Pilot structure has been split into several main sections and each section is generated almost independently to the rest of the code. After the general parsing of the graph file is done, the tool can walk through the sections of the Pilot structure and start generating schematic code. Pilot bundles are a special case and handled as several channels.

Specific snippets of code are produced for each section of the Pilot code. These generic code samples form snippets of code with set macros in each file to be replaced by the graph information. An example is shown in Code Listing 4.2. Furthermore, templates exist which are the user specified code for each Pilot process that is presented in the graph file. If no user specified template is found then a generic blank file is entered.

The overall result is a Pilot program that has the wiring information inputted from the graph file and the user created PIProcess functions added to the end of the file. Adding the PIProcess functions in this manner allows the file to be easily regenerated from static code pieces in the templates. The general usage of the tool is shown in Listing 4.1.

```
graph_map.py [-h] graph.dot [outfile.c]
```

Listing 4.1: Usage for the graph_map tool for generic map operations and Pilot code generation.

Because the Pilot parser relies on a general Pilot structure, difficulties can arise when this structure is abnormal. For this reason general skeletons are used to produce different forms of code from the graph files. Similar to algorithmic skeletons [16], users can determine which form most closely represents the usage of their program and use a general skeleton for the automatic code generation. This idea of applying general skeleton structures is common in parallel setups such as Intel’s Threading Building Blocks [65] or
4.2. Graph representation of MPI processes

Concurrent Collections [36]. Currently a chain skeleton is used for the Pilot farm example which was shown in Section 4.1.2. Skeletons are ordered instructions and loops from the cmdlist which are repeated several times. By changing the ordering in the cmdlist, different skeletons can be created for other situations such as trees or blocks. These skeletons are user creatable.

1 for (i=0; i<\{GPI_MODULE\}\nummodules; i++)
2 {
3  \{GPI_NODECHANSLIST\}
4 }

Code Listing 4.2: Snippet definition for a repeated module initialization.

1 for (i=0; i<\{thefarmnummodules\}; i++)
2 {
3  thefarm[i].outTask = PI_CreateProcess(outTask, 5*i+1, &(outTaskchans[i]), &pg);
4  thefarm[i].inResult = PI_CreateProcess(inResult, 5*i+2, &(inResultchans[i]), &pg);
5  thefarm[i].inTask = PI_CreateProcess(inTask, 5*i+3, &(inTaskchans[i]), &pg);
6  thefarm[i].outResult = PI_CreateProcess(outResult, 5*i+4, &(outResultchans[i]), &pg);
7  thefarm[i].doTask = PI_CreateProcess(doTask, 5*i+5, &(doTaskchans[i]), &pg);
8 }

Code Listing 4.3: Filled in snippet definition for a repeated module initialization.

The wiring diagram of the Pilot program is constructed similarly to other sections in the cmdlist. From the rules related to the skeleton structure, the cmdlist generates macros which will result in the wiring diagram. The wiring diagram uses the edge and node information from the graph files to populate the correct macros. After this is done, the wiring diagram is produced by substitutions as the tool walks down the cmdlist. Certain optimizations are possible in the production of the code and this is reflected
4.2. Graph representation of MPI processes

in the skeleton structure. For example, in the chain skeleton a repetition
of independent modules is done to produce a chain. A user could optimize
this procedure by creating a loop which will simplify the initiation process.
Such a snippet definition is defined in Code Listing 4.2 and the resulting
substitution shown in Code Listing 4.3.

Since this is just the initialization of the module code and is being auto-
generated, there is no need to create a loop. A user could hard-code a large
number of modules verbatim. However, by using a user definable skeleton,
the user can use a code snippet and the cmdlist module to generate an array
to define the modules, and initialize this information in a loop to simplify
the auto-generated code. Many of the snippets are static definitions of the
Pilot program which will go in-between the auto-generated code sections.

4.2.3 Explicit and abstract graph declarations

One challenge in representing a large number of processes in a graph file is
the notion of defining nodes explicitly or abstractly. A sample visual repre-
sentation of a GraphViz file is shown in Figure 4.4. An explicit definition
is always possible, as in the Figure 4.4 example, which represents a chain
module with the same module repeated three times.

As the number of modules increase, the task of generating a graph file
can become as cumbersome as physically creating the code or the resulting
mapping. There are several solutions to this problem. First, because of the
popular use of GraphViz, users never need to create these GraphViz files
by hand, instead users would use a graphical interface to create and modify
such graphs. In this case, storing the explicit definition of the resulting
graph in the file is fine. The graphical interface can be thought of as a
graph preprocessor for generating these large graph files.

Another option is to provide an abstract definition language so users may
repeat modules in the graph specification. This would make the resulting
graph file easier to read, as well as the resulting graphical representation
easier to manipulate. This task is difficult and would require formalizing a
representation to define these repeatable abstract modules in the attributes
4.3. Function-level mapping

of the GraphViz file. A custom graphical front-end for recognizing these special attributes and producing a visual representation would also be necessary. For the scope of this work only an explicit definition is used and an abstract definition is left for future work.

Providing this graphical specification to MPI simplifies the management of many processes and encourages simpler function-level programmability. The portability constraints of the tool allow the use of many popular MPI implementations that have a basic round-robin allocation scheme, such as MPICH2 and Open MPI.

4.3 Function-level mapping

The function-level programming paradigm is a process-oriented model that exposes small units of parallelism. Because of the many MPI processes which are present, particular care with managing MPI processes, and the overheads that arise from this model, must be examined.

FG-MPI does not add parallelism; it simply makes it possible to express additional concurrency. It does this by adding a new layer of abstraction which allows the user to further collocate MPI processes within OS-processes. This adds another degree to the mapping system.

Exploring these new mapping dimensions can increase performance by reducing communication overheads. Increased communication can be reduced by collocating MPI processes that communicate often, and ideally MPI processes can be collocated with compute intensive MPI processes to maximize efficiency.

4.3.1 FG-MPI mapping system

A basic overview of FG-MPI is contained in Section 2.2. FG-MPI allows for a smaller unit of parallelism compared to that of regular MPI. By providing a function-level unit of parallelism instead of a program-level one, FG-MPI is able to express additional concurrency not previously available.

FG-MPI uses coroutines to provide this fine-grain processing model. The
4.3. Function-level mapping

ability to use these coroutines are provided at the user level through the addition of a \texttt{-nfg} flag on the \texttt{mpiexec} command. By using the \texttt{-nfg} flag, a user can control the number of fine-grain MPI processes within a single OS-process. For example, the command:

\begin{verbatim}
mpiexec -f hostfile -n X -nfg Y ./my_fgmpi_app
\end{verbatim}

Listing 4.4: A command showing the execution of an FG-MPI application.

starts up $X \times Y$ MPI processes with $Y$ MPI processes inside each of the $X$ OS-processes. Figure 4.6b shows a simple MPI process mapping where 14 MPI processes are mapped across four OS-processes. The MPI processes are sequentially ranked within a collocated OS-process. From this, users have the ability to launch collocated MPI processes, up to memory limitations, inside OS-processes. More complicated launching commands are possible, such as that shown in Listing 4.5, where 14 MPI processes are spawned, 8 in the two OS-processes, and 6 in the remaining two OS-processes, as shown in the layout of Figure 4.6b. This provides a flexible mapping architecture that can represent various bindings to hardware. The only current limitation is that collocated MPI processes are numbered sequentially within an OS-process.

MPI binds OS-processes to hardware via the hostfile. A sample hostfile is shown in Code Listing 3.1. MPI will read the hostfile and bind the first MPI rank to the first entry, the second rank to the second entry, and so forth. It will wrap this hostfile to provide a round-robin type of binding until it runs out of MPI ranks to assign. This simple mapping scheme allows any binding of MPI processes to hardware by explicitly assigning each MPI rank manually.

FG-MPI uses the same technique to bind OS-processes to machines. To assign specific functions to each MPI process a mapping function is necessary. In a simple SPMD program, all MPI processes would be mapped to the same function where all MPI processes would execute the \texttt{main} function. An example FG-MPI mapping function is shown in Code Listing 4.6. The purpose of this function is to take an MPI rank and map it to a function.
4.3. Function-level mapping

Figure 4.6: Illustrating the binding of MPI processes to OS-processes. The large dotted circles represent OS-processes, while the small solid circles represent MPI processes. (a) is the normal MPICH2 case, while (b) is the FG-MPI case.

Listing 4.5: An advanced command showing the execution of an FG-MPI application.

```bash
mpiexec -f hostfile -n 2 -nfg 4 ./my_fgmpi_app : -n 2 -nfg 3 ./my_other_fgmpi_app
```

Mapping functions to processes and binding MPI processes to hardware can become complex, particularly in a process-oriented model. Exploiting attributes that are represented within a graph easily allows for binding schemes to be represented. Furthermore, users can provide optimization modules which can express how to bind MPI process to hardware with more flexibility.

```c
FG_ProcessPtr_t fgproc_map( int argc, char** argv, int rank ) {
    return ( &mainfg );
}
FG_MapPtr_t map_lookup(int argc, char** argv, char* str) {
    return ( &fgproc_map );
}
```

Code Listing 4.6: The mapping functions FG-MPI uses to assign MPI ranks to functions. The listed example maps all ranks to a single function `mainfg` providing an SPMD behaviour.
4.3. Function-level mapping

4.3.2 Formalization of the mapping

MPICH2’s current binding specification is detailed in Section 2.1.2. It uses a process manager called Hydra [35] to map MPI ranks to hardware. The current representation can be expressed as a collection of tuples

\[ Q_{\text{mpich2}}: r_i \rightarrow (p_i, m_i) \] (4.2)

where \( Q \) is the desired mapping function, \( r \) is the MPI rank, and the tuple \((p, m)\) represents the OS-process and machine respectively. \( Q \) in this case is application specific and depends on the hostfile used in the process. In words, \( r_i \) is a consecutive list of MPI ranks, and each rank will have a processor number \( p_i \), on some machine \( m_i \). In this way, Equation 4.2 provides a simple specification for binding MPI ranks to hardware, where only one MPI rank can be assigned to an OS-process on a particular machine.

The number of MPI processes \( N \), can be represented with a collection of tuples \((P_k, M_k)\) where \( M \) is the number of machines with \( P \) processes each. Formulating it as so allows \( N \) to be determined by

\[ N = \sum_k P_k M_k \] (4.3)

where \( k \) is the number of unique collections which exists. If there is an even number of processors per machine, then in this case \( k = 1 \).

When introduced to FG-MPI’s collocation of MPI processes, this breaks the construct that only one MPI rank can be assigned to an OS-process. In FG-MPI, several coroutines can exist within an OS-process which allows for a smaller unit of parallelism. This means a new mapping formalization must be provided. A possible representation is a collection of triplets

\[ Q_{\text{fgmpi}}: r_i \rightarrow (c_i, p_i, m_i) \] (4.4)

where a third number \( c \) will represent a coroutine within an OS-process. This states that each MPI rank must be assigned to a single coroutine \( c_i \), which is in an OS-process \( p_i \), on a particular machine \( m_i \). Calculating \( N \)
4.3. Function-level mapping

follows naturally from this specification as well by adding another number to the triplet \((C_k, P_k, M_k)\). This expresses the calculation of \(N\) as

\[
N = \sum_{k} C_k P_k M_k
\]

where \(C\) is the number of coroutines per OS-process. Again \(k\) is the number of unique collections which may exist. If it is a uniform setup with the same number of coroutines in every OS-process, and an even number of processors per machine, then in this case \(k = 1\).

4.3.3 Fine-grain mapping architecture

FG-MPI allows for a smaller unit of parallelism and added concurrency when compared to MPI. This fine-grain process paradigm in FG-MPI causes MPI processes to be tightly collocated within OS-processes. Multiple stacking strategies are possible when many MPI processes are collocated together.

For example, the Pilot farm application in Section 4.1.2 has collections of coroutines which represent modules. These modules are stacked in a long chain to provide a farm operation. A chain skeleton can be used in many different applications; this provides the same strengths and weaknesses in each case. One such advantage of using a chain is that the simplicity of the structure provides easier understanding of the communication that occurs throughout the application. Since the structure used in this example is a chain, and at times many of the chain modules will not be computing particularly when filling and emptying the chain, modules stacked on hardware by optimizing computation may be ideal. Furthermore, communication patterns in a chain may prefer different stacking techniques to optimize communication.

For example, two binding strategies which could be exploited are stacking neighbours and round-robin. These are illustrated in Figure 4.3. Stacking neighbouring chain modules as shown in Figure 4.3a will optimize for communication, but computation may not be easily distribution over machines. Providing a round-robin strategy as shown in Figure 4.3b will allow for all
4.3. Function-level mapping

computational resources to be utilized quickly, but maximizes communication between machines. A clear trade-off between computation and communication can be seen in each example. A more flexible way to optimize performance can be achieved by allowing for this extra flexibility degree. Varying task size in the farm will depend on the best trade-off and this is further explored in Chapter 5.

Other such examples, such as the prime number application described in Section 1.1.3 shows other needs where a large number of MPI processes are necessary. This example again represents a chain structure, which to maximize the computation an optimized binding strategy is necessary; one that maximizes computation but reduces communication as much as possible. To achieve this, extra information about the structure of the application as well as hardware information is necessary. This process will not necessarily be automatic; however by utilizing all the information available, user input can be significantly reduced. A detailed example of this is presented in Section 4.3.4.

This fine-grain architecture also reflects a function-level programming style discussed briefly in Section 1.1.3. Programmability for parallel languages [13] has been an important topic with the recent popularity of multi-core processors, and the design of scalable applications must often be done from the start [23].

Code Listings 4.7, 4.8 and 4.9 provide a small example illustrating how fine-grain function-level programming may be executed. For example, three forms of a simple operation, such as shifting elements in an array, are presented. Code Listing 4.7 shows a standard sequential version that most programmers are familiar with. Code Listing 4.8 provides an OpenMP [55] version which uses #pragmas in the code to represent parallelism. OpenMP has been a standard tool for exploiting multi-core processors and often is used in converting sequential code.

Code Listing 4.9 shows a function-level approach. This approach operates by representing numbers as messages and array elements as MPI processes. Writing the array shift like this allows for scalability to become a fundamental design in the creation of the application. Small elements
communicate together and when the number of elements increase, so does concurrency and communication; the trick with such an application is controlling fast communication mechanisms and providing optimal mapping of the components. Furthermore, this approach has the added benefit of expressing the parallelism outside the program through mapping instead of expressing parallelism inside the code with #pragmas such as OpenMP. This gives the benefit of exploiting a dynamic remapping of communication for different system architectures instead of hard coding loops or other common parallel structures.

```
for (i=0, i < n, i++) {
}
```

Code Listing 4.7: A sequential example of shifting an array in C.

```
#pragma omp parallel for
for (i=0, i < n, i++) {
}
```

Code Listing 4.8: An OpenMP example of shifting an array in C. Care must be made with shared loop variables.

```
// For every process (i->n) in COMM_GROUP arrayA:
MPI_Send(&reqI, 1, MPI_INT, (myrank_i+2), tag, arrayA);
MPI_Recv(&recvI, 1, MPI_INT, (myrank_i+2), tag, arrayA, &s);

MPI_Send(&reqI, 1, MPI_INT, (myrank_i), tag, arrayB);
MPI_Recv(&recvI, 1, MPI_INT, (myrank_i), tag, arrayB, &s);
myval += recvI; // A[i] += B[i]
```

Code Listing 4.9: A function-level MPI example of shifting an array in C MPI. Array elements are MPI processes and the moving of numbers is done through message communication.
4.3. Function-level mapping

4.3.4 Exploiting the graph information

Representing the communication and node structure as a graph allows further information to be extracted, assisting in the collocation and binding of MPI processes. The hwloc [34] and PLPA [61] tools represent similar techniques exploited by MPICH2 and Open MPI.

One crucial difference between MPI and FG-MPI programming is the unit of parallelism. Having smaller tasks presents more MPI processes; this coupled with the need to collocate processes makes this task more difficult. Because a graph is used, a lot of information can be extracted. Communication patterns are already illustrated in such a structure. Determining the collocation of MPI processes can be determined by analyzing groups within the graph. The information presented provides a large potential for mapping. Other information may be added to the graph as attributes which the user can further exploit.

The user may exploit all this added information in user creatable binding modules. How these custom modules affect the architecture of the tool is shown in Figure 4.5. These binding modules must be created by the user since only the user may know the optimal scheme for binding MPI processes to hardware for their application. Users may use information from hwloc, the hostfile, and the graph communication structure to determine a collocation pattern. Since this is a user generated binding module, the output can be anything the user determines. For example, after the optimal binding strategy is determined, constructing a binding result via a hostfile with the corresponding mpiexec command may be outputted.

By using the Pilot farm example, a simple binding strategy could be envisioned. First the computation and communication trade-off, or the task size each module is to compute, is important to consider. A five module chain could be constructed and the resulting GraphViz graph made. The user may be aware either from the topology hardware or task distribution how computationally expensive each task pushed into the farm may be. If so, using this information is important in computing an optimal binding. One can add the estimated time each module will take computing as a hidden
4.3. Function-level mapping

A communication to computation task size may be known as well, such as that shown in Section 5.1.2 for a chain. The user may now construct a binding module which will read the estimated computation time each module will take, and use a precompiled formula to determine if the neighbouring module should be stacked with this module or put onto the next machine. After this, a `hostfile` is produced and `mpiexec` command is displayed to show an optimal binding given the information. Information from the graph can be further updated and this command may be reproduced quickly for different scenarios.

Providing the ability to construct this input module allows for more complicated scenarios. For example, if a user wanted to provide some special FG-MPI map function, discussed in Section 4.3.1 that is related to the graph structure, then this code could be auto-generated and inputted into the FG-MPI program. In most circumstances generic binding strategies such as the CPU and cache isolation modules could be used. A module to collocate small communication-heavy MPI processes is provided as well.

Automatic graph optimizations are also possible, however may require more sophisticated graph algorithms for discovering patterns. Because the binding modules are user written, more advanced graph optimization algorithms can be done with little modification and is left for future work.
Chapter 5

Evaluation

The evaluation presented consists of different process-to-core binding strategies for function-level programming. Using FG-MPI presents a new problem where a large number of MPI processes may be present.

Using the tools presented in Chapter 4 allows for the creation of thousands of MPI processes in a graph, producing automatic wiring Pilot code, and providing a collocation mapping strategy. An interesting question relates to the possibilities of such a function-level paradigm and any limitations that are still expressible in such a setup.

Evaluation of the unit of parallelism used in a process-oriented model is presented. Comparing the differences of a function-level program to a standard or sequential MPI program was discussed in Sections 1.1.3 and 4.3.3. Much of the evaluation uses the Pilot farm presented in Section 4.1.2. Using a simple structure, such as a chain, is important for evaluation because it provides an understandable analysis of communication.

The test setup consists of a cluster with 26 compute nodes connected by a 10GE interconnection network. Each of the nodes in the cluster is a dual socket, quad-core (8 cores per node) Intel Xeon® X5550, 64-bit machine, running at 2.67 GHz. All machines have 12 GB of memory and run Linux kernel version 2.6.18-274.3.1.el5. The FG-MPI [22] used is modified from MPICH2-1.0.8, using a modified Hydra-1.4.1p1 process manager. The Pilot-2.0 [60] library is used.

5.1 Minimizing the unit of parallelism

An important measurement to be aware of in a function-level programming space is the smallest amount of parallelism that is achievable within a certain
setup. To analysis this the Pilot farm discussed in Section 4.1.2 is used.

\(N\) is decoupled from the hardware in FG-MPI, meaning it is no longer bounded by the number of cores in the machine but by properties of the application. In the Pilot farm example, \(N\) represents the number of modules. An \(N\) module farm consists of \(5N + 1\) MPI processes where ideally the farm is mapped to \(N\) OS-processes.

![Diagram](image)

Figure 5.1: Showing a reducing unit of parallelism from (a) to (b) to (c) for each consecutive OS-process. Large dotted circles represent OS-processes, while small solid circles represent MPI processes.

Since this farm is a chain, for each module the flow of in-tasks must equal the flow of out-results, thus in a steady-state one can solve the flow equations to determine the maximum throughput of the farm in terms of the task execution time. Because of the simple structure, the start-up, steady-state, and wind-down phases of processor farms are well understood [84]. By experimenting with different task sizes, or computation costs, on a fixed size chain mapped to eight nodes of the cluster, it is possible to determine the task size at which communication begins to dominate for this particular chain structure.

### 5.1.1 Cost of an MPI_Send

One important property for a function-level programming paradigm is a small unit of parallelism; thus it is important to know how small these units of parallelism may be. Because communication in a function-level
5.1. Minimizing the unit of parallelism

A parallel program is large, it is also essential to provide a fast communication mechanism to support such a paradigm.

Table 5.1 provides the communication time for a one-way MPI.Send in FG-MPI. For the send, a message size of 8192 bytes was used. In a process-oriented model small isolated entities communicate and this communication usually reflects a large amount of small messages. 8192 bytes is the upper limit of the small message size in MPICH2. Larger and smaller messages sizes show similar trends. The round trip for a send and receive was measured and the number presented is half this time. This number should represent the latency and communication overheads FG-MPI or MPICH2 provides in each scenario.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Compute</th>
<th>Communication</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Between Nodes</td>
<td>Parallel</td>
<td>TCP</td>
<td>62.47µs</td>
</tr>
<tr>
<td>Between OS-Processes</td>
<td>Parallel</td>
<td>Shared memory</td>
<td>3.13µs</td>
</tr>
<tr>
<td>Within Collocated Processes</td>
<td>Shared</td>
<td>memcpy</td>
<td>1.41µs</td>
</tr>
</tbody>
</table>

Table 5.1: Communication costs for a single one-way MPI.Send in FG-MPI for a message size of 8192 bytes on the test setup.

For the communication between nodes, FG-MPI uses the default MPI-CH2 TCP communication. This number is dominated by the latency between the two machines. For intra-process communication, FG-MPI uses MPICH2’s default Nemesis [9], a fast communication mechanism which operates on cache. Other tools, such as KNEM [41] used by Open MPI, actively work to improve intra-node communication in MPI. For collocated processes, FG-MPI uses an optimized message passing interface that can even bypass the middleware to communicate in some cases since both MPI processes exist in the same OS-process. This essentially should mimic a memcpy and function return, which provides the fastest communication of the three cases. In this case, the collocated MPI processes must concurrently share the core, while the other two cases run in parallel. Even in the pure MPI case between OS-processes good performance can be seen. Furthermore, because of power hungry operations, such as cache coherence, passing a message can

55
5.1. Minimizing the unit of parallelism

sometimes be faster than shared memory access between cores associated on different caches, as shown in the Barrelfish [73] project.

This shows that to reduce the heavy communication presented in function-level programming, it is ideal to collocate MPI processes that communicate often. One bonus in using FG-MPI is that utilizing this quick collocated communication time can be achieved by stacking processes together. Some of these stacking techniques are explored in this chapter.

5.1.2 Minimum task size for inter-node

Knowing the smallest unit of parallelism possible is an important property in exploiting a function-level paradigm. The numbers presented in Table 5.1 show a lower bound on the minimize task size possible. This means it is quicker for the process to compute the task than to send the message. Using the Pilot farm allows for further evaluation to determine a more accurate value.

The variables of the Pilot farm are the amount of work (computational task size time and number of tasks), chain length (number of modules), and the number of cores. Over 64 cores, a 63 module chain is run where one core is dedicated to the Manager process. The modules are mapped to the cores so that all inter-module communication is identical and uses the TCP communication mechanism. All intra-module communication is done by the FG-MPI optimized communication mechanism. To separate communication between modules, these are mapped round-robin across nodes as shown in Figure 4.3b. In this example, the worst-case mapping with uniform communication is measured, rather than taking advantage of having fast intra-node communication; this would be when mapping neighbouring modules to the same node was performed to optimize communication. The workload consists of iterating over a set of floating point operations. This was chosen to provide an easy ability to adjust task size from the sequential task to a number of smaller tasks doing an equal amount of work overall; care was taken to prevent compiler optimizations over the loop and buffering values in the cache. Furthermore, a workload that is sufficiently large to achieve
5.1. Minimizing the unit of parallelism

Figure 5.2: A module Pilot farm varying task size. The amount of work per OS-process remains constant by changing the number of tasks accordingly. 63 and 126 modules were used in the one module and two modules per OS-process cases respectively. 63 cores were used over eight machines.

steady-state was used, meaning that starting and emptying the chain does not represent a large amount of the overall time.

Figure 5.2 shows that by reducing the task size and increasing the number of tasks to maintain a constant amount of work, a computational task size of 12.6 ms per task can be achieved with a communication overhead of around 6.5%. Beyond 12.6 ms the task size becomes too small and Workers in the chain finish tasks too quickly. This causes the tasks to not forward work often enough to keep all processes busy, and thus not utilizing all the computation resources. It is possible to MPI_Yield more frequently, which tells the Workers to stop and do other tasks in the collocation group. However, the combination of the increased number of MPI_Yields, and the difficulty in keeping processes busy, cause the overhead to rise sharply.
5.1. Minimizing the unit of parallelism

The smallest effective task size theoretically possible is related to the time `MPI_Send` takes to complete, which is shown in Table 5.1. On this setup, it was measured to be around 0.062ms, much lower than our calculated minimum task size of around 12.6ms.

These experiments have shown that promoting a function-level paradigm, and reducing the unit of parallelism, is only useful if it is possible to effectively execute small enough tasks without hitting overheads in MPI. For this particular chain structure, which provides an easy evaluation of communication, a specific unit of parallelism is necessary to keep MPI processes busy. The minimal computational task size was calculated to be about 12.6ms of work. One way to further push this number is to add more communication into the collocated OS-processes; this can be done through stacking.

Figure 5.2 also shows that by stacking two modules per OS-process, effectively doubling the chain length, allows a smaller task size of 8ms before hitting an overhead wall. By increasing the chain length, yet keeping the same number of processors, care must be taken to maintain the same amount of work for each OS-process. This was done by keeping the task size per OS-process the same, thus each pair of modules compute the same amount of work compared to the one module per OS-process.

Figure 5.2 shows a static overhead compared to the one module per OS-process case, which is related to the chain structure used in this example. Because of the increase in chain size, the start-up and wind-down to initially reach a steady-state is increased: which is to be expected. This was also hindered from the smaller actual task size. However what is interesting is that the minimum task size for the OS-process was able to reach around 8ms, with around 4ms actual task size per module. The decrease of the effective task size is due to the optimized communication for the neighbouring module pairs, coupled with the increase of tasks each module must execute, which causes each module to become busier. This experiment can be repeated by collocating more modules together and reducing the computational task size even lower. Trade-offs in maximizing computational resources (cores), or minimizing communication (collocating processes), starts to become a
5.1. Minimizing the unit of parallelism

The interesting result is that a very small task size is achievable by stacking processes. Using many fine-grain tasks in the Pilot farm, at the worst-case, suffers a communication overhead of around 6.5%. The constant portion of the overhead compared to the sequential time is due to the uneven non-steady-state distribution of work throughout the chain. The frequency of the MPI_Yield provides control of this overhead, but becomes unstable at small task sizes. Comparing the steady-state of the chain shows near sequential time operation with only a small communication overhead.

5.1.3 Minimum task size for collocated processes

From Section 5.1.1, the communication of a single MPI_Send within collocated processes was determined to be 1.41μs. Thus this is the absolute minimize task size that can be achieved within FG-MPI. The following experiment attempts to further push the minimal task size in function-level programming using FG-MPI.

First, by putting all the modules into one large collocated OS-process insures that only fast optimized communication is occurring between MPI processes. Stacking a seven module chain, which has a total of 35 collocated MPI processes inside one OS-process, allows for the reduction of the computation task size. Table 5.2 shows such a setup.

<table>
<thead>
<tr>
<th>Task size</th>
<th>Total time</th>
<th>Amount of comm.</th>
<th>Average time per comm.</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.6ms</td>
<td>1648.0s</td>
<td>2,000,040</td>
<td>1.52μs</td>
</tr>
<tr>
<td>2.9ms</td>
<td>1660.2s</td>
<td>9,999,920</td>
<td>1.52μs</td>
</tr>
<tr>
<td>1.8ms</td>
<td>1667.8s</td>
<td>16,000,040</td>
<td>1.43μs</td>
</tr>
</tbody>
</table>

Table 5.2: A seven module chain with 35 collocated processes within one OS-process. Task size and number of tasks were varied to maintain a constant amount of work. Average time per intra-process communication is shown.

The amount of work for each case remains constant by varying the number of tasks and the computation size of each task. This is shown by the total time, which is constant, minus some communication overheads related
5.2 Scalability of the unit of parallelism

to the increase in the number of tasks. Because the communication pattern is easy to analyze for the chain, the total amount of communication can be determined and is shown in Table 5.2. In each case the average $1.49\mu s$ for collocated process communication confirms the result found in Section 5.1.1. Overall, a minimal computation task size of $1.8ms$ was achieved. This number was computed with 800 thousand tasks and over 16 million communications occurring within one OS-process. Because of memory limits, a task size lower than $1.8ms$ was unable to be executed.

The trade-off with optimizing for so much communication is that this limits the amount of hardware resources that can be utilized. Ideally a combination of communication and computation is necessary. Users who attempt to program in a process-oriented model should try to maintain a task size of at least $1.8ms$ within a collocation process setup.

5.2 Scalability of the unit of parallelism

From Sections 5.1.2 and 5.1.3 it was determined that the minimal actual task sizes were $12.6ms$ and $1.8ms$ respectively for inter-node and collocated modules. These numbers represent the specific setup which was used in a purely distributed or purely collocated setup. In reality, most users will use a combination of these two situations over a large distributed machine.

When expanding to more hardware, the scalability factor of these small units of parallelism is important. Figure 5.3 shows the speed-up that is achievable by varying the chain length of the Pilot farm. As the chain length increases (number of modules and number or cores), a better speed-up is achievable for a larger task size. Decreasing the chain results in a smaller task size at the cost of less nodes or computational power.

As expected, as more computational hardware is added, a larger task size is needed to keep the hardware busy. This is due to the heavy inter-node communication that uses expensive communication. Overall the result shows that for scalable applications that need a lot of computational power, a minimum specified task size should be respected. Furthermore this also shows that the heavy communication patterns that the chain farm demonstrates
5.2. Scalability of the unit of parallelism

Figure 5.3: Speed-up for a seven module Pilot farm for a fixed task size and amount of work. Each module is mapped to its own OS-process and core on a distributed machine.

does not significantly affect the speed-up, and that achieving speed-up in such an application does not required reducing communication, but instead increasing the task size each module must execute.

Furthermore, the data in Figure 5.3 represents the worst-case scenario where every neighbouring module is on a different node. This is similar to the stacking strategy shown in Figure 4.3b. This was done to maximize the communication that is executed. Within modules, five MPI processes communicate via the fast collocated sending mechanism.

Another way to look at scalability is within a single node. Figure 5.4 locks the amount of computational resources and looks at the increase in time for the same amount of work. By using the formula, \( W = T \cdot S \), where \( W \) is the overall work, \( T \) is the number of tasks, and \( S \) is the task size, the overall work remains constant when increasing the number of tasks \( T \)
5.2. **Scalability of the unit of parallelism**

by providing a smaller task size $S$. The workload in this case consists of iterating over a set of floating point operations, where the iteration number could be used to control the task size; care was taken to prevent compiler optimizations over the loop and buffering values in the cache. This differs from Section 5.1.3 in that the modules are not stacked within one collocated group, but instead spread over seven collocated groups on a single node: using all the computational resources of the node. This shows the shared memory scalability.

![Figure 5.4: Shows the linear relationship for the increase in communication computed over one node with constant work. The increase in time represents the overhead in the communication cost for the increased number of tasks.](image)

The actual time difference between the cases are not large, since the same amount of work is being computed. This is similar to taking 1450s of work and dividing it up into smaller pieces; the more tasks there are the smaller each piece is, but overall the same amount of work is used. A small linear overhead is observed as the number of tasks increase from *100 thousand* to
5.2. Scalability of the unit of parallelism

800 thousand over the single node. This small linear overhead is directly represented by the communication increase from the increase in the number of tasks. Overall in this case, scaling to a very large number of tasks within a single machine only provides a communication overhead of around 3.5s or around 1.7% of the sequential time.

Summary  In the evaluation, care was taken to narrow down the minimal task size, or calculate the smallest unit of parallelism within the test setup described at the beginning of this chapter. This value is important for the paradigm described and the tools created throughout this thesis.

Section 4.1 demonstrates that Pilot is an MPI library that can be used to provide an easy CSP channel interface. Section 4.2 describes a tool for wiring large diagrams in a many-process application via a graphical interface. Section 4.3 talks about the mapping FG-MPI gives and the small unit of parallelism it provides to Pilot. With these three ideas, a framework is presented that provides a flexible and easy to manage system for a large scale process-oriented model. Evaluation in this section shows that very small units of parallelism are possible, even with the increase in communication, which is a characteristic of function-level programming. Furthermore, scalability has been shown to be related to this unit of parallelism and that the system is scalable to both the number of tasks and the computational resources.
Chapter 6

Conclusion

Function-level programming provides easy programmability aspects by building and combining small components to make more complicated applications. However, MPI was developed for heavy-weight program-oriented applications which do not necessarily fit well with function-level concepts. By porting Pilot to FG-MPI, a smaller unit of parallelism is expressed. With the CSP-like communication mechanism of Pilot and the small unit of parallelism from FG-MPI, a solid foundation to express a function-level programming architecture in MPI is established. Challenges with communication, scalability, flexibility, and ease of use for managing many processes are quickly revealed.

A graph specification was developed, extending MPI to bind processes graphically instead of through the standard hostfile. This graph structure uses a popular graphical format for usability and is flexible by providing external information into the structure. This well-defined graphical structure provides a solution to the mapping problem for a process-oriented model. Furthermore, the extra layer of binding that FG-MPI exposes is easily expressed in this graph structure and can be used to provide further collocation mappings.

A modular tool was developed to provide an interface between the graphical representation and several operations users may want to perform. These tasks are represented as user created modules that can read the graph structure. Currently a Pilot wiring module is constructed that auto-generates the task of creating Pilot wiring and communication code. This task becomes daunting when introduced to function-level programming which can have a large number of complicated communication patterns.

The graphical representation can include external information to perform
many operations. For example, complicated binding strategies or extended graph format modules could be constructed. Since these strategies are user defined, this is a generic tool that can be used with many other run-times.

Evaluation was performed to determine the smallest unit of parallelism. Experiments involving collocated, intra-node and cluster setups were executed. Scalability of this unit of parallelism was also explored. It was concluded that the communication overheads are not significant when programming in a process-oriented model, as long as fast communication mechanisms exist.

6.1 Future work

A modular tool allows for flexible and extensible advantages. Possible future work with the tool involves more advanced graph optimization mapping and binding techniques. Because the tool was written in Python, complicated graph algorithms may be used on the GraphViz file to optimize several factors. For example, it can be difficult to find an optimal collocation pattern for a large graph. By using a more advanced mapping module, groups of processes with large communication could be collocated together. Furthermore, computational information can be stored into the graph format and used to help optimize the collocation process. Binding strategies can then be calculated to determine OS-process to machine mappings, or collocation of MPI processes to OS-processes. The output is not set to a specific implementation of MPI, instead it can use any user-defined output and thus can construct the equivalent hostfile, FG-MPI mapping function, or any other independent format.

More work is possible within Pilot, the third party CSP-like library that was used. Pilot was developed for an MPI environment. Even though the code was altered in the porting to FG-MPI, several memory-optimizations are still possible. Pilot has many large structures that every MPI process stores. Because of the collocated nature of FG-MPI, it is further possible to optimize these memory structures by identifying the collocated groups and have them share the static data among themselves. This will make only
6.1. Future work

one copy of the large structures per group. Work has been done on the Pilot process structure, however this could be repeated for the channel and bundle structures. Because FG-MPI is memory bound with the number of MPI processes it may create, this will optimize the number of possible MPI processes that can be launched.

More future work to the Pilot library may involve removing the global variables from Pilot in a more backwards compatible way. Currently these global variables are converted to a structure and passed through the API of each Pilot function call. A more structured way could be to rewrite portions of the Pilot code to prevent that change in the API calls. This will make FG-MPI Pilot programs backwards compatible with the original Pilot software.

Currently the graph format chosen to represent the communication and process structure of the application is explicit. This means that for a 3000 process system, 3000 processes must explicitly be declared in the graph file. In most cases this is not an issue since the graph format is well known and users create these graphs with graphical front-end tools. However, one way to reduce such a large graphical file is to provide an abstract definition for some of the graph information. This is complicated and would also require a custom visualization tool to express the abstract format. An abstract format was not developed for use with the tool because of the many drawbacks presented in such an approach, such as the loss of using a well-establish graph library, and communal tools that work with the standard graph format.

Furthermore, a custom GUI front-end could be developed allowing for easier manageability of the graph files. A custom front-end would also solve the issue with the abstract definition. This would allow for a user interface that more closely represents the parallel programming environment.

Summary Overall a flexible, maintainable, and extensible system was developed to extend function-level programming practices to MPI. Using FG-MPI promotes a more natural extension to Pilot that allows for this new programming paradigm. Expressing applications in this format provides easier programmability, scalability advantages, and lower overheads.
Bibliography


Bibliography


Bibliography


Bibliography


Bibliography


[47] Benoit B. Mandelbrot. Fractal aspects of the iteration of $z \to \lambda z(1-z)$ for complex $\lambda$ and $z$. Annals of the New York Academy of Sciences, 357(1):249–259, 1980. →[pg. 36]


[67] Carl Ritson, Adam Sampson, and Frederick Barnes. Multicore Scheduling for Lightweight Communicating Processes. In John Field and Vasco Vasconcelos, editors, *Coordination Models and Languages*, volume 5521
Bibliography


Appendix A

Code Appendix

Code Listing A.1: The wiring code for the Pilot farm program.

```c
// wire first module to the manager - CreateChannel(fromProcess, toProcess)
managerPutchans.out = intaskchans[0].in = PI_CreateChannel(
    managerPutProcess, thefarm[0].inTask, &pg);
managerPutchans.in = NULL;
outresultchans[0].out = managerGetchans.in = PI_CreateChannel(
    thefarm[0].outResult, managerGetProcess, &pg);
managerGetchans.out = NULL;

for (i = 0; i < nummodules; i++)
{
    // inter-module wires for results and tasks, the last one is special -- NULL terminated
    if (i == (nummodules - 1))
    {
        inresultchans[i].in = NULL;
        outtaskchans[i].out = NULL;
    }
    else
    {
        outtaskchans[i].out = intaskchans[i + 1].in =
            PI_CreateChannel(thefarm[i].outTask, thefarm[i + 1].inTask, &pg);
        outresultchans[i + 1].out = inresultchans[i].in =
            PI_CreateChannel(thefarm[i + 1].outResult, thefarm[i].inResult, &pg);
    }
    intaskchans[i].outlocal = Workerchans[i].in =
        PI_CreateChannel(thefarm[i].inTask, thefarm[i].Worker, &pg);
    intaskchans[i].out = outtaskchans[i].in = PI_CreateChannel(
```

78
Code Listing A.2: A prime number solver written in Pilot. Only the Worker process is shown.

```c
int Worker(int pid, void* parameters, void *pg)
{
    // Pilot broiler code
    PI_PROCENVT *pgg = (PI_PROCENVT*)pg;
    doTaskparameters_t *p = (doTaskparameters_t *) parameters;

    ulong num,myprime=0;
    mpz_t integ;

    int notdone = TRUE;
    while ( notdone )
    {
        PI_Read(p->in,pgg, "%lu", &num);
        if ( num != (ulong)TERMINATE_TAG ) {
            mpz_init_set_ui(integ, num);
            if ( myprime == 0 ) {
                myprime=num;
```
Appendix A. Code Appendix

```c
fprintf(stderr,"%lu, ",myprime);
}
else if ( !mpz_divisible_ui_p(integ,myprime) ) {
    //i.e. ( num % myprime != 0 )
    if ( p->out != NULL ) PI_Write(p->out,pgg,"%lu",num);
}
else {
    notdone = FALSE;
    /* Send the terminate_TAG*/
    if ( p->out != NULL ) PI_Write(p->out,pgg,"%lu",num);
}
mpz_clear(integ);
return 0;
```

Code Listing A.3: A GraphViz representation of a farm module.

```graphviz
digraph PILOT {
  managerPut [PIdesc=manager];
  managerGet [PIdesc=manager];
  managerPut -> inTask1;
  outResult1 -> managerGet;
  managerGet -> NULL;
  NULL -> managerPut;
  subgraph cluster_1 {
    PIdesc=thefarm;
    inTask1 -> outTask1;
    inTask1 -> doTask1 [PIdesc=outlocal];
    doTask1 -> outResult1;
    inResult1 -> outResult1;
    inTaskBundle1 [PIdesc=inTask1,PIdesc=inreq,PIdesc= bundletask1 =outTask1,PIdesc= outreq,PIdesc=bundle out2=doTask1,PIdesc= outreq]
  }
}
```
Appendix A. Code Appendix

Code Listing A.4: Portions of a cmdlist structure from a Pilot farm graph.

```plaintext
('1', [])
('2', [( '{GPI_NODENAME}', 'outResult' )])
...
('2', [( '{GPI_NODENAME}', 'doTask' )])
```
Appendix A. Code Appendix

5 ('3', [('\{GPI_MODULE\}', 'thefarm'), ('\{GPI_NODEPROCESSLIST\}', '
\tPI_PROCESS * outTask;
\tPI_PROCESS * inResult;
\tPI_PROCESS * inTask;
\tPI_PROCESS * outResult;
\tPI_PROCESS * doTask;\n')])
6 ('4', [('\{GPI_NODENAME\}', 'outResult'), ('\{GPI_NODEEDGELIST\}', '
\tPI_CHANNEL * in;\n\tPI_CHANNEL * out;\n')])
7 ...
8 ('6', [('\{GPI_NODENAME\}', 'doTask'), ('\{GPI_MODULE\}', 'thefarm')])
9 ('6a', [('\{GPI_NODE\}', 'manager'), ('\{GPI_NODENAME\}', 'manager')])
10 ...
11 ('8a', [('\{GPI_WIRE\}', '\tinTaskchans[0]. out = outTaskchans[0].
in = PI_CreateChannel(thefarm[0]. inTask, thefarm[0]. outTask
, &pg);')]])
12 ('8a', [('\{GPI_WIRE\}', '\tinTaskchans[2]. out = outTaskchans[2].
in = PI_CreateChannel(thefarm[2]. inTask, thefarm[2]. outTask
, &pg);')]])
13 ('8a', [('\{GPI_WIRE\}', '\tinResultchans[2]. out = outResultchans
[2]. in = PI_CreateChannel(thefarm[2]. inResult, thefarm[2].
outResult, &pg);')]])
14 ('9', [])