PRESTO: A PARALLEL RUNTIME ENVIRONMENT FOR SCALABLE TASK-ORIENTED COMPUTATIONS

by

DAVID M. OZOG

A THESIS

Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Master of Science

June 2013

THESIS APPROVAL PAGE

Student: David M. Ozog

Title: PRESTO: A Parallel Runtime Environment for Scalable Task-Oriented Computations

This thesis has been accepted and approved in partial fulfillment of the requirements for the Master of Science degree in the Department of Computer and Information Science by:

Dr. Allen Malony	Chair
and	
Kimberly Andrews Espy	Vice President for Research & Innovation/ Dean of the Graduate School

Original approval signatures are on file with the University of Oregon Graduate School.

Degree awarded June 2013

© 2013 David M. Ozog

THESIS ABSTRACT

David M. Ozog

Master of Science

Department of Computer and Information Science

June 2013

Title: PRESTO: A Parallel Runtime Environment for Scalable Task-Oriented Computations

While the message-passing paradigm, seen in programming models such as MPI and UPC, has provided a solution for efficiently programming on distributed memory computer systems, this approach is not a panacea for the needs of all scientists. The traditional method of developing parallel applications in C/C++ and Fortran potentially leaves behind high-level and heterogeneous environments which are the most conducive for supporting modern computational science endeavors. PRESTO alleviates this problem with an easy-to-use framework which provides multi-language adapters to a flexible MPI middleware supporting common computational models such as the asynchronous master/worker and ring pipeline in heterogeneous environments.

CURRICULUM VITAE

NAME OF AUTHOR: David M. Ozog

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon, Eugene, OR
Whitman College, Walla Walla, WA
Boston University, Boston, MA

DEGREES AWARDED:

Master of Science, Computer Science, 2013, University of Oregon
Master of Science, Chemistry, 2009, University of Oregon
Bachelor of Arts, Applied Mathematics, 2007, Whitman College
Bachelor of Arts, Physics, 2007, Whitman College

AREAS OF SPECIAL INTEREST:

Parallel and Distributed Computing
High Performance Computing
Scientific Computing

PROFESSIONAL EXPERIENCE:

Research Aide, Argonne National Laboratory, Chicago, IL

Software Developer/Project Manager, Concentric Sky, Eugene, OR

GRANTS, AWARDS AND HONORS:

DOE CSGF, 2013

Graduate Research Fellowship, Computer & Information Science, 2010 to present

Cum Laude, Whitman College 2003-2007

Honors Physics Thesis, Whitman College, 2007

Research Experience for Undergraduates (REU), University of Oregon, 2006

ACKNOWLEDGEMENTS

I would first like to thank Allen Malony for providing guidance, motivation, and support during the design and development of this project. Without his expertise, the most interesting parts of PRESTO would never have been implemented. I would also like to thank the team members of the NeuroInformatics Center, the Performance Research Lab, and the University of Oregon Department of Computer and Information Science. In particular, Craig Rasmussen, Chris Hoge, and Rob Yelle have been very helpful in providing technical advice and systems support. I would also like to thank Charlotte Wise for taking the time to proof read documents, and providing much needed guidance. And finally I extend thanks to Douglas Toomey, David Hammond, and Christopher Bone for their work and patience while being the first real-world users of the PRESTO environment.

TABLE OF CONTENTS

Chapter Page

I. INTRODUCTION 1

1.1. A Problem in Parallel Computational Science 1

1.2. What Do Science Application Developers Need? 4

1.3. How Do We Solve This Problem? 6

II. DESIGN AND ARCHITECTURE 8

2.1. Design Considerations 8

2.2. Performance 12

2.3. Scalability 12

2.4. Usability 13

2.5. Other Considerations 14

III. INTERFACE 15

3.1. Launching the PRESTO Environment 15

3.2. Computational Models 15

3.3. Language Interfaces 16

Chapter Page

IV. APPLICATIONS 19

4.1. Stingray: A Seismic Geophysics Application 19

4.2. Head-Modeling: A Neuroinformatics Application for Conductivity Analysis 22

4.3. Head-Modeling's Computations 23

4.4. Head-Modeling and PRESTO Integration 24

V. PERFORMANCE 27

5.1. Relative/Absolute Performance 27

5.2. Weak Scaling 29

5.3. Strong Scaling 30

VI. CONCLUSIONS AND FUTURE WORK 37

6.1. Future Work 37

6.2. Conclusion 39

REFERENCES CITED 42

LIST OF FIGURES

Figure Page

1.1. Parametric Simulation Sweep - A hypothetical heterogeneous application . . 7
2.1. Software Design 9
2.2. Scalable operations 13
3.1. Master/worker interface 17
3.2. Ring 17
3.3. Ring interface 17
3.4. Java interface 18
4.1. Stingray dataset 20
4.2. Head Modeling Conductivity Inverse 24
4.3. Heterogenous Environment 26
5.1. Weak scaling comparison between PRESTO and the DCS 31
5.2. Weak scaling for PRESTO beyond 256 workers 32
5.3. Strong scaling comparison between PRESTO and the DCS 34
5.4. Strong scaling of the Stingray geophysical application 36
6.1. Task to task 40
6.2. 2D ghost-cell application 40
6.3. PRESTO cluster environment 41

LIST OF TABLES

Table Page

5.1. ACISS cluster specifications 27

CHAPTER I

INTRODUCTION

This thesis discusses the motivation, design, implementation, and application of a Parallel Runtime Environment for Scalable Task-Oriented Computations (PRESTO). This chapter introduces a common and important problem in the field of parallel and distributed computing systems, then considers what sort of software framework might alleviate the problem. Finally, the ideas behind PRESTO design goals and implementation are presented in the context of mitigating such problems.

1.1. A Problem in Parallel Computational Science

Current trends in scientific application requirements and massively parallel hardware systems suggest that efficiently exploiting large-scale resources will provide a substantial challenge in the future of computing. While parallel frameworks, tools and languages exist and are ever being developed to aide scientists in conducting parallel computations on distributed memory systems, such tools still require advanced knowledge of performance optimization techniques and distributed algorithm efficiency. For instance, the simple optimization involving the overlap of computation with communication requires a fairly strong familiarity with a particular parallel programming API, and often non-blocking routines (which are needed in this case) are not available in languages that typical domain scientists are most comfortable using. Even if such interfaces exist and are easy to work with in a particular context, they may lack portability, reliability, and performance across different architectures and operating systems. Many domain scientists therefore require assistance from expert parallel programmers to maintain efficiency and scalability of their cutting-edge scientific codes.

Established traditions in parallel programming place a burden on scientists even when their requirements are simple. Current supercomputer architectures are loosely based on the notion of prioritizing parallel capability over ease of programming. This results in many modern systems following something akin to a commodity cluster model where more parallel hardware at low cost is preferable to less parallel hardware at high cost. An example would be relatively slow yet highly energy efficient commodity hardware in the Blue Gene\Q system [10]. In order to work within such Non-Uniform Memory Access (NUMA) systems, the ubiquitous architecture for conducting computations on modern parallel hardware, one must use some sort of a message passing library. While the Message Passing Interface (MPI) standard [9] has proven itself as a powerful player that will stick around for quite some time, there is an unnecessary burden on scientists to learn an interface that is generally unrelated to the problems that scientists are trying to solve. Furthermore, being able to design and implement a working MPI application is far different from writing MPI programs that are optimized for efficiency: the latter task is considerably more challenging and time consuming.

While MPI and 3rd party MPI libraries have unequivocally powered distributed memory parallel computing during the last two decades, using these tools efficiently requires application developers to use low-level antiquated languages, which can result in relatively poor scientific productivity. This results in developers being forced to work within the realm of C and Fortran, while placing a formidable hurdle on those working in more modern and flexible languages. While MPI implementations do exist in other languages (Python, C++, OCaml, etc.) they are typically subsets of the official MPI specification. Furthermore, application scientists, who may have relatively little experience and interest in the intricacies of efficient parallel computing, should not be required to learn an entire message-passing interface if it is not necessary. Instead, they should be provided

with clear abstractions for conducting parallel computations in terms of common models found in scientific computing.

It can also be difficult for parallel programmers to write software effectively even when working within simple computational models. For example, a plethora of embarrassingly parallel solutions are implemented across a variety of applications using the master-worker paradigm. However, each software system is unique, presumably because this model is easy to understand and implement. However many applications show subtle inefficiencies that are only noticed after careful performance profiling, or via inspection by an expert that is familiar with load balancing techniques and other optimizations. This is only one example of a common parallel computational model. Others include ring pipelines, Bulk Synchronous Parallel (BSP), MapReduce, SIMD/MIMD, and other forms of data and task-based parallelism [22]. For this problem there is a void of tools in the community that provide a common and easy-to-use interface for scientists to solve concurrent software problems in a portable, reliable, and elegant way. Collaboration between groups of researchers in computational science is therefore limited because people are forced to work in homogeneous environments of tools that are typically not interoperable or supportive of dynamic analysis.

While the modern solution may be to architect new parallel applications to use flexible and high productivity parallel languages such as Chapel [2], this leaves behind legacy applications, because code translation typically needs to be done manually. This solution also does nothing to support computational environments that can easily expose parallelism, but are not in a place to exploit modern supercomputer clusters because of computational environment constraints. Example are codes written in the languages most common in scientific development: Python, Java, Matlab, R, etc.

In summary, there are three essential problems caused by the status quo in science-based parallel computing mentioned in this section:

1. In order to do parallel computing efficiently and portably, domain scientists are forced to work within the constraints of MPI: using low-level languages and needing to learn effective utilization of the API.

2. Because of such constraints, collaborative efforts are restricted to homogeneous computational environments, which is counter productive for the needs of most application scientists.

3. Common parallel computational models (such as master worker, ring pipeline, and DAG scheduling) are often re-implemented from application to application in instances which a general framework could exist, but does not. If a well-suited tool does happen to exist, then problem #2 often prohibits progress.

Although the solution to each of these problems is daunting to consider, the purpose of this paper is to propose the design and partial implementation of a parallel programming environment, called PRESTO, that attempts to solve all three concerns.

1.2. What Do Science Application Developers Need?

In alignment with the core problems mentioned in current parallel computing from 1.1., there are three goals for the development of a tool that provides a solution:

1. Support the execution of parallel programs in heterogeneous environments in this library (see Chapter II for more information).

2. Create a library with a *common interface* for writing parallel programs in a variety of popular programming languages (see Chapter III)

3. Provide an interface to common parallel computational models such as master worker, ring pipeline, and DAG scheduling (Chapter III) with optimized performance and scalability.

These goals provide the motivation and requirements for the development of such a tool. These goals are addressed and partially resolved in the first implementation of the parallel runtime environment design discussed in this thesis.

To fully appreciate why users need Goal #1 and the power that it would enable, consider Figure 1.1. which represents a hypothetical scientific workflow. This workflow requires a parametric sweep of various input to a simulation. Such parameter sweeps are very common in application evaluation, for instance when statistical analysis might describe system properties. In Figure 1.1., there are nodes Sim_i for $i = 1..p$ which are separate and independent instances of a simulation run. Because these execution instances are independent of each other, they can execute in parallel. For the sake of this example, we shall assume that the computational engine that each Sim_i requires for execution is MATLAB. When running multiple instances of MATLAB across cluster nodes, a sweep program written in a another language (such as Python) could potentially launch separate instances of the simulations for each set of inputs. This program is represented as the “sweep" node in Figure 1.1. The sweep program launches each instances of a simulation with varying input data (represented by blue bars in the figure). As soon as a simulation finishes, it sends its output data back to the sweep application (yellow bars) for 3 reasons:

1. So the sweep program knows when all jobs are finished.

2. The program can make dynamic scheduling decisions.

3. The user and can collect all results in one place.

A subset of the output data (green bars) are then sent to a centralized statistics (or visualization) program which computes user-specified analysis on the fly. Let us assume that this is a separate routine written using the R programming languages. In real-time, the user can monitor the progress of the simulation system and make decisions about the state of the execution by either monitoring the status of the R statistics output, or by viewing a visualization (or both).

1.3. How Do We Solve This Problem?

The presented framework, a Parallel Runtime Environment for Scalable Task-Oriented Computations (PRESTO), is an attempt to solve the three problems mentioned at the end of section 1.1. It solves problem #1 by providing plug-and-play capability for heterogeneous compute environments. In other words, if an application or workflow requires multiple computational engines to complete an overall problem, PRESTO supports an environment to do so. In the case of Figure 1.1., if the sweep, sim, and stats programs all execute in different interactive, interpreted languages (such as Python, Matlab, and R), then PRESTO allows users to piece together an overall execution environment that utilizes these different computational engines in a combined way.

PRESTO solves problem #2 by providing a common programming interface for a handful of popular programming languages (Python, Matlab, Java, C++, and R). This paper will present this interface from two perspectives: the user's and the developer's. Users are provided a simple to use interface which abstracts parallel computational models (i.e., master-worker and parallel pipeline), allowing for the utilization of common programming paradigms without the need to implement them from scratch. Developers can also add such functionalities to other programming languages because of the clear specifications laid out by the interface itself. After implementing the common interface for the languages just

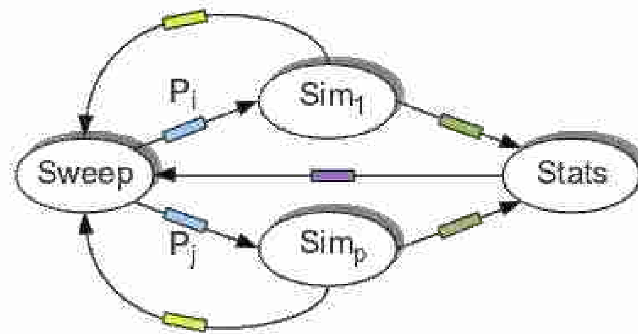

Figure 1.1. Parametric Simulation Sweep - A hypothetical heterogeneous workflow. A sweep program launches multiple instances of a simulation with varying input data (represented by blue bars). When each simulation (Sim_i for $i = 1..p$) completes, they send the output data back to the sweep application (yellow bars). A subset of the output data (green bars) are then sent to a centralized statistics (or visualization) process which computes user-specified analysis on the fly. In real-time, the user can monitor the progress of the simulation system and make decisions about the state of the execution.

mentioned, it is clear that adding new languages to the interface can be done relatively easily by following the same algorithms. The interface specification and requirements shape the implementation; and, as long as some sort of network message passing is possible, the interface can feasibly be created for any programming language. In this paper, we consider the cases of C++, Matlab, Python, and Java because of their ubiquity in the field of scientific parallel programming. PRESTO currently supports an optimized master-worker implementation in these languages, and a ring pipeline in Matlab.

CHAPTER II

DESIGN AND ARCHITECTURE

This chapter presents the high level design of PRESTO by discussing the three most important software design considerations for the target domain: performance, scalability, and usability. Other considerations are important, such as compatibility, extensibility, maintainability, modularity, reliability, reusability, robustness, which are considered in the final section of this chapter.

2.1. Design Considerations

The high-level design of the PRESTO environment follows the diagram in Figure 2.1. The diagram represents the PRESTO environment, which supports a collection of heterogeneous computational engines that work together to accomplish a goal. In this diagram, the top components (labelled Matlab, Java, R, Python, and Fortran/C/C++) are referred to as the *computational engines* in this thesis. These components can be thought of as adapters to the common interface provided by PRESTO. In a typical use-case, the computational engines are components of a user application, such as a scientific workflow, that are a part of a global task, analysis, and/or execution to be done. In PRESTO, the job performed by the computational engines is not specific - it can range from a simple kernel of execution to a centralized visualization process.

The bottom rectangular portion of Figure 2.1. represents the internal implementation of the PRESTO framework. The framework can be abstracted into two primary software layers: the Python middleware and the MPI implementation layer. These two layers are considered from the bottom up.

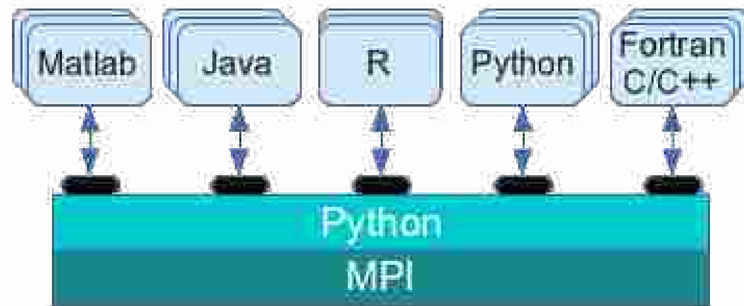

Figure 2.1. A representation of the high-level software design for PRESTO. A Python middleware manages MPI computations (in the form of common parallel computational models) to support the requirements of heterogeneous environments, potentially running a collection of programs written in different languages.

At the heart of the system is an MPI implementation which handles the execution of common parallel computational models, such as master-worker. As mentioned in Section 1.1., MPI is a key component in a large number of applications in the realm of scientific parallel computing. Because PRESTO implements its parallel operations on top of MPI, many doors are opened to support both new and legacy applications that utilize this message passing paradigm. This property also readily enables further development and optimization of new and existing components of PRESTO.

The software layer that connects the parallel MPI implementation with an interface supporting heterogeneous computational environments is a Python middleware infrastructure. This design decision is employed for several reasons. Firstly, a middleware in an interpreted and widely used programming language such as Python makes it easier for software developers to extend the functionalities of PRESTO, both in terms of supporting new language interfaces and in terms of supporting new parallel models of computation. The idea is also that this layer is a *programmable middleware* which allows for customization of currently supported adapters and computational models, such as in specific load balancing techniques, data passing, and message protocols. Furthermore,

writing this layer of the framework in Python is a decision that benefits development productivity, because it can be far easier to write certain snippets with the modules provided by a high level language such as Python. However, performance is a crucial consideration for the target applications of this framework, so one can consider this Python layer as a prototype for something written in a more efficient language, such as C/C++. To this end, a C++ implementation of a subset of the current functionality of the Python-based PRESTO environment currently exists, and has been tested for Matlab computational engine environments. While there is nothing special about the Python middleware capabilities, the performance of this middleware is surprisingly on par with the current C implementation. This is because network latency, application kernel execution time, and data buffering are the unequivocal performance bottlenecks in this context. For this reason, the Python middleware may be considered something more than simply a prototype.

Using Python as a light wrapper around MPI is by no means a new idea [6, 5, 14]. Because of Python's status as a powerful, mature and easy to learn language, it is not surprising that a few different wrappers exist that emulate the C++ MPI-2 bindings (which no longer exist in the new MPI-3 specification), such as `Mpi4py`, `MYMPI`, and `pyMPI` [15]. For this project, `Mpi4py` is the chosen tool because of its maturity and relatively strong documentation. `Mpi4py` supports point-to-point (sends, receives) and collective (broadcasts, scatters, gathers) communications of any picklable Python object [6] making it far easier to work with the standard C/Fortran implementations which perform best on regular arrays of primitive types (although more complicated structures are technically supported in the derived datatypes interface). `Mpi4py` also supports optimized communications of Python objects of single-segment buffers represented by NumPy arrays, which can greatly increase efficiency and performance.

There are, however, important drawbacks to consider when using such a convenient wrapper such as `Mpi4py`. For instance, as described by Dalcín et al. [6], “The inherently asynchronous nature of nonblocking communications currently imposes some restrictions in what can be communicated using MPI for Python. Communication of [regular] memory buffers is fully supported. However, communication of general Python objects using serialization, is possible but not transparent since objects must be explicitly serialized at sending processes, while receiving processes must first provide a memory buffer large enough to hold the incoming message and next recover the original object." In other words, nonblocking communication, which is vital for achieving optimal performance, is not supported by `Mpi4py` when using standard Python objects because of unpredictable receiving buffer properties. As we shall see in the PRESTO implementation description, this means that certain portions of the middleware infrastructure are inherently blocking, when in fact they need not be. Although this has a relatively minor performance toll, it is something to keep in mind about the drawbacks of abiding to an easy to use and easy to program middleware, such as PRESTO entails.

Design considerations for any software project must consider issues of compatibility, extensibility, maintainability, modularity, reliability, reusability, robustness, security, usability, performance and scalability. For PRESTO, the latter three properties are of crucial importance because of the expected application targets of deployment. As such, this chapter discusses those considerations in detail in the following sections. However, the other considerations are also important and deserve discussion in the context of PRESTO. They are explored in the final section of this chapter.

2.2. Performance

Performance is of primary concern in the design goals of PRESTO, simply because the target audience intends to improve the performance of their applications by parallelizing code. It is important to balance the trade-off between the overhead of managing parallel task scheduling and migration operations with the benefit of doing computations locally. Certain restrictions may need to be placed on the interface in order to abide by a reasonable balance between overhead and absolute performance. Fortunately, Matlab's Distributed Computing Server provides a convenient standard of comparison for certain parallel operations in PRESTO for low numbers of processes. This will be discussed in the performance evaluation section in chapter V.

2.3. Scalability

Scalability is perhaps the most important design consideration, because the target audience for users of the PRESTO environment desire the ability to exploit available parallel hardware without having to port their code to another language. If the time taken to execute the application is no faster when scaling out to higher numbers of processor cores, then all is lost in terms of our design goals.

One of the primary goals of PRESTO, as mentioned in section 1.2., is to provide an interface that supports common computational models with comparable performance and scalability that an optimized library might provide. While absolute performance, as discussed in section 2.2., is also very important, without satisfactory scalability, there is a serious problem with the applicability of PRESTO. This is why weak and strong scaling measurements are carefully considered in chapter V.

In order to accomplish sufficient scalability, the architecture of PRESTO is planned out in accordance with the diagram of Figure 2.2. In this diagram we see that collections

of computational engines (CEs) are interfaced with the PRESTO middleware via hubs. Within each hub of CEs, a collection of tasks are assigned to CEs autonomously (i.e., without interaction from the PRESTO middleware). This architecture drastically improves the scalability of the framework, especially in the master/worker paradigm. More of this is discussed in chapter V.

2.4. Usability

Because most parallel applications (or components thereof) can be reduced to a common parallel computational model, PRESTO's interface to apply such models should be simple and easy to use. For instance, the most common parallel model, master/worker, consists of nothing more than some task(s) (usually just one) that needs to be executed on different sets input. It is easy to imagine something like a function call which simplifies the requirements of this model, and presents a clean interface to the user for applying master/worker computations. This is also true for most, if not all, other computational models, so usability in PRESTO is an extremely important design goal. Chapter III is dedicated to the design, development, and presentation of PRESTO's interface.

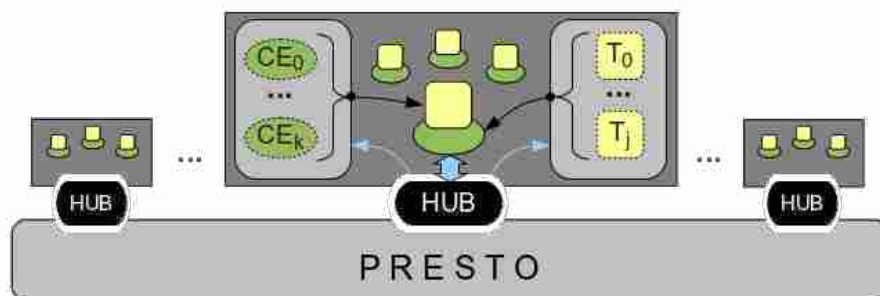

Figure 2.2. PRESTO's scalable operations architecture

2.5. Other Considerations

Besides the most important design goals mentioned, the notions of extensibility, maintainability, modularity, reliability, reusability, and robustness are also very important. However, because of the scale of this project (and the fact that the development team is only one person), these goals are considered secondary. Though they will become extremely important for any version of PRESTO beyond a prototype, for the scope of this paper these goals are only mentioned, not discussed.

Implementation of a parallel Matlab has a rich and diverse history [4, 7, 12, 17, 16, 26, 3, 11, 23, 13, 27]. This is perhaps because of the unreasonable cost of academic licenses for Matlab's Distributed Computing Server, which allows for parallel loop constructs to be computed on remote Matlab processes (licenses are assigned on a per process basis). While this fact is important to consider, especially when comparing absolute performance of PRESTO with DCS (as is done in chapter V), the goals of this framework are orthogonal and more ambitious than simply finding a replacement for DCS. The goals are instead to fill the void for an interface to common parallel computational models (which the DCS only supports parallel for loops) in a high-performing, scalable, and usable way.

CHAPTER III

INTERFACE

This chapter presents a brief overview of the interface provided by PRESTO, both in terms of the available computational models (master/worker and ring pipeline), and in terms of the supported languages (Matlab, Python, Java, C++).

3.1. Launching the PRESTO Environment

The launching of PRESTO currently supports integration with the Portable Batch System (PBS), a popular job scheduling system found on Linux clusters. The launch is a simple command line interface:

```
presto [-h] [-n N] [-engine ENGINE] [-app APPNAME]
```

where currently `ENGINE` can be either of `"matlab"`, `"java"`, `"python"`, or `"cpp"`. The `APPNAME` parameter is required for all languages except Matlab and is the name of the application program/binary to execute. If `APPNAME` is not included when `ENGINE` is `"matlab"`, then the user is dropped into an interactive environment which has `N` PRESTO worker processes at his/her disposal. As a side note, it is definitely possible for such an interactive environment to be provided in Python, however that is currently not supported.

3.2. Computational Models

PRESTO currently supports the master/worker parallel computational model in Matlab, Python, Java, and C++. Figure 3.1. shows the interface for Matlab. The entry point to PRESTO in Matlab is essentially a function call denoted by `master_split` which is intended to replace the parallel “forloops" provided by Matlab’s Distributed Computing Server (described in more detail in chapter V). The arguments to this call are 1) the name

of a Matlab function to be executed by the workers, 2) a list of Matlab object arrays that are to be split across the workers, and 3) a list of Matlab objects that are to be sent to each worker. The way the 2nd list of arguments works is specific and deserves a description. The PRESTO master process loops over this list of object arrays and removes the ith item from each array. This collection of n items (where n is the number of object arrays and incidentally, the number of distinct inputs to the worker function) consists of the “split" data needed for a single task of execution. The shared data, on the other hand, are sent in their entirety to each worker. This is a clean interface that replaces the parfor loop (also shown in Figure 3.1.) and provides for background optimizations by explicitly defining shared data, much like certain OpenMP pragmas. For example, the shared data is sent to workers via an efficient call to `MPI_Broadcast`.

A parallel task-pipeline is also supported for Matlab. A diagram of this model is shown in Figure 3.2., and the interface itself is shown in Figure 3.3.

3.3. Language Interfaces

The master/worker computational model is supported in several languages, and have almost identical semantics. The Java and Python interfaces for example, are quite similar (with the exception of differences in language syntax) and are shown in Figure 3.4. The C++ interface is also similar, but has crucial differences. Specifically, “`function_name`" is only an ID in C++ (not a function name) because there is no reflection as there is in Matlab, Python, and Java. There is also the requirement to link C++ programs with the PRESTO library, and users must specify how to serialize/de-serialize objects (more in section 4.1.2).

Note the similarity of the interfaces is not explicitly a design goal for PRESTO, but it does make reference to the fact that such an interface to common parallel computational models is a powerful and much-needed tool for the parallel computing community.

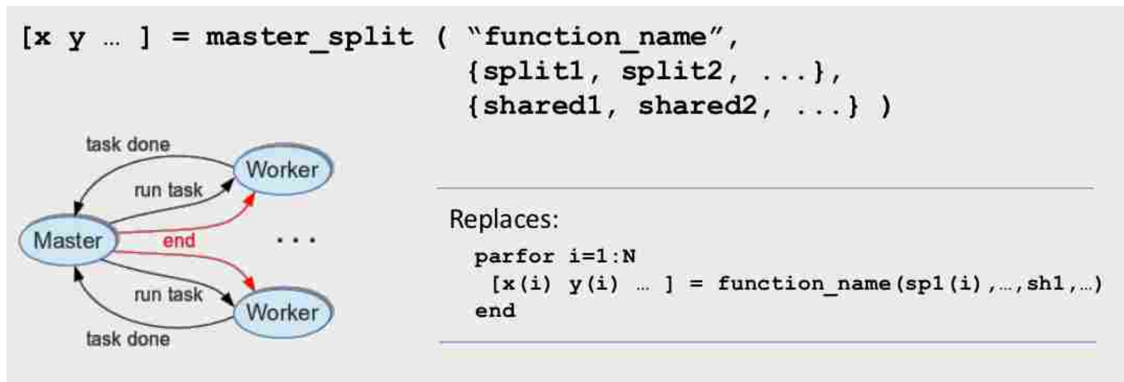

Figure 3.1. PRESTO's master/worker interface in Matlab: The first argument is a string reference to the worker task function, the second argument is a cell-list of general Matlab objects to be "split" across the pool of workers, and the third argument is a cell-list of shared objects that each worker requires to conduct its computation.

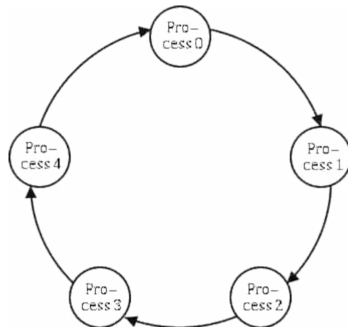

Figure 3.2. Ring: A diagram of a parallel task-pipeline. Each task can be computed independently of the others, but must be applied in a particular order.

```
[x y … ] = task_ring ( {"function1", "function2", ...},
                       {input1, input2, ...} )
```

Figure 3.3. Ring interface: PRESTO's Matlab interface for computing parallel task-pipelines as represented in Figure 3.2. The first argument is the cell-list of function references, and the second input is the cell-list of inputs for each respective task.

```
YourInClass *inputObjectArray = new YourInClass[numTasks];
YourOutClass *outputObjectArray = new YourOutClass[numTasks];
YourSharedClass *sharedObject = new YourSharedClass();

// Initialize the inputs

Master M = new Master();
M.Launch();
outputObjectArray =  M.master_split ( "function_name",
                                      inputObjectArray,
                                      sharedObject      );
M.Destroy();

// Carry on…
```

Figure 3.4. PRESTO's java interface for the master/worker computational model. Similar to the Matlab case in Figure 3.1., input data is separated into lists of "split" objects and "shared" objects. Unlike in the Matlab case, users must allocate a list of output objects and must launch the Master instance separately. Respectively, this is because 1) specifying a single output class makes for a much cleaner back-end and 2) Java is not an interpreted language like Matlab, and thus the PRESTO runtime needs to be launched when the Java application starts.

CHAPTER IV

APPLICATIONS

Even in its infancy, PRESTO has been used for several real-world applications. This chapter looks closely at the usage and deployment for two specific real-world applications in geological physics and neurological informatics. We begin by introducing some background for each of the research projects, considering the parallel requirements, and finally describing specifics of the PRESTO implementation, setup, and deployment for each of the application environments.

4.1. Stingray: A Seismic Geophysics Application

The goal of the Stingray application is to investigate the 3-D structure of the Earth's crust and topmost mantle beneath a volcanically active seabed. In particular, many questions regarding oceanic magma chambers such as where they are, how large they are, and what forms them, remain unanswered. Stingray evolved out of a need for seismic analysis of a dataset that could help answer such questions. This data comes from a 2008 seismic experiment that intended to investigate the 3D structure of the crust and topmost mantle beneath the Juan de Fuca ridge (see Figure 4.1.). The scientific objectives were to: 1) determine if the segmentation and intensity of the magma-hydrothermal systems at the ridge are related to magma supply or to the magma plumbing between the mantle and crust, and 2) constrain the thermal and magmatic structure underlying the ridge's hydrothermal system in order to understand the patterns of energy transfer [24]. The experiment involved the use of Ocean Bottom Seismometers to record seismic energy from an array of airguns deployed on the ocean surface via a large ship whose purpose is to conduct seismic research (the *R/V Marcus G. Langseth*).

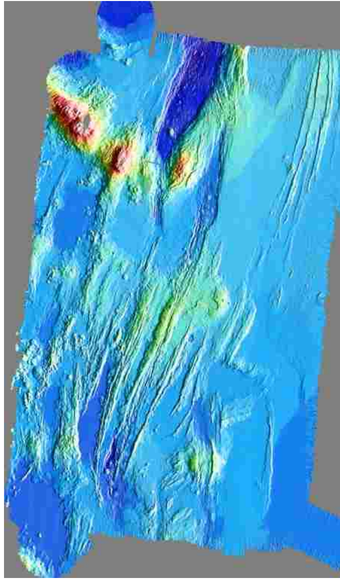

Figure 4.1. Stingray dataset: A visualization of the ocean bottom topology for a region examined in the volcanically active Juan de Fuca ridge.

4.1.1 Stingray's Computations

In essence, Stingray is a computational application that performs minimal travel time ray tracing in order to determine the most likely paths for seismic waves. It involves solving a large sparse system ($Ax = b$) using Matlab tools for handling sparse matrices and computing least squares. It is an iterative algorithm that first computes a forward problem by performing ray tracing in a loop over the Ocean Bottom Seismometers stations. Then, a similar loop optionally occurs over specific events. Each of these loops are independent of each other, and have the potential to execute in parallel. Finally, with this collection of forward solutions, an inverse problem is solved by performing a least squares solution of the linear system using the current model parameters. After this point, if the system has converged to an acceptable tolerance value (user-defined), then the computation performs another overall iteration until convergence is achieved.

Stingray is an excellent use-case for the efficient use of Matlab. While some criticize the language for its bulkiness, poor string manipulation utilities, and relatively

poor parallel performance, Stingray uses the best parts of Matlab to it's advantage. For instance, the computational forward problems are run by using Matlab's *mex-file* interface, which allows users to call their own code from high-performance, low level languages (C/C++/Fortran) as if they were built-in functions for Matlab. Because each forward problem is independent of each other and its Fortran code (interfaced via `mex`) is well-established and optimized, the Stingray/Matlab environment has the potential to perform many high-performing computations in parallel, all the while keeping the application self-contained in an environment suitable for statistical analysis, visualization, and object manipulation. The Stingray data structures, which include input parameters and output results from the low-level *mex* executions, are extensible and easy to work with.

4.1.2 Stingray and PRESTO Integration

Because of the design goals and implementation which prioritize usability of the interface in PRESTO, integration with Stingray was quite simple. For most applications that utilize the Matlab's Parallel Compute Toolbox (PCT), integration is not difficult because the parallel constructs (in the form of generic PCT `parfor` loops) are already in place, and they are the insertion points for PRESTO, as described in chapter III. For Stingray, both the loops over Ocean Bottom Seismometer "stations" and over "events" are independent and already written to use `parfor` loops. Therefore, in the simplest case we trivially replace the two `parfor` loops with calls to PRESTO's `master_split` function.

In a more sophisticated integration, a PRESTO `master_split` is applied to a kernel that wraps the original forward solution `forloop`. The advantage in this case is that individual Matlab worker instances are able to use on-node parallelism via the PCT. Therefore, if the PRESTO environment were to launch one Matlab computational engine process across a cluster, then this environment would have much greater potential for

exploiting parallel resources. This situation has a two-layer hierarchy of parallelism: one at the MPI level and one below at the Matlab PCT node-level. We will see in chapter V just how important this hybrid parallelism can be for large-scale performance.

4.2. Head-Modeling: A Neuroinformatics Application for Conductivity Analysis

The second application that can make use of the interface provided by PRESTO is the Head-Modeling simulation code out of the University of Oregon's NeuroInformatics Center (NIC). The overall goal of this research is to better understand human brain dynamics and conditions by means of non-invasive methods that allow for high-temporal and spatial fidelity. The current state of affairs in measuring brain activity relies heavily on technologies such as function Magnetic Resonance Imaging (fMRI), which have good 3D spatial resolution ($1mm^3$), but poor temporal resolution ($\leq$ 0.5 seconds). Electromagnetic measures (such as EEG or magnetoencephalography (MEG)) provide high temporal resolution in the order of neural activities ($\leq$ 1 msec), but their spatial resolution lacks localization accuracy. The crucial problem of enabling high spatial and temporal resolution can hypothetically be addressed by incorporating a priori knowledge and assumptions about the sources of electrical current by imposing accurate constraints on the problem [20].

The primary use of this code is as a Finite Difference Method (FDM) simulation of the electrical activity inside the human head during an Electrical Impedance Tomography (EIT) experiment. One primary goal of this bridging between electrodynamic theory and electroencephalography (EEG) experiments is to accurately determine sets of subject-specific conductivities for the individual head tissues. This is important because of the well-known observation that head-tissue conductivities vary from person to person, and also vary within a particular person across developmental age.

4.3. Head-Modeling's Computations

The Head-Modeling code consists of several modules of computation (conductivity, Lead Field Matrix (LFM), and forward template generation) that all revolve around the forward solution kernel. In the execution of a single forward solution, input consists of a single head model (usually extracted from an individual's CT/MRI images), the input current source and strength, and parameters regarding the solver algorithms. Using an FDM solver to compute solutions to the Poisson equation, the forward solution results consist of the potential field inside the entire head volume. This information can then be used to compare with the EIT experimental measurements. The idea is to match the theoretical potential values at EEG sensor locations with the same sensor measurements as found in the experiment. If the values are identical, then we have increased confidence that the head model geometry and set of segmented tissue conductivities are accurate, and can be used for source localization and further study of human brain dynamics.

In this application, the determination of segmented conductivity values involves exploring a large space of forward solutions and optimizing the model to match experiment. The overall process is represented in Figure 4.2., where an individual's head is processed using an inverse problem technique. In this approach, many forward solutions for an individual head model are computed - one for each set of conductivity values. For instance, if there are only three tissues in the segmented head model, skull, scalp, and brain, then each forward solution consists of trial values for each of those conductivities. By comparing the measured theoretical potential values at the sensors with actual measurements made during the EIT experiment (using an l^2-norm function), we can reasonably determine any given individuals appropriate conductivities [25].

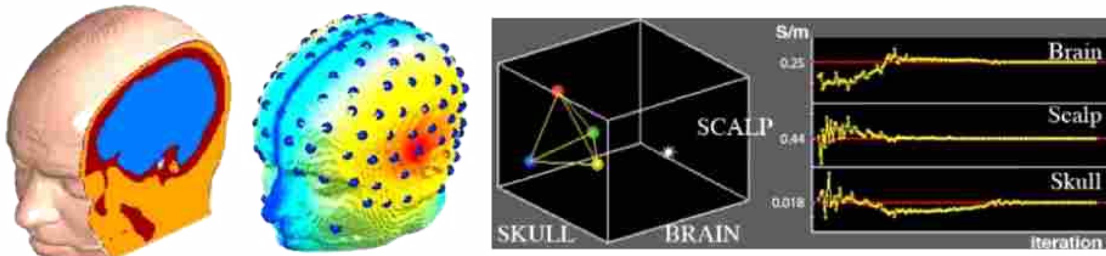

Figure 4.2. The Head-Modeling conductivity inverse using simplex (Nelder-Mead) optimization [25]. The blue spheres on the head model show the location of the EEG sensors. Using a EIT injection current as input to the simulation, different combinations of skull, scalp, and brain tissues are used to compute a forward solution. The optimization is complete when all three tissues converge to a local minimum.

4.4. Head-Modeling and PRESTO Integration

The Head-Modeling code provides a real-world use-case for the development of both a Python and a C++ master/worker interfaces in PRESTO. This is because the Head-Modeling tools make use of a high-level Python wrapper for launching forward solutions, conductivity inverse problems and LFM generation. In the simplest usage of PRESTO, we wrote a Python script that launched a large set of forward solutions, each with a different set of inputs (specified by a file with the *.hm* extension). This application environment utilizes the Python interface which is similar to the Java interface discussed in chapter III. While this same approach is possible when executing conductivity inverse problems, it only allows for parallelization based on different current-injection pairs in the EIT experiment, *not* parallelism based on independent forward solutions.

In order to explore the potential for massive parallelization of both the conductivity and the LFM modules, the PRESTO C++ interface implementation was developed hand-in-hand with the Head-Modeling code. Certain complications arose due to the fact that the C++ standard library does not provide means to serialize generic objects. This is a considerable problem because certain C++ objects in the Head-Modeling application either

need to be sent across the network to workers, or somehow constructed on the worker-side based on more simple parameters. Because the former possibility is more elegant, portable, easy-to-use, and likely applicable to a number of other C++ applications, that approach was implemented in PRESTO and integrated with the Head-Modeling code. In order to accomplish C++ object serialization easily and effectively, the Boost C++ library was used [21]. With this approach, any collection of C++ objects can potentially be supplied as input to C++ computational engines being run on the worker-side.

A heterogeneous Head-Modeling workflow environment was also deployed using the tools provided by the PRESTO middleware. In this environment, MPI process #1 was in charge of collecting forward solution results from each of the simulations being executed by the workers. This process was simply an R plotting program that received input and placed them in a scatter plot throughout the overall application's execution. This accomplishes the design goal displayed in Figure 1.1. where the *sweep* program is the Head-Modeling conductivity code, each *Sim_i* program, and the *Stats* program is the R visualization process. This heterogeneous compute environment has the major advantage of being able to analyze a Head-Modeling workflow in real time as it executes. This could forseeably save much power and effort, for instance if a simulation needs to be re-run based on bad input, which can clearly be displayed in the online visualization. Before the creation of this PRESTO-integrated environment, NIC scientists occasionally may need to wait for the entire execution to complete before seeing useless results based on user-error or data incompatibilities. The heterogeneity of this Head-Modeling/PRESTO environment is represented in Figure 4.3.

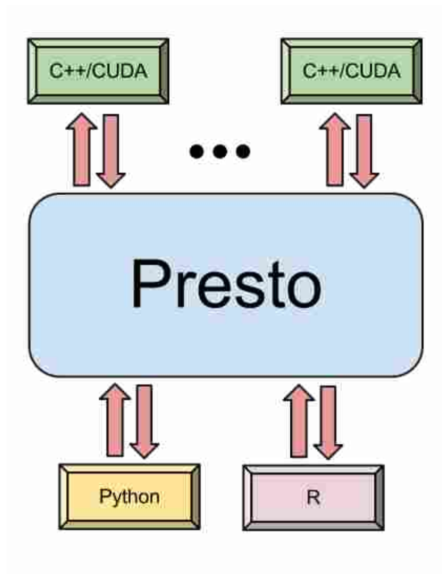

Figure 4.3. The heterogeneous Head-Modeling/PRESTO forward computation environment.

CHAPTER V

PERFORMANCE

This chapter presents performance measurements of PRESTO in terms of relative improvements during development, and more importantly in terms of an absolute comparison with Matlab's Distributed Computing Server (DCS). The DCS provides an excellent standard of comparison for PRESTO's performance because of its establishment in the parallel computing community, its strong overlap in functionality, and the sheer *cost* of DCS, even for academic licenses (approximately $43,000 for a 256 workers). The experiments in this section were conducted on the University of Oregon ACISS cluster, with specifications shown in Table 5.1.

5.1. Relative/Absolute Performance

The implementation of PRESTO followed an iterative development cycle for which a completely functional system was initially deployed, then further enhanced during each iteration. For the first few iterations of a relatively naive implementation, there were serious performance problems that deserve comment. Specifically, the issue that had the most detriment on scalability in the master/worker module was the fact that PRESTO initially consisted of a separate buffer for each task, where the master iterated over the full set of

Aciss-fatnodes : 16 compute nodes (4x 2.27GHz 8-core Intel X7560 CPUs w/ 384GB DDR3 RAM), 512 total cores
Aciss-generic : 128 compute nodes (2x 2.67GHz 6-core Intel X5650 w/ 72GB DDR3 memory), 1536 total cores
Aciss-gpunodes : 52 compute nodes (2x 2.67GHz 6-core Intel X5650 w/ 72GB DDR3 memory), 624 total cores; 3 NVIDIA Telsa M2070 GPU, 156 GPUs total

TABLE 5.1. ACISS cluster specifications

buffers in order to distribute the data to the workers. While this model worked quite well with applications such as Stingray, for which the time to execute a single task (in that case, a forward ray tracing solution) greatly outweighs the overhead of handling a relatively small number of tasks (for Stingray, a maximum of about 5,000 “events"). If some other application were to require the solution to millions of tasks that each last on the order of milliseconds, then the initial PRESTO implementation broke down, and suffered extreme overheads and overall performance degradation.

In order to emulate this situation which displays poor scalability, a synthetic application was constructed that performs a computation typically found in many physical applications, such as computational fluid dynamics. The Matlab code for this simple kernel is quite easy to write and fits on one line:

```
y = sqrt(1+x*x) * besselj(.25, x) + exp(1/(x+1))*ellipke(1/(1+log(x)^2))
```

which represents the following equation:

$$y = \sqrt{1+x^2} * \left(\frac{x}{2}\right)^{1/4} \sum_{k=0}^{\infty} \frac{(-x^2/4)^k}{k!\Gamma(1/4+k+1)} + e^{\left(\frac{1}{1+x}\right)} * \int_0^1 [(1-t^2)\left(1 - \frac{t^2}{1+\log^2 x}\right)]^{1/2} dt \tag{5.1}$$

and can be evaluated across the domain space for many different values of x in the range $[0,1]$. The execution of this computation takes on the order of several milliseconds to complete. So when the overhead of packaging a single task and sending it across the network to a worker computational engine takes relatively more time (dozens or hundreds of milliseconds), then this small kernel becomes a serious performance problem.

The solution to this problem is simple in theory yet relatively sophisticated in implementation: course-grain the collection of tasks into bundles to be distributed to workers. For example, in the synthetic equation above, it makes sense to buffer several

values of x into a single task bundle to be sent to the worker. Implementing this required reconfiguring the system so that workers were able to respond to any number of tasks, where the task input data is any Matlab object, not necessarily a simple collection of x values. This also required making few changes to the task delivery protocol discussed in chapter II, and substantial changes to how input and output data are handled by PRESTO behind the scenes. Overall, this change showed drastic improvements to system performance, and allows for absolute performance comparisons to Matlab's Distributed Compute Server, as discussed in the following sections.

5.2. Weak Scaling

This section shows performance measurements of the synthetic physics kernel presented in equation 5.1 in terms of weak scaling. Weak scaling is a particularly useful metric for performance ability because it shows how efficiently an application scales in terms of increasing input data size. For instance, if a dataset of size s is run across p processors, then a weak scaling measurement would consider a dataset size of $2s$ across $2p$ processors, and so on.

The weak scaling of PRESTO is shown in Figure 5.1. for the kernel from equation 5.1. This graph shows two comparisons, one with a small input size (the red/green lines) and one for a much larger input size (the blue/purple lines). Specifically, the red/green lines correspond to a collection of computations of equation 5.1 with 1,000 tasks across 12 worker processes (with each process having an exclusively dedicated core). Because this is a weak scaling plot, the data points for 24 workers and 48 workers compute 2,000 and 4,000 tasks, respectively. The larger data set shown in blue/purple is for an experiment in which 1 million tasks were computed across 12 worker processes, and the data size changes similarly across the horizontal axis. The diagonal black line is the optimal speedup

that is possible when there is exactly zero overhead (which in a master/worker model, is essentially impossible to reach).

Because the University of Oregon only has a DCS license that goes up to 32 Matlab worker processes, the data in Figure 5.1. can only go so far. One significant advantage of PRESTO is that users are not limited on the number of multiple Matlab instances that can be executed using the university's network license. In this light, Figure 5.2. shows the weak scaling of PRESTO out to 384 workers. While we see significant degradation in speedup compared to the optimal, we still see an advantage to scaling to a large number of processors. Unfortunately we cannot compare absolute performance to the DCS for such large numbers of workers at this time, but it is not unrealistic to expect that performance will be comparable because of the high-overhead, fine granularity of the problem, and the necessarily centralized nature of the solution.

5.3. Strong Scaling

While the weak scaling discussed in the previous section is a great metric for measuring the efficiency of PRESTO itself, in most real-world applications, strong scaling matters most. This is because parallel resources are generally widely available (especially in academia and in scientific research communities). Therefore, it matters greatly how well you can speed up a particular application given a certain amount of hardware resources. This is what strong scaling intends to measure by choosing a fixed problem size and comparing the time of application execution across a number of different processor cores. This section considers the synthetic physics kernel from equation 5.1 and the Stingray geophysics application discussed in section 4.1.

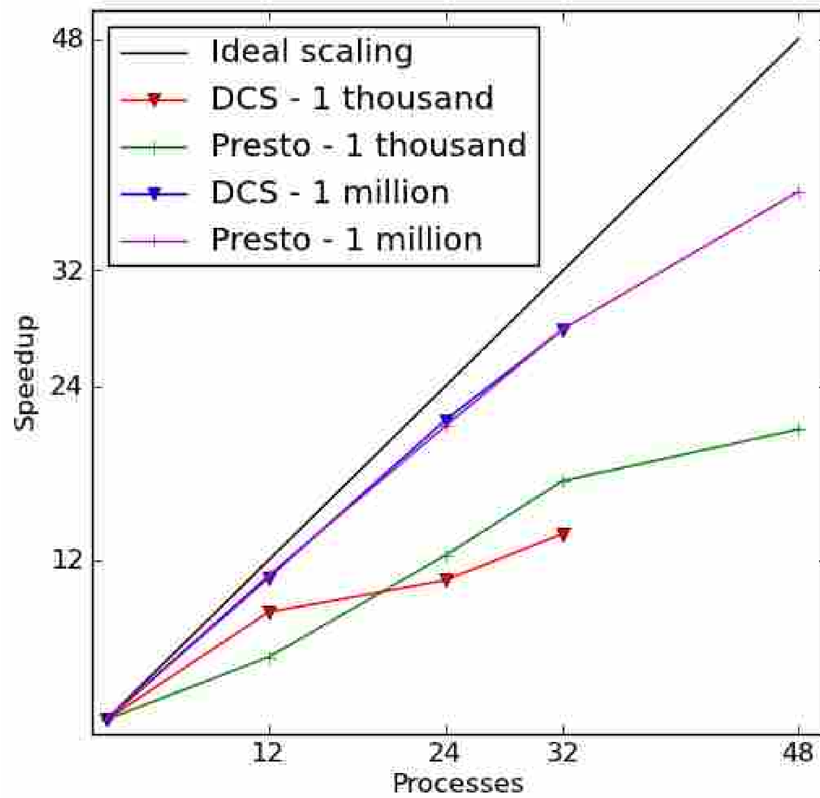

Figure 5.1. Weak scaling comparison between PRESTO and Matlab's DCS. For the experiment shown in red/green, 1,000 tasks from equation 5.1 were computed across 12 workers (each with a dedicated processor core). Each other data point along those lines corresponds to a data size proportional to the number of cores (24 cores $\rightarrow$ 2,000 tasks, 48 cores $\rightarrow$ 4,000 tasks, and so forth). An experiment with 1 million tasks was also conducted and is shown in blue/purple.

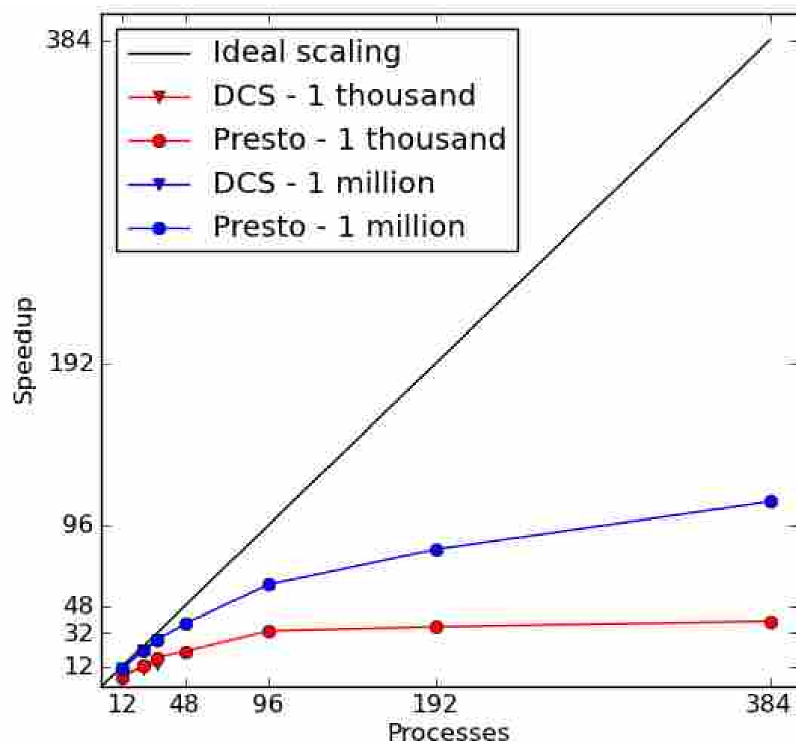

Figure 5.2. Weak scaling for PRESTO beyond 256 workers. In this plot, we scale out past the capabilities of the DCS on the universities current license. While there is a scaling degradation compared to optimal speedup, this might also occur in DCS at these scales because of the nature of the problem and the inherent centralization of both parallel runtimes.

5.3.1 Synthetic Application

The strong scaling for the synthetic application represented by equation 5.1 is shown in Figure 5.3. In this plot, we fix the problem size to 1 million values of x, and vary the number of worker process from 1 to 48 (1-4 nodes with a max of 12 processes per node) where each process has a dedicated core (we limit the number of total processes in PRESTO to be able to compare to the 32 worker maximum in the DCS).

Figure 5.3. shows some interesting characteristics. For instance, we notice that PRESTO always outperforms the DCS, but the margin is most prevalent for lower numbers of processes per node. Because the DCS is a proprietary piece of software, we can only speculate as to the cause of this phenomenon. One possible explanation is that the DCS may be using heavier-weighted Matlab processes to do on-node parallel computations, whereas PRESTO is making use of the Parallel Compute Toolbox, as explained in chapter IV. Another interesting observation is that the performance appears to converge to the same value as you scale to a larger number of nodes/processes. Again, we can only speculate as to why, but it is a good sign that PRESTO keeps pace with such a well established parallel computation as the DCS, especially in this synthetic application which emphasizes the detrimental effects of overhead when working with a large number of small tasks.

5.3.2 Stingray

Finally, we consider the strong scaling of the real-world Stingray application, which is described in detail in chapter IV. This application has heavy computational requirements, judging by the execution time taken per task (approximately 20 minutes for stations and 10 seconds for events). In a typical high-fidelity dataset, Stingray computes on about 30 stations and 4,000 events. The performance measurements shown in Figure 5.4. are from such an execution for both the DCS and PRESTO. Because the DCS is limited in our case

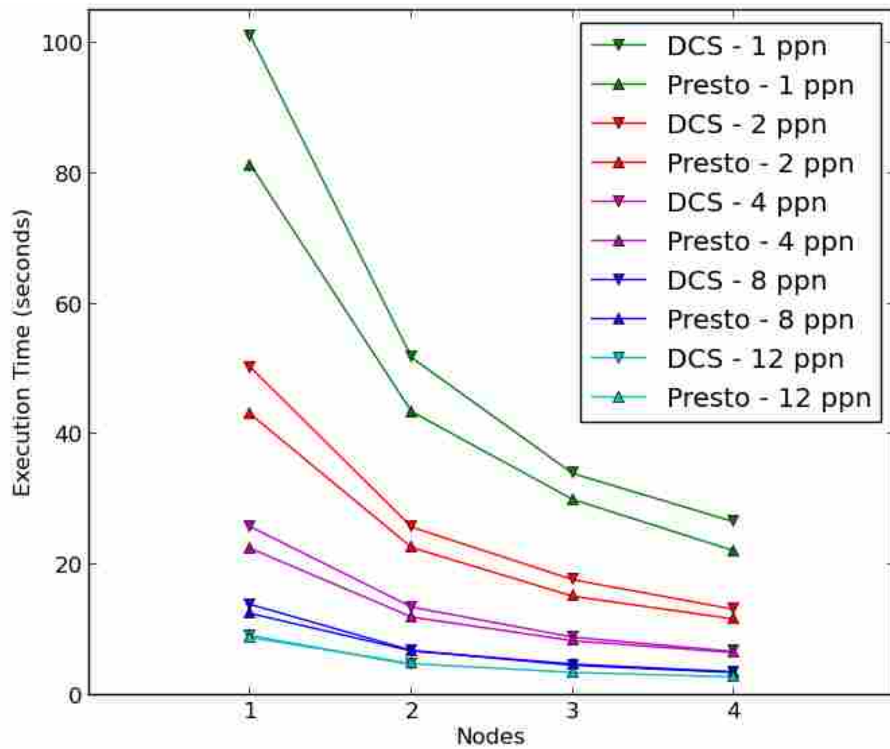

Figure 5.3. Strong scaling comparison between PRESTO and the DCS. Each color is for a different number of processes per node (ppn). PRESTO always outperforms the DCS, but less-so at higher numbers of overall processes and ppn.

to 32 workers, it only has 1 data point in the graph. PRESTO, on the other hand, can scale out to a much larger number of processors at a far lesser cost ($0).

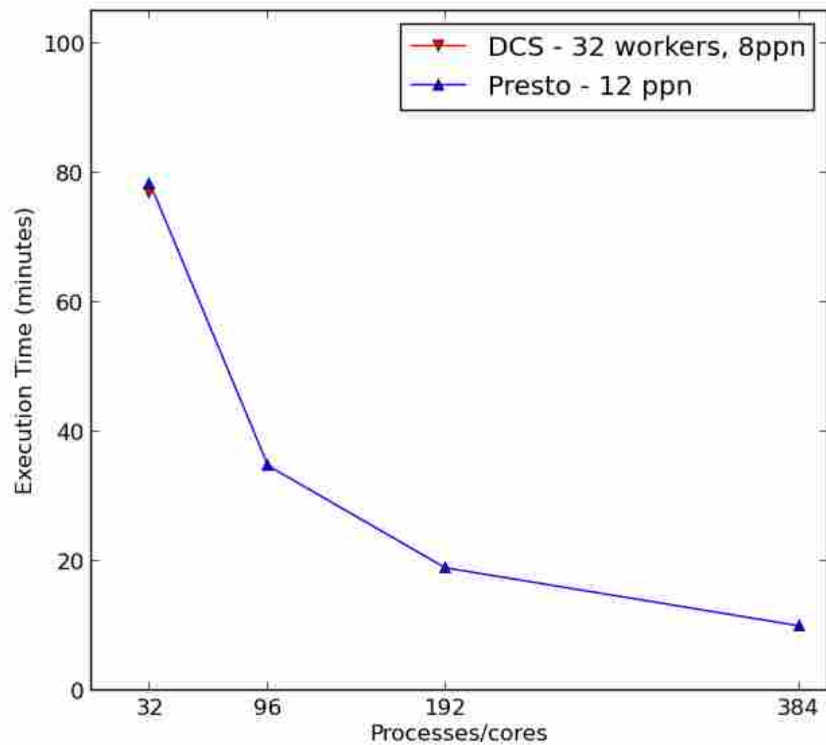

Figure 5.4. Strong scaling of the Stingray geophysical application. The university's license is limited to 32 total workers, but PRESTO can scale out to 384 (and more). Note that the DCS measurement is for 4 nodes at 8 processes per node which was determined to be the best performing resource allocation that utilizes all 32 workers.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

This chapter brings the paper to a close by briefly discussing further improvements to PRESTO (future work), and summarizing PRESTO's accomplishments and contributions (conclusion).

6.1. Future Work

PRESTO is a work-in-progress, and as such, much is left to be done to improve the system. For instance, only the master/worker computational model is fully supported across multiple languages. While this model is extremely common and a very formidable first step towards a multi-model framework, there are a plethora of other modules that can be added to PRESTO. The ring pipeline model, for example, has a preliminary implementation in Matlab, but it is not load balanced as is the master/worker, and the interface has not yet been created for other languages. The framework would be more complete if other models were added, such as MapReduce, Ghost-Cell computations, and general directed acyclic graph (DAG) scheduling.

One major step towards such non-centralized models has been implemented. Figure 6.1. shows a diagram of how tasks can directly be sent between computational engines. Having this ability would enable some very important computational models, as found in grided ghost-cell applications. Such an application is represented by the graph in Figure 6.2., which is similar to the master/worker model, with some critical improvements relevant to certain applications. In particular, consider an application whose domain is a 2D grid. Some cells of the grid need to communicate with its neighbors in order to perform a computation (this is a common model found, for example, in weather

and traffic simulations). These applications are decomposed into parallel sub-grids where communication occurs between sub-grids by accessing data in the adjacent ghost cells. PRESTO supports the master/worker and task-to-task data migration, so it would be quite easy to extend such a ghost-cell model.

Because of its generality, the holy-grail of task scheduling models would be an easy-to-use DAG scheduler. Support for this model has been considered and partially implemented in PRESTO, but further work needs to be done to support a usable interface such as the master/worker model presented in this thesis. While implementing the internal communication and progress engine in an efficient way is one matter, data representation is also an important hurdle. There is much literature and related work that could provide tools and support for DAG scheduling [1, 8, 18, 19]. The difficulty in data and graph representation between different programming languages is a less common topic in the field. Some work in data representation within domain specific languages [13] might provide valuable information for working through this crucial part of the DAG scheduling problem.

Cleanup of the plug-and-play interface to PRESTO for supporting heterogeneous environments is also an implementation priority. The specific Head-Modeling based environment discussed in section 4.1.2 is quite interesting and powerful, but it was accomplished by a somewhat manual manipulation of the Python middleware. While the choice of using Python for the middleware was deliberate for the sake of programmability (this example being an excellent case where that became handy), the plug-and-play style of heterogeneous environment configuration is effective enough to deem further improvements to the modularity of this component of PRESTO. It is foreseeable that an internal module could interface with PRESTO's current MPI layer in order to support general configuration of heterogeneous application environments, without having to think

too much about specifics, such as which MPI processes will do which things in the environment.

6.2. Conclusion

The design of PRESTO allows for flexible execution of common parallel computational models in a variety of programming languages. The notion of modularizing these models for general use in scientific applications relieves strain on developers who occasionally need to re-invent the wheel when they need to work within parallel models such as master/worker. PRESTO has implemented an optimized and dynamically load balanced master/worker model in Matlab, Python, Java, and C++, which enables domain scientists to use an efficient task scheduling tool without having to dedicate time and know-how towards writing optimized code for distributed memory programming. PRESTO also shows strong performance characteristics when compared to DCS. This is true for both the real-world Stingray application and a synthetic task pool that emphasizes the overhead inherent to both centralized task schedulers. Overall, PRESTO has provided a reasonable solution to a major problem in parallel computing, but requires sufficient contribution from the community to fully accomplish the goals presented in section 1.2.

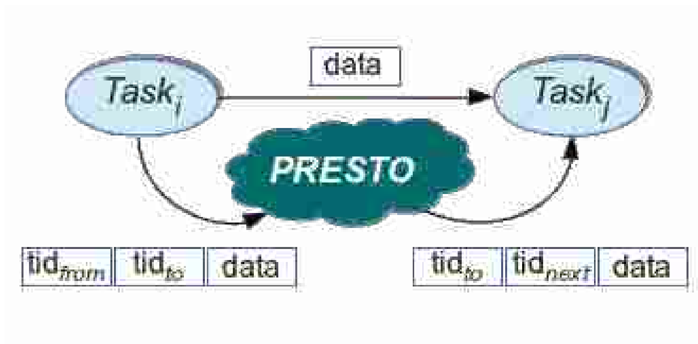

Figure 6.1. Task-to-task: PRESTO supports sending tasks between computational engines, although this ability is not exposed by the master/worker interfaces. This ability suggests that PRESTO can be extended to support other more sophisticated and non-centralized computational models.

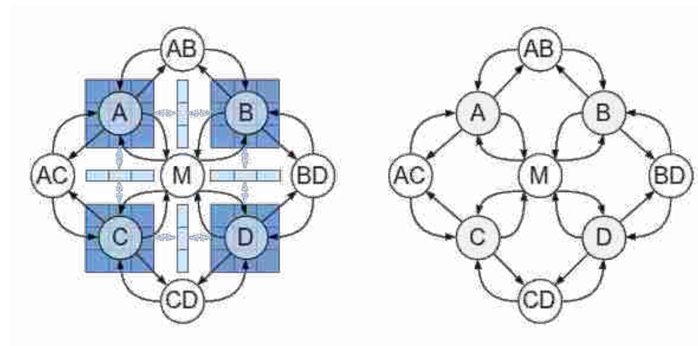

Figure 6.2. 2D ghost-cell application. This model is very similar to master/worker, but requires some task-to-task for efficient communication across the ghost cells. The AB, AC, BC, and CD engines could be communication components between adjacent grids.

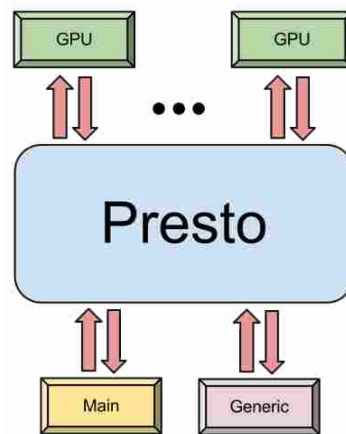

Figure 6.3. PRESTO cluster environment. One possible extension to PRESTO is to allow for the registration of cluster node-types which could be mapped to different kernels. For example, GPU nodes could be utilized for GPU tasks automatically.

REFERENCES CITED

[1] D.I. George Amalarethinam and G.J. Joyce Mary. Article: A new dag based dynamic task scheduling algorithm (dytas) for multiprocessor systems. *International Journal of Computer Applications*, 19(8):24–28, April 2011. Published by Foundation of Computer Science.

[2] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[3] R. Choy and A. Edelman. Parallel matlab: Doing it right. *Proceedings of the IEEE*, 93(2):331–341, 2005.

[4] Ron Choy, Alan Edelman, John R. Gilbert, Viral Shah, and David Cheng. Star-p: High productivity parallel computing. In *The 8th Annual Workshop on High-Performance Embedded Computing (HPEC 04)*, 2004.

[5] Lisandro Dalcín, Rodrigo Paz, and Mario Storti. Mpi for python. *J. Parallel Distrib. Comput.*, 65(9):1108–1115, September 2005.

[6] Lisandro Dalcín, Rodrigo Paz, Mario A. Storti, and Jorge D'Elía. Mpi for python: Performance improvements and mpi-2 extensions. *J. Parallel Distrib. Comput.*, 68(5):655–662, 2008.

[7] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[8] Mark Goldenberg, Paul Lu, and Jonathan Schaeffer. Trellisdag: A system for structured dag scheduling. In Dror Feitelson, Larry Rudolph, and Uwe Schwiegelshohn, editors, *Job Scheduling Strategies for Parallel Processing*, volume 2862 of *Lecture Notes in Computer Science*, pages 21–43. Springer Berlin Heidelberg, 2003.

[9] W. Gropp and E. Lusk. The mpi communication library: its design and a portable implementation. In *Scalable Parallel Libraries Conference, 1993., Proceedings of the*, pages 160–165, 1993.

[10] R.A. Haring, M. Ohmacht, T.W. Fox, M.K. Gschwind, D.L. Satterfield, K. Sugavanam, P.W. Coteus, P. Heidelberger, M.A. Blumrich, R.W. Wisniewski, A. Gara, G.L.-T. Chiu, P.A. Boyle, N.H. Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *Micro, IEEE*, 32(2):48–60, 2012.

[11] Christopher W. Harrop, Steven T. Hackstadt, Janice E. Cuny, Allen D. Malony, and Laura S. Magde. Supporting runtime tool interaction for parallel simulations. In *In Proceedings of ACM/IEEE Supercomputing 1998 Conference*, 1998.

[12] G.C. Hulette, M.J. Sottile, and A.D. Malony. Wool: A workflow programming language. In *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 71–78, 2008.

[13] Geoffrey C. Hulette. *Twig: A Configurable Domain-Specific Language*. PhD thesis, University of Oregon, 2012.

[14] Timothy H. Kaiser. Mympi - mpi programming in python. In *In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 458–464. CSREA Press, 2006.

[15] Jon K et al Nilsen. Simplifying the parallelization of scientific codes by a function-centric approach in python. *Comput. Sci. Disc.*, 3 015003, 2010.

[16] Rajkiran Panuganti, Muthu Manikandan, Baskaran Ashok Krishnamurthy, Jarek Nieplocha, and Atanas Rountev. An efficient distributed shared memory toolbox for matlab, 2007.

[17] Radu Prodan and Thomas Fahringer. Dynamic scheduling of scientific workflow applications on the grid: a case study. In *In 20 th Symposium of Applied Computing (SAC 2005*, pages 687–694. ACM Press, 2005.

[18] A. Saifullah, K. Agrawal, Chenyang Lu, and C. Gill. Multi-core real-time scheduling for generalized parallel task models. In *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, pages 217–226, 2011.

[19] R. Sakellariou and Henan Zhao. A hybrid heuristic for dag scheduling on heterogeneous systems. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, page 111, 2004.

[20] A. Salman, A. Malony, S. Turovets, V. Volkov, D. Ozog, and D. Tucker. Next-generation human brain neuroimaging and the role of high-performance computing. *HPCS*, 2013.

[21] Jeremy Siek, Lie-Quan Lee, and Andrew Lumsdaine. Boost random number library. http://www.boost.org/libs/graph/, June 2000.

[22] David B. Skillicorn and Domenico Talia. Models and languages for parallel computation. *ACM Comput. Surv.*, 30(2):123–169, June 1998.

[23] MatthewJ. Sottile and AllenD. Malony. Interlace: An interoperation and linking architecture for computational engines. In Patrick Amestoy, Philippe Berger, Michel Dayde, Daniel Ruiz, Iain Duff, Valerie Fraysse, and Luc Giraud, editors, *Euro-Par '99 Parallel Processing*, volume 1685 of *Lecture Notes in Computer Science*, pages 135–138. Springer Berlin Heidelberg, 1999.

[24] Douglas Toomey. Etomo: Endeavour seismic tomography experiment. *http://pages.uoregon.edu/drt/Research/ETOMO/index.htm*, 2009.

[25] K Uutela, M Hamalainen, and R. Salmelin. Global optimization in the localization of neuromagnetic sources. *IEEE Trans Biomed Eng*, 45:716–23, 1998.

[26] Herbert Utz Verlag. *Work Efficient Parallel Scheduling Algorithms*. Informatik, 1998.

[27] Min-You Wu, Wei Shu, and Jun Gu. Efficient local search for dag scheduling. *IEEE Trans. on Parallel and Distributed Systems*, 2001:617–627, 2001.