A Parallel Processing Library for User-friendly Applications

by

Aleksander Borys Demko

A Thesis submitted to the Faculty of Graduate Studies of

The University of Manitoba

in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba, Canada

Abstract

Clusters of commodity, “off the shelf” workstations have given developers and users access to scalable and affordable computing resources. However, unlike large, symmetric multi-processing machines, these clusters have an up front cost in complexity, both for the developer and the user. Existing software frameworks have attempted to mitigate this complexity with varied success. In most frameworks, the user is forgotten and left to deal with an unwieldy application.

This thesis presents the design, development and testing of a new C++ computer programming library, Scopira Agents Library (SAL). SAL is a message passing interface and implementation suitable for building parallel applications, with a focus on developer ease of use and user application deployment specification. The target developers and users of such a solution would be those who prefer an easy to develop library, with simpler deployment and application integration options with acceptable sacrifices to performance and scalability.

The novelty of this parallel programming library is that it is more user-friendly than other existing libraries. This novelty has two major facets: (*i*) programmer-usability and productivity and (*ii*) application integration. Together, they permit a wider range of programmers to utilize parallel programming in a wider range of new and existing applications. This goal, user-friendliness, is rare among current parallel programming libraries.

The result of the novelty is that parallel programming can be embedded into more

applications, especially desktop applications. The user base and use cases for parallel applications can be increased, resulting in more efficient use of resources in a variety of applications. With increased efficiency, work can be performed in less time and larger problems can be tackled.

Acknowledgments

I would like to thank my advisor, Dr. Nick Pizzi for his invaluable advice, direction, patience and perseverance throughout this adventure.

Finally, I would like to thank my wife, Luba, for her motivation and unwavering support during the final stages of this work.

Table of Contents

List of Figures

List of Tables

List of Abbreviations and Acronyms

API Application Programming Interface, a contract and specification by one service application (usually a code library) to another application (usually the client application).

C++ A multi-paradigm programming language invented by Bjarne Stroustrup, who started its development in 1979. It is a superset (although not strict) of the C programming language.

CORBA Common Object Requesting Broker Architecture (CORBA) is a software standard that enables software components distributed on multiple hosts and written in multiple computer languages to cooperate and work together.

COTS Common off-the-shelf hardware, refers to hardware that can easily be purchased from common, mass-market vendors (as opposed to specialized vendors) at usually cost effective prices (due to mass-market competition).

COW Cluster of workstations, a collection of workstations connected by a network.

CPU Central processing unit, the general processing core in a workstation.

FIFO First in, first out, a standard queue-like data structure in Computer Science. Alternatively, in UNIX, a FIFO refers to a type of interprocess communication available to processes within one host.

GPU Graphics processing unit, processors tuned for 2D and 3D graphics

rasterization.

GUI Graphical User Interface.

GUID Globally unique identifier. A 128-bit integer value that can be randomly or systematically generated at different hosts. Due to their large range, they are assumed (but, of course, cannot be guaranteed) to be unique within the universe.

IDL Interface definition language, used to define the interface of programming objects and functions in a language-neutral fashion.

IOR Interoperable Object Reference, a token or opaque reference to an object instance in a distributed object system such as CORBA.

IP Internet protocol, the core packet protocol for the Internet.

ITK The Insight Segmentation and Registration Toolkit, a software toolkit by Kitware, Inc.

LOC Lines of code.

MPI Message Passing Interface, an API specification for process intercommunication.

NOW Network of workstations, similar to COW.

OpenGL Open graphics library, an API specification for 2D and 3D graphics.

PID Process identifier, a operating system specific token or opaque reference to a running process, usually a simple integer.

PVM Parallel Virtual Machine, a library and software system for combing a NOW

into a single, virtual machine for computation purposes.

RAII Resource acquisition is initialization, a design pattern used in C++ where objects (and their deterministic lifetimes) are used to safely control access and handles to another resources.

RTTI Run-time type information, a C++ system that keeps information about an object's data type at run-time.

SAL Scopira Agents Library, the extension to Scopira that provides an embeddable, distributed processing system and the main body of work of this thesis.

SFS Stochastic feature selection, a method of feature selection used for pattern classification developed by Nick Pizzi.

SSI A single-system image is a cluster of computers appearing (to users and applications) to be one single system.

TCP Transmission Control protocol, a stream protocol built on IP and one of the core protocols of the Internet.

Threads A lightweight process. A feature of modern operating systems that permits multiple instructions within one process to be executed concurrently while sharing the same memory space.

UDP User Datagram Protocol, a datagram (packet) based protocol built on IP and one of the core protocols of the Internet.

UNIX A computer operating system developed by AT&T in 1969. Today, it is a

specification and trademark that can be applied to a wide variety of operating systems that share certain common functions.

URI Uniform resource identifier, an imprecise synonym for URL.

URL Uniform resource locator, an address of a resource (usually document) on a network with protocol information. In popular language, these usually refer to web addresses.

UUID Universally unique identifier. A synonym for GUID.

VTK The Visualization Toolkit, a software toolkit by Kitware, Inc.

1 Introduction

1.1 Motivation

Cluster and parallel computing continues to be the domain of experienced algorithm developers and power users. Developers must be skilled in many areas to correctly and efficiently write parallel programs. To write efficient code, familiarity with a low level language such as C is crucial, giving the developer full control over memory use and processor utilization. The developer must be familiar with Linux (or some UNIX variant) as it is the operating system of choice for computation clusters, even though many developers write their applications on their desktop computers, which are often running Microsoft Windows. The developer must be familiar with a message passing library, its programming interface, deployment methods and debugging

systems. Finally, programmers must be familiar with how to decompose and design their algorithm in a parallel fashion.

For many developers, these upfront learning and development costs outweigh the benefits. Either the algorithms have moderate computational processing demands, running within long (but acceptable) time frames on single processors, or the developer's program is simply tied to an interactive desktop application paradigm.

Users often prefer to use desktop applications over harder to use, specialized parallel applications. Ease of use, familiarity and availability of their workstations and laptops make them preferable, even if their performance is less than optimal.

Typically parallel applications provide a contrasting experience to the user. The parallel application typically runs on a local Linux cluster. This often requires being on site (for example, due to network firewall restrictions or bandwidth requirements), eliminating mobile and off-line access. The user must remember and use an additional user name and password to access the system. The user must then login to the potentially unfamiliar system and use its interface. The user might have to remember the commands for some obscure non-graphical application or re-familiarize themselves with an alien graphical desktop. The files accessible by the compute cluster may or may not map to files that the user's desktop can access, which may involve further manual file copy operations. This experience is in stark contrast to that of local desktop applications.

Clearly, there is room for improvement. How can parallel application

development be made easier for the application developer and who can then utilize parallel processing in user-friendly applications? How can developers become more productive parallel algorithm developers so that users will accept higher-performing applications without losing the user-friendly graphical interfaces to which they are accustomed? The contribution of this dissertation will be the introduction and implementation of one possible solution.

1.2 Target User

There are many parallel programming solutions and message passing libraries in existence today (see Chapter 2 for a background overview). However, none of the existing message passing libraries focuses on ease of use. The focus is usually on performance and scalability, with little concern paid to application deployment and maintenance.

The proposed solution described here will attempt to address this shortcoming, by presenting a library designed to be intuitive and easy to use for both developer and user. This library will have acceptable performance trade-offs especially for its intended developer audience.

For the developer, the library API itself will be designed to make development fast, safe and robust. This is done through a concise and powerful API that utilizes C++ language features, to minimize the amount of development needed and to maximize the amount of compile-time verification. For the user, the self contained thread-based

(rather than process-based) library implementation will be easy to embed into applications, with no setup requirements, permitting the deployment of easy to use parallel processing applications.

This type of library targets the following developers:

- Developers with moderate parallel processing requirements who value development time could use this library to quickly develop parallel applications. The developed applications, could fall back (without deliberate user control) to utilize only the resources of the user's workstation when multi-node processing is not desired, required or possible.
- Developers and users with existing applications could use this library to add parallel processing, permitting the reuse of existing application code and interfaces while extending the usefulness and scalability of their applications.
- Developers who want to target their parallel algorithms at less technical users may use this library to build applications that are easier to use and deploy. In fact, the built applications would be (from the user's perspective) no different that a non-parallel processing capable application.
- Developers in any of these situations would greatly benefit from using the proposed programming library.

1.3 Problem Definition

This problem has a few facets. The first issue is how to make programming

parallel algorithms and applications, via a message passing library, *easier and less error prone* for the developer (usability). The solution must be *embeddable and multi-platform*, so as to transparently cope with existing desktop applications, on whatever platform they may be. Finally, the solution must be reasonably efficient (compared to other base line message passing libraries), so as to reap the benefits of parallel programming. These three key issues can be further expanded:

A library is easier to use (when compared to another library) when it requires less time and research to use. The library interface could minimize the amount of information required from the developer (especially redundant information), reducing the chance for programmer errors. Less code takes less time to write. The library could provide more compile and runtime checking, giving the developer error feedback before the bugs manifest themselves as difficult-to-debug output errors or mysterious program crashes. This reduces debugging and ultimately, development time significantly.

The library should be embeddable completely into existing applications. That is, it should be usable in the developer's existing applications, rather than forcing the developer to write new applications, dedicated to the task of parallel computing. The applications could be available on many platforms, so the library must be multi-platform. To further couple with the existing applications, the library should minimize (or eliminate) the need for management programs and setup procedures. Infrastructure programs such as these detract from the user's needs to simply run the core algorithms, and are often poorly understood and annoying. Any setup programs that are absolutely

required should be easy to use, perhaps part of the application, and preferably graphical and menu driven.

Finally, the library must be relatively efficient. It should maintain respectable scalability for many problem types and sizes. If the library is too inefficient, and introduces too much runtime overhead, the user may ignore the offered parallel solution, or perhaps demand a better one written using another library. This library will of course make trade offs, and certain problem types and large data sizes (and large processor sets) may require other parallel programming libraries with dedicated and perhaps specialized computation and communication hardware.

1.4 Research Questions

The research work in this thesis aims to design and implement a new programming library that will attempt to solve our stated problem: *How can parallel application development be simplified, permitting the construction of more user-friendly parallel processing applications?*

We will assess the effectiveness of the proposed approach using three evaluation criteria: (*i*) programmer usability, (*ii*) computational performance and (*iii*) application integration.

First, how can the solution be made easier and less error prone? How do we assess (programmer) usability? Usability often is a subjective metric, and can be assessed through surveys, focus groups and other traditional forms of usability testing.

Focus group testing can be error prone, expensive and time consuming, so for this thesis a more objective metric will be designed. This metric will attempt to look at the programming interface of the newly developed library and attempt to assess its usability, introducing further questions: *How do we assess the usability of a particular programming interface? How do we compare the usability, error reduction and general programmer aid of a pair of programming interfaces?*

Second, the focus of this project is not communication performance, but rather delivering an easy to use parallel programming library to allow parallel computing to be deployed in a wider array of applications. Nevertheless, efficient performance must still be achieved, at least for many common work loads, so as to not negate the benefits of parallelization and the time invested into these solutions. To maintain acceptable levels of performance, we must be able to objectively assess this efficiency. *What performance metrics do we use? What other libraries, algorithm types and work loads do we compare and test against? What levels of performance are considered acceptable?*

Finally, we must question the solution's embeddability or integration, that is, its ability to work with existing application code and application deployments. *What features are required to make the library most embeddable? What platforms must be supported? How do we minimize external and cumbersome infrastructure software? What techniques can be used to infer network and other configuration options?* Answering these questions will help the developer integrate the library seamlessly with

their existing code, letting users leverage parallel computing with their existing, familiar applications. To supplement the previous two evaluation criteria, we will provide anecdotal evidence demonstrating the effective integration capabilities of the proposed approach.

The answers to these questions will then collectively direct us to an answer to the original, primary question: *How can parallel application development be made easier for the application developer, who can then utilize parallel processing in user-friendly applications?*

1.5 Thesis Statement

The resulting work of this thesis will be the creation and assessment of a new, parallel programming library. This library, although not as high-performance as other well established libraries, will be decidedly more user-friendly, allowing the embedding of the library into existing applications, making for a more seamless user experience.

By utilizing various C++ programming language features, the library's interface will be easier to use (when compared to typical C or C++ parallel programming libraries) and by inferring redundant information automatically thereby being less prone to programming errors. The library's interface will be designed with object-orientation in mind, allowing it to be embedded naturally into larger graphical applications.

Second, to show that the library performs within acceptable performance characteristics for a variety of common work loads, various objective tests and

benchmarks will be performed. This will be done to show that the various novel features introduced by the library do not have an overly detrimental effect on performance, when compared to more established, performance-focused competitors.

It will be shown that the newly developed library is more seamless to embed into existing application than existing libraries via: various network auto-detection techniques; elimination of infrastructure programs; and generally less configuration and maintenance requirements on the user.

1.6 Thesis Objectives

1.6.1 Grand Objective

The grand objective of this thesis is to create a new software programming library for parallel application development, that is easier to use (program with), embed into existing applications and deploy to users in a seamless fashion when compared to major, existing parallel programming libraries. The newly created library must also be efficient for some work loads, allowing the developer to fully realize the benefits of parallel processing.

1.6.2 Method Objectives

The experimental methods will attempt to objectively assess the thesis answers to the various research questions. By showing measurable results, the work can, by answering these questions, show that the grand objectives have been achieved.

For programmer usability, deterministic code analysis techniques will be used to show how algorithms implemented for competing systems can vary in complexity and execution. Computational performance will be measured along many setups and configurations, hopefully painting a clear picture of the various performance characteristics of the competing packages. Finally, application integration will be demonstrated in one real-world application, with accompanying analysis and discussion.

1.6.3 Novelty and Contributions

The main contribution of this work is the design, development and implementation of a new, simpler parallel programming library that provides adequate performance (efficiency) for a useful range of parallel programming problems.

The novelty of this parallel programming library is that it will be more user-friendly than other existing libraries. This novelty has two major facets: (*i*) programmer-usability and productivity and (*ii*) application integration. Together they permit a wider range of programmers to utilize parallel programming in a wider range of new and existing applications. This goal, user-friendliness, is unique among current parallel programming libraries.

The result of the novelty is that parallel programming can be embedded into more applications, especially desktop applications. The user base and use cases for parallel applications can be increased, resulting in more efficient use of resources in a variety of

applications. With increased efficiency, work can be performed in less time and larger problems can be tackled.

1.7 Thesis Organization

The thesis is organized as follows.

Chapter 1 contained an overview and introduction to the thesis topics, including its goals and expected contributions.

Chapter 2 contains a background overview of parallel programming, a short history, its goals and various software and hardware trends. This chapter also outlines the current state of the art and various issues with certain key technologies.

Chapter 3 gives a background overview and analysis of the C++ programming language. This chapter outlines some of the features of C++ that justify its choice for the work of this thesis.

Chapter 4 gives a background overview of the Scopira programming library, a library that this thesis work utilizes for various (non-parallel programming) utility functions. This thesis work is often combined with the Scopira programming library to build a wide range of applications.

Chapter 5 outlines the design goals and considerations of the thesis work. This covers both the design of the application programming interfaces as well as an overview of the networking/threading back-end implementations.

Chapter 6 outlines the experiments that were conducted. These experiments

provide objective analysis of the thesis work, assigning various metrics to the work along its various thesis goals.

Chapter 7 presents the results of the experiments. An analysis is provided to summarize the results, with some discussion.

Chapter 8 summarizes the complete thesis work, its contributions and discusses some directions for future work.

Finally, appendices and a bibliography are also included.

2 Background: Parallel Programming

2.1 Introduction

Individual computer processors have limits to their performance, and only through parallel programming can scientific and high performance developers scale their work loads. These limits are both practical (there is an absolute limit to processor speeds) and financial (the cost-performance ratio tends to grow exponentially with processor speed), forcing users to grow the number of processors (rather than just their speed), if they want to achieve practical scalability.

Parallel programs, through this scalability, execute faster than those constrained to single processors. This has many user benefits. Users can get their results faster, which results in less wait time. In time-constrained environments, new algorithm options

become available. Users can process more data, giving more accurate results. Finally, globally deployed parallel applications (like the SETI@home [5][60] or Folding@home [64] projects) can make the seemingly impossible problems possible.

At first, parallel processing was the domain of expensive super computers, with many processors and expensive interconnects and infrastructure. However, over time, economies of scale have helped common consumer processors to, when connected together appropriately, reach impressive throughputs usable for a growing number of workloads, for a fraction of the price of specialized hardware. Users coupled common off the shelf (COTS) hardware with free UNIX-like operating systems (usually Linux) to form *Beowulf* [90][103] clusters. Less dedicated deployments are sometimes called a cluster or network of workstations (COW or NOW, respectively). These clusters brought parallel computing to many new groups of people, spurring new interest in parallel computing research and applications.

Over time, as the economics of scale continued to push consumer processors to impressive new speeds, conventional super computer manufacturers such as Cray [25] and SGI [98] watched more and more of their business go to Beowulf-specializing or regular computer vendors. COTS hardware and free software simply provided a much better value (in terms of processing power per cost) for many problems and domains.

The next large wave of parallel computing moved to the desktop market itself. Moore's law (which made predications concerning the ever increasing capabilities of processors [70][71]) may have reached its limit, prompting large, mass market

processor vendors such as Intel [53] and AMD [1] to sell processors with multiple cores (processing units) in each processor, effectively bringing parallel computing to the desktop. Now, common compute intensive consumer applications require parallel processing (at the very least, non-distributed parallel processing usually via operating system threads) should they want to take advantage of all the processing power.

Programmers have come up with a variety of tools to help tackle the challenge of developing parallel programs. These solutions can include parallel-specific language extensions, or operating system enhancements, or new code libraries. This chapter will outline some of the more prevalent packages.

2.2 Design and Organization

Parallel programming and *distributed programming* are two basic approaches for achieving concurrency in software. Parallel programming assigns work to two or more processors within a single or virtual computer. A dedicated cluster of compute nodes is considered a virtual computer according to this definition. Distributed programming assigns work to two or more processors, which usually reside on different computers. These computers may differ by location, hardware architecture and operating system configuration, resulting in a more heterogeneous configuration.

The parallel programming approach is concerned with dedicating processors (and their network interconnections) to solve mostly compute-intensive problems, and as such is more beneficial to the demanding scientific programmer. The distributed

programming approach is more ad-hoc, and, although it could be used for some high performance computing, it is better suited for solving many more general tasks. These tasks include the building of distributed applications that must utilize the resources of other machines to perform a task. These resources may include specialized hardware (such as printers and scanners), databases, file repositories and dedicated terminals.

Flynn [31][36] introduced a classification scheme for parallel programs and computers. His key classes of parallel machines were SIMD (Single Instruction, Multiple Data) and MIMD (Multiple Instruction, Multiple Data). SIMD applies the same algorithm to different pieces of data (sometimes known as "divide by data") across processors, while MIMD assigns different tasks or algorithms (and different data) to each processor (sometimes known as "divide by task").

SIMD lends itself more to scientific, and thus parallel programming, while MIMD and its division of tasks (especially at less granular scales) is more analogous to distributed programming. The dividing line between both schemes is not distinct. The schemes often overlap, resulting in hybrid systems. For example, an application could use distributed concepts for setup and overall application design while utilizing parallel programming for the high performance work.

2.2.1 Design Methodology

Parallel programs, with their many benefits, do come with a cost. These costs include additional challenges and considerations during program design, an

undoubtedly longer debugging process and finally a more involved deployment process.

The design process of parallel programs includes issues such as: decomposition, communication and synchronization.

Decomposition is the process of dividing the problem and its solution into parts. Often in parallel programs, the most scalable technique is to decompose the problem by data. Different parts of the data are sent to different processes that then all proceed to apply the same algorithm in parallel. The specifics on how to decompose the data could also have many options, each with different performance results. For example, the division of a numeric matrix could be done by rows, columns or both. More unbalanced structures require other considerations. The problem may also be decomposed by logical/functional steps, for example: input, searching, calculating, sorting, output, etc. Finally, for distributed applications, the problem must be decomposed by resource, for example: printer, databases and file repositories.

Communication issues arise in deciding how these decomposed solution parts will interact. Do the parts share memory? Which pieces of the partial solution do they exchange, and in what order? Who (which processor) manages the whole process? Are communications between senders and receivers synchronized? Are there collective broadcast operations? These issues have to be considered when designing a parallel program.

Synchronization issues involve the coordination, scheduling and operating order of the various processors. Do all the parts start at the same time, or must they be primed

individually? What happens when there is resource contention? What happens if a processor finishes its part before all the others (very typical in heterogeneous clusters and with irregularly structured data)? Are there algorithm dependencies that must be considered? Does the processor wait (and waste time being idle) or does it get more work? Who assigns this work? All these concerns must be considered when maximizing the efficiency of a particular parallel program.

2.2.2 Programming Challenges

Parallel programming also brings with it a host of new challenges to the development process. The concurrent interaction of many tasks brings a host of new and subtle issues.

Data race conditions occur when multiple tasks access a shared data resource, and the results depend on the order of access. The scheduling and interaction of tasks depends on the non-deterministic (from the software's view) behaviour of operating system process scheduling, network traffic, etc. Yet the application itself must mitigate these factors to retain determinism within itself. To resolve data race conditions, the application must enforce rules of access, such as enforcing a domain-specific access order or simply using mutual exclusion constructs to serialize access to shared data.

Indefinite postponement occurs when a task waits indefinitely on some event that never occurs. This is a fundamental development concern as potentially all data receive operations in a task are suspect. Programming bugs may cause a sending task to miss

sending a required message, branch to some other code path, or crash and cease to exist. Entire computer nodes may crash due to some hardware, operating system or other software bugs.

Deadlock is a subtler problem related to indefinite postponement. Two or more tasks may be so intertwined in their communication that they end up waiting for events from each other. Since they are all in a waiting state, none of the tasks resolves the deadlock.

Communication concerns are also introduced by parallel programs. Will the program use shared memory or message passing? Shared memory programs have seemingly simpler communication models, but they still must be aware of data race, indefinite postponement and deadlock pitfalls. On certain architectures (such as MOSIX [11][72] and SGI's NUMA [19]), not all memory accesses have the same access times, and programs must try to localize their working data set. Message passing algorithms must also be concerned with indefinite postponement and deadlocks. Furthermore, messages may be lost, delayed or interrupted; all challenges that must be met.

All these issues (in addition to the regular challenges of algorithm development) could manifest as bugs in the software, leading to crashed, frozen or otherwise unable to function software instances. When this occurs, at the very least, users should be able to restart or continue the program without leaving any persistent data in an unused state or leaving any stray task instances on remote nodes (“zombies,” in operating system parlance). Ideally, the program should be built with fault tolerance in mind and continue

to run. However, fault tolerance comes with a large development cost, potential run-time overhead and is usually not necessary for many applications.

Specialized applications running on untrusted nodes also have other considerations. This approach is popular with *volunteer computing* applications (projects where anyone can volunteer processor time) such as SETI@home [5] and other applications such as those based on the BOINC [6] software package. In addition to fault tolerance requirements (nodes abruptly disconnecting is the norm, not the exception), these applications must also double check their results to protect against malfunctioning, or more probably, malicious nodes. Malicious nodes may be motivated by sabotage, curiosity and/or the urge to deceptively accelerate up through the contribution charts (in cases where rewards for participation are offered). Checking a node's work commonly involves sending its work unit to another node (or two, in the case of triple checking). The comparisons must have some tolerances, to account for marginal differences in results due to different processor architectures. This can be done on all work units, a random subset, or a random subset that prefers to check new nodes' work (as a form of trust establishment). All this requires more development work and possibly much more runtime overhead.

Finally, all parallel programs and deployments have limits to their scalability. At a certain point, adding more processors will not give any more meaningful speedup (Amdahl's law [4]). By definition, only a fraction of a parallel program is able to be parallelized. The sequential parts, the cost of communication and synchronization will

eventually dominate the application's runtime eliminating the benefits of adding any more processors. Certain subsets of "embarrassingly parallel" applications can avoid this up to a large numbers of processors, but few "interesting" problems are in this class. These applications have no intercommunication requirements and are able to scale globally, such as SETI@home and Folding@home [6].

2.2.3 Organizational Models

The programmer must also decide how best to organize and interconnect the various task processors in the software. Certain types of algorithms lend themselves to certain organizational models. Some of these organizational models (*design patterns*) are presented here.

In the *delegation* (*boss-worker*, *master-slave*) model (Figure 1 (a)), one process is the boss, while one or more processes are the workers. The workers do nothing but *pull* work from the boss and return results. The boss is responsible for distributing work, controlling overall application flow and terminating the workers. The boss does not do any computational work itself and simply sits idle (i.e. does not consume processor resources), waiting for requests and results from the workers. The boss may create workers as needed, or may keep a stable of workers throughout all jobs (this minimizes the cost of worker creation). Genetic algorithms [61] are often implemented using this model.

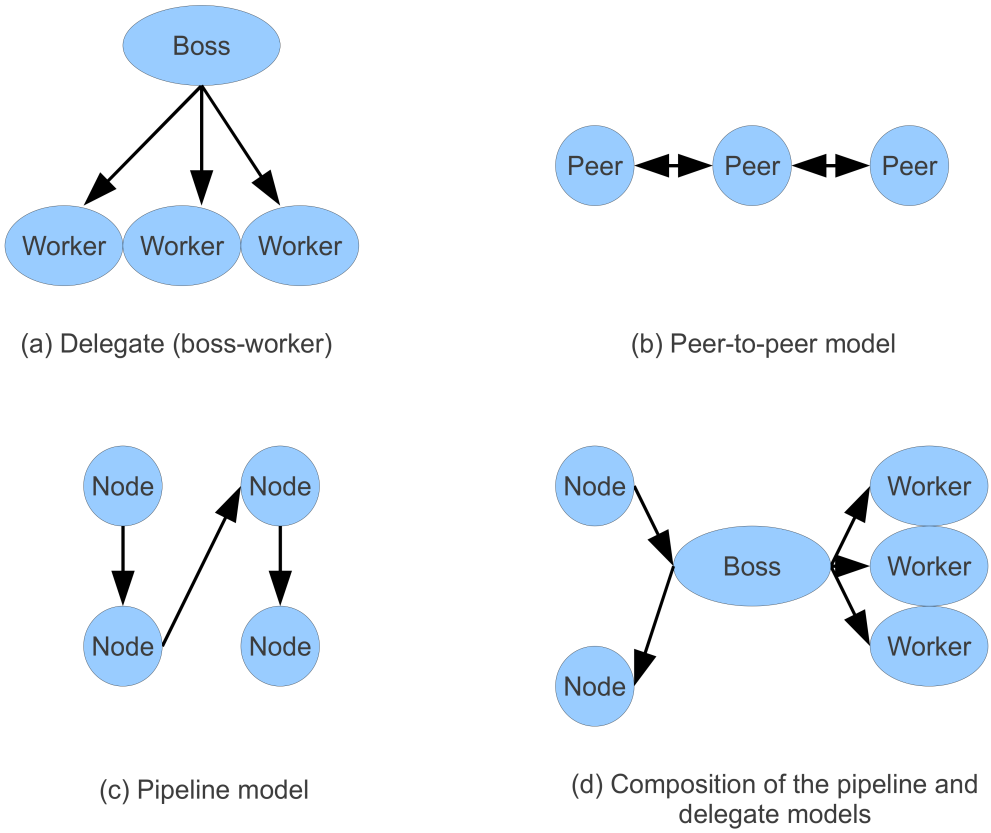

Figure 1: A sample of program organizational models. Arrows indicate data flow.

The *blackboard* model can be considered a variation of the delegation model. In this model, there is no boss process, but rather a collection of mostly autonomous workers who communicate via a common, shared data "blackboard". Workers, when ready, access the blackboard, determine what they should do, and return later with results and new units of work. This model is common in autonomous agent systems.

In the *peer-to-peer* model (Figure 1 (b)), all the processes are more or less equal, each doing the same work. This implementation is the closest to a pure SIMD (or

MIMD) implementation. Each worker in this model performs the same algorithm on their own subset of the dataset, exchanging data at various intervals. One worker (usually the first, initial process), must create the others, parcel out data and combine the results. This little bit of sequential overhead cannot be avoided and does not detract from the overall algorithm design. Image processing often lends itself to this model.

In the *pipeline* model (Figure 1 (c)), each process or thread handles one step in a multi-step process. This is an example of dividing by task, rather than by data, and can be considered a type of MIMD implementation. Nodes may be assigned tasks due to their unique resources (for example, database stores) or specialized hardware (such as some data acquisition instrument) or powerful processor configuration. The pipeline model, due to its serial nature, is highly prone to developing bottlenecks. One node will be the limiting factor in the chain, decreasing the efficiency of the whole application. This presents a load balancing challenge to the developer, who cannot simply add more processors. The pipeline model is often used in distributed applications, where nodes are brought together for their specialized resources, rather than just processor time contributions.

The *producer-consumer* model can be considered a subset of the pipeline model. One task produces data (gathered from some source) to be processed by the consumer. This is often used in *client-server* configurations for distributed applications.

Finally, the models can be combined into a *composition* (Figure 1 (d)). This is often the result of combining algorithms, either sequentially or in a nested (caller-callee)

fashion. This model comes in many configurations.

2.3 Standard Libraries

Many modern operating systems contain multi-tasking and multi-programming features. These features may be utilized by any program, without the need for specialized libraries or configurations. This section will discuss some of these built-in features that many operating system provide.

2.3.1 Multiple Single Instances

The most basic form of multi-processing is starting and executing multiple instances of the same program, but with different data, on one host computer. The instances themselves work in isolation and do not communicate with each other. The programs are SIMD and are divided by data. Each instance either has different data or is running the same data with different parameters. They usually do not process different parts of the same data (as this would require inter-instance communication). When these tasks are executed on a multi-processor computer, then the total work performed often scales linearly (assuming no other bottlenecks) with the number of processes run simultaneously, up to the number of processors.

This simplistic technique has drawbacks. Single work tasks are not split up, and the individual jobs themselves are not performed in parallel. As there is application specific no process manager, there is no automated load balancing to optimize resource utilization. The user, in fact, must often manage the processes manually, a tedious

process that can be somewhat helped by the use of shell scripts.

This technique requires a modern, *preemptive* multi-tasking operating system. All desktop computers sold today include operating systems with this capability, including Microsoft Windows, Apple's OS X and Linux. This capability allows the operating system itself to preempt a process and take back control of the processor. The operating system then gives control to another process, time slicing the processor's time between all running processes. This gives the impression (to the processes) that they are running simultaneously. On multi-processor computers, they actual do execute simultaneously, resulting in a work load speed-up.

Certain operating systems use *cooperative* multitasking. These include older desktop operating systems (such as Microsoft Windows 3, or Apple's System 9) and many embedded operating systems, such as those found in cellular phones. These operating systems require the process itself to explicitly yield control of the processor. Such operating systems, by design, lack the capabilities to utilize multiple processors and provide desktop parallelism.

2.3.2 Threads

Threads are an important feature of a preemptive, multi-tasking operating system. They are standard on all modern, computer operating systems, and provide a lightweight and powerful mechanism for achieving parallelism within a single program instance. Under UNIX and UNIX-like operating systems, the POSIX threads API [21] is

often used, while Microsoft Windows has its own API. There exist many other interfaces over these interfaces, such as Boost Threads [16][57] (and soon standard C++), Qt [14][87] and Scopira Threads (Section 4.1.4). These all help in reducing typical thread programming errors as well as provide a generic interface for writing portable programs.

A thread is a scheduled operating system execution stream. Typically, all processes have one main thread and an address space. Processes each contain an instruction pointer, stack, and some state registers. Additional threads are sometimes known as lightweight processes, as they are much less resource intensive than multiple processes but still are scheduled like normal processes, and (on multi-processor hardware) actually run concurrently with other threads in the same process and memory space.

Threads have their own stacks but share the same address space, global variables, and dynamic memory heap with other threads. There is only one instance of the address space among the threads in the same process: changes in this address space are immediately visible to other threads. This makes intercommunication between the threads fast, especially for shared data, which does not need to be explicitly transferred or duplicated at all. In some ways, this also makes them easier to program but this is not always the case, as described below.

Threads may “communicate” with each other by placing messages and data in some shared data structure. Access to shared memory must be synchronized and

sequenced so as to prevent race conditions that may occur if one thread reads the memory before another associated thread has finished writing it. For this, the thread API will provide a collection of synchronization primitives that threads can use to coordinate access. These typically include:

Mutex (mutual exclusion or critical section) objects provide the concept of a lock, to protect and sequence shared data access. A mutex can either be unlocked or locked by a thread. When another thread attempts to *lock* a mutex, it will block (wait) until the existing thread *unlocks* (releases) the lock. Alternatively, a thread may timeout waiting for a lock, and may try again (polling) or perform some other action. A thread with an active lock may relock the same mutex: this aids in the development of certain (recursive) algorithms.

A *condition variable* provides a method of thread signaling. One or more threads may *wait* on a condition, while other threads may signal them. The alternative, constantly polling a mutex, is both computationally wasteful and inaccurate, as there is some latency between lock attempts during which the receiving thread will never operate on messages. Conditions are often coupled mutexes. A *wait* operation is used to unlock a mutex and wait for the condition to be signaled (and re-acquire the lock on receiving the signal). This is a common programming pattern as a condition is often used to signal the arrival of some data, which is then stored in some shared variable.

Finally, *read/write locks* provide a specialized variant of mutexes. Read/write locks allow either one *writer* exclusive control of a resource, or many *readers* to share

it. This increases concurrency over standard mutexes, as readers are now no longer forced to wastefully wait for each other to release locks. Obviously, writers and readers cannot share the resource at the same time. Read/write locks can be implemented using normal mutexes. However native read/write locks are preferred, as they are able to solve subtle scheduling and contention issues that may not possible with standard mutexes.

For certain classes of simpler communication problems, some libraries (such as QtConcurrent [88]) provide some additional concepts around threads that do not require the use of the previous primitives. They include the concept of *futures* [45], which represent the results of some future operation. The main thread spawns a function in the background, is given a *future token* and uses the token to poll and, when ready, retrieve results from the completed background thread. Primitives for the map-reduce [27] functional paradigm may also be provided. If an algorithm can be broken down into mappers (data converters) and reducers (consolidators/mergers of converted data), then the framework can take care of setting up and applying the user-provided functions until algorithm completion. Google [40] is a major proponent of this technique used in their search engine and other applications.

Despite the availability of these primitives, concurrent programming with threads can still lead to many subtle and difficult to debug programming errors. This is especially true for the inexperienced programmer, who may be tasked to implement complex numerical or other algorithms.

Threads also have some disadvantages. Any thread that crashes or faults, may

bring down the whole process and all the other threads in it. This makes threaded applications, in some sense, less fault tolerant. All threads have complete and equal access to the address space, therefor requiring all threads to be trusted and preventing any sandboxes of untrusted or foreign code.

Programs that do not want or cannot have (because of lack system support) native, preemptive threads, may use *pseudo-threads* (*or user-level threads*). Pseudo-threads may be implemented in the operating system or in the application itself. These threads only return control back to the scheduler when they either explicitly yield control or call an I/O or other blocking function (which in turn yields control). After control is returned, another thread is given control. Obviously, these threads (within a single process) cannot utilize multiple processors, as they share only one actual operating system thread. Pseudo-threads are particularly useful if the thread programming model is desired, for example in I/O programming, but preemption and multi-processing is not required. It is particularly useful on smaller (e.g. embedded) platforms that do not have a preemptive multi-tasking operating system or capable processor, such as those found in smaller devices such as cellular phones. Many runtimes or virtual machines of interpreted languages also start with a pseudo-thread model, due to its portability, simplicity and low thread switching overhead.

2.3.3 Interprocess Communication

Many operating systems provide additional methods for interprocess

communication. This section will give a brief overview of the methods commonly provided in UNIX and UNIX-like operating systems (such as Linux).

Pipes are specialized file-like objects that allow two related processes to communicate. Pairs of processes may be chained from the command shell by the user when they are launched, or with the **`popen`** system call. Pipes are basic, untyped binary streams, although text is usually sent.

A *FIFO* (first-in, first-out) is a bidirectional pipe offered by a program with an on-disk "name". This allows unrelated processes to exchange data. The **`mkfifo`** system call may be used to create them.

UNIX-Domain Sockets are similar to FIFOs in that they are local to one host and have an on-disk name or handle. However, they offer a more featured API than simply a plain file stream, utilizing the socket interface that is also used for TCP/IP and UDP/IP. This way, they can be thought of as an efficient, local machine-only network socket system.

System V IPC is a collection of interprocess communication primitives that includes: messages queues, shared memory, and semaphores. Message queues are lists of messages (each with a fixed maximum size) for a receiver, where order is preserved. Shared memory allows processes to create and share segments of memory, where all may examine and change the contents of the shared segment. Finally, semaphores are counters (with special operations defined on them) that are used to provide synchronized access to shared data objects across multiple processes.

These techniques, although functional and fast, tend not to be popular with parallel algorithm developers. For developers who go to the trouble of partitioning their algorithms into multiple processes, there exist better library-based solutions for message passing. Such libraries (discussed in the next section) bring many benefits over the esoteric and platform-specific APIs discussed here, including enhanced ease of use via a simpler API. They are often more tuned for group and numerical computing and multi-platform support and data manipulation.

2.3.4 TCP/IP

The *Transmission Control Protocol/Internet Protocol* (TCP/IP) is one of the standard intercommunication protocols of the Internet and of many local area networks. It is a globally deployed networking standard that is understood by the vast majority of networked devices. All modern operating systems provide programming interfaces for communicating with TCP/IP. The TCP/IP protocol provides an error free, ordered and reliable network stream. Its sibling protocol, the User Datagram Protocol/Internet Protocol (UDP/IP) provides connection-less datagram messaging, which does not guarantee arrival or delivery order, and as such is more efficient for applications that do not require these guarantees.

Programmers may use TCP/IP directly to do cluster and parallel computing; however, it only provides a reliable, bidirectional, binary data pipe between two processes. Programmers themselves therefore must manage how their objects get

transferred over this binary pipe, including connection setup and maintenance, message signaling/framing, message routing (if not direct), and data marshaling. Parallel application programmers are often better off using a messaging passing library, which performs all theses functions in a well tested and standard manner. Message passing libraries may also transparently utilize other transports, such as those tuned for local host communication or on specialized communication hardware, further increasing communication efficiency in certain cases.

2.4 Message Passing Libraries

A message passing library is a software library that provides an API for sending and receiving messages, and possibly other auxiliary functions. These libraries provide several benefits over operating system specific libraries. A platform independent API does not lock the programmer into one operating system, and provides data collection and translation (marshaling) functions for transferring data between different computer architectures. These libraries are able to adapt to various communication requirements all behind the same API, such as using a thread-implementation for within-process communication or network sockets for basic inter-host communication. Such a library also performs many of the common setup and maintenance functions required for cluster computing, removing this task from the programmer. Finally, the API may be better tuned for numerical computing, increasing programmer productivity as many common functions do not need to be redeveloped.

2.4.1 MPI

The *Message Passing Interface* (MPI) [68][99] is perhaps the de facto standard in message passing libraries. It is an API standard [73][74] defined by a committee, the *MPI Forum*. There exists free (such as MPICH [42] and LAM [20]) and commercial (such as Scali/Platform MPI [84]) implementations as well as specialized implementations (such as USFMPI [22] which has a threaded implementation). Some implementations include optimizations for specific communication hardware, such as Infiniband [51]. The specification is language independent with C and Fortran implementations being the most common. The C++ implementation tends to often be ignored by C++ programmers, as it contains only minor differences over the C version and shies away from using more ambitious C++ features.

The standard is designed for communication for both workstation clusters and specialized parallel processing super computers. The flexible API contains many constructs for dealing with a variety of communication types (such as broadcasting, scattering, and gathering) and striped array configurations, permitting library implementation optimization opportunities. The plethora of options can sometimes be confusing and error-prone, but are necessary for completeness.

The popularity, completeness, and ubiquity of the MPI standard and its implementations makes it a solid foundation on which to build parallel processing applications.

2.4.2 MPI (C++ API)

The MPI standard provides a C++ version of the API potentially useful for C++ programmers. Although enticing for C++ programmers, this API is a basic port of the C API. It converts some core MPI data types (and their functions) to objects (and methods), but still retains a very C-centric approach to pointers and numeric arrays. It is rather conservative in its use of C++ features, ignoring such facilities as generic programming or object serialization. C++ programmers may sacrifice the few features that the C++ MPI API does provide and simply use the C API, as it affords them greater source code compatibility with existing C algorithms.

2.4.3 PVM

PVM, *Parallel Virtual Machine* [37][85], is a software package for the parallel networking of computers. It permits a group of processes to cooperate on a network (via message passing) to solve problems in parallel. It includes an API for message passing, process management and fault tolerance (as well as other services), a multi-platform implementation and management software.

PVM supports many platforms and enables the connection of dissimilar computers to form heterogeneous clusters. The PVM implementation takes care of data marshaling and otherwise hiding platform differences. This encourages users to pool all available computational hosts together, increasing performance and efficiency. Programs are free to use platform-specific optimizations, however, when they run on

preferred hardware.

Unlike MPI, PVM has one primary implementation. This focuses the community and testing on one package, at the risk of diversity, specialization, and optimization issues. PVM, via its process manager programs permits processes (if so configured) to join and leave the system at will. This tends to permit more ad-hoc use, and is particularly useful for small to medium sized clusters..

2.4.4 Charm++

Charm++ [65] is a C++-based object-oriented parallel programming library developed by the Parallel Programming Group at the University of Illinois based on Charm [35][56]. It aims to enhance C++ programmer productivity yet still retain good performance. Charm++ is available on most desktop and workstation platforms and also includes support for many super computing architectures such as: BlueGene, Origin2000, and various Cray systems.

Charm++ programs and algorithms are decomposed into a number of cooperating objects called *chares* (concurrent objects) that communicate with other chares via messages (communication objects). Processes are dormant and only awakened (and assigned to a processor) when messages arrive. This delayed scheduling approach minimizes scheduler use and complexity. Load balancing, in general, is dynamic and adaptable but static load assignments are also available.

Charm++ uses its own Interface Description Language (IDL) to define the

messages remote objects may receive. Given a user supplied object specification in this IDL, Charm++ will produce various standard C++ files that are subsequently integrated with the application. Although cumbersome at first, this permits programmers to treat remote objects almost as if they are local objects via an asynchronous CORBA-like [96] calling system, as Charm++ marshals and packages the calls as message objects and sends them to their destination.

Charm++ also includes *Adaptive MPI* (AMPI) an implementation of a significant subset of MPI 1.1 over the Charm++ system. This permits many MPI programs to be used and tested on Charm++ without significant change. The system reuses the dynamic and adaptive nature of Charm to bring load balancing to MPI applications, giving MPI applications additional deployment options.

Charm++ shares many technical similarities with the work described in the thesis, but with significant differences in design goals. For example, developer usability/application integration is not a focus of the Charm++ library, resulting in cumbersome deployments and integration approach. In particular, Charm++'s use of an IDL to specify and formalize messages complicates embeddability and integration by requiring that the developer learn a new language and use additional utilities during the build process.

2.4.5 CORBA

CORBA [96] is a standard for cross-platform object-oriented programming

defined by the Object Management Group [76]. It permits the programmer to distribute objects within a program to different processes, usually on different hosts connected via a network. The programmer is then able to interact with these objects, almost as if they were typical, in-process objects.

In practice, the interaction with remote objects is not completely transparent. There are some significant time commitments required for the design and development of the distributed objects. After the objects are instantiated, the programmer must also be aware of many limitations when dealing with the remote objects. The remote objects may reside in different processes and thus in different memory spaces, so the programmer may not use traditional pointers to interact with them. This limits the interaction with remote objects to what is defined by the objects specification – usually to public method calls only. Performance (latency and total throughput) with remote objects is also limited by the network. Finally, method calls on remote objects can *fail* in drastically different ways from in-process objects, usually because of network, hardware or software issues. The programmer must account for this by designing the application to be robust and able to account for catastrophic failures in core, common objects. Using *exceptions*, an error handling feature provided by many languages, including C++ (but not C or Fortran), developers can make their applications more robust, but without the clutter of error checking tests after each remote method call.

CORBA is a standard with many different implementations, some tuned to different deployment environments and priorities. The objects themselves are specified

in a language-neutral IDL, which is later compiled into target programming languages. The generated code provides data marshaling (gathering and conversion of data parameters for network transport) for the clients, and skeleton implementations to aid in implementing the objects on the server.

Pointers within one processes' address space, a typical method of retaining a handle to object instances, cannot be used in distributed applications. CORBA uses opaque Interoperable Object References (IORs) as handles to distributed objects within a network. These handles contain all the information needed (such as host address and port) to access and use a distributed object instance.

CORBA provides facilities to convert well known names or signatures into IORs. This allows applications to find their remote components more easily, without the manual propagation of IORs. The basic name service provides basic name to IOR resolution. This service is contextual and allows the grouping of similar names into contexts with a scalable recursive look up. A more dynamic and decentralized trading service learns and discovers objects within a network. Objects advertise (“exports”) services while clients search for (“imports”) services. The trading service introduces clients and services by matching their service requests and advertisements.

Distributed computing (as encouraged by CORBA and similar technologies) has notably different goals than parallel programming message passing libraries, which are designed for groups of processes to quickly and efficiently exchange loosely defined, ad-hoc data with each other. Distributed computing is useful for connecting and sharing

resources (as in the consumer-producer or pipeline program organization models) spread over different machines, connected via a network and achieves this flexibility through higher runtime overhead. Distributed computing may also be used to unify objects or application pieces developed by different programming teams or at different times. Distributed computing is also well adapted to agent programming, which sometimes requires the ad-hoc, multi-platform, multi-languages facilities that are provided.

2.5 Task Based Libraries

Task based libraries present a different approach to parallelism than that of message passing libraries. In a message passing library, the programmer defines both the tasks and the intercommunication sequences between them. In a task based library, the programmer simply provides the tasks, which are assumed to have a simplistic input-output processing model. The library then performs all the resource management, data partitioning and transport, and scheduling for the programmer.

The message passing library approach is more flexible and is able to handle more communication models at the cost additional complexity. For algorithms that can be decomposed to independent tasks however, a purely task based approach may be beneficial as it requires less library-specific setup and communication code. Task based libraries and approaches can also be layered (and used) over message passing libraries. Some examples of task based libraries are now discussed.

2.5.1 BOINC

The Berkeley Open Infrastructure for Network Computing (BOINC) [6] middleware package is a collection of software to aid in the building and distribution of *volunteer and grid computing* projects. These types of projects involve large numbers of nodes (potentially millions) with no inter-node communication. BOINC was originally part of the SETI@home [5] project but it broke out into its own project when its utility in other work became apparent.

BOINC provides a programmer API for job and results submissions, data transfer, software for job management, account management and web site administration. Contribution tracking and ranking is particularly important as it provides feedback and motivation to volunteers. Developers need only supply the application code specific to their algorithm.

Volunteer computing projects depend on the donation of computer time from desktop computer users on the Internet. These worker nodes are untrusted and anonymous, and special considerations must be implemented when utilizing them. In addition to fault tolerance requirements (nodes abruptly disconnecting is the norm, not the exception), these applications must also (at least) double check their results to protect against malfunctioning, or more probably, malicious nodes. Malicious nodes may be motivated by sabotage, curiosity and the urge to deceptively accelerate up through the contribution charts.

BOINC provides facilities to help manage untrusted nodes in volunteer computing

projects. This includes tracking and rechecking nodes' work by resubmitting jobs to other nodes, and comparing results for validity. Doing this for every job, although thorough, would be inefficient. For this BOINC includes many options for performing this check on only a subset of the submitted jobs. BOINC can also use a credit point system to assign a trust reputation to users. As the users gain trust, their work is checked less.

2.5.2 QtConcurrent

The Qt Library [14][87] is primarily a library for multi-platform graphical user interface (GUI) desktop application development. It contains an API for drawing on screen graphics, managing interactive widgets and many other utility areas that are useful to developers who want to build cross-platform applications. One of these areas includes a threading module that provides a consistent multi-platform API around threads and threading primitives such as mutexes and semaphores.

In addition to basic thread primitives, Qt offers the QtConcurrent framework that provides high-level APIs that make it possible to write multi-threaded applications without dealing with lower-level primitives. The framework also provides some features specific to the Qt GUI library such as asynchronous function calling that frees the GUI thread from doing work, resulting in more responsive GUIs.

The QtConcurrent API uses a task concept, where programmers supply the basic algorithm task code, a data set, and then lets QtConcurrent partition and execute the

algorithm. The library performs the thread management (usually via flexible thread pools) and scheduling. This lets programmers concern themselves more with their specific algorithm code rather than thread management and proper thread primitive usage.

Background tasks can be managed by the calling threads via a *future* concept. When a background thread or thread set job is launched, the calling thread is given a future token that represents the future (not yet computed) return value of the background computation. The caller may query or wait on the future when it is ready, and upon completion can obtain the results of the background operation. This basic but powerful concept frees the user from having to manage thread processes.

QtConcurrent is useful for applications already utilizing the Qt library that need a small amount of concurrency features. However, it lacks many features found in other dedicated task libraries, and also (by design) does not contain any support for cluster or distributed computing

2.5.3 Threading Building Blocks

Threading Building Blocks (TBB) [89][113] is a C++ template library from Intel Corporation. With the advent of multiple processing cores in consumer desktop machines (rather than increasing clock rate), Intel wants to increase multi-processing capabilities in standard desktop applications. It hopes to encourage such multi-threaded programming via the TBB library.

TBB, like QtConcurrent, provides a task based concept rather than thread and thread-primitives (mutexes, semaphores) approach to multi-programming. Programmers supply the basic algorithm task code, a data set, and then lets the library partition the data and run the algorithm. The library performs the thread management (usually via flexible thread pools) and scheduling. This, again, lets programmers concern themselves more with their specific algorithm code rather than thread management and proper thread primitive usage.

This approach is similar to OpenMP (discussed in Section 2.6.1) in theory, but in practice is much different (in implementation). TBB is purely a C++ library using standard C++ constructs and features. Unlike OpenMP, it does not require a specialized compiler or non-standard language extensions.

TBB is able to work with other threading libraries and with OpenMP. It is also designed with nesting in mind, allowing all levels of a program to be parallelized. It uses a flexible, dynamic scheduling algorithm that supports *work stealing* (moving work from overloaded processors to idles ones). TBB, however, is only for threading and does not scale beyond one host such as for a cluster of workstations.

2.6 Language Extensions

Most modern and mainstream programming languages were not designed with parallelism in mind. The easiest way to add parallelism to an application using such languages is via a code library, extending the functionality of a language without

changing the language itself.

An alternative to code libraries is to extend the language (or create a new language) with parallel concepts, such as iterators and synchronization operations. Parallelism constructs become a natural and integrated part of the language, fully checked during the compiling process with instant error feedback.

Switching (or updating) a user's programming language requires a larger commitment from the programmer. The programmer must now use a specialized compiler, which may be costly, or may not perform as well as the non-parallel compiler in other areas. Unless the extensions are optional, the programmer is now committed to this (possibly) niche compiler for all future projects and platforms.

Alternatively, language extensions may be implemented as code translators that transform extended code to standard code. Although complicating the build process, this technique allows the continued use of existing (and trusted) standard compilers for parallel projects.

Finally, there is research (such as SUIF [3][44] and the Intel Compilers [52]) into making compilers automatically parallelize serial code. This would be a panacea for parallel code development: free parallelization without any added development work. However, this challenging problem has had limited success as it is often difficult (due to the inherit dynamic nature of many programming languages) to fully statically (at compile time) deduce a program's structure without some input from the programmer. The programmer's understanding of an algorithm's intent seems critical to being able to

decompose and partition an algorithm for parallel execution.

This section outlines some language extension-based packages.

2.6.1 OpenMP

OpenMP (Open Multi-Processing) [24][78] is a shared-memory (via threads) multi-programming API standard created by the OpenMP Architecture Review Board (ARB). The first version was released for Fortran in 1997 followed by a C/C++ version in 1998.

Under C/C++, OpenMP permits code to be augmented with OpenMP directives. These directives direct an OpenMP compiler to partition and parallelize segments of code using threads. Thread management is done automatically, using a variety of scheduling schemes, such as dynamic, static and guided scheduling, with respect to the data.

These directives are implemented as **`#pragma`** preprocessor directives. Compilers that do not support these specialized directives simply ignore them, permitting OpenMP code to be compilable by conventional compilers for execution on serial machines.

OpenMP provides a support library that is linked with OpenMP programs. Programmers may use the API provided by this library to perform additional dynamic (at run-time) tuning and configuration. Users may also influence OpenMP-enabled programs by setting various OpenMP-specific environment variables.

OpenMP support has often been implemented in specialized compilers such as

those by PGI [112] and Intel [52]. Recently, however, more mainstream compilers such as the GNU Compiler Collection (GCC) [38][41] and Microsoft Visual Studio [47] have added support for OpenMP, providing opportunity for wider adoption.

OpenMP provides a solution only for shared-memory (single host) multi-processing. Although it can be combined with cluster computing solutions, OpenMP itself does not provide multi-host parallel computing features.

2.6.2 Unified Parallel C

Unified Parallel C (UPC) [12][23][107] is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The language's model is usable on clusters of machines (distributed memory architecture) but the programmer is presented with a single address space. Variables are grouped to processors but any processor may transparently access any other processor's variables that are marked as *shared*.

Thread scheduling is set at program startup, usually one operating system thread per physical processor (or processing core). UPC makes no implicit assumptions about the memory and synchronization model. The programmer must explicitly use the various provided threading primitives to synchronize access to shared data. These primitives include typical mutexes (locks) and barriers (synchronization points).

UPC requires special, upgraded compilers to compile its extended C code. A modified version of the GNU C Compiler (GCC), GCC UPC [39] supports UPC.

Various research compilers also support UPC.

2.6.3 Erlang

Erlang [9][10][33] is a programming language for the development of highly concurrent, robust and fault tolerant software systems. The research work on Erlang started in 1981 at the Ericsson Computer Science Lab with production deployments starting in 1988. Development and interest continues to this day.

The language focuses on developing highly concurrent applications through message passing, with a strong emphasis on boss-worker and client-server topologies. Any function can be made into a concurrent task: giving it a PID (process identifier) with which it can receive and send messages. In-language primitives are provided for sub-task spawning, asynchronous message sending, receiving, parsing, and queuing. The runtime has native support for clustering, allowing multiple Erlang process instances to intercommunicate for performance scalability on one host, or a network of many hosts. Finally, the language has support for multiple versions of functions, building a foundation to allow in-place updates of live software resulting in no downtime, a feature critical to demanding, high-availability domains.

The language, its libraries and runtime have been successfully used and deployed in a variety of areas. These include Ericsson's AXD301 scalable telephone switch, CouchDB [7][8] a schema-free database and the X2000 satellite control system developed by NASA [2].

Erlang has been deemed a success in its particular niches. Unfortunately, widespread adoption has been slow. Its Prolog-inspired syntax can seem alien to most programmers, giving it an actual and psychological learning curve. Erlang does not integrate well with C code, requiring quite a bit work to adapt and interface non-Erlang code with the Erlang messaging model. Finally, it does not support generics and other numerically optimized types making it cumbersome and slow for numerical computing.

2.7 Other Solutions

This section describes parallel processing solutions that do not fit in the previous sections.

2.7.1 Mosix

Mosix [11][72] is management software for Linux clusters. Development started in 1977 and continues on various platforms. In 1999, a Linux version was released and immediately capitalized on the popularity for cost-effective cluster computing.

Mosix extends and enhances the Linux kernel software so that multiple Linux kernels on separate machines can combine and present one large system image to processes. The processing and memory resources are merged together presenting a large, single-system image (SSI) to users and applications.

Mosix is not a programmer's library and does not need to introduce new APIs for applications. Rather, applications run unmodified on a Mosix cluster as if they are on a large computer. The Mosix Linux kernel provides all the standard, expected operating

system features and functions. Older applications may be reused directly, extending their life. Usability is simplified by presenting the user with one system interface.

Multi-processing is performed via the standard operating system threading interfaces. Applications that take advantage of single-host parallelism via threads would scale to multi-host parallelism on a Mosix cluster. Applications can now scale to clusters with no added development time.

Unfortunately, a Mosix cluster can only emulate a single system in interface but not in performance. On a true single system all processes have high speed (via the system bus) access to memory, often with uniform latency. Under a Mosix cluster each processor only has system bus-speed access to the memory in the same node. Accessing memory in another node requires network communication. This creates a latency bottleneck as the much slower network is used to simulate memory reads and writes.

Most threaded applications assume very fast, random access to memory, as is typical in most workstations. As a result, many such applications have intricate memory access patterns or non-local (per processor) memory working sets. When scaled to a Mosix cluster, such applications may "thrash" (abuse) the network, making the network a performance bottleneck and severely limiting overall system efficiency, negating parallelism speed-up.

These performance issues can be somewhat mitigated by using faster, possibly specialized, intercommunication hardware such as Infiband [51] or SGI NUMA [19]. Persistent performance problems, however, may require some software redesign. The

redesign need not necessarily be major – developers can still use single-image threads for multi-programming – rather the application should attempt to better localize data and memory access per thread (which results in better local working sets for processors and thus less network communication overall).

2.7.2 OpenCL

OpenCL [59][114] provides a standard for utilizing graphic processing units (GPUs) in consumer 3D accelerator hardware for fast, parallel computing in the form of *GPU-computing*. OpenCL is vendor neutral standard, unlike previous vendor-specific solutions such as NVIDIA's CUDA [92]. Although not directly related to cluster computing, GPU-computing, like cluster computing, utilizes COTS hardware to realize large performance-price gains. However, GPU-computing can be combined with cluster-computing to combine their respective benefits.

Thanks to the continued performance push of consumer video games, high-performance, dedicated hardware graphics accelerators have reached mass-market adoption. No longer is high quality 3D graphics the exclusive domain of specialized workstations for vendors such as SGI [98]. These graphics accelerators contain highly specialized graphics processing units (GPUs) that are capable of rendering visual scenes orders of magnitude faster than general processors (CPUs).

By its nature, the process of rendering and rasterization of graphics onto a display lends itself to parallelization. As such, GPUs attain their high-performance by applying

parallel processing to this problem. GPUs are composed of many smaller, simpler processors that perform the rendering in parallel. Together, these simple processors can easily outperform a single, but much faster general processor at this particular task.

More recently, the various parallel processing elements in GPUs have become user-programmable, and may be used for non-graphics related functions. Developers may compile specialized mini-programs for the parallel units and have the GPU execute their combined programs in a highly parallel fashion, with speed-ups of an order of magnitude or more over conventional CPUs. Hardware vendors first introduced their own APIs and standards for these programs, such as NVIDIA with CUDA. Standards such as OpenCL have emerged to unify APIs and provide a common language and interface for developers.

Currently, due to hardware limitations, OpenCL programs have many restrictions, such as: program size, memory accessibility, variety of data types, stack-less local variables (eliminating recursive functions) and no heap (eliminating dynamic memory). Even with these constraints, developers have been clever in applying GPU-computing to non-graphical, but computationally demanding areas. Even so, adoption is only in the preliminary stages and is expected to increase when some of these restrictions are relaxed, features are added to the standard, and development tools mature.

OpenCL and GPU computing provide an additional parallel computing option for algorithm developers. Although not directly related to general CPU parallel processing, GPU programming requires similar proficiencies in algorithm decomposition and

design. GPU algorithms may also be combined with general parallel computing, using traditional parallel computing to link (via a network) GPU-enabled compute nodes.

3 Background: The C++ Language

3.1 Introduction

For all but the most short term ("throw away") projects, the choice of which programming language to use when implementing an algorithm or developing an application is important. The language must be relatively modern (that is, still maintained and used) yet show that it will last (and still be maintained and used in the years to come). This requires that the language not be obscure (for finding future developers and maintainers may be problematic). Finally, in high performance computing and especially cluster computing, the language must be efficient, or more specifically, allow for the creation of efficient run time executables.

Traditionally, C [58] or Fortran [63] are used when implementing computationally

demanding algorithms. Developers simply require the pure speed offered only by languages that are efficiently compiled to machine-specific code. More popular interpreted languages – those that compile to an intermediate representation that is then interpreted at runtime – can introduce a significant amount of computation overhead. These interpreted languages (such as Java [55], Python [86] and Ruby [91]) are ruled out when such delays translate to longer run-times.

Contrast this to desktop or web application developers who are more interested in programming languages with ease of use features (such as automatic memory management) and many software library options (fostering code reuse) rather than pure performance.

When application and algorithm developers mix (for example, providing an interface or visualization options to an algorithm), often a two-language approach will be used. The computation core will be written in C and the interface in an interpreted language, such as Java. The two mix either via an embedded approach (in Java's case, via JNI, the Java Native Interface) or via a network communication approach. Examples of this approach can be found in MATLAB [97][111] and Maple [67].

The C++ language provides benefits to both application and algorithm developers. To the algorithm developers, it provides various features (some of which will be enumerated in the following sections) that make code more robust, concise and flexible, yet still compile to fast machine code. To the application and algorithm development teams, it offers a unified language that may straddle both the domains of algorithms and

interfaces in one efficient and flexible package.

C++ is often overlooked by programmers with the perception of it being too large a language, *bloated* with unnecessary features, ignorant of their uses and applications. With enough patience and time, developers will learn of each feature's particular use and how it makes for writing better program and libraries, without sacrificing performance.

3.2 History

Bjarne Stroustrup began designing the C++ (at the time, *C with Classes*) programming language in 1979 while at Bell Labs with the hope that it would aid in the development of a network-distributed UNIX operating system kernel [105]. Having had previous positive experience with Simula in his Ph.D. work, but negative experiences with its performance and scalability for larger systems, Stroustrup vowed never again to tackle large projects with inadequate programming tools and languages.

The C language [58], chosen as the base for C++, is flexible and efficient, and its implementations widely available and highly portable [105]. C is *efficient* as its low-level operations, such as bit-manipulation and unchecked type conversion mirror the fundamentals of traditional computers, crucial for performance and access to hardware (the latter an absolute requirement when writing operating system software). C++ was deemed a programming structure and organizational enhancement to the language, and priority was given that they not introduce any run-time overhead compared to pure C.

This *no-worse-than-C* (often termed as the *zero-overhead* rule) approach to performance would prove to be an important feature, allowing the language to be used in many performance-critical applications. Being built on an existing, well-tested language meant that C++ introduced no limitations to the programmers and immediately offered them a familiar programming style in which they could reuse much of their existing code.

C++ was first publicly released in 1985 [105], with the publication of *The C++ Programming Language* [104] and the commercial *Cfront* C++ compiler. Although sometimes confusingly referred to as a preprocessor, rather than emitting machine code it emitted C code, Cfront was a full compiler front-end. Cfront did full C and C++ syntax and semantic checking (with immediate error feedback), built and analyzed an internal representation of the input and finally emitted the final C code, using C as if it was a portable assembler. The emitting of C code allowed Cfront to use the wide availability of various C compilers for the final machine code generation stage, increasing the available platforms for C++ and reused the compiler optimizations research in existing compilers.

Cfront development continued, adding multiple inheritance to version 2.0 (1989), while version 2.1 (1990) brought the compiler in sync with *The Annotated C++ Reference Manual* [32] the first official standards document for the language, which would become the starting point for official standardization. Release 3.0 (1991) added templates and exception handling. In 1991, the second edition of *The C++*

Programming Language [104] was published.

Since 1990, the ANSI/ISO C++ standards committee has been the primary forum for the effort to complete C++. This was required as the user-base for C++ quickly attracted the interest of various groups of users, tool implementors and educators. To scale to these new demands and responsibilities, the committee was used as a forum to debate and flesh out the needs of the various stake holders.

In 1994 the ANSI/ISO Committee Draft was registered as an official standard, giving users and implementors a common reference or contract. Non-standard language extensions were still created as certain niche users required, or certain vendors thought they required. With the publication of a standard however, these extensions were made obvious, giving users an explicit line to cross when they entered non-standard territory.

3.3 Object-oriented programming

Object-oriented programming involves the concept of grouping data and functionality into "objects" (packages of state variables and functions) when designing computer software. Although Simula [26] is often considered the first object-oriented language, the paradigm did not gain mainstream popularity until the early 1990s. C and Fortran do not support this paradigm within the languages themselves. However, most, if not all, new programming languages do provide an object-oriented paradigm.

Traditionally, an object-oriented programming language provides the following features [69]: (*i*) modularity: the concept of grouping functions and data or state (*ii*)

encapsulation: the ability to protect or restrict data to key functions and (*iii*) polymorphism: the ability to transparently treat particular object variations as some common, abstract, ancestor type.

The first goal of C++ was to add the object-oriented paradigm to the C language, and as such implements these concepts completely. The **`class`** construct allows the programmer to group functions (methods) and data together (providing *modularity*), the **`private`** and **`protected`** directives allow the programmer to protect data and methods within those classes (providing *encapsulation*) and finally the **`virtual`** keyword combined with class inheritance permits the programmer to utilize *polymorphism*.

The C programming language does not support object-oriented programming. The paradigm could and has been simulated (for example, in the GTK+ [62][110] widget library) with varying degrees of success. Without help from the language, however, the programmer is often left with a more tedious and verbose system. Method calls in GTK+ for example must include an explicit reference to the class as well as a type cast. Creating a new class in GTK+ requires dozens of lines of error-prone setup code, compared to one in C++ (a **`class`** construct).

Scientific algorithm programmers do not gain many benefits from an object-oriented programming paradigm. The translation of mathematical functions and algorithms to computer code already maps nicely to the separate functions and data model already present in all programming languages, including C and Fortran. However, these concepts are useful for library developers in these domains, giving

library authors the ability to present new objects (such as complex numbers, or new array structures) to algorithm developers without having to extend the language.

Applications developers, on the other hand, benefit greatly from the code organization benefits of object-oriented programming. Desktop applications tend to be larger (in terms of lines of code) projects, often with more authors employing a more disparate collection of software libraries than algorithm developers.

3.4 Class Destructors and RAII

The C++ language is one of the few mainstream object-oriented paradigm supporting languages that explicitly supports deterministic class destructors. A class destructor is a special method function that is called immediately when an object instance is destroyed, either explicitly from the heap via the **`delete`** operator or implicitly as the instance exits the scope of the containing function, block or class. The destructor is guaranteed to be called no matter how the execution path leaves scope, including via a **`return`** or **`break`** statement, or via a thrown exception. An annotated C++ example of this is:

```
void some_function(void)
{
    FileObject F;
    // assume that F's destructor will call F.close(),
    // to terminate any resources

    // open and work with F here...

    if (...) {
        throw some_exception(); // (1) exception thrown
    }
```

```
        if (...) {
            return; // (2) return immediately exits scope
        }

        // (3) F falls out of scope naturally
    }
```

In the above example, the object **`F`** will have its destructor (which in turn calls **`F.close()`**) implicitly called in all three locations. In another language, programmers would have littered their code with three calls to **`F.close()`**, and calls to any other cleanup functions for any other object they may be using. This clutters the code and increases the chances for bugs (especially memory leaks) to creep in. In C++, this technique is also required if the code is to be *exception safe* [108].

This use of destructors to automatically clean up resources implicitly has been encapsulated in *Resource Allocation Is Initialization* (or RAII [109]). This concept, where the acquisition and releasing of some resource (in this case, a file) is directly linked to the lifetime of an object is applicable to many resources. These include dynamic memory allocations, reference counts, thread mutexes and semaphores, SQL queries and transactions, and graphical resources.

Interpreted object-oriented languages such as Java tend to be coupled with a powerful garbage collection mechanism, making memory management for the application developer much easier. This however means that object destruction is decidedly non-deterministic, making predictable clean up code cumbersome. Java provides a partial solution, allowing classes to clean up resources via an optional **`finalize`** pseudo-destructor. This method will be called when the garbage collector is

disposing of the instance, which occurs after some unspecified delay after its end-of-use.

Non-object oriented languages such as C have no concept of object-methods at all. The programmer must make all resource allocations and deallocations explicit, unnecessarily increasing code size and the potential for bugs.

3.5 Generic Programming

Generic programming [106] (using templates or parameterized types) is a programming paradigm that supports the design and development of functions and types (classes) that operate on yet-unknown types. This is similar in idea to object-oriented programming, but with a significant difference. In generic programming, the types are reconciled at *compile-time* (rather than *run-time*) affording the programmer huge performance gains (as the generated types are custom built to the desired types) and compile-time type checking (reducing programming errors early in the development cycle).

C++ is the only mainstream language that implements full generic programming concepts with compile-time in-lining (where the bodies of functions are inserted right in the caller's code), where generated types, classes and algorithms receive the same support and features as native types [105] and functions. Generated types using the template mechanism can be made to be no-worse than if the same concept was programmed explicitly by the programmer (or via macros). This ability is vital for

numeric computing where performance is critical.

In the C language, macros can be used to simulate the most basic use-cases of template programming. However, in even those instances, macros lack such features as proper type checking and smart linking provided by C++ templates.

Combining templates, function in-lining and operator overloading (being able to redefine operators such as "+" or "()") library programmers can make very powerful numerical array types that have no-worse-than-C performance characteristics [105]. For example, given the following C example, a 10 by 10 matrix of complex numbers of type **`double`**, lets assign -1 to the imaginary component in the matrix element 5,5:

```
double the_array[10*10*2];          // 100 elements
the_array[ ((6*10)+5)*2 + 1 ] = -1;
```

The same much more intuitive declaration in C++ might be:

```
narray<complex<double>,2> the_array(10,10);
the_array(5,5).imag_part = -1;
```

From a performance standpoint, the two versions are identical. Utilizing templates, function in-lining, and operator overloading, the C++ version performs the same steps and operations as the C version but with a cleaner, more robust syntax. The C++ version could also have run-time range checking in the element look-up that can be quickly disabled (for well tested, post-debugged public release builds) to further enhance code robustness. The next C++ update, C++0x will introduce *concepts* to help alleviate some of the vague and verbose errors compilers sometimes emit during template programming. Concepts, similar to interface classes in object-oriented programming will provide an interface specification for new types. Should a type fail to

fulfill a concept, a sensible error can be immediately emitted rather than emitting a more convoluted error later, when the type is used.

3.6 Memory management

Dynamic memory allows applications to request and utilize memory allocations of varying size as it is needed. This functionality is critical in many types of algorithms and applications as it allows them to scale to any data set size, without wasting memory through overestimated preallocations.

The C language's standard C library provides explicit functions **`malloc`** and **`free`** for the allocation and deallocation of dynamic memory. The language and library provide no aids in managing memory, the programmer must make sure to properly manage all dynamic memory allocations. This is error prone and often leads to many subtle and not so subtle memory errors (such as memory leaks).

The explicit managing of memory is so error prone that many new languages such as Java and C# tout their automatic memory management as one of their key features. These languages provide *garbage collection* (automatic memory management) services to manage memory. Through the coordination of language and runtime services, all dynamic memory allocations (and their references) are managed by the runtime and library systems. The runtime is then able to determine when memory is no longer being used (“garbage”) and then proceeds to *collect* (free) the unused memory automatically and in the background. Although this allows some temporary memory waste, as there is

some delay between when memory is no longer needed and when it is actually freed, the convenience and reliability of automatic memory management results in a huge boon for programmer productivity.

C++ takes a different approach to memory management than that of Java and C#. Although the basic and explicit memory management options of C are offered (as well as the newer type-aware versions **`new`** and **`delete`**), C++ allows library authors to utilize the existing language facilities to create automatic memory management systems.

Utilizing generic programming, operator overloading, and RAII, programmers can create type-safe *smart pointers* that behave like normal pointers, but perform additional checks and other functionality on assignment and termination. Smart pointers can immediately release unused memory automatically and do not require background processing (or a specialized runtime), a feature important for memory intensive algorithms. Smart pointers greatly enhance programmer productivity and reduce errors, while at the same time, giving no-worse-than-C performance.

The standard C++ library provides a basic smart pointer, **`auto_ptr`** for basic one-owner semantics and as an example of a smart pointer interface. For complex semantics, in particular, shared ownership, developers have had to go to other libraries (such has Boost's [16][57] **`shared_ptr`** or Scopira's [28][29][30][94] **`count_ptr`**). The new C++ update, C++0x, will include a **`shared_ptr`** class (using Boost's implementation), providing a standard shared-ownership shared pointer implementation.

Unlike Java and C#, user types in C++ (such a complex number class) do not

have to be dynamically allocated on the heap. They can be allocated on the stack, within other objects directly or as contiguous sets in array allocations. This is vital for numerical computing as it means a large array of *n* complex numbers, for example, could be allocated as one large contiguous block of memory. In Java, this would have to be *n* individual allocations of small complex instances or as two separate arrays of doubles, each of length *n* (in essence, breaking the array of complex numbers into two arrays of real parts and imaginary parts). The former case wastes (and potentially fragments) memory and processor time, while the latter case forces the programmer to restructure their program into primitives for performance reasons, defeating a major benefit of using higher level languages.

3.7 Parallelism In C++

The C++ standard language and library lacks facilities for parallel multi-programming. In the 1980s, during the early days of C++ development, multi-programming was not in demand as it is now. Multi-processor configurations were strictly the domain of expensive workstations and mainframes. Local computer networks were only beginning to be widely used and Internet connectivity was typically available only at academic and government institutions. This is in stark contrast to today where multi-core desktops have hit the mass-market and network and Internet connectivity is widely deployed.

The C++ standards committee is also very conservative with respect to adding

new APIs. They are aware that once an API is standardized, many vendors and users will invest in it. Changing poorly thought out or inadequate standards after they are published wastes resources and investments and adds to user confusion. The C++ standards committee did not want to prematurely commit to untested parallel APIs and designs.

As such, the C++ standards committee left it up to third party vendors to provide parallel programming libraries. Although this provided competing non-standard libraries, it did permit ideas and APIs to test and prove themselves among users. Some C++ programmers chose C libraries, out of compatibility with C code or for other reasons. The libraries span all the abstraction levels and ideas of parallel programming, from operating specific threading libraries, task-oriented libraries and message passing libraries to language extensions.

With the advent of mainstream multi-core computing, many desktop applications (not just scientific applications) are expected to be capable of multi-processing. As such, the C++ standards committee will include a threading API in the next C++ standard, C++0x. This API is based on (and is almost identical to) the Boost library's [16][57] threading implementation, and as such has had wide user-testing and feedback. This implementation is now available in the C++ Technical Report 1 (TR1), a preview of various new features in the upcoming update.

3.8 The Standard C++ Library

The C++ standard library is relatively small compared to the libraries in Java or C#. This is the result of C++ being under the control of standardization committees and boards, rather than single companies (Sun and Microsoft, in the cases of Java and C# respectively); however what is supplied is well tested and vetted, and generic enough to be applicable to all programmers. The standard C++ library also provides a style of library implementation and design that can be used by other libraries.

The standard C++ library provides basic string facilities via **`string`** and **`wstring`** (wide-character), a requirement in all applications. Basic input/output facilities via an extensible *iostreams* systems is included. This allows the formatting and processing of the core data types to streams, as well as any user-data types. The streams are able to operate on disk files and in-memory, with other sources provided via third party libraries.

The C++ library also provides a generic (template based) container collection that works on any data type. This collection includes re-sizable arrays (**`vector`**), linked lists (**`list`**), associative arrays (**`map`**), sets (**`set`**) and other common containers. Using generic programming and compile-time type generation, the resulting containers are specific to their contained-types, resulting in the best possible performance. Programmers never again have to re-implement these structures for their types.

General algorithms are also supplied. Functions such as searching, sorting, partitioning, iteration and counting are provided. There are also generic (template

based) functions and algorithms that can work on any data type (mostly *iterators*, discussed below).

The containers and algorithms in the C++ library are brought together via the *iterator* concept. An iterator is a programming object that can move or iterate through some data set. Iterators can vary in their interface and capabilities, while their implementations are specific to their container. The iterator concept was made to mimic the interface of standard C pointers and pointer iteration. This has the tremendous benefit of being able to use standard C arrays and pointers with the C++ library's algorithms. The mixing of algorithms and containers via iterators is done at compile time, again resulting in efficient code comparable to hand-coded solutions – vital for algorithm developers.

Developers are of course free to use the plethora of third party libraries that build on this foundation. Due to the power of the language itself, library developers are able to create some high-performance compile-time based libraries without needing to update the language.

The Boost [16][57] library is one such library that prides itself on its high code quality standards by providing high performance, multi-platform and flexible, general C++ libraries. It tends to follow the standard C++'s library ideals of using a broad range of the language's features to achieve its goals. Some notable libraries in this collection span the areas of: threading, random number generation, graph construction, an MPI [68][99] layer, image manipulation, Python interfacing, smart pointers, regular

expressions, serialization and multi-dimensional arrays. Boost's quality standards are so high that it is often used as a testing ground for libraries and features under consideration for feature updates to the C++ libraries. In fact, Boost's implementations for threads and smart pointers are among some of the updates to the next C++ standard, and may already be used in TR1.

4 Background: The Scopira Library

This chapter outlines the Scopira Library, a programming library that is used extensively by the work in this thesis.

The initial driving force for Scopira was to develop a comprehensive, object-oriented programming architecture using C++ for the development of applications relating to exploratory data analysis of magnetic resonance images (MRI), especially functional MRI [49]. Subsequently, the architecture was expanded to deal with confirmatory and exploratory biomedical data analysis, visualization, and interpretation, in general. This approach strikes a balance between slow interpreted languages such as IDL [17][115] and MATLAB [97][111] and fast compiled languages such as C and Fortran. Although well suited for algorithm prototyping and ad-hoc data visualization, interpreted languages are simply not suitable for application development. Conversely,

C and Fortran, although efficient, lack basic and expected language features such as object-orientation or basic memory management required for building large scale applications. C++ was chosen to straddle the two extremes, and even though it has been somewhat overshadowed by newer languages such as Java and C#, it is still the only language with such features as generics and object-orientation that still compiles to efficient machine code.

The emphasis with Scopira [28][29][30][94] has been on high performance, open source development and the ability to easily integrate other C/C++ libraries used in the biomedical data analysis field by providing a common OOP API for applications. This library provides a large breadth of services that fall into the following three component categories:

Scopira Tools provide extensive programming utilities and idioms useful to all application types. This category contains the reference counted memory management system, flexible/redirectable flow input/output system, which supports files, file memory mapping, network communication, and check sum calculation, as well as object serialization and persistence, reproducible and tunable random number generation, universally unique ids (UUIDs) and XML parsing and processing.

The *Numerical Functions* all build upon the core n-dimensional **`narray`** concept. C++ generic programming is used to build custom, high-performance arrays of any data type and dimension. General mathematical functions build upon the **`narray`**. A large suite of biomedical data analysis and pattern recognition functions is also available.

Finally, a *Graphical User Interface Library* based on GTK+ [62][110] is provided. This library provides a collection of useful widgets including a scalable numeric matrix editor, graph plotters, image viewers as well as a plug-in platform and a 3D canvas based on OpenGL [46][77].

The next three sections describe each of these Scopira component categories in turn. This is followed by a section presenting a few biomedical data analysis applications developed using Scopira and is followed by some concluding remarks.

4.1 Scopira Tools

Scopira consists of modular subsystems that can be used as needed by developers. The *Scopira Tools* subsystem provides generic facilities useful in many programming domains, not just numerical and scientific computing.

4.1.1 Memory Management

An *intrusive* reference counting scheme provides the basis for memory management. The scheme is considered intrusive as it records an object's reference count within the object itself, typically by having the object descend from a common base class. Many libraries, such as VTK [93][116] and GTK+ [62][110] implement similar reference counting systems.

Scopira implements a template class **`count_ptr`** that emulates standard pointer semantics while providing implicit reference counting on any target object. Alternatively, the **`intrusive_ptr`** from the Boost library may also be used, as

Scopira's reference counting scheme is compatible with its requirements. With either smart pointer, reference management becomes considerably easier and safe, a vast improvement over C's manual memory management.

4.1.2 Input/Output

Scopira provides a flexible, polymorphic and layered input/output system (Figure 2). Flow objects may be linked dynamically to form I/O streams. Scopira includes *end flow* objects, which terminate or initiate a data flow for standard files, network sockets and memory buffers. *Transform flow* objects perform data translation from one form to another (e.g., binary-to-hex), buffer consolidation and ASCII encoding. Future transformers will include CRC calculators, compressors and cryptographic ciphers. *Serialization flow* objects provide an interface for objects to encode their data into a data stream. Through this interface, large complex objects can quickly and easily encode themselves for storage to disk or transmission over a network. Upon reconstruction, the serialization system re-instantiates the objects from type information stored in the stream. Shared objects – objects that have multiple references – are serialized just once and properly linked to multiple references.

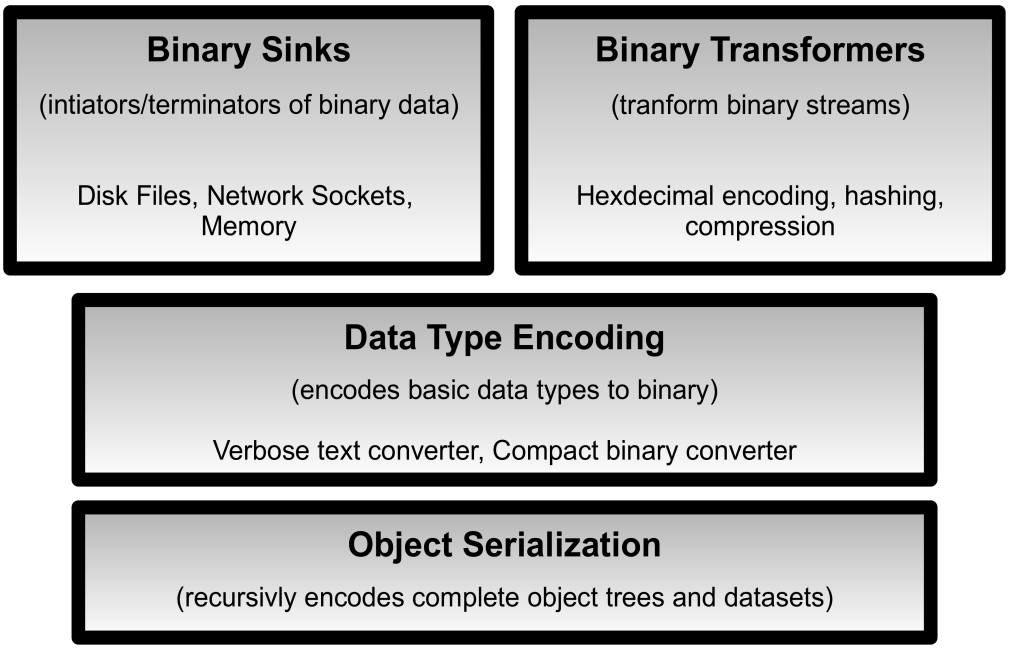

Figure 2: Scopira input/output stack

4.1.3 Configuration Handling and Plug-ins

A platform independent application-preferences handling system is supplied via a central parsing class. This class is able to accept input from a variety of sources (configuration files, command line parameters, etc.) and present them to the programmer via one consistent interface. The programmer may also store settings and other options via this interface, as well as build GUIs to aid in their manipulation by the end user.

Using a combination of the serialization type registration system and C++'s native RTTI (run-time type information) functions, Scopira is able to dynamically (at runtime)

allow for the registration and inspection of object types and their class hierarchy relationships. From this, an application plug-in system can be built by allowing external modules (e.g. dynamic link libraries) to register their own types as being compatible with an application, providing a platform for third party application extensions.

4.1.4 Other Utilities

Finally, the tools subsystem provides a variety of other services and interfaces. Native operating system threads (via the POSIX threads interface) are presented as C++ objects, with mutex locking and shared areas accessed via classes that follow the *Resource Acquisition Is Initialization* (RAII) (Section 3.4) principle. Generic arrays provide a lightweight (yet still STL-like) array class that is simpler than the STL's vector class and not specific to numeric computing as is Scopira's **`narray`**. Random number generation (inspired by Boost's random library) is also included. Universally unique identifies (UUIDs) and uniform resource locators (URLs) are also provided. XML processing (provided by the libxml2 library) is an optional feature, allowing one to build open and easy to use data file formats.

4.2 Numerical Functions

The central area of Scopira that is relevant to numerical and parallel computing is its array class, **`narray`**. This section will describe the background, reasoning and design of these arrays in depth.

4.2.1 Background: Arrays

The C and C++ languages provide the most basic support for one dimensional arrays, which are general and are closely related to C's pointers. Although usable for numerical computing, they do not attempt to provide the additional functionality that scientists demand, such as easy memory management, intuitive mathematical operations, or fundamental features such as storing their own dimensions. Multiple dimensional arrays are even less used in C/C++, as they require compile-time dimension specifications, drastically limiting their flexibility.

The C++ language, rather than design a new numeric array type, provides all the necessary language features for developing such an array in a library. Generic programming (via C++ templates, that allow code to be used for any data types at compile time), operator overloading (e.g. being able to redefine the plus "+" or array access "[]" operators) and inlining (for performance) provide all the tools necessary to build a high performance, usable array class.

The C++ standard template library (STL) uses these facilities to create the **`vector`** class. This class, along with its sibling containers and variety of generic functions provides an example of how to design the interface and implementations of flexible generic containers and algorithms. The STL **`vector`** class is a general vector class, designed to support all data types. Although a significant improvement to raw C arrays, these "arrays" still lack many features useful in numeric computing, such as multi-dimension arrays and subset-slices. One method of creating multi-dimensional

arrays with the **`vector`** type involves having a vector of vectors (and so on for higher dimensions). Although this works in limited situations, it has many disadvantages such as being less efficient, non-contiguous memory storage (eliminating the useful ability to treat the multi-dimensional array as a single one-dimensional array), inconsistent interfaces and verbose type names.

The C++ STL touches on the topic of numeric arrays with its **`valarray`** concept. This is a generic array container designed for numeric computing with hooks for providing high performance element-wise operations. These classes were designed for specific vector operations, specifically high performance bulk operations, and were not intended to be general numeric arrays with ease of use as a goal. However, this container can be used as a building block for building an end-user array class (examples are even provided in [106]), if only indirectly (that is, as a guide for interfaces and implementation).

The **`valarray`** types introduce another concept not addressed by the standard **`vector`** type or C arrays: the concept of slices. Via the **`slice_array`** type (and **`mask_array`** and **`indirect_array`** types, which take this idea to different ends), slices allow the program to view subsets of an array via an array-like interface. By storing basic information such as strides (that is, which n^{th} element does the slice use the original array), general slices operating on any dimensions within the host array can be made. This powerful concept is incredibly useful and is necessary for any serious numerical array framework.

Users have created their own libraries to fill the void left by the lack of standardized multi-dimension array classes in C++. These libraries vary in performance, API style, and focus. Some of the better established packages will now be discussed here.

The highly regarded Boost C++ libraries [16][57] contain not one, but two numerical array libraries, both introduced in version 1.29 of the library collection: *Boost.MultiArray* and *uBLAS*.

Boost.MultiArray provides a basic, but complete n-dimensional array class with support for views and slices. The library, like many of those in the Boost collection, utilize advanced C++ features and idioms to achieve their goals of performance and completeness, sometimes sacrificing ease of use for newer C++ programmers. This library, at its core has the most in common with the Scopira **`narray`** classes, differing mainly in their notions of element access and use of temporaries.

uBLAS is a C++ library that provides BLAS (Basic Linear Algebra Sub-programs) functionality for a variety of different matrix types. Building on BLAS Fortran library, *uBLAS* is designed with performance in mind (especially with the goal of being no worse than its Fortran predecessors) and focuses on linear algebra operations and matrix data types. The library supports a variety of matrix types (including dense, packed and sparse matrices) but does not generalize at all to larger dimensions.

The Blitz++ library [15] is an older library that provides an n-dimensional array class, complete with slicing. The API focuses on the array classes itself, and does not

offer a collection of algorithms, or interpolation aids with visualization systems or other libraries. The development of Blitz++ has slowed after a decade, and has switched to a maintenance mode without reaching a seminal 1.0 version.

Although there are numerous implementations of n-dimensional array classes, algorithm developers and users often need not be too concerned with over committing or being locked into one particular implementation. Due to the large influence of the C++ STL on the various library developers, there are only a small set of element access styles that are used. Many also offer raw C-array like access to ease interfacing with other libraries. Using simple adapter classes or systematic source code refactoring, developers may quickly update their code to work with any new libraries.

4.2.2 The nindex Class

The core Scopira array type **`narray`** uses an **`nindex`** type to generalize arrays to any dimension. This **`nindex`** type can be thought of as the coordinates or reference of an element in an array. This is a template type that is generalized by the dimension only (it does not specify the element type). For example, **`nindex<2>`** is a 2-dimensional array index (matrix) and contains two values, *x* and *y*. Similarly, **`nindex<1>`** only contains the *x* value, and **`nindex<3>`** contains *x*, *y,* and *z* values. Internally, these are generalized to small, non-resizeable arrays with specialization for the first few dimensions. In addition to storing the coordinate values, this class provides operations that are needed in building a generalized array type such as returning the product of all

the values or calculating stride arrays.

4.2.3 The narray Class

After defining the **nindex** concept, building basic **narray** array types becomes relatively straightforward. A simplified definition of **narray** is:

```
template <class T, int DIM> class narray {
    T* dm_ary;           // actual array elements
    nindex<DIM> dm_size; // dimension sizes

    T get(nindex<DIM> c) const {
        assert(c<dm_size);
        return dm_ary[dm_size.offset(c)];
    }
}
```

From this code snippet we can see that an **narray** is a template class with two compile time parameters: *T*, the element data type (**int**, **float**, etc.) and *DIM*, the number of dimensions (1, 2, 3, etc.). The actual elements are stored in a dynamically allocated C "array", **dm_ary**. The dimension lengths are stored in an **nindex** type, building on that generalization.

A generalized accessor is provided, which uses the **nindex**-offset method to convert the dimension specific index and size of the array into an offset into the C array. This generalization works for any dimension size.

Another feature shown here is the use of C's **assert** macro to check the validity of the supplied index. This boundary check verifies that index is indeed valid otherwise failing and terminating the program while alerting the user. This check greatly helps the programmer during the development and testing of applications, and during a high

performance/optimized build of the application, these macros are transparently removed, obviating any performance penalties from the final, deployed code.

More user friendly accessors (such as those taking an *x* value or an *x* and *y* value directly) are also provided. Finally, C++'s operator overloading facilities are used to override the bracket "[]" and parenthesis "()" operators to give the arrays a more succinct and natural feel, over explicit **`get`** and **`set`** method calls.

Although technically a violation of encapsulation in object-oriented design principles, the **`narray`** class provides an accessor to get at the internal C array. This access is invaluable when interfacing with other libraries or data structures, despite bypassing all the programming checks in **`narray`**.

4.2.4 The nslice Class

The **`nslice`** template class is a virtual n-dimensional array that is simply a *reference* to an **`narray`**. The class only contains dimension specification information and is easily copyable and passable as function parameters. Element access translates directly to element accesses in the host **`narray`**. An **`nslice`** must always be of the same numerical type as its "host" **`narray`**, but can have any dimensionality less than or equal to the host. This flexibility is very powerful; one could have a one-dimensional vector slice from a matrix, cube or five-dimensional array, for example. Matrix slices from volumes are also quite common (e.g. Figure 3). These sub slices can also span any of the dimensions/axes, something not possible with simple pointer arrays (for example,

matrix slices from a cube array need not follow the natural memory layout order of the array structure).

The **`nslice`** implementation is inspired by the STL's gslice_array types. That is, in addition to basic source **`nslice`** reference and dimension size information, the **`nslice`** contains an array of *strides*. These strides indicate how many raw array elements are between the user elements in the **`nslice`**.

Programmers who wish to write more general code, should use **`nslice`** in their interface. Not only is obtaining an **`nslice`** representation of an **`narray`** trivial (in terms of both use and performance), but code that uses **`nslice`** is able to operate on a wider variety of source arrays and sub slices.

For maximum flexibility, programmers should write their algorithms in a type-free manner using C++ templates and generic programming. The **`nslice`** type has the same "form" as an **`narray`**, that is, it has all the same accessor methods and other operations, including producing **`nslices`** of itself. Programmers can then use templates to allow their algorithms to take any **`narray`**-like "form" which includes **`nslice`**. These algorithms can also generalize the actual element type allowing them to be used on any precision real numbers or integers (if applicable) as needed.

For applications that take vectors or sequences of elements, the more general STL style begin/end iteration is encouraged. Both **`narray`** and **`nslice`** support this, as do the STL containers and countless other third party libraries.

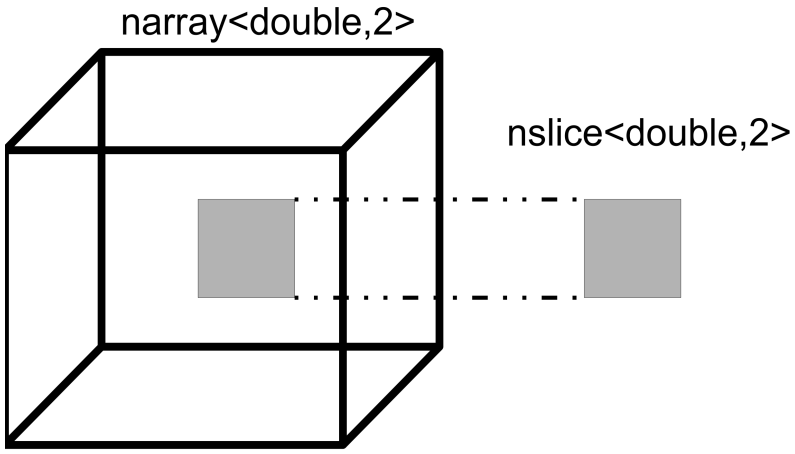

Figure 3: An nslice reference into an narray data set

4.2.5 Memory Mapping

The **`narray`** class provides hooks for alternate memory allocation systems. One such system is the *DirectIO* mapping system. Using the memory mapping facilities of the operating system (typically via the **`mmap`** function on POSIX systems), a disk file may be *mapped* into memory. When this memory space is accessed, the pages of the files are loaded into memory transparently. Writes to the memory region will result in writes to the file.

This allows files to be loaded in portions and on demand. The operating system will take care of loading and unloading the portions as needed. Files larger than the system's memory size can also be loaded – the operating system will keep only the working set portion of the array in memory. However, mapping files that are larger than physical memory must be done with care, programmers should still keep the working set within the memory size of the machine. If the working set exceeds the available

memory size, performance will suffer greatly as the operating system pages portions to and from disk (excessive juggling of disk-memory mapping is sometimes called “page thrashing”).

Furthermore, as the **`narray`** class is 64-bit clean, on 64-bit architectures very large files may be used as datasets and the operating system will page portions of the file into memory as needed. One caveat that large-dataset programmers must be aware of, however, is that if one element is accessed in the array, then the operating system will load that element's complete page from disk (each page is usually a few kilobytes). Slices that access many sparse elements will end up paging many sections to disk, ballooning the actual working space size of slice operations that do not follow the natural C array order.

4.3 Graphical User Interface Library

This subsystem provides a basic graphical API wrapped around GTK+ [62][110] and consists of widget and window classes that become the foundation for all GUI widgets in Scopira. More specialized and complex widgets, particularly useful to numerical computing and visualization, are also provided. This includes widgets useful for the display of matrices, 2D images, bar plots and line plots. Developers can use the basic GUI components provided to create more complex viewers for a particular application domain.

The Scopira graphical user interface subsystem provides useful user-interface

tools (widgets) for the construction of graphical, scientific applications, with particular focus on the biomedical research domain. A matrix/spreadsheet like widget is able to view and edit arrays (often, but not limited to matrices) of any size. This extensible widget is also able to operate on Scopira **`narrays`** natively. The widget supports advanced functionality such as bulk editing via an easy to use, stack based macro-language. This macro-language supports a variety of operations including setting, copying and filter selecting data within the array. A generic plotting widget allows the values of Scopira **`narrays`** to be plotted. The plotter supports a variety of plotting styles and criteria, and the user-interface allows for zooming, panning and other user customizations of the plot. An image viewer allows fully zooming, panning and scaling of **`narrays`**, useful for the display of image data. The viewer supports arbitrary colour mapping, includes a legend display and supports a tiled view for displaying a collection of many images simultaneously. Miscellaneous widgets such as a "joystick" control (that permits discrete, cardinal direction panning), VCR buttons (that present "play," "pause," etc. type buttons) and a random seed editor are also provided. A simplified drawing canvas interface is included that permits developers to quickly and easily build their own custom widgets. Finally, Scopira provides a *Scopira Lab* facility to rapidly prototype and implement algorithms that need casual graphical output. Users code their algorithm as per usual, and a background thread handles the updating of the graphical subsystem and event loop.

4.3.1 Model-View Plugin Framework

Scopira provides an architecture for logically separating models (data types) and views (graphical widgets that present or operate on that data) in the application. This model-view relationship is then registered at runtime. At runtime, Scopira pairs the compatible models and views for presentation to the user. A collection of utility classes for the easy registration of typical objects types such as data models and views are provided. This registration mechanism succeeds regardless of how the code was loaded; be it as part of the application, as a linked code library, or as an external plug-in.

Third parties can easily extend a Scopira application that uses models and views extensively. Third party developers need only register new views on the existing data models in an application, then load their plug-in along side the application to immediately add new functionality to the application. The open source C++ image processing and registration library ITK [50][54] has been successfully integrated into Scopira applications at run time using the registration subsystem.

A *model* is defined as an object that contains data and is able to be *monitored* by *views*. A *view* is an object that is able to bind to and listen to a model. Typically, views are graphical in nature, but in Scopira non-graphical views are also possible. A *project* is a specialized model that may contain a collection of models and organize them in a hierarchical fashion. Full graphical Scopira applications are typically project-oriented, allowing the user to easily work with many data models in a collective manner. A basic project-based application framework is provided for developers to quickly build GUI

applications using *models* and *views*.

4.3.2 3D Visualization

A complementary subsystem provides the base OpenGL-enabled widget class that uses the GTKGLExt library [43]. The GTKGLExt library enables GTK+ based applications to use OpenGL for 2D and 3D visualization. Scopira developers can use this system to build 3D visualization views and widgets, which allows for enhanced data exploration and processing. Integration with more complete visualization packages such as VTK [93] [116] is also possible.

4.4 Applications

Several biomedical data analysis applications have been implemented using Scopira [66][79][81][82][100][101][102]. Some are in-house, proprietary, and highly specialized systems, while others are open source applications that are available to the biomedical research community at large. These applications run the gamut from confirmatory to exploratory data analysis, image processing, pattern recognition, classification, and visualization. We briefly present three applications developed using Scopira. As this thesis work uses Scopira, this demonstrates possible types of applications that could benefit from this work.

One Scopira-based application is EvIdent® [79], an exploratory data analysis system for rapidly investigating novel events in a set of two- or three-dimensional images (e.g. MRI, infrared, spectroscopic maps, etc.) as they evolve over time or

frequency (or any other analysis dimension). For instance, in a series of functional magnetic resonance neuroimages, novelty may manifest itself as neural activations over a time course (Figure 4). The core of the system is an enhanced variant of the fuzzy c-means clustering algorithm [13]. Fuzzy clustering obviates the need for models of the underlying requisite biological function, models that are often statistically suspect. EvIdent® offers several innovations: (*i*) biomedical researchers may probe for unanticipated but domain-significant structure in the data; (*ii*) flexible generation of unbiased, testable models; (*iii*) rapid analysis of data in complex cognitive experiments; and (*iv*) excellent precursor and complement to any model-based inferential method.

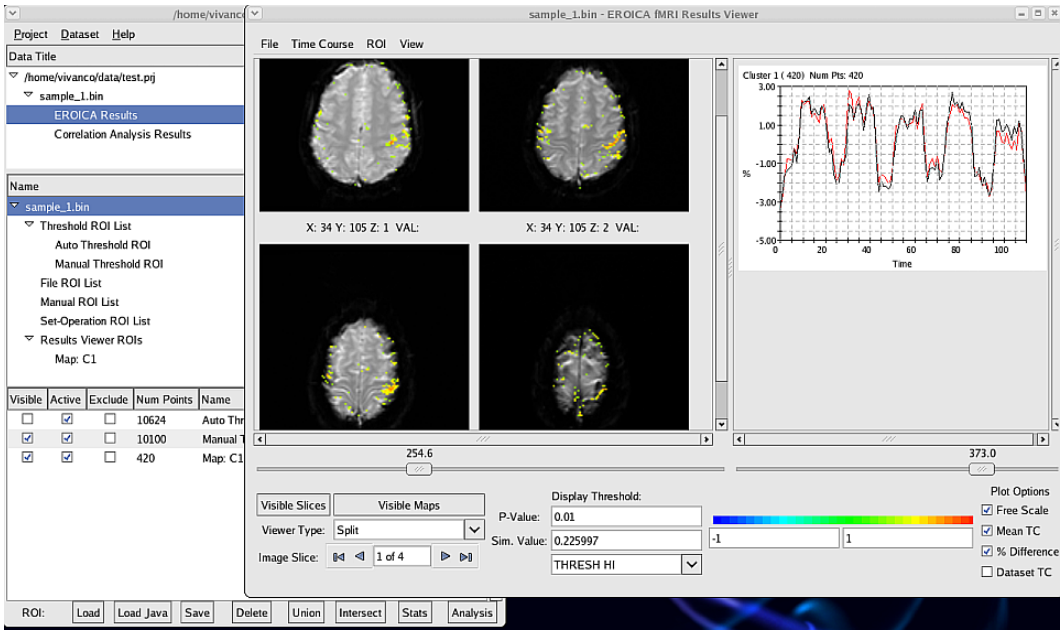

Figure 4: Functional MRI activation map viewer in EvIdent®

Visualizing high dimensional patterns and their relative relationships, is a useful and challenging technique that is important in data exploration and confirmation. A Scopira-based application was developed to implement a new projection strategy, the Relative Distance Plane (RDP) [66][100][101] which uses a similarity-based mapping requiring only a single computation of a distance matrix, for the visualization of high dimensional patterns and their relative relationships. RDP allows an investigator to visually inspect (Figure 5) datasets for anomalies prior to subsequent analysis (e.g. classification, regression, clusters). An important aspect of RDP is that certain distances are exactly preserved in a new 2D (or 3D) coordinate system. Give two (or three, in the 3D case) reference patterns selected from the dataset, all other patterns are displayed without any distortion of their original relative distance to the reference patterns. RDP is a projection pursuit variant using directions defined by pairs (or triplets) of patterns from the dataset.

Another Scopira-based application involves the analysis, visualization (via Scopira and VTK [93][116]), and interpretation of biomedical images using optical coherence tomography (OCT) [48], an optical imaging modality that provides micrometer scale resolution morphological images. OCT is similar to ultrasound in operation except that low coherent near infrared light is used instead of sound. The light is focused onto a sample and back reflections from within the sample are recorded to create a morphological image of the interior structure of the sample. The back reflections occur from changes in optical density at tissue boundaries and cellular

structures. The three dimensional morphological images have an axial resolution of 10 µm and a transverse resolution of 25 µm that is superior to standard ultrasound images. The coherence requirement of OCT in highly scattering biological tissue limits penetration depths to 2 mm. However, the method is fully implemented in fiber optics, allowing sub-millimetre probes to collect images via catheters and endoscopes [18].

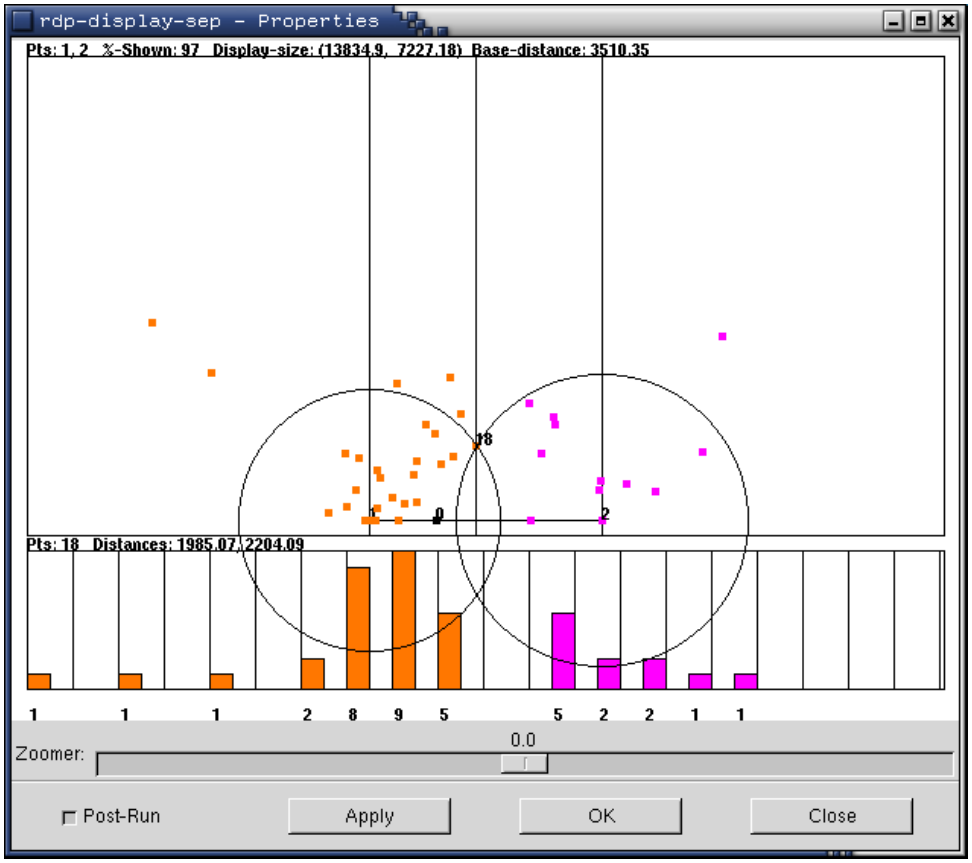

Figure 5: RDP Separation Display

The elevation and transfer of skin flaps is essential in reconstructive surgery. Clinical prediction of eventual tissue viability at the time of elevation can be inaccurate and lead to reconstructive failure. A common example is that of mastectomy skin flap necrosis in the setting of immediate breast reconstruction. A Scopira-based application was made to help specialists delineate demarcation lines that separate dead and viable skin areas for further processing [102].

5 Design

5.1 Overview and Goals

This chapter presents the design overview and goals of the work.

5.1.1 Relation to Scopira

The *Scopira Agents Library* (SAL) is the name given to the message passing library that is the result of this work. *Agents*, in this context, refer to the objects that manage groups of SAL-tasks, and has no relation to agent-based computing (e.g. [75]). However, despite sharing a name with the Scopira library itself, SAL is a separate and library. To underscore, Scopira is a general library for application development while SAL is the message passing library that is the result of the research in this thesis.

SAL does use the Scopira library for certain general functions such as file and

network I/O and object serialization, and also has similar API styles and structures. However, SAL's core concepts and its implementation are independent of Scopira and SAL could be made to use any other library for these basic facilities.

5.1.2 Goals and Limitations

SAL's goals are to be an easier-to-use and deploy message passing library with adequate performance for a variety of use-cases. The target audience for such a library includes interactive application developers and parallel algorithm developers with moderate performance and scalability needs. The algorithms should have moderate communication needs, that is, overall algorithm performance should not be highly sensitive to messaging throughput or latency.

Interactive (for example, GUI or Web) application developers (with new or existing application code bases) that wish to utilize parallel processing in their applications, quickly and seamlessly, would ideally use SAL. Their applications would retain the same ease-of-use yet still be able to utilize multi-processor and (if detected) multi-host parallelism, increasing performance without application complexity.

Parallel algorithm developers who have moderate performance and scalability requirements may choose to use SAL for its ease of use and ability to quickly make parallel applications. Utilizing SAL also gives these developers the option of embedding their algorithms into deployable applications later on, if desired.

SAL's advantages are of course not without their trade-offs. By design and

implementation, SAL may be less efficient and less scalable than other libraries. For its target audience, these sacrifices are acceptable, however, for some users other options may be preferable. For example, SAL is not designed for grid computing and communication intensive algorithms. SAL also does not, in its current implementation, utilize specialized communication hardware or protocols.

SAL's object-oriented design, error checking and buffering makes SAL have higher CPU and memory overhead than other optimized libraries, resulting in lower communication throughput and higher latency. SAL's current direct message routing implementation and simplistic API limits scalability, making it ill-suited for grid computing or similar large-scale applications.

5.1.3 Implementation Goals

SAL, by design, borrows a variety of concepts from both MPI and PVM. SAL, like PVM, attempts to a build a unified and scalable “task” management system, with an emphasis on dynamic resource management and interoperability. The tasks themselves are coupled with a powerful message passing API inspired by MPI. Unlike PVM, SAL also focuses on ease-of-use: emphasizing automatic configuration detection and deemphasizing the need for infrastructure processes. Using operating system threads and C++ objects, SAL emphasizes multi-programming within single OS processes (which are fastest for same-host communication) and embedding: providing the complete implementation with the library (and thereby, the application). Applications

always have an implementation of SAL available, regardless of, or the availability or access to, cluster resources.

SAL introduces high-performance computing to a wider audience of users by permitting developers to build standard cluster capabilities into desktop applications, allowing those applications to pool their own, as well as cluster resources. This is in contrast to the goals of MPI (providing a dedicated and fast communications API standard for clusters) and PVM (providing a virtual machine architecture among a variety of powerful platforms).

SAL extends the core Scopira C++ library (Figure 6). It provides everything needed for developers to make cluster-aware applications, including a message passing API, implementations of this API and a host of services and other facilities. Developers may use SAL to make their Scopira applications multi-processor and cluster-aware. Although SAL development activities and research is ongoing, the core components have been used in a production environment.

In SAL terminology, an "agent" refers to the "task"-managing engine in the library that represents a node in the agent network. Tasks, as in PVM, are individual processing entities within the system that have their own identifier and message passing abilities. This agent object is the key broker between the application code, task processes and the agent network. An agent delegates the actual task management and message passing responsibilities to an internal "engine" object. The specific engine implementation is chosen at application startup and can be based on user preferences

(for example, the user may choose to not use an available cluster) and the local network configuration. The engines can differ by the services provided and by scheduling policies. Although only two engines (a single-host and network-enabled multi-host engine) are initially provided, additional engines (e.g. decentralized network topologies) may be added in the future.

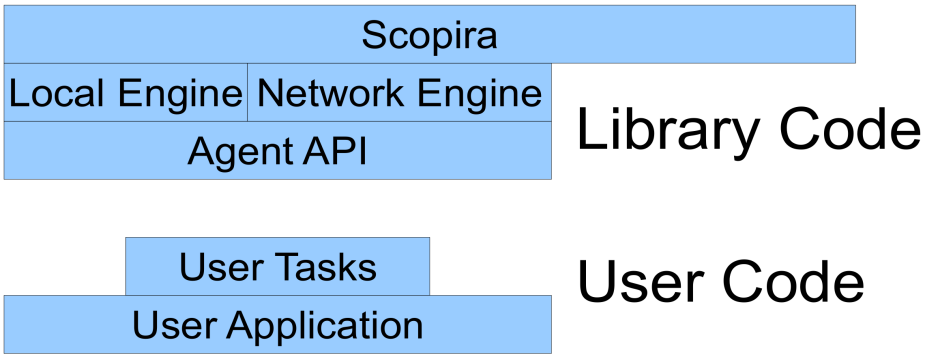

Figure 6: The SAL API Stack

5.2 Messaging API

SAL provides an object-oriented, packet based and routable API for message passing (like PVM, but unlike MPI). This API provides everything needed to build multi-threaded, cluster-aware algorithms embeddable in their applications.

The API uses a few key object-concepts to form the API stack. This API stack contains the following objects: tasks (algorithm processes), contexts (a collection of methods that a task uses to communicate with other tasks) and **`send_msg`** objects (corresponds to a single messaging transaction). An overview of how these objects

interact is as follows:

- The SAL engine chooses a user-task to run (usually from a queue of tasks)
- The user's task object has its **`run`** method called and is passed a context object
- The task uses this context to create **`send_msg`** objects (a complimentary object, a **`recv_msg`** object, must be used to receive such messages).

This is illustrated in Figure 7:

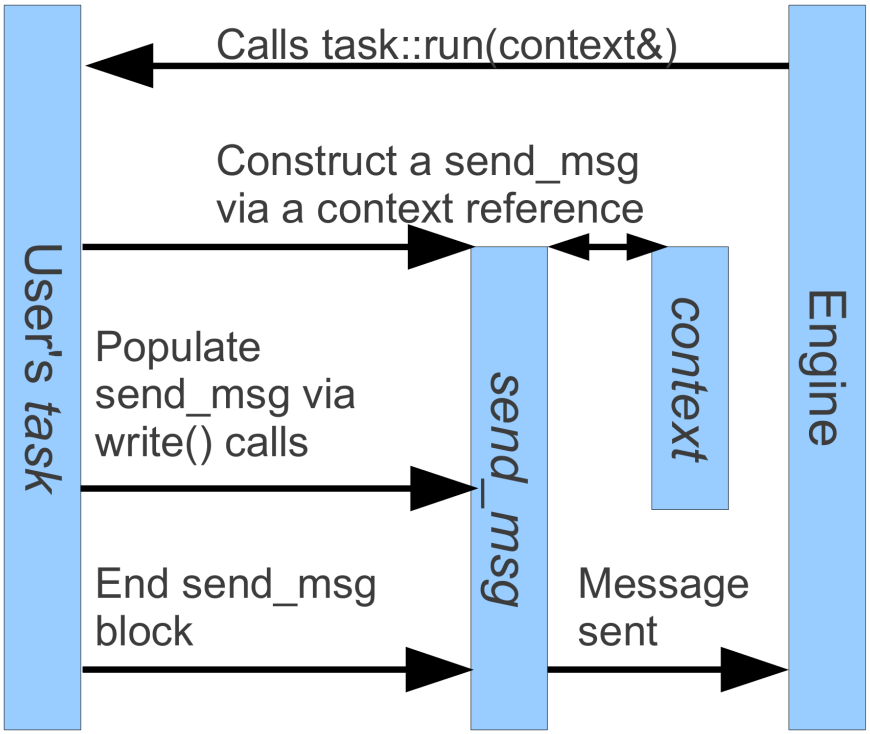

Figure 7: A typical call sequence (proceeds from top to bottom)

5.2.1 Tasks

Tasks are the core objects that developers build for the SAL system. A task represents a single job or instance in the agent system, which is analogous to a process in an operating system. However, they are almost never separate processes, but rather grouped into one or more agent processes that are embedded into the host application. This is unlike most existing parallel APIs, that allocate one OS process per task concept, which, although conceptually simpler for the programmer, incurs more communication and startup overhead, and is OS dependent. The tasks themselves are language-level objects but are usually assigned their own operating system threads to achieve preemptive concurrency.

Tasks have the following features and properties:

- Logic initialization and shutdown in their constructor and destructor (as with any C++ object);
- A core "run" method, which is the central method that is called when a task should perform its work.
- User specific methods and state variables.

The developer's focus is primarily with the task's run method. It is this method that is passed a reference to a "context" object, which provides access to the core messaging API. The run method signals the agent system its result code (e.g. whether it is done or should be run again), by returning integer code. Possible signals include:

process completion, process yielding, sleep until message arrival and sleep for a time period. The API in detail:

```
// interface and parent class of all tasks
class agent_task_i {
     // main run method in the task interface
     int run(task_context &ctx);
};
```

5.2.2 Context Interface

A context object is a task's gateway into the SAL message passing system. There may be many tasks within one process and each will have a different context interface – something not feasible with an API with a single, one-task-per-process model (as used in PVM or MPI). Being able to embed all the tasks as threads in one process is vital for application embeddability, a core goal of SAL. This class provides several facilities, including: task creation and monitoring; sending, checking and receiving messages; service registration; and group management. It is the core interface a developer must use to build parallel applications with SAL. A selected, annotated API list is shown:

```
class task_context {
     // returns the number of CPUs is the system(s)
     int universe_size(void);
     // returns this task's UUID
     uuid get_agent_id(void);
     // spawn sub tasks and form a task group
     uuid launch_group(int num_processes);
     // is the task with the given ID still “alive”
     bool is_alive_task(uuid taskid);
     // is there a pending message from the task with the
given ID
     bool has_msg(uuid taskid)
     // get this task's group id/index (when in a group)
     int get_index(void);
     // get the size of the group (when in a group)
     int get_group_size(void);
```

```
        // a barrier synchronization call, for a group
        void barrier_group(void);
        // wait until all others in my group terminate
        void wait_group(void);
};
```

Data is actually sent and received via **send_msg** and **recv_msg** objects, which take a context object as a parameter during their construction. These objects are outlined in the next section.

Central to the messaging system in SAL is the concept of *Universally Unique Identifiers* or UUIDs (sometimes known as GUIDs). UUIDs are 128-bit integer identifiers that can be considered, for all practical purposes, to be "unique within all keys in the universe." More importantly, they need not be centrally generated or managed, allowing for distributed systems to generate UUIDs without a central source, yet still be reasonably confident that keys may intermix. These are analogous to PVM's Task IDs (TIDs) in concept, but differ in implementation. In PVM, a PVM server encodes its sequence ID within all the task IDs it generates, a sequence number it does not know until the master server assigns it upon joining the virtual machine. Within SAL, UUIDs may be generated at anytime.

UUIDs have wide appeal and are used in a variety of systems, from distributed software systems to OS level services. The concept of universally unique identifiers that can be generated in a distributed fashion is powerful and applicable to many domains and problems. To generate these IDs, developers first employed the technique of hashing various machine characteristics (such as a network machine address (MACs),

Internet address (IPs), etc.) and combined with a time stamp and a random number. Over time, privacy concerns over the traceability of UUIDs containing MACs or IPs lead to the use of strong random number generation facilities in many operating systems. The operating system monitors a variety of random events in the system, such as mouse movement or network noise to build an entropy pool from which strong random numbers can be made.

In SAL, all objects such as agents and tasks have associated UUIDs. Tasks can then publish and share this ID with other tasks or with the user. UUIDs in Scopira are represented as small, convenient, opaque C++ objects that can be manipulated, compared and stored, similar to primitive data types in the language.

Developers often launch a group of related instances simultaneously, and then systematically partition the problem space for parallel processing. To support this popular paradigm of development, SAL's identification system supports the concept of *groups*. A group is simply a collection of N task instances where each instance has a $groupid \in [0,N\text{-}1]$. The group concept is analogous to MPI's communicators (albeit without support for complex topologies) and PVM's named groups. This sequential numbering of task instances allows the developer to easily map problem work units to tasks. Similar to how PVM's group facility supplements the TID concept, SAL groups built upon the UUID system, as each task still retains – and may use – their underlying UUID for identification.

5.2.3 Message Sending Objects

In SAL, the sending and receiving of data is done via dedicated **`send_msg`** and **`recv_msg`** objects, which utilize the context interface to perform their work. It is these objects that have a collection of writing and reading methods for sending data over the network, not the context object itself. These objects reuse the underlying Scopira serialization system, allowing the developer to reuse their object serialization code for both SAL and for regular file I/O.

For example, the **`send_msg`** class itself does its specific work in its constructor (setting up its destination) and destructor (actually sending the data). All the writing method implementations are reused from Scopira, specifically the **`bin64oflow`** class, which implements the method in the **`otflow`** interface. It is to this interface that object-serialization code is written too. A selected, annotated API of **`send_msg`** follows:

```
class send_msg {
    send_msg(task_context &ctx, int destination);
    // destructor, does the transfer via RAII:
    ~send_msg();
    // inherited type serialization methods
    void write_bool(bool);
    void write_char(char);
    void write_short(short);
    void write_int(int);
    void write_size_t(size_t);
    void write_int64_t(int64_t);
    void write_long(long);
    void write_float(float);
    void write_double(double);
    void write_string(const std::string &);
    template <class T> void write_generic(const T &);
    void write_bool(bool);
    size_t write(const byte_t *, size_t);
    size_t write_byte(byte_t);
```

```
        template <class T>
            size_t write_array(const T*, size_t);
        size_t write_void(const void *, size_t);
    }
```

Any object may be sent in a type-safe manner, from basic primitive variable types to compound objects. Unlike MPI (and similar) message passing interfaces, this decidedly object-oriented design provides send and receive functions that are usable at any time, outside of any transactions. This design has various benefits:

- **Serialization**: Any data types or objects may be sent in a type-safe manner, drastically reducing programmer errors. This reuses the powerful serialization mechanism in Scopira (Section 4.1.2), enabling programmers to reuse their serialization-compatible objects for other tasks.
- **Packets**: Data is transparently collected, grouped and sent in discrete packets, which simplifies programming and debugging.
- **Scoped transactions**: Utilizing RAII (Section 3.4), packet sending and receiving are done via dedicated code blocks. In particular, a **`send_msg`** object is constructed at the start of the scope and then populated with data. When execution leaves the dedicated scope block, the **`send_msg`** object's destructor is called, triggering the actual sending of the message. This has all the typical benefits of an RAII application: the user does not need to remember to call explicit commit-like methods, and may exit the scope in a variety of ways (for example, via a **`return`** or **`break`** statement).

These concepts are best illustrated with a short code example. The following

annotated code snippet demonstrates a simple task's run method that sends some data. Figure 7 provides a visual illustration of the various objects and their interactions:

```
int my_task::run(task_context &ctx)
{
     narray<double, 2> a_matrix;  // a matrix of doubles

     // the following is a messaging sending block
     {
          // construct the message object
          send_msg M(ctx, 0);

          // write a basic integer
          M.write_int(100);
          // write a whole object, in a type-safe manner
          // no need to specify array length or type
          a_matrix.save(M);
     }    // message is sent as execution leaves this scope

     // at this point the data is sent and another message
     // transaction can begin
}
```

5.2.4 Task Creation and Monitoring

Tasks may spawn or launch other tasks (Figure 8). In the basic case, one task is spawned, which is useful for client-server pairings or when one task “calls” (and expects an answer from) another task (or tasks) to perform a certain computation. As in PVM, there is no rigid relationship between tasks, allowing this flexible mechanism to be used to build a variety of systems. Finally, tasks in SAL are language-level objects, requiring the creating task to specify the C++ class names of the new tasks. In PVM and similar systems that model tasks around OS processes and applications, the caller would have to specify file path names to actual programs, a value that would vary by OS and by installation.

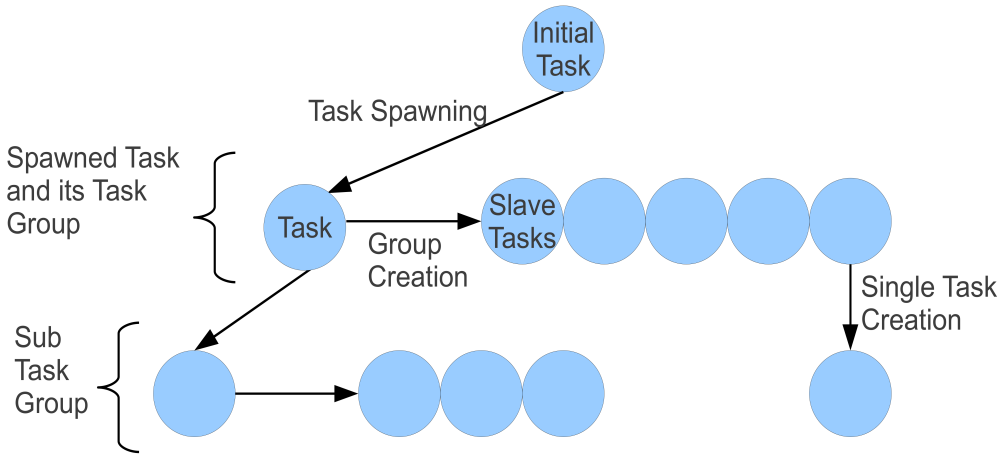

Figure 8: Example of nested task group spawning and communication

Groups of tasks may also be launched. A task group is simply a collection of task instances of the same task type that can refer to each other via sequential IDs as well as UUIDs. This permits the developer to use simpler notation when partitioning a problem space into parallel processes.

In all cases, the context interface allows any task to monitor the lifetime of another process. A task may also request that another task be interrupted and destroyed – however SAL can only do this between a task's run calls as thread cancellation is usually neither safe nor portable.

The task launching mechanism, combined with the scalable UUID-based identification system, permits the construction of a variety of communication topologies. For example, each task within a task group can also spawn its own group,

creating large processes hierarchies. Another situation includes coordinator-like processes that can orchestrate a collection of groups and other processes, basically facilitating disjoint tasks to perform a greater goal. Certain tasks may be persistent or server-like, providing standard services, such as storage or random number generation, to new tasks. All these options present a certain dynamic flexibility within the system, where tasks live, die and spawn within the system, while being members of a global, universally addressable messaging universe.

5.2.5 Messaging

The messaging system in SAL is built on both the generic Scopira I/O layer as well as the UUID identification system. SAL employs a packet-based (similar to PVM) message system, where the system only sends and routes complete messages, and not the individual data primitives (as MPI can and often does) and objects within them. Only after the sending task completes and commits a message is it processed by the routing and delivery systems. The SAL agent uses OS threads to transport the data, freeing the user's thread to continue to work. In contrast, MPI users that wish to utilize overlapping IO require an implementation that specifically supports it, such as USFMPI [22] (this can be somewhat emulated in standard MPI by using non-blocking functions).

The Scopira I/O system (from which the message system API is based) uses a three level object-oriented system for data serialization (the process of converting objects to a stream of bits).

At the bottom level of this system is the binary interface and its various implementations. Fundamentally, this level has two types of implementations, “sinks” or final stream terminators and binary filters. A sink begins (or ends) any stream chain by taking the data out of or putting it into the stream system. Examples include files, network sockets, and memory blocks. A filter simply converts one binary stream to another, for example a cryptographic cypher, or a lossless data compressor.

The second interface level introduces the concepts of primitive types to the I/O interface. This interface presents various methods for writing and reading a variety of primitive types (such as integers and strings) and converting them into binary streams. Implementations include an ASCII-representation converter (useful for debugging), a compact binary converter and a binary converter that always stores data in 64-bit format (useful for 32-bit and 64-bit interoperability).

The final interface level builds on the primitive type API and adds full object serialization. This allows any object that implements the serialization interface to be written to an I/O stream. The Scopira object serialization implementation includes support for object-caching and reference bookkeeping, which allows objects that have multiple references to be correctly serialized.

To send data packets with SAL, the task instantiates a **`send_msg`** object. The sender provides the destination task(s) either by UUID, group index number, or a broadcast flag. The **`send_msg`** message packet is then populated with data via its “type-level” standard Scopira I/O interface. Finally, only when the message object is starting

the process of its destruction, are its contents sent to the routing system.

Sending (committing) the data during the **`send_msg`** object's destruction (that is, via its destructor) was the result of an intentional design decision. In C++, stack objects are destroyed as they exit scope. The user should therefore place a **`send_msg`** object in its own set of scope-braces, which would constitute a sort of "send block". All data transmissions for the message would be done within that block, and the programmer can then be assured that the message will be sent at the end of the scope block without having to remember to do a manual send commit operation.

Similarly, the receiver uses a **`recv_msg`** object to receive, decode and parse a message packet, all within a braced "receive block."

Finally, the message system includes a complete recipient and polling API. The programmer may specify a filter for incoming messages from a specific sender, any task, any task within the same group, or a more complex specification using a basic boolean logic based query expression. Recipients may also poll for messages rather than block waiting for them, allowing for concurrent processing and error condition checking.

5.2.6 Services

The SAL system permits users to build services: facilities that are provided by persistent or long-running tasks within the system. SAL enables tasks to register themselves as service providers and provides configurable searching facilities so that

other providers may be discovered. Services can provide any number of functions, such as random number generation, centralized data set storage, task management services, etc. Their persistence between client-task runs makes them useful for a variety of domains. As tasks are lightweight processes (namely threads) rather than OS processes (as they are in PVM), services such as a name server (which is built into PVM itself) are implemented as service-providing tasks within SAL.

A task is said to provide some well known user-defined service if it supports that service's messaging protocol. SAL may find service providers on behalf of a task, but after the initial introduction, the initiator must then further probe the resulting tasks for more specific information. Services are a protocol level contract rather than a new interface or type and, as such, SAL itself cannot verify or enforce the completeness of any task's service implementation.

5.3 Scheduling Engines

The SAL scheduling engines implements the SAL API. The engines are responsible for task management, message transport and processor management. SAL currently includes two types of engines, a “local” engine that uses operating system threads on a single host machine and a “network” implementation that is able to utilize a network of workstations. The network engine is a functional superset of the local engine.

5.3.1 Local Engine

The “local” engine is a basic multi-threaded implementation of the SAL API that is embedded completely within the user's application process (Figure 9). It uses the operating system's threads to implement multiprocessing within the host application process. The engine lacks the networking abilities to manage separate nodes and intercommunication but is able to use all the processing cores on the host machine by using operating system threads within the host process.

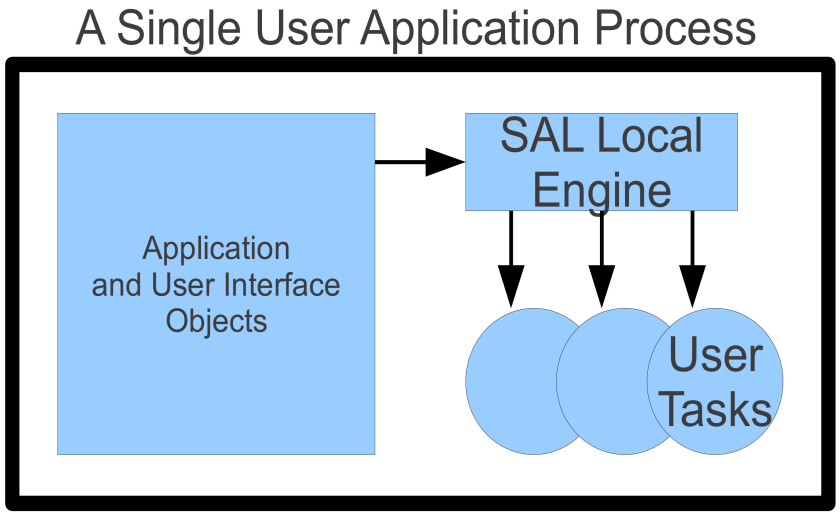

Figure 9: Embedded local-engine in a user application process

As this engine is contained within a single-process, it is the fastest to use for application development and debugging. Using the local engine, the programmer may fully design and test their parallel algorithm and its messaging logic before moving to a multi-node deployment. Furthermore, as multi-processor and multi-core desktop

systems become more commonplace, this basic engine is perfectly suited for single-host deployments and users who may not need full cluster resources. The local engine is always available and requires no configuration from the user. Developers need not write dedicated non-message passing versions of their algorithms simply to satisfy users that may not want to go to the trouble of deploying a cluster.

The implementation of the local engine is relatively straightforward, as it is contained within the host application process. The engine itself protects all the administrative information in a mutually exclusively accessed area, protected by thread "mutex" primitives. The engine maintains:

- A list of "worker" threads and their operating state;
- The next task to check for processing;
- A mapping of service UUIDs to tasks that implement specific services;
- A process table of tasks, indexed by their UUIDs. For each task, the engine maintains the current running state, group peers (if any), and the incoming message queue.

Task instantiation is straightforward. A new process entry is created and associated with a task with a running state set to *ready*. The engine then adds new worker threads to the thread pool, maintaining at least as many worker threads as active tasks. Finally, the thread pool is notified, so that an idle worker thread selects and runs the new task. The thread pool size is never reduced, only increased – this is done to

prevent resource intensive thread creation/destruction cycles. While for many process communication patterns there may exist some optimal thread pool size that is less than the number of active processes, finding this number is non-trivial, as the engine would have to determine if blocked tasks are waiting for stopped tasks that require new work threads. For example, as soon as any task waits for a message from another task, the engine quickly deteriorates into the worst case: as many threads as active tasks.

Message routing is straightforward and similar to PVM. The engine wraps a small message header object around the sender's data packet and appends this directly to the destination task(s) event queue. Each event queue is itself protected by a thread mutex and notification condition object, so that the waiting task may immediately process the new data without a chance of thread conflicts (i.e. race condition). Each message header object treats its data payload in a read-only manner. This allows the various destination tasks for a broadcast message to share one copy of the data payload, greatly reducing memory duplication.

The local engine does no load balancing. As the engine provides as many worker threads as active tasks, it relies on the operating system's ability to manage threads within the processors. This works quite well, when the number of tasks instantiated into the system is a function of the number of physical processors, as encouraged by the API (via reasonable defaults). Since there is only one primary user/initiator in a local engine (that is, the host application's user), the number of task groups in the system is predictable (often, one).

In summary, I found the implementation of the local engine to be relativity straightforward. Without the complexities of network communication, the engine implementation itself is simply a collection of shared associative arrays with various levels of mutexes and conditions all shared by a group of worker operating system threads. This makes for a reasonable reference implementation of the API, useful for both debugging and for production deployments where the user's desktop machine is of sufficient processing power.

5.3.2 Network Engine

The network engine implements the SAL API over a collection of machines connected by an IP-based network; typically Ethernet. The cluster can be a dedicated compute cluster, a collection of user workstations, or a combination. The engine itself provides inter-node routing and management, leaving the local scheduling decisions within each node up to a local-engine derived manager. The network engine is a functional superset of the local engine, and uses the same local engine scheduling at the single host level.

The network engine implementation is a functional superset of the local engine. That is, the network engine also manages multiple tasks on a single host using operating system threads and basic message queuing. A single-host network deployment is functionally similar to a local engine deployment.

5.3.2.1 Topology

A SAL network stack has two layers (Figure 10). The agent transport layer contains the agents themselves (objects that manage all the tasks and administration on a single process) and their TCP/IP based links. The agents virtualize and present the messaging layer, where tasks can send messages to each other using their UUIDs, ignorant of the IP layer or the connection topology of the agents themselves. For simplicity and efficiency, a SAL network (like PVM) has a master agent residing on one process. This master agent, in addition to participating in compute activities, is responsible for the allocation, tracking and bookkeeping of all the tasks in the system. It is assumed that within a single site deployment of an SAL network, at least one stable server (i.e. non-user desktop) machine can be found to assume this role. This master role need not be deliberately assigned by the user – the first network based agent will automatically assume this role if no other master agent is found. A centralized master allows for simpler and faster task administration.

The network engine uses a combination of URL-like direct addressing and UDP/IP broadcast based auto-discovery in building the agent network. The simplest sequence is to start an application in *auto discovery mode*. When a network engine starts, it searches the local network for any other agent peers and, if found, joins their network. If no peers are found, then it starts a network consisting of itself as the only member and assumes the master agent role. Users may also specify the master's URL directly, connecting them explicitly to a particular network.

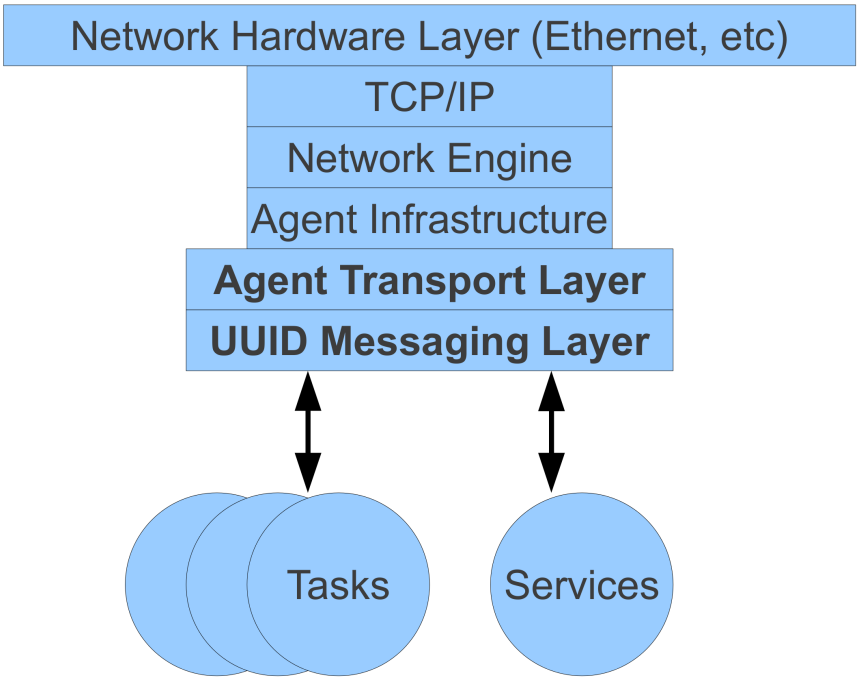

Figure 10: The SAL Network Stack

The Scopira package includes a SAL shell application that can be used in more specialized deployments. This application simply loads any external application modules and proceeds to join or create a SAL network. It is intended for system administrators and users who want to launch worker processes to which the desktop applications connect.

Agents within the system utilize two different routing policies as needed: fully connected agents that perform direct-routing and agents that proxy all their communication via a master agent. The former are usually dedicated compute nodes, which can be considered relatively stable and network connected, and would benefit

from increased speed and reduced latency by direct TCP/IP communication with other peers. The latter type – agents that communicate via the master – are more for nodes that are "unreliable" (worker agents deployed on user desktops, for example) or for desktop applications and graphical front ends that do not need the superior performance of direct communication. Routing through the master also permits communication when firewall or network issues (such as the cluster nodes being on their own private network) would otherwise limit communication.

The master agent is critical for message routing within the network. First, the master tracks the physical location of all the other agents within the system. Specifically, it is able to map an agent's UUID to their TCP/IP addresses, which is required for peers to do direct peer-to-peer routing. Finally, the master does proxy routing for agent nodes that are not doing direct routing. This design makes routing straightforward in the agent network, as all messages are sent directly to the peer or directly to the master agent (to which all agents always have a direction connection).

These flexible routing and deployment options permit a variety of different network topologies. Figure 11 contains a montage showing four types of routing topologies: dedicated compute cluster, desktops as cluster clients, ad-hoc desktop cluster and volunteer (idle time) computing.

The traditional compute cluster topology (Figure 11 (a)) has a collection of dedicated compute hosts (usually on a dedicated network switch or possibly other specialized communication hardware) directly connected for optimal performance.

Users connect to the cluster directly to submit and monitor running jobs.

A more seamless connection method involves users running SAL-based desktop applications (Figure 11 (b)) on their desktop hosts that automatically find and connect to an already running cluster master agent. The user's agent submits jobs to the master for computation, who then assigns sub-jobs to work nodes within its cluster. The user's agent itself usually does not do any computational work (and definitely not the work of other users who may also be connected to the same master agent) but simply monitors and retrieves results of running jobs. Finally, the user agent may disconnect from a running network and reconnect later, making long running jobs independent of the reliability of desktop clients.

Desktop users can also form their own ad-hoc compute cluster (Figure 11 (c)) by simply starting their SAL-enabled desktop applications. This is useful, for example, if the users lack the hardware resources for a dedicated compute cluster, lack the know-how or are mobile users. The first application instance will, upon not finding any other instances, start a new agent network with itself as the master. Subsequent application instances will connect to this master, building up the ad-hoc compute network. The users may now run jobs from their applications that will automatically be deployed on this ad-hoc cluster.

The ad-hoc desktop and dedicated cluster topologies are similar, in that all the nodes are (logically) interconnected. However, in practice, the ad-hoc cluster may be more dispersed, with many intermediate routers and switches separating the instances.

Volunteer-based computing (Figure 11 (d)) uses the idle processing time on desktop (and other hosts) at a site to perform computation work. A SAL-enabled application would be installed (either by the user, or site-wide by a system administrator) to run at start-up with a low scheduling priority (so as to not interfere with the host's regular duties). The application would find and connect to the local dedicated master and begin requesting and processing work. This topology effectively gives a site free computing resources, as it harnesses otherwise wasted processor cycles.

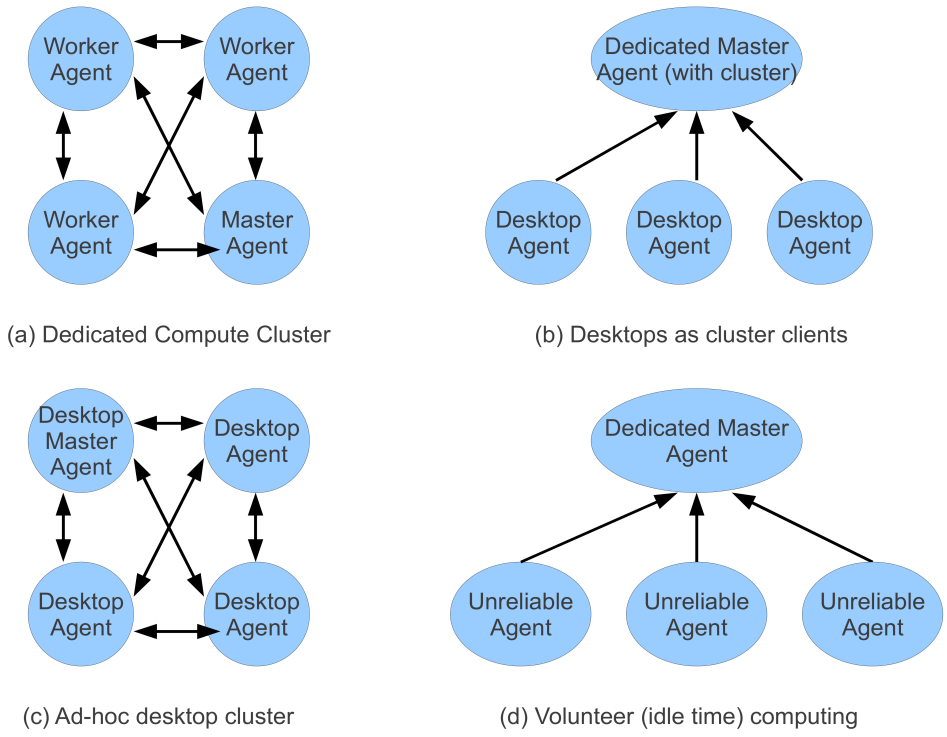

Figure 11: A sample of possible network deployment topologies

Finally, the same SAL-enabled application may be run in all these topologies giving the end-user flexibility in deciding how they would like to deploy their

processing tasks. The topologies can also be combined, further adding to deployment options.

5.3.2.1 Task Management

In addition to its critical routing functions, the master agent is also responsible for all the task tracking and management within the network. By centralizing this information, load and resource allocation decisions can be made instantly and decisively.

For each agent peer, the master tracks its load, routing policy (direct or indirect) and task running policy. Specifically, each agent is able to specify what types of jobs it is willing to accept: all jobs, no jobs (useful for desktop nodes or front end nodes) or only self-initiated jobs (for agents that are present only for their own jobs).

For each task, the master tracks its network location (on which agent it resides) and its running state. The master gives this information to slave agents on demand, as it is required for direct message routing. The slaves then cache the location information, greatly reducing unnecessary redundant requests while occasionally flushing or updating their caches as needed. The master's records are definitive and always reflect the real state of the network.

All task instantiation requests are handled by the master agent. When a task within an agent requests the creation of more tasks, the request is routed by the hosting agent to the master agent. Based on the current loads and hosting policies of the various non-master agents, the master relays the request to the chosen agents. The agents then create

the actual tasks, report back to the master, which in turn reports back to the initiating task's agent and task.

The master agents use the number of active tasks on each agent as the primary metric of processor load. This tactic is sufficient for most scenarios; however for more compute loaded systems (that is, systems with other programs and users), a more sophisticated allocation policy will be required. In particular, examining the operating system reported system load may give a better picture of the load of the machine. Of course, the agents would then be susceptible to the operating system's metrics, its variances and any reporting anomalies, but it would factor in other applications running on the machine, as well as appropriately rate threads and processes that are not processor bound (as opposed to disk or network bound).

5.4 Sample Services

Services or service tasks within SAL are tasks that provide well known functions and services to other tasks. These services are typically persistent (much like a server process in an operating system) and wait patiently to process requests from client tasks. They may be started at network boot time or demand-loaded as needed. The tasks themselves receive no special treatment nor use any special APIs; they are normal tasks within the agent system. An agent is defined by the services it provides via a well known and published messaging protocol. Service providers may be application-specific or general utility function providers.

Service providers may play a variety of roles. A *monitor service* allows tasks to register themselves as monitors of other task(s), either being notified or perhaps killed when the watched tasks terminates. This service forms the basis for fault tolerant computing, providing cleanup services for when key tasks within a job abruptly terminate. An *administration service* can provide the basic functionality needed for general system monitoring and administration. Client tasks can perform automated, routine maintenance as well as present this information to the user, both graphically and in a report. A *job manager service* (where "job" refers to a collection of cooperative tasks) is used to track user-visible jobs in the system. This allows a user to "detach" or disconnect their client application from the agent system and leave their jobs running unattended. Upon return, the user is presented with a list of jobs (and their completion states). The user then resumes interacting with a selected job. Specific devices, instruments, and license-limited software can be accessed through a *representative service*. This allows a unique resource to be protected and managed by a sole process, to whom all tasks must submit requests. For specific applications, *pseudo-random number generation* may also be centralized. This allows job reproducibility (critical for algorithm testing, development and scientific publishing), as a distributed set of tasks must still contact a single, managing source for their random number sequences. Finally, a *file or data set service* may provide centralized access to data files. This may be done for ease of use (consolidation of all the files into one name space), access control or simply because the files are only available at fixed agents/hosts (this is particularly

useful for cluster configurations without a shared file system). Arbitrary user authentication and access control may also be implemented to further refine the files available to a particular task or job set.

5.5 Deployment

SAL is designed to allow developers to make parallel applications that require no special configuration from end users. This is vital to the goal of making parallel applications easy to use. This is achieved using a variety of techniques. First, the complete messaging and routing engine is included in the programming library itself, and is thereby embedded in the application. Users do not need to install and configure additional framework or infrastructure software. Users without cluster resources can still use the always-available local engine, which provides parallelization through operating system threads. This vastly increases an application's potential user base by lowering the requirements to install and run the software, specifically, the need for cluster hardware and software.

SAL-based applications may be configured (by the user or their system administrator) to automatically seek out other agent peers on the local network. If other peers are found, then the agent will automatically join the existing network. If none is found, then the user's desktop can either start its own network (with itself as the only member) or proceed to use the local engine. This is useful when dedicated compute resources are unavailable or network access is inconsistent – particularly useful to

laptop users and for smaller institutions with less information technology resources.

The SAL networks themselves may be formed in an ad-hoc manner by the user's desktop application instances, via auto network discovery. System administrators or users may setup stable agents on dedicated, always-available hardware providing a pool of reliable compute resources to all SAL-driven applications within a site.

Finally, Scopira and SAL are multi-platform. Primary platforms include Microsoft Windows and Linux, with Apple's OS X and various UNIX operating systems as secondary platforms. Users now no longer need to bother with learning how to access and use the local Linux-driven computer cluster, but can instead run their Windows based desktop application, which will seamlessly communicate with the cluster and other peer nodes. This is possible as SAL is designed to be a multi-platform library suitable for developing applications on all the major desktop platforms. The network and object-serialization layers in SAL (inherited from Scopira) take care to specify data type sizes and byte-order, permitting data messages to be transferred between nodes of differing processors and operating systems.

6 Experiments

6.1 Introduction

This chapter presents the experimental design for testing the performance, usability and application integration of SAL. Through a variety of metrics, these experiments will gauge the Scopira Agents Library (SAL) along three major axes:

- **Performance:** how fast are SAL applications. A set of metrics will be used to objectively quantify the communication and management overhead a SAL-built (compared to other offerings) application will incur.
- **Usability:** how easy it is to develop a parallel application using SAL, compared to other offerings. A set of metrics will be used to correlate the overall ease-of-

use, amount of work and debugging time a programmer will experience when using the library. In addition to these objective metrics, several subjective case studies will be presented to evaluate difficult-to-quantify benefits.

- **Application Integration:** how easy is it to integrate SAL into a stand-alone application that could be used by non-technical users. Embedding SAL into desktop applications shares the performance benefits of parallel computing with groups of users that may have lacked the programming or technical background that may be necessary to run parallel algorithm implemented with more established libraries. Subjective measures and anecdotal use-cases will be used, as such benefits are difficult to quantify.

If the design and implementation goals of SAL are successful, these experiments should show that SAL is more usable than comparable libraries with sufficient performance in its targeted use cases. SAL will be shown to integrate into stand-alone applications, usable by non-technical users. The SAL will be somewhat less efficient in more demanding configurations but for its intended configuration it should provide acceptable performance (negligible differences in overhead) compared with the leading message passing libraries. Recall that, by design, SAL sacrifices some performance for usability.

6.1.1 Setup

6.1.1.1 Compared Libraries

The experiments will compare SAL-created programs to:

- **Uniprocessor:** a standard, non-parallel implementation. This version is a basic, non-parallel version that is free of any communication or setup code. This version gives the optimal performance on one processor and is the baseline for all other comparisons.
- **Threaded**: a single-machine, preemptive threaded version, using POSIX Threads. This version cannot scale past the processors in a single host. However, utilizing its shared memory architecture, it is expected to be the fastest (have the least overhead) for single host cases.
- **MPI (Messaging Passing Interface):** a version created using an open-source MPI implementation (such as LAM [20] or MPICH [42]) library, which conforms to the MPI specification. This API is the current standard library for writing parallel programs. This version will be tested in all cases, both multi-process with a single host and with multiple networked hosts. The standard C version of MPI will be used, rather than the lesser used C++ version. The C++ version adds little in the use of object-oriented or generic programming styles to the MPI API, and keeps the same overall style of communication functions as those in the C version.

- **PVM (Parallel Virtual Machine):** a networked version using the established PVM library. This version will be tested in all cases, both multi-process with a single host and with multiple networked hosts.

The core of the experiments will be the comparisons with MPI and PVM, as these libraries (MPI more so) are the leading message passing libraries for parallel computing. The uni-process and threaded cases provide ideal performance baselines.

6.1.1.2 Test Programs

Two basic algorithms will be developed for the tests:

- **Boss-worker Random Search:** This algorithm implements a rudimentary version of Stochastic Feature Selection (SFS) [80], a feature-reduction strategy that aids in the classification of biomedical data sets. This "embarrassingly parallel" algorithm lends itself to the boss-worker organization model (Figure 1 (a)). The workers request the data set and initialization information from the boss, perform the required amount of work, and submit their results on completion. There is no communication between workers, leading to less demand on the communication hardware and software. The organizational model has built-in load balancing: faster workers will simply do more tasks and will not be held up by slower workers.
- **Peer to peer Conway's Game of Life:** This algorithm implements Conway's Game of Life, a classic, deterministic cellular automaton played out on a two-

dimensional matrix. Each iteration of the algorithm produces a new matrix based entirely on the previous one. This task is implemented in a peer to peer fashion (Figure 1 (b)), where each peer processes a subset of the matrix and exchanges border information with its neighbors. This lock-step algorithm is somewhat communication intensive, as is common in many image processing algorithms, and will serve as a more rigorous test of the communication performance of the various libraries.

These two test programs will compare how the libraries handle two different parallel program organizational models with different communication requirements.

A pseudo-code outline of the algorithms is provided in Appendix A: Algorithm Pseudo-Code, while the full source code is available in electronic form (Appendix B: Electronic Files).

6.1.1.3 Test Hardware

The following hardware will be used:

- **"Single-Node," One 8 Core Node:** For the single-node tests, an eight core (via two four-core Intel Xeon processors) machine running Linux, will be used.
- **"Multi-Node," Fourteen 2 Core Nodes:** For the cluster/networked hosts tests a cluster of 14 nodes, each with two AMD Opteron processors each running Linux, will be used. The nodes will be connected via a standard, but dedicated, gigabit Ethernet switch.

These two configurations will test single-node and multi-node parallel scalability, showing the effects of network transmission overhead on performance.

6.2 Assessing Performance

Performance of the libraries will be assessed by submitting the programs to various tests. The work loads will be of fixed size, sufficient to return measurable timing results. The tests will vary the number of processors to give insight into the scalability of each program and library combination. All performance results will be normalized to work done per processor, for easy comparisons.

There will be five implementations (one for each library) of each of the two programs, for a total of ten programs. Each of these programs will be submitted to the following two sets of tests (corresponding to the two hardware configurations):

- **Single-node:** Runs with processors P=1, 2, 4 and 8 will be performed on the single-node. This will test processor scalability without network overhead.
- **Multi-node:** Runs with processors P=1, 4, 8, 16 and 24 will be performed on the multi-node cluster. For each run, $N=P/2$ nodes will be used, as each node contains two processors. This will test scalability with network overhead.

Note that the uniprocessor programs will only be run with P=1, while the threaded programs will only be run under the single-node hardware configuration.

6.2.1 Performance Comparisons

Given the variables that can be adjusted in these experiments, it is important to frame and classify the experiments, along their various dimensions such that meaningful conclusions can be made.

At the core of these experiments, we will compare the *performance* (work units completed per second) of the libraries against each other. To assess the *scalability* of the libraries, we will sample their performance over a range of *P*, the number of processors used. We can then compare the normalized per processor performance (work units completed per second, per processor) over the various libraries. This will be considered a *run*. Because the performance metrics are normalized per processor, a run will also highlight the overall efficiency of a library: its ability to scale (ideally, linearly) overall performance with the number of processors.

For the peer-to-peer job type, multiple runs will be performed over various job sizes. This highlights the effects job size has over communication characteristics and thus overall performance. Finally, comparisons will be made between the runs on the single-node computer and a multi-node cluster. This gauges the effect that a real network has on communication latencies and thus overall performance. General comparisons will be between the two algorithms, illustrating the effects of organizational models on performance.

6.3 Assessing Usability

Blind, focus group-like testing methodology was considered. In such testing, groups of programmers would be subjected to implementing programs in the various libraries. The programmers would implement various algorithms using the libraries and after, attempt to objectively score the usability of all the various libraries in a survey. This method has many drawbacks. Cost and time would be a factor in running all the developer focus groups. There would be a heavy bias for some programmers, especially for libraries they may have seen or used. Documentation and the general availability of program samples would also favour some libraries over others. Finally, the inexperience of some programmers either with the programming language, the algorithms, or parallel program design and implementation may further obscure the results.

Rather, a more objective method (inspired by [83]) was desired and will be used. The experiment will take the programs written by the author and compares them objectively, using the methods described below. The author is experienced with all the libraries, the C++ programming language and parallel program decomposition and design. All attempts will be made to write the best, safest and most concise programs afforded by each respective library.

Each program type group (boss-worker and peer to peer) will have five different implementations, one for each library. The programs will be compared within their group.

Each program will be divided into the following categories:

- **Boot strap code:** This code is part of the standard infrastructure code to start and shutdown the program. This code includes the C++ entry point, any library start and stop code, as well as any data file loading and algorithm initialization. This code is relatively constant (with respect to program complexity). Differences between this type of code among the libraries can mostly be ignored, as it is usually not a concern for developers, given its trivial nature. This type of code will not be included in the analysis.
- **Algorithm code:** this is algorithm code that does not include any parallel constructs or communication commands. It is specific to the algorithm and is common to all versions of the program. For the most part, it is the same for all program versions, and any differences will be slight and negligible. This type of code will not be included in the analysis.
- **Communication code:** this is the core communication code that is responsible for packaging and exchanging data between processors. This code varies significantly between libraries, and as such, will be scrutinized and compared.

For communication code segments, the following objective metrics will be used:

- **LOC:** Lines of code (LOC) measures the number of code lines (ignoring comments and blank lines) in the program. This will give a rough estimate of the size (and usually the complexity) of a segment of code.
- **Tokens:** The token count measures the number of language tokens in a segment

of code. Language tokens include variables, reserved words, operators, and string literals. Comments will be ignored. Longer lines of code will be detected and penalized by this measure, usually a signal of complex code. This metric is preferable to character-lengths of lines as it ignores whitespace and variances in identifier lengths, which usually are not considered contributors to code complexity.

- **Average Tokens per LOC:** The average number of tokens per line of code will give an idea to the average complexity (in terms of length) of a line of code.
- **Dangerous-Operators:** "Dangerous" operators will be counted. Dangerous operators are functions or operations that the library makes the developer use, but cannot be checked at compile time. Mistakes committed during the use of a dangerous operator result in subtle and difficult to debug run time errors or erroneous output.

  The following dangerous operators will be counted:

  - **Use of pointers:** any operation that uses pointers or pointer arithmetic. Pointer operations are dangerous as they are subtle in their use, unchecked by the compiler and could easily corrupt or crash a running program.
  - **Type casting:** using any C++ cast operator. C++ strives for proper type consistency and safety. Using a cast operator usually means the programmer wants (or needs to, because of a library deficiency) to override the system, ignoring the checks provided by the compiler.

- **Type specification:** occurs when the user needs to explicitly state the type of a parameter. This forces the developer to repeat information, and since this information is not checked at compile time, an error could lead to runtime errors. This is a particular problem when the actual type and stated type can easily change, such as when switching platforms or if the object type is specified in another compilation unit, away from the communication code.
- **Element counting:** occurs when the user needs to explicitly state the size of structures or elements. This duplication (again, not checked at runtime) is error prone and could easily cause buffer overruns.

- **Extra functions:** Extra operators are function calls that perform cleanup or other required maintenance functions. These types of operators are pure-maintenance (overhead) code required by the programmer – they never add functionality to the program or algorithm. The programmer must remember to always use them as required, as they are not checked by the compiler. Failure to include these operators at best, "leak" or waste resources (such as memory), or at worst, cause run-time errors.

6.4 Assessing Application Integration

To analyze how well SAL integrates with a desktop application, a test application will be built, and several test scenarios will be run. The results will be subjectively analyzed from a user's point of view.

The existing boss-worker algorithm code will be built into an interactive graphical desktop application. This application will be run on three platforms (Microsoft Windows, Apple Mac OS/X, and Ubuntu/Fedora Linux). To emulate possible uses of a SAL-enable application, these test applications will be run in three scenarios:

- To show how SAL provides no-worse than threading utility, the three applications will be run independently on their respective platforms, using only the processors in their host workstation.
- To show how the automatic cluster group features could be useful in small deployments, the three applications will pool their processing power and form a small group network. This is an example of desktop ad-hoc cluster computing, as noted in the SAL design chapter (Figure 11 (c)).
- Finally, to show how non-technical users could automatically utilize the resources of a Linux cluster, the Windows client will be connected to a cluster. This is an example of desktop workstations as cluster clients, as noted in the SAL design chapter (Figure 11 (b)).

7 Results and Discussion

7.1 Introduction

This chapter presents my experimental results with analysis and concluding discussion. Appendix C: Experiment Protocol describes the experiment process in detail.

7.2 Results: Performance

An objective analysis of the performance of the various libraries is presented here. The performance results (per processor, normalized to the uniprocessor implementation) was compiled and presented as eight plots. Two plots were produced for the boss-worker program type (one per single-node and multi-node hardware types). Six plots were produced for the peer-to-peer program type (three job sizes for each of the two

hardware types). Each graph plots all the various implementations and their efficiency over the number of cores, showing how the implementations scales with processing power availability.. The graphs show the differences between the libraries with respect to performance and scalability. Optimal scalability was shown with a value of 1.0 ("as good as the non-parallel version, per processor"). Comparing to the efficiency of the optimal case provides a sense of the overhead incurred by the libraries.

7.2.1 Boss-worker

Overall, Figure 12 shows that all libraries have near ideal efficiency over all numbers of processors, *P,* in the single-node experiment. In fact, due to the embarrassingly parallel nature of the boss-worker algorithm, most differences between implementations can be attributed to timing errors and perhaps contention with other processes on the test system. The SAL and threads implementation seemed to show an even more level scalability growth, but with PVM attaining near optimal efficiency for *P*=8, a definite difference could not be concluded.

The glaring exception to this is the MPI implementation for *P*=8, with an efficiency rate of about 0.88. MPI's aggressive message polling scheme (see Section 7.2.3 for in-depth analysis of this feature) is the cause of this inefficiency. The normally idle boss thread is turned into a processor-bound thread that competes and takes away processing power from the actual worker threads. This leaves nine (eight workers and one boss process) to contend over the eight processing cores in the test machine

resulting in suboptimal efficiency. The optimal efficiency of such a setup is 8/9=0.89, which is about what the MPI implementation reports at P=8.

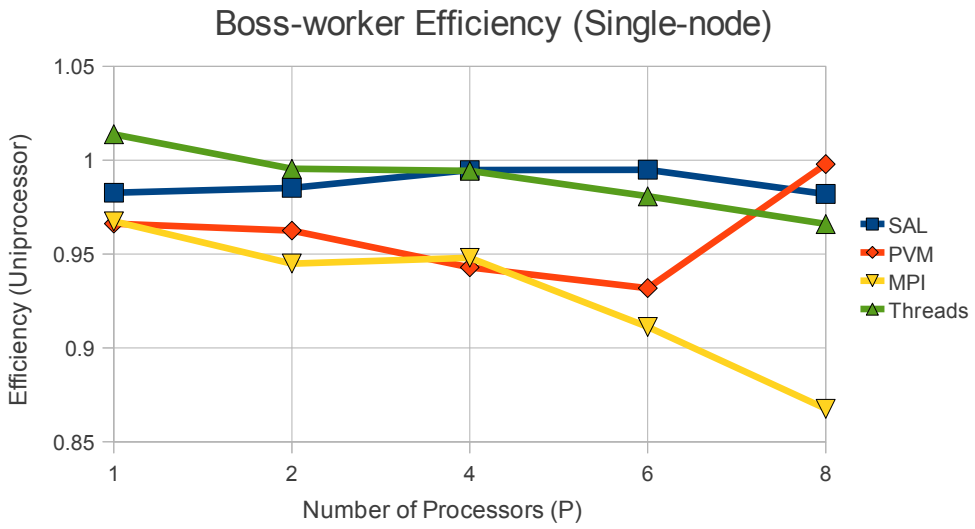

Figure 12: Boss-worker library efficiency on an SMP computer

Overall and unsurprisingly, all libraries performed nearly optimally for this algorithm across all P.

In the multi-node case (Figure 13), all boss-worker implementations effectively have an efficiency near 1.0 for all P. This is unsurprising, as the algorithm, by definition, is embarrassingly parallel. Interestingly, for many cases, the MPI implementation seems to have super-efficient (greater than 1.0 efficiency). However, this difference is so slight, that it may be explained by timing errors.

Due to the embarrassingly parallel nature of this algorithm, one expected that

efficiency will be near 1.0 for any value of *P*. This makes this algorithm a good candidate for volunteer computing other large scale deployments.

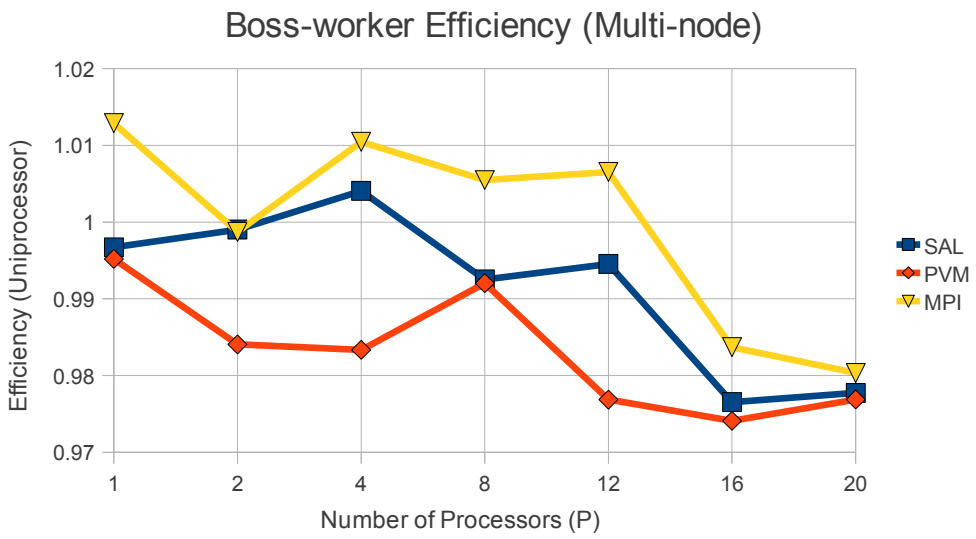

Figure 13: Boss-worker library efficiency on a network cluster

In conclusion, we can see that SAL is no worse than any other libraries for an embarrassingly parallel algorithm such as our sample boss-work instance. For such algorithms, SAL can be used with confidence as it does not provide undue communication overhead.

7.2.2 Peer-to-peer

In the pass-to-peer case (Figure 14), the experiment ran on a single-node with image size *N*=1. This is the smallest tested image size and results in the highest communication frequency as it has the lowest per-iteration computation requirements. This high frequency tests the libraries the most, bringing their weaknesses to light.

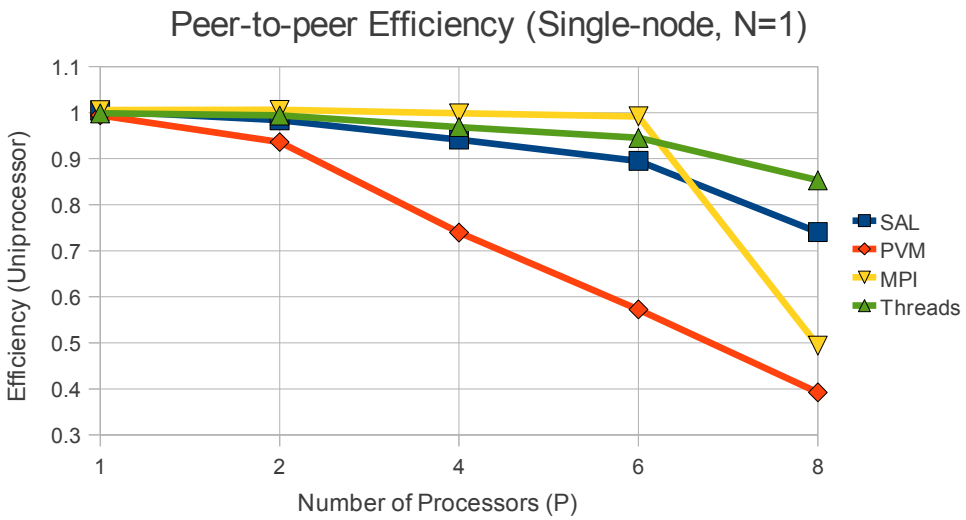

Figure 14: Peer-to-peer library efficiency on an SMP computer (N=1)

From the plot we can see that the threads and SAL implementations have roughly comparable and reasonably scalable (to 8 processors) results for all number of processors *P*. This demonstrates the speed of SAL's shared memory implementation in single-node configurations. In contrast, PVM's efficiency drops off significantly resulting in poor scalability with increasing *P*.

MPI, despite being a multi-process/message passing library (with all the assumed overheads that this entails) has superior performance to the shared memory implementations (SAL and threads). However, for the last *P*=8 case, the performance of MPI drops off significantly, due to its aggressive message polling feature (Section 7.2.3). This weakness hinders the performance of the MPI implementation for all the single-node runs. This limits MPI's ability to efficiently use all the processors on a

single-node deployment (which would be one of the simplest and most popular cases of parallel algorithm deployment) unless developers and users utilize the proper work-arounds.

The next experiment (Figure 15) increases the image (job) size by 10, resulting in more (work) processing time per iteration and thus less inter-peer communication. Less communication results in less dependence on the communication libraries, reducing their effect on overall performance. This is clearly evident in the plot as scalability and efficiency for all the libraries are more or less identical.

Finally, increasing the image size to N=100 removes any visible differences (Figure 16) between the libraries as the worker peers spend most of their time working on the image, rather than doing inter-peer communication.

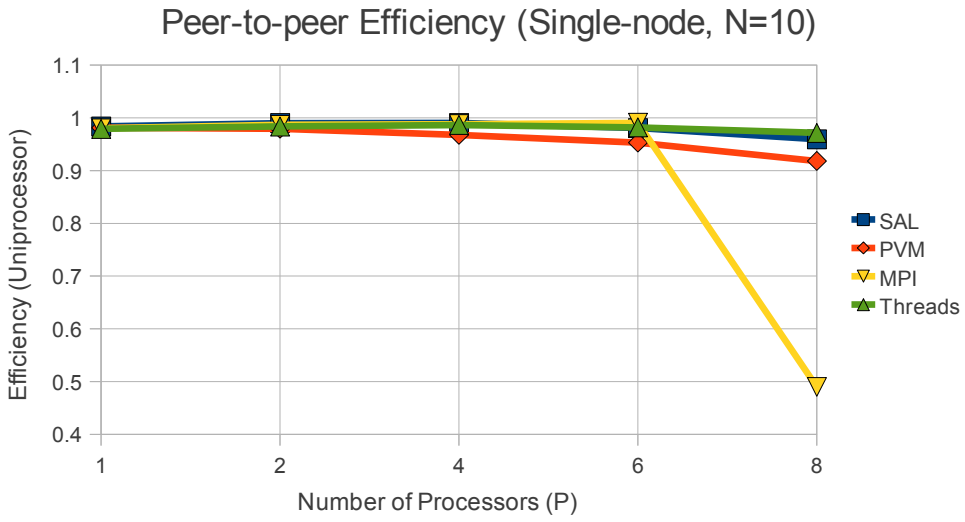

Figure 15: Peer-to-peer library efficiency on an SMP computer (N=10)

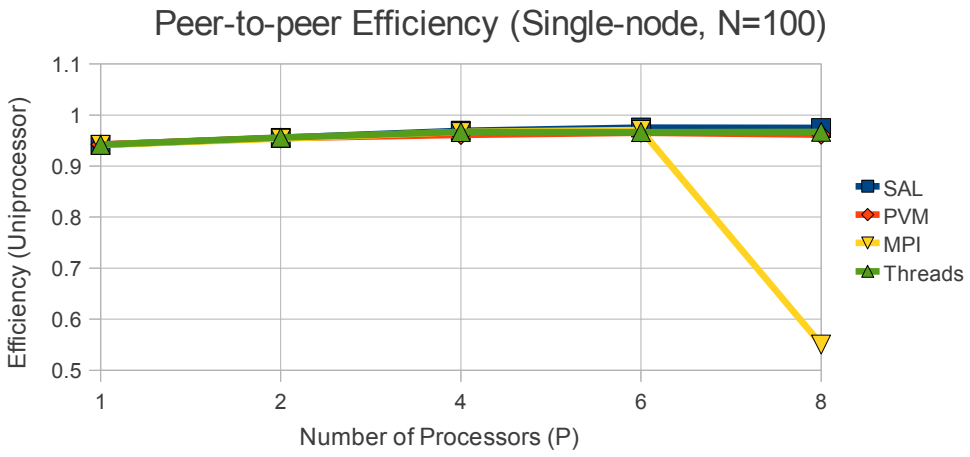

Figure 16: Peer-to-peer library efficiency on an SMP computer (N=100)

In the multi-node case (Figure 17), a cluster of machines connected by a network is used to test the communication overhead of the various libraries. The network is expected to introduce noticeable overhead compared to the single-node case, making the introduced latency and overhead of the communication libraries more important in the overall performance of the algorithm.

PVM and SAL have comparable degradation curves, with PVM having a consistent efficiency advantage for mid-range *P* values.

At *P*=8, MPI had a curious dip in its efficiency curve. This could not be conclusively explained, but one possible explanation could be attributed to some kind of process deployment inefficiency (Section 7.2.3) for this particular algorithm. For the most part though, MPI has the best performance especially at the highest processor counts (*P*=16,20). Optimization-centric features in the MPI library seem to have positive results on performance.

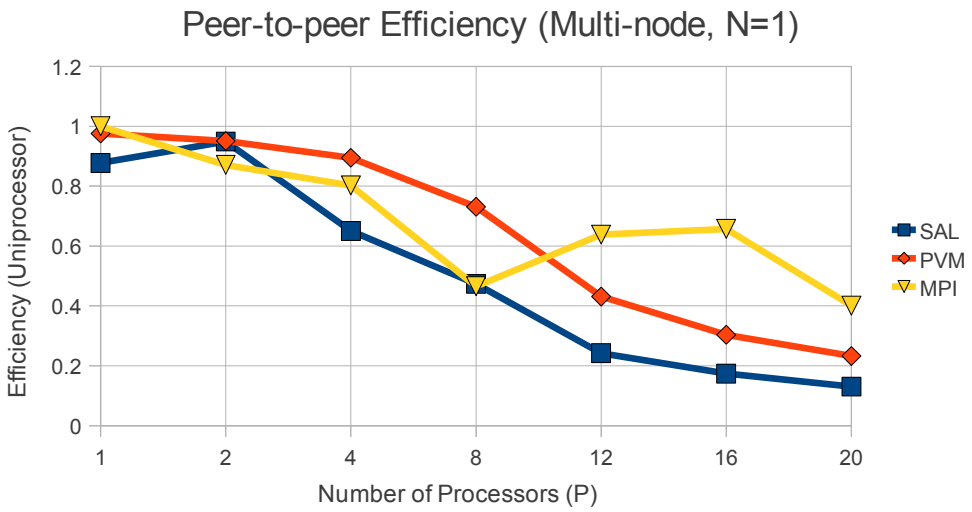

Figure 17: Peer-to-peer library efficiency on a network cluster (N=1)

The overall work performance (work being defined as $W=EP$, where E is efficiency and P is the number of processors) of all implementation peaks before reaching P=20. That is, utilizing additional processors after the peak actually has a detrimental effect on overall performance. MPI has the best overall work performance, peaking at P=12 with W=0.64(12)=7.68. SAL and PVM both peak at P=8, with performance rates of 0.47(8)=3.76 and 0.73(8)=5.84 respectively. MPI, through its aggressive communication latency optimizations, is able to squeeze more absolute work performance out of this configuration utilizing more processors in the process.

As the image size increases, overall scalability of all the implementations increases (Figure 18) resulting in performance peaking at larger P values. PVM seems to have the best efficiency for P=4,8, but MPI overtakes it for large P while SAL attains

roughly comparable efficiency to PVM for large *P*.

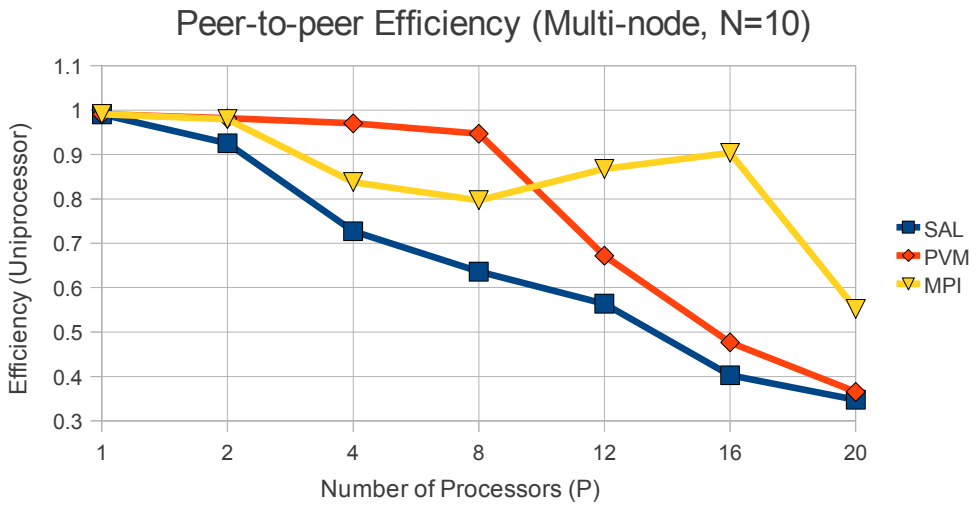

Figure 18: Peer-to-peer library efficiency on a network cluster (N=10)

Finally, for large image size *N*=100 (Figure 19), communication is so infrequent (usually about 10 messages per second) that the messaging overhead differences between the message passing libraries has little effect on overall efficiency. SAL and PVM have comparable efficiencies, while MPI, overall, is the most efficient.

For all image sizes of *N* however, one expects that the efficiency of the peer-to-peer algorithm will continue to drop as *P* increases, until it reaches nearly 0. The ratio of communication-to-work increases with *P*, and will eventually cause the system to spend most of its time performing communication operations rather than compute work.

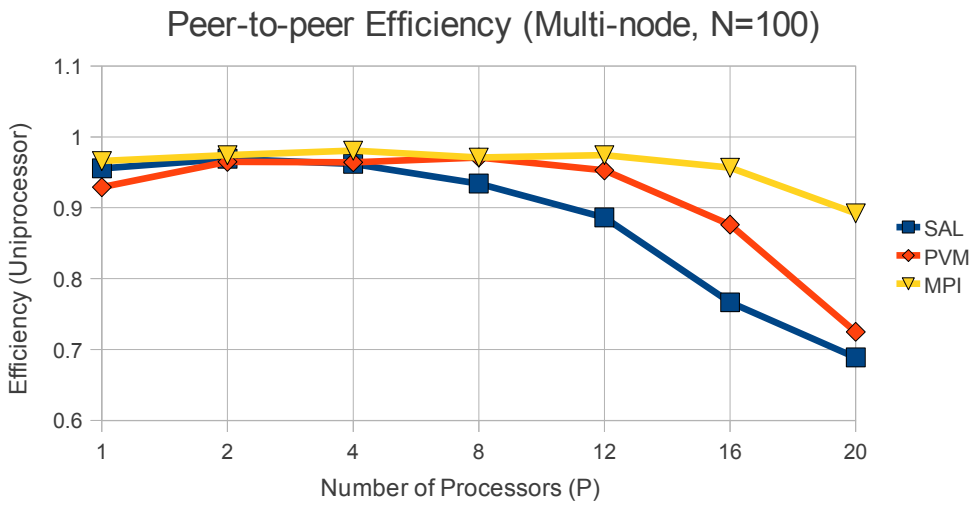

Figure 19: Peer-to-peer library efficiency on a network cluster (N=100)

In conclusion, for this more communication intensive peer-to-peer algorithm, we find that SAL is an adequate solution with respectable performance. For single-node cases, SAL is vastly superior to PVM, giving thread-like performance via its shared memory architecture. For multi-node, communication-intensive configurations, SAL is last but trails closely behind PVM in overall performance. As the communication sensitivity reduces due to larger data set sizes between communication events, the differences become less apparent. Except for the full-processor/single-node case where it has quirks (attributed to its aggressive message polling), MPI, as expected (due its focus on performance), tends to be the best overall in multi-node configurations.

7.2.3 MPI Performance Notes

Several notable characteristics of MPI surfaced while developing the MPI implementations of the algorithms. There are various features that MPI has implemented to permit optimized communication performance for large algorithm deployment with experienced developers. These features all come with trade-offs that can actually confuse or hinder the development of smaller, simpler parallel algorithm deployments. Some of these aspects may be in all MPI applications or just specific to the particular implementation used (LAM 7.1.1 running under Linux).

MPI assumes synchronous communication with fixed buffer sizes. This could lead to subtle bugs for developers that are not aware of these implementation details. MPI uses internal buffers (or perhaps the buffers of the communication protocols directly) in sending messages. When a developer requests to send data that exceeds this buffer, the send call is blocked (paused) until the receiver consumes the message. Certain job configurations, for example those that do interleaved communication (common in peer-to-peer configurations) or for when a node communicates with itself (handy for when there is only one worker node and processor) will become dead locked. These dead locks may not manifest themselves until the data sizes exceed the buffer sizes, making them harder to debug and understand. Developers must therefore understand these limitations when implementing their algorithms. MPI provides specific non-blocking, but more complex, data sending routines that can be used to alleviate problems in certain cases.

MPI's default behaviour prefers to spread out nodes over the cluster, rather than clustering them on individual SMP machines. This scheme has its benefits and trade offs, one of which is increased communication latency for processor bound work loads, hindering performance. MPI offers the end-user many options when configuring the size of their MPI cluster and its allocation strategy. The end-user must be aware of these options and trade offs when deploying their algorithms. Occasionally, the user may have to experiment with various options to find the optimal setup. Although these options are powerful and needed to cover all the potential high-performance use-cases of MPI, they may be a burden to the casual parallel computing user.

Finally, MPI often does “aggressive message polling.” Rather than wait for the operating system to signal and wake up a process that is waiting for a message, the process aggressively polls the operating system for the message in a loop. Although this decreases latency for processes that can expect data in the near future, it wastes processor cycles that may be usable by other users on a shared system. Furthermore, in certain program organization models, such as boss-worker, the boss, by design, spends most of its time in a message-wait loop. Under this scheme the boss process now unnecessarily consumes a full processing core, which could have been used by workers and other users.

7.3 Results: Usability

For an objective assessment of programmer usability, the code counts of the

sample program's communication code segments are presented and compared. The uniprocessor counts are listed for completeness. As the uniprocessor by definition does no communication, they have no communication code to be counted and thus contain all-zero counts.

The tables contain the following headings:

- LOC: lines of code.
- TOK: number of tokens.
- ATL: Average tokens per line of code.
- POP: Number of pointer operations.
- COP: Number of cast operations.
- ETS: Number of explicit type specifications.
- EEC: Number of explicit element counts.
- EFC: Number of extra function calls.

7.3.1 Boss-worker Usability Results and Discussion

Table 1 shows the analytical break down of the communication code in the various boss-worker algorithm implementations. The threaded version of the algorithm contained a surprising amount of code. Despite not needing data communication code (i.e. code to encode the data objects to the communication layer), a notable amount of code was required to setup and synchronize access to the shared variables between the threads. The standard POSIX Threading API [21] was used, which required not only the

use of pointer operations but also many explicit locking and unlocking functions, as shown in the table. Forgetting these extra functions leads to subtle runtime errors or data corruption that are difficult to debug. Most C++ programmers would choose to use another API (like Scopira Threads (Section 4.1.4), Qt [14][87] or Boost Threads [16] [57]), which use basic C++ features (such as RAII, Section 3.4), that would eliminate all the extra functions and pointer uses.

Boss-worker	**LOC**	**TOK**	**ATL**	**POP**	**COP**	**ETS**	**EEC**	**EFC**
Uniprocessor	0	0	0.0	0	0	0	0	0
POSIX Threads	46	284	6.1	17	0	0	0	14
SAL	33	265	8.0	0	0	0	0	0
PVM	50	596	11.9	34	11	0	43	4
MPI	47	968	20.6	57	9	44	44	0

Table 1: Communication code analysis of the boss-worker algorithms

The SAL implementation required the least amount of code (both in terms of lines and tokens). This is not surprising as this was one of the design goals of the library. RAII was used to deliberately remove all possible error-prone extra functions, while type safety and type-deduction was used to further remove tedious coding burdens from the programmer.

The SAL library also reuses the Scopira object serialization constructs to perform data marshaling. This allows the programmer to implement data serialization once for their objects, and reuse the same code in other serializations activities such as file I/O. In contrast, the communication/marshaling code in MPI or PVM is specific to those

libraries and is unusable in other contexts. Scopira numeric arrays in particular are already serializable. SAL users need not specify size information when transmitting arrays or slices, a contributing factor to the zero *EEC* operations for the SAL implementation.

The PVM implementation required more than twice the number of tokens as the SAL version, and 50% more lines of code. However, this is still much less than the MPI version. PVM contains a function-per-type communication API, enforcing some type safety, as noted by the lack of *ETS* operations. The API is also packet based, and although this is not quantified here, makes for an easier to use API than MPI. The non-RAII based packet API however includes functions that may be easily forgotten, as noted by the *EFC*. Luckily, these types of errors would manifest themselves quite quickly at run-time, unlike the errors associated with the use of extra functions in the POSIX Threads implementation. The PVM implementation required a few dangerous type casting operations to coalesce object data types to PVM data types. However, the bulk of these casts (*COP*) were **`const`**-type casts, which was required as the PVM API was not **`const`**-correct. **`Const`**-correctness requires that C or C++ functions mark any parameters that they do not change (such as in data sending functions) as **`const`** or constant. Failure to do so may invoke type-errors that the programmer must override with casts – a tedious and error-prone process.

Finally, the MPI implementation required almost four times the tokens and 50% more code than the SAL implementation. This is by far the worst, in terms of

programmer usability. This may be a design decision, choosing to sacrifice usability for better programmer control over communication customization and optimization. MPI functions calls are not packet based and require that the programmer repeat information for each function call, such as data destination and communication group. This is noted in the large value for ATL, and is evident with more verbose, cluttered code. Maintenance costs are also increased since the programmer must manually keep all the parameters in sync during any changes.

MPI is the only API to use generic **`void`** pointers in their communication functions, requiring the user to specify the type as an option, as denoted by the large value for *ETS*. This is particularly error prone when the programmer must verify (usually by consulting various references) that the language types exactly match the MPI types.

Both PVM and MPI required pointer operations and element count specifications even when sending or receiving single elements. This contributed to their *POP* and *EEC* counts greatly and increases the amount of tedious and superfluous code the programmer has to write.

In conclusion, of the three message passing libraries, SAL provides by far, according to the metrics, the most usable and least error-prone API interface. In contrast, despite being the de facto standard of message passing interfaces, MPI provides the most verbose and tedious API by a large factor. PVM provides a comfortable middle ground between SAL and MPI. A pure POSIX threads

implementation required a surprising amount of code, perhaps signaling developers to choose more usability-friendly threading libraries.

7.3.2 Peer-to-peer Usability Results and Discussion

Table 2 shows the analytical break down of the communication code for the various peer-to-peer algorithm implementations. For the most part, all the discussions and conclusions of Section 7.3.1 apply, and will not be repeated here. There are some noteworthy differences between the boss-worker results and those of peer-to-peer.

Peer-to-peer	**LOC**	**TOK**	**ATL**	**POP**	**COP**	**ETS**	**EEC**	**EFC**
Uniprocessor	0	0	0.0	0	0	0	0	0
POSIX Threads	40	270	6.8	27	1	0	0	15
SAL	35	530	15.1	0	0	0	0	0
PVM	47	691	14.7	13	0	0	19	14
MPI	33	702	21.3	31	0	17	17	2

Table 2: Communication code analysis of the peer-to-peer algorithms

The peer-to-peer algorithms perform some complex merging of peer data between communication calls. This code is common to all message passing implementations (SAL, PVM, and MPI). The complexity of this code is about 100 tokens, and is the most significant premium (in terms of tokens) that all the message passing libraries pay over the pure threads implementation. Despite this cost, SAL still has a notable advantage over PVM and MPI in terms of token counts. For algorithms that work on a single, shared data object such as this, the shared memory architecture of a threads

implementation is quite advantageous in terms of complexity reduction.

PVM, in this case, required higher LOC and EFC values than either SAL or MPI. PVM requires certain extra maintenance functions to be called per message block. The peer-to-peer algorithm required many different types of messages to be sent (compared to the boss-worker algorithm), resulting in PVM accruing more extra function calls and lines of code counts.

Finally, one issue from Section 7.2.3 regarding MPI performance issues also has an impact on usability that only subtly shows up in these metrics. The MPI assumption of synchronous communication with fixed buffer sizes forces the programmer to use alternate, non-blocking send calls for MPI in the peer-to-peer communication code. This code required two additional clean-up functions that manifested themselves as two extra calls (EFC) in the results table. When using MPI, developers must be mindful of its performance optimizations when developing their algorithms.

Drawing similar conclusions as in Section 7.3.1, we see that SAL is superior to MPI and PVM in terms of tokens and dangerous operations. With respect to LOC, SAL is comparable to MPI, but only because MPI's lack of packet based grouping of communication functions sacrifices token counts for lines of code counts. Surprisingly, the pure threading implementation also had a comparable LOC values, but with fewer tokens. A more usability-friendly thread-based API (such as those listed in Section 2.3.2) would help to reduce the lines of code and dangerous operator counts to more reasonable levels.

7.4 Results: Application Integration

The results of the application integration scenarios will be presented here.

7.4.1 Application Design

The test application for assessing application integration issues uses the core boss-worker algorithm code, SAL for message passing and the Qt [14][87] library for the user interface. All libraries are well tested on the three test platforms, including both 32-bit and 64-bit configurations.

The worker code snippets are similar to that of the non-GUI boss-worker algorithm. That is, they receive work (via SAL), perform it (using the core algorithm code) and return the results to the boss (via SAL).

The boss code is quite different, as it is now integrated into the GUI. There is no separate boss thread: the GUI thread, in addition to responding to the usual GUI events periodically (via a Qt time object) checks the boss SAL message queues and processes the corresponding message events as needed. Unlike MPI, SAL is message based and will queue events asynchronously permitting such a configuration. This configuration is sufficient, as the worker's algorithm performance is not dependent on the latency of the boss's reply.

In algorithms where the boss process latency is integral to the performance of the system, the GUI thread would spawn the boss process separately, giving the algorithm boss its own dedicated thread. This is a required trade-off in avoiding any performance

penalties.

The GUI/boss code itself (by design) was relatively straightforward to implement and understand. Perhaps counter-intuitively, the application's GUI/boss code is actually simpler than the basic thread code, as SAL forces programmers to compartmentalize their communication code in explicit message passing blocks. A thread-like implementation, although potentially faster, involves shared areas locked by mutexes that are more prone to developer error and race conditions.

Furthermore, development of the desktop application uses the local, threaded implementation of the SAL message passing engine. This greatly simplified debugging, as it was restricted to one process. In other message passing libraries, one may need to debug multiple processes simultaneously, perhaps on separate hosts, greatly increasing development time.

The application contains a *Cluster Status* tab to enumerate all the agents (machines) of the current SAL cluster and their processor counts. This information is strictly for illustrative purposes and is not required for the operation of the application. SAL-enabled desktop applications do not need to present this information to the users.

7.4.2 Application Use Cases

In the first use case, the three applications (one per platform: Windows, Mac and Linux) ran independently. This scenarios represents the lone desktop user, who may not have, want, or need the computation resources of other hosts. This includes mobile

laptop users who are disconnected from their host network.

Figure 20 shows the three applications running on the platforms. The application is a typical GUI application with menus and interactive output. The user needs only to start the application via an icon and click through the various functions and options. The algorithm, via SAL, automatically uses all the processors and cores in each workstation, without any user configuration or input.

Performance between the three applications of course varies greatly, as the machines themselves vary in processor power and other factors. In particular, the Windows workstation is the most powerful with eight processor cores (performing 95 iterations/sec), followed by the Mac with two cores (30 iterations/sec) and Linux with only one core (and only 8 iterations/sec). The performance per core, of course, is dependent on the CPU type itself and the optimization capabilities of the respective C++ compiler.

In the second use case, the three applications (one per platform: Windows, Mac and Linux) were run as a group. This scenario represents a user or group of users with moderate computation requirements. They can simply pool their desktop processors into one computation group. This may be popular for groups without the resources (human or financial) to support a dedicated compute cluster. Older, idle workstations may be quickly (without having to change the configuration of the existing operating system) re-purposed to contribute resources to the group. Traveling laptop users may also form such groups amongst themselves in an ad-hoc fashion via a wireless network.

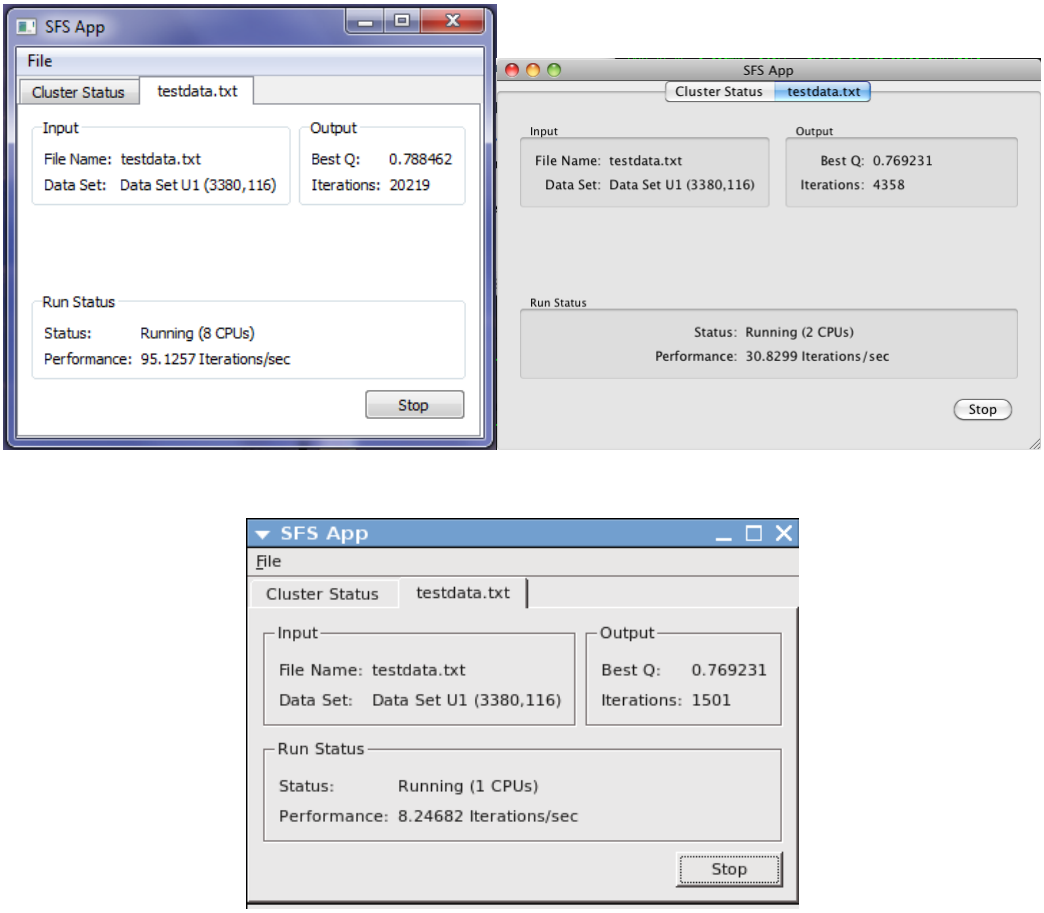

Figure 20: Application integration demonstration for individual runs

The cluster groups are formed in an automatic manner, using network discovery. When the second and third instances launch, they automatically discover (and form a group with) the first application instance. The user does not need to start or stop any ancillary programs, use the command line or understand anything about networks.

Once an application instance joins the network, it will be able to use all the processors in the group to perform its tasks. As shown in Figure 21, we see each instance utilizes all 11 processors in the group and all have about the same performance:

130 iterations/sec. Due to SAL's multi-platform support, the weaker machines workstations (Linux and Mac in this case) are able to use the many processors of the Windows machine, drastically increasing their performance. This adds to the scientific utility and extends the life of older hardware by letting them reuse the resource of newer, more modern workstations.

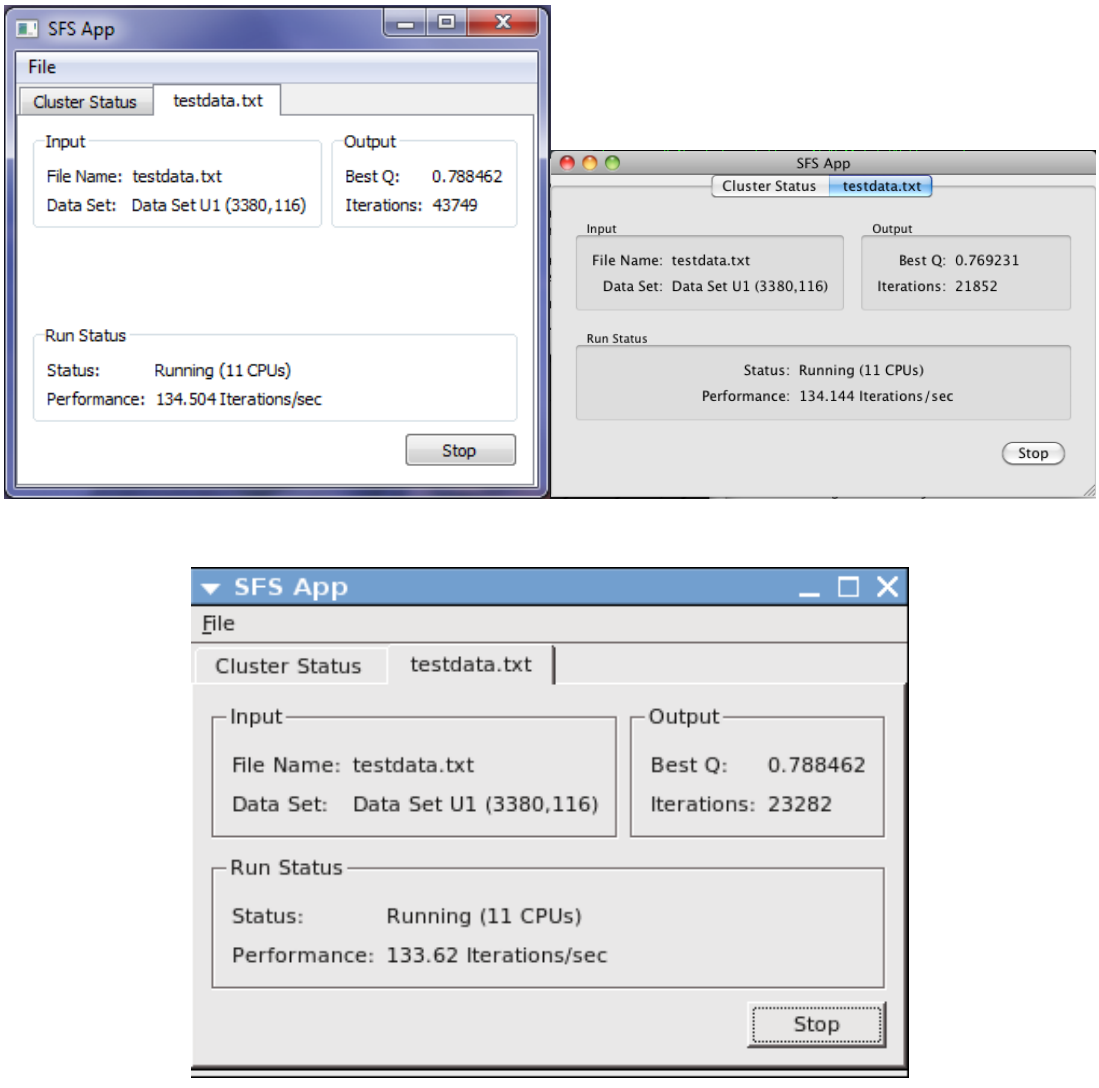

Figure 21: Application integration demonstration for group runs

Finally, in the last case the Windows desktop application is shown (in Figure 22) using the resource of a dedicated computational Linux cluster. In this scenario, a desktop Windows user is able to automatically use the resources of a locally managed Linux cluster greatly increasing the performance (to a total of 436 iterations/sec) of their algorithm.

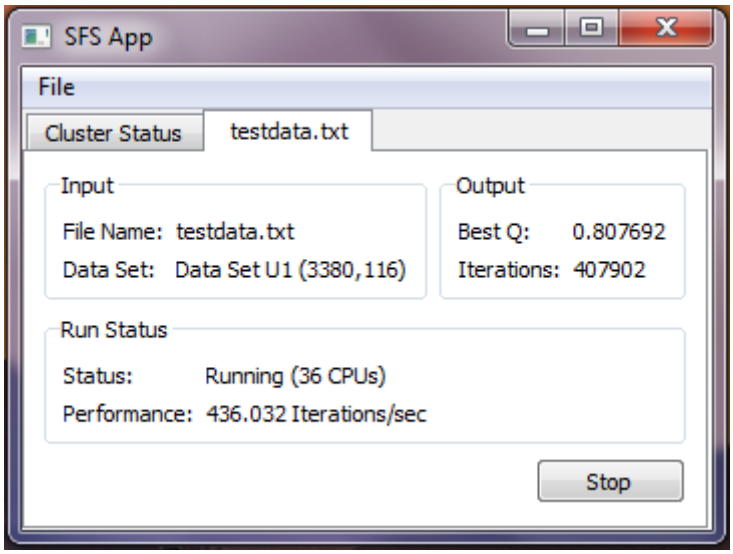

Figure 22: Application integration demonstration for cluster runs

The user needs only to start their application in the usual manner (usually by clicking an icon). The user does not need to be aware of the Linux cluster at all. This frees the user from having to learn to login, operate and transfer data to and from a system they may not be familiar with.

The automatic discovery mechanism of SAL currently only works with machines that are on the same IP subnet. If the cluster is on another subnet (as was the case here), the user's SAL-enabled application must be given the cluster's host URL (link) string. This link string looks like a web address (although rather than **`http`**, it specifies a **`scopira`** protocol). The application can be easily given this link, either as a command line parameter (which could be embedded in a desktop shortcut icon) or via a configuration screen in the application itself.

A SAL application, upon failure to connect to any peers of clusters, will automatically run in a local mode that uses all the processors/cores on the user's desktop. This means that even if the cluster is unavailable (due to hardware or network issues) the application is still usable without the need for additional user configuration. This is particularly handy for mobile laptop users who may be away from their host cluster, yet still want to be able to use their software.

The SAL processes on the cluster must still be managed by someone. This does not have to be the developer or otherwise technical user and can be another user or the system administrator. By design, SAL applications may be easily grouped into cluster groups by any user. The cluster also does not have to be Linux based, any type can be assembled by grouping together a variety of workstations as a computation cluster when using SAL-enabled applications.

7.4.3 Application Conclusions

This real-world application, through use cases, presented some of the more subjective benefits of the Scopira Agents Library. These benefits include:

Embeddability into an existing application as a library. The complete SAL system can be embedded in the host application as a library requiring no external setup or management programs (unlike PVM and MPI, which require management programs). The user does not need to login to unfamiliar systems, or use unfamiliar software, but instead may run standard desktop applications.

No manual setup for desktop parallelism. The embedded library, when it does not find any peers or clusters, will immediately use its built in, always available, single-host threaded engine. This engine is always available to the user and developer, giving the application multi-processor scalability (using locally available processor cores) without any setup. This is useful for users who do not have cluster resources available, are mobile (with a laptop, for example), or have work loads that do not require cluster computing.

Transparent cluster parallelism. The detection and discovery of other SAL processor providers is transparent to the end user. A system administrator, the user, or other users may launch worker SAL processors that automatically provide computation resources. Clients need not specifically enter the location of these providers, their SAL library will automatically discover them through standard local area network broadcast techniques. Should no resources be found, the internal threaded implementation will be

used to provide single-host multi-processor/core parallelism.

Multi-platform and ad-hoc computing. Although all parallel libraries are available on a multitude of platforms, the embeddability of the SAL approach provides new options for casual and ad-hoc cluster computing. Users may launch multiple instances of their applications on any major operating system, and through the transparent discovery mechanism, they will group together and share resources. Users need not dedicate hardware, consult system administrators or use unfamiliar operating systems to utilize cluster computing. This is useful for smaller or mobile groups of users that require less formal network and hardware configurations.

7.5 Other Applications

We used SAL for a number of projects, some of which I now describe.

A full version of stochastic feature selection (SFS) [80] was developed to use Scopira and the SAL. SFS is an iterative feature dimensionality reduction technique for the classification of complex voluminous biomedical data. SFS randomly assigns the original dataset samples (e.g. magnetic resonance spectra) into design and test sets. Once the design phase is complete (i.e., classification coefficients have been determined), the test set is used to externally validate the classification performance. The stochastic nature of SFS is controlled by a feature frequency histogram (Figure 23) whereby the performance of each classification iteration is assessed using a fitness function. An ad hoc cumulative distribution function, constructed from this histogram,

is iteratively used to randomly sample new features (rather than each feature having an equal likelihood of being selected for a new classification iteration, only those features used in previous "successful" iterations are selected). Via SAL, SFS bundles classification iterations to minimize inter-process communication and maximize CPU loads. Furthermore, while SFS exploits parallelism, it remains (optionally) strictly deterministic, that is, results are perfectly reproducible regardless of computational load (an extremely useful benefit for biomedical research).

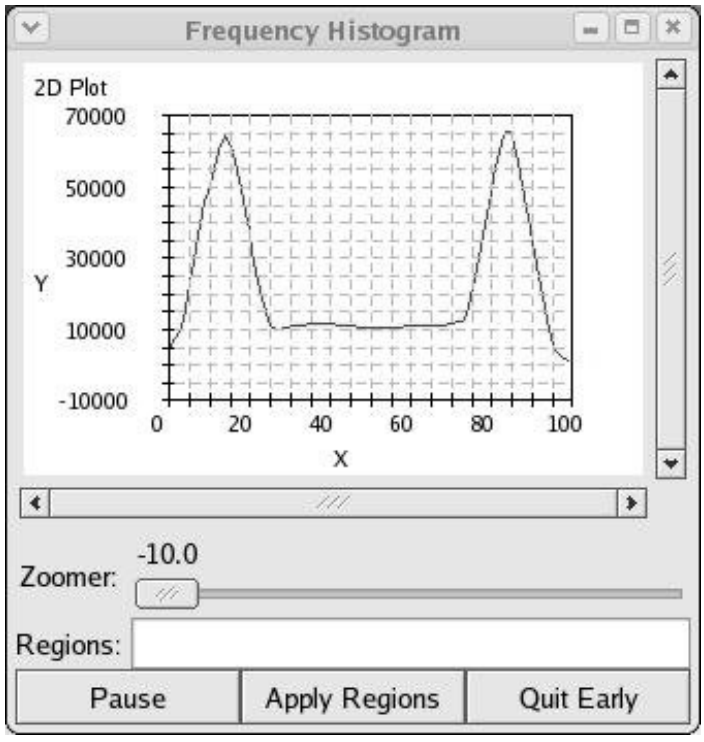

Figure 23: Feature frequency histogram used by SFS

The SFS algorithm uses an extensive combinatorial search operation and was originally implemented using MPI (specifically, the LAM/MPI implementation). The

MPI version has been well tested and extensively used. Although functional, the MPI version was comparatively cumbersome to use, requiring knowledge of logging into the Linux cluster and slave node startup. Command line usage with graphical monitoring was clumsy at best requiring a Linux workstation. These usability shortcomings made it an interesting candidate for converting to SAL.

SFS was subsequently ported to SAL in a relatively straightforward manner. As the algorithm had already been designed in a parallel fashion, porting simply required that all the communication/data transport code segments (the segments that utilize MPI) be converted to use SAL message packets. The application itself was restructured as a task object as SAL deals with objects rather than whole processes. The new version maintained the same performance characteristics as the MPI version but now sported an easier to use interface (available from the user's Windows desktop as a native Windows application) with seamless access to the compute cluster (forgoing the necessities of logging into Linux). As an added bonus, the application is also able to run without a cluster, albeit limited to the resources available at the user's desktop. This allows the application to be run when the cluster is unavailable, either temporarily due to system issues or due to temporary unavailability (such as when the user is mobile via a laptop computer) or when the cluster's resources are simply not needed.

Another application, *Raygun*, has been developed from the ground-up as a Scopira application, subsequently using SAL. This application was initially developed to provide extensive 3D visualization capabilities to data sets produced by a third party

application. This application uses a Monte Carlo based simulation to simulate light rays traveling through a variety of user defined mediums and finally terminating at a detector. The visualization component allows the researcher to examine large collections of these rays, with real time 3D capabilities and speedy analysis tools – much faster and more powerful than the previous MATLAB-built tools.

Along with the visualization component, it was quickly realized that replacing the propriety ray generation software would be beneficial. Not only was the cost of the software rather high and its algorithm implementation details unknown, the software was tied to a Windows machine and required long, often multi-day run times to complete jobs. Being restricted to a single Windows machine reduced remote access and left it vulnerable to the general instabilities of the host OS. More importantly, however, was the lack of cluster support within the application, requiring one machine to spend multiple days on a problem, ignoring the vast compute resources of any available local Linux cluster. It was then decided to implement our own Monte Carlo based ray generation algorithm within the Raygun application, utilizing SAL for parallelization.

The core algorithm itself was written as a single concrete C++ object. Interfaces (or "views") were then added to Raygun. The first version simply ran the algorithm without SAL and therefore used one processor. The second version utilized SAL (and therefore, all the processors in the available cluster). The version was implemented as three types of SAL tasks, a worker task, a master task and the monitoring task within the GUI. The workers run the Monte Carlo simulation algorithm itself, reporting successful

rays back to the master task. The master task tracks and coordinates the worker tasks, but does no work itself. It reports rays and general performance metrics back to the monitor task, which then displays them to the user in real time.

The Raygun application provides a good use-case of a complete SAL solution. The application provides a full suite of cluster-aware data simulation and modeling capabilities, combined with high-performance visualization capabilities. The investigator gets all of this within a single desktop application environment native to their familiar operating system, blissfully unaware of the details of cluster computing and management.

In all cases, the algorithms chosen for implementation in SAL were relatively friendly to parallelization. That is, they were not communication bandwidth intensive or sensitive to network delays and latencies. This was deliberate as our network infrastructure can be considered to be of a typical nature of mostly “fast” (“gigabit”) Ethernet. Such algorithms' total performance are not significantly affected by the speed of the transport layer. This is particularly important as the SAL transport layer is slower than most stock MPI implementations, by design, and much slower than any hardware-tuned MPI implementations. Communication intensive algorithms are simply out of the scope of the SAL framework and typically better left to the specifically-tuned frameworks and hardware.

8 Conclusions

Although SAL is still in development, it is already proving to be a useful library for cluster computing. The SAL transport layer has yet to undergo aggressive optimization and, as a result, is slower than a tuned MPI (or similar) implementation, especially if the competing implementation is tuned for a particular OS or communication network infrastructure. However, speed was not the primary goal of SAL. Nevertheless, for a large collection of low-communication applications, SAL provides more than acceptable performance. This was verified in our own algorithms, as the SAL implementations of our low-communication applications are linearly scalable as a function of the number of processors used. For medium-to-high communication applications, scalability was only slightly worse than PVM.

In the area of developer usage, SAL may be considered to be a general success.

For non-Scopira developers, the API provides yet another message passing API, requiring parallel programmers to still decompose their applications. The C++ features provide a more object-oriented approach to a universal task system than that of PVM, while the embedded non-cluster implementation offers easier debugging and development. Further, for existing Scopira developers, the library provides a natural API that is a seamless addition to the Scopira library.

Finally, the users get a significant benefit. Their desktop applications are able to use behind-the-scenes cluster resources yet still function and behave like their native desktop applications. They need not learn the technical details of cluster access and application usage. This lowers the entry barrier to cluster computing allowing all users within a site to utilize cluster resources, expanding their computational capabilities.

Future research and work includes further developing and deploying more SAL-based applications. This will allow more testing of both the implementation of the various SAL engines, as well as test the completeness of the SAL messaging API. The services concept will further need to be explored and defined, with several services (initially, the task monitor and job monitor services) slated for development. More demanding applications will be developed to fully test the task allocation and fault tolerance capabilities of the SAL system. Various advanced and specific load balancing algorithms will also be explored. Currently applications only touch on stressing the basic scheduler. Finally, more general debugging and network monitoring software will be made to inspect the status and configuration of an active SAL network, helping in

SAL development and providing visual feedback to developers and administrators.

8.1 Answers to Research Questions

This section will outline some of the results framed as answers to the research questions posed in section 1.4, Research Questions.

This work has shown that through a variety of programming language features and implementations design choices, *parallel application development can be simplified, permitting the construction of more user-friendly parallel processing applications:*

This solution is *easier and less error prone* through the use of a variety of C++ programming language features that permit for a more concise library API, that requires *less developer code with more compile-time error checking*. I *assessed this usability objectively* through a variety of source code metrics that offer a representation of code complexity. We can then *compare these metrics directly*, giving a sense of usability and general programming aids.

With regards to performance, I used *straightforward performance measurements*, and c*ompare the proposed solution to that of existing, established libraries, using various algorithm types and work loads*. Acceptable levels of performance are difficult to determine, as relative performance varies with algorithm type and work load. But for almost all the tested configurations, the solution shows acceptable (compared to existing solutions) performance results. *This shows that there are some algorithm types and*

work loads for which this solution's performance is acceptable.

The solution implements all the characteristics of a fully embeddable library. It is a self-contained library, utilizes in-process threads and is multi-platform. Supporting the *three major desktop operating systems was deemed a requirement*, as this solution targets desktop and commodity hardware. External and cumbersome infrastructure software was *minimized by containing all the needed management functionality in the library itself*. Basic network broadcasting functions and a straightforward URI-based connection specification system were implemented to *infer network configuration options for the user.*

8.2 Contributions

The main contribution of this work is the design, development, implementation and assessment of a new parallel programming library, SAL, that provides adequate performance (efficiency) for a range of parallel programming problems.

The novelty of this parallel programming library is that it is more developer- and user-friendly than other existing libraries. This novelty has two major facets: (*i*) programmer-usability and productivity and (*ii*) application integration. Together, they permit a wider range of programmers to use parallel programming in a wider range of new and existing applications. This goal, user-friendliness, is unique among current parallel programming libraries.

The result of the novelty is that parallel programming can be embedded into more

applications, especially desktop applications. The user base and use cases for parallel applications can be increased, resulting in more efficient use of resources in a variety of applications. With increased efficiency, work can be performed in less time and larger problems can be tackled.

8.3 Limitations and shortcomings

SAL's advantages are of course not without their trade-offs. By design and implementation, SAL may be less efficient and less scalable than other libraries. For its target audience, these sacrifices are acceptable; however, other options may be preferable to other users. For example, SAL is not designed for grid computing and communication intensive algorithms. SAL also does not, in its current implementation, utilize specialized communication hardware or protocols.

SAL's object-oriented design, error checking and buffering causes SAL to have higher CPU and memory overhead than other optimized libraries, resulting in lower communication throughput and higher latency.

8.4 Future Work

SAL is a complete, fully tested and usable message passing library that is immediately available. However, even though its original mandate has been fulfilled, there are still many areas of possible expansion and feature improvement, some of which I will now discuss.

8.4.1 Performance Optimizations

The current SAL network engine strived for correctness and ease of source code maintainability, while sacrificing performance. Future work includes optimizing this engine for better network performance, through various API and implementation enhancements. For example, a UDP-based transport layer could be considered, sacrificing the guaranteed reliability of TCP for the speed of UDP. This sacrifice may be acceptable in error-free networks such as those in controlled compute cluster environments. Interfacing with specialized, high-performance communication hardware such as Infiniband [51] could be considered.

8.4.2 Load balancing and task migration

Network wide load-balancing can be added by allowing long running tasks to be *migrated*, mid-run, to less loaded hosts. This would permit on the fly per-host load adjustments, a necessity when machine loads change. Programmers would follow an object-oriented style in utilizing this feature: they would permit their task objects to be serializable (that is, its state reduced to a byte stream) by implementing **`load`** and **`save`** methods in their task objects. This same concept is used to permit compound data structures to be sent over the network. SAL would not use any specialized (and thus hardware or at least platform specific) stack-saving functions – the **`save`** methods would only be called between **`run`** method calls.

This same serialization concept could be used to do check-pointing for fault

tolerance. The state of a running task would be periodically saved to disk (or another host). If the task should subsequently fault – disconnect or crash due to software, hardware or network issues – the master will restart the task from its last recorded state on a new agent, giving the task another chance to finish its work. Agents could be configured to only run one user task per process, permitting operating system enforced security and protection between tasks.

Some of these functions could be implemented now, either in user code or as a service task. This would permit experimentation without having to make potentially tricky engine code adjustments.

8.4.3 Decentralized Networking

The master-agent architecture of the current network engine implementation could be expanded to be more scalable with secondary *master* agents, or backup agents. This would permit better fault tolerance for when the master agent's host is lost and permit the use of larger networks as it distributes the routing and task management bottlenecks.

Eventually, this concept can be taken to its natural conclusion in the form a *decentralized engine* implementation. Unlike the current network implementation, which has a master agent, the decentralized version would not have any designed masters and be purely peer-to-peer. This would be a significant challenge, as it requires much more complexity with respect to task management, message routing and scalability. The resulting system however would have grid-computing like scalability

and fault tolerance.

8.4.4 Algorithm Exceptions

SAL permits the development of fault tolerant algorithm implementations. An implementation is fault tolerant if it can continue processing (or at the very least, fail gracefully) after the loss of a processing peer.

SAL provides, with its API, the functionality needed to explicitly monitor peers. However, for algorithm implementations that simply want to fail gracefully (rather than attempt to recover and continue processing), the SAL API could be more automatic and helpful.

Using C++ exceptions, the SAL API could trigger termination of the user's task code when it detects a error. Errors could be defined as attempting to communicate (either by sending or waiting for data) with an abruptly terminated peer node (terminated either because of algorithm bugs or because of system faults). The error condition would eventually be discovered by all the member tasks in the task group, effectively (and safely) terminating the group. The initiator of the task group (usually the user via a GUI) could then choose to rerun the algorithm with a new task group. However, the exception mechanism must be used judiciously to avoid unintended consequences in algorithm implementation design. Many use cases and scenarios would have to be considered and tested.

8.4.5 Additional Message Interfaces

For applications that would like to use the transport services and communications system but not necessarily the SAL messaging API, SAL provides various wrapper APIs. These interfaces are not part of the core body of research but are available in an experimental fashion for possible future work. They were used to test the scheduling engines with existing MPI user programs. Future work could look into extending these APIs, in the hopes of further widening the use of SAL. However, using alternative messaging APIs does eliminate one of the main benefits of SAL: providing an easier to use messaging API.

A minimal MPI interface is provided that bridges MPI-enabled applications to the agents transport layer. It does this by mapping the basic MPI communicator concept to an agent group and encapsulating every MPI message in a SAL message packet. Although this API and implementation combination would be slower than a plain MPI implementation (and much slower when compared to an implementation optimized for particular communication hardware), it is still quite useful. For applications that are not bandwidth or latency sensitive, the difference in performance would be negligible with respect to total run times. Furthermore, this bridge-API allows the application code to be reused and utilize a SAL network. This serves as a way to test and benchmark the SAL implementation against a reference MPI implementation. Finally, programs using this MPI API can be made to communicate with standard agent algorithms and services in a straightforward manner.

SAL also includes a small API for MATLAB applications. This API permits data passing between running MATLAB and SAL-enabled applications using the agents network transport. Although basic, this API does allow developers to build parallel applications with MATLAB without requiring the *Cluster Toolkit* (from MathWorks, Inc.), a native toolkit that provides cluster computing facilities for MATLAB. Finally, these MATLAB processes, by virtue of the agents network, may communicate with C++-built applications and thereby offload processor-intensive computations.

Scopira also provides a PVM layer that helps Scopira applications utilize the PVM API. This library is currently being used to quickly compare PVM and SAL applications with respect to performance and may eventually form the basis for a PVM layer over SAL. Other relationships between PVM and SAL may also be explored, such as allowing the PVM server daemons to launch SAL agents, similar to the use of using PVM to bridge disjoint MPI instances [34].

Appendix A: Algorithm Pseudo-Code

This appendix contains an overview of the algorithms used in the test programs. This is only a brief overview (*pseudo-code*), for a detailed list, the source code should be consulted.

Boss-worker

The boss-worker algorithm is a rudimentary version of Stochastic Feature Selection (SFS) [80], a feature-reduction strategy that aids in the classification of biomedical data sets. The input to such an algorithm is a dataset of patterns, where each pattern has a set of features and a class label. The goal of a feature reduction algorithm is to find only the discriminating features of the dataset that help a classification algorithm predict the class labels. This is vital for many types of datasets that contain a

large ratio of features to patterns, which can cause problems for classification algorithms. The pseudo-code of the algorithm is:

- As input, take in a matrix (two dimensional array) of features (where each column is a feature and each row is a pattern) and a vector of class labels. Choose some subset of the patterns to be the *training set*, leaving the others to be the *test set*.
- **`While`** the done criteria has not been met:
  - Choose a random subset of the features, following certain selection rules (for example, some selected regions may be combinations of subregions in the original dataset).
  - *Train* a classifier on the training patterns and this selected subset of features, and then *test* the classifier on the testing patterns. This implementation uses Linear Discriminant Analysis (LDA) [95], but other classifiers can be easily (and in the full version of SFS are) used.
  - If the percentage of correct classifications is better than the best result thus far, then note this percentage (and its regions) as the current best.
  - Stop the loop after some threshold has been met, for example, an iteration limit, time limit, or classification accuracy.
- Finally, report the run time, iteration count and the best feature subset with its accuracy percentage.

Peer-to-peer

This peer-to-peer algorithm is an implementation of Conway's Game of Life, a classic, deterministic cellular automaton played out on a two-dimensional matrix. Each element (*cell*) in the matrix has one of two states, *alive* or *dead*. Subsequent states of a cell depend on the previous state and its immediate eight-connected neighbors' states. The algorithm is completely deterministic except for the initial random generation of the cell matrix. The pseudo-code of the algorithm is:

- Randomly create the first two-dimensional integer matrix of logical values (0 for dead, 1 for alive). This becomes the first *current* matrix..
- **`While`** the done criteria has not been met:
  - Prepare the *next* matrix by applying the following deterministic rules to each new cell:
    - A *dead* cell becomes alive if it has exactly three alive neighbors.
    - An *alive* cell continues to live if, and only if, it has two or three neighbors.
  - This new matrix becomes the *current* matrix for the next iteration of the algorithm.
  - Stop the loop after some threshold has been met, such an iteration limit, time limit, or if the system has reached a stable or cyclic state.
- Finally, report the run time, iteration count and the final state matrix.

Appendix B: Electronic Files

The source code to Scopira and the Scopira Agents Library (SAL) as well as all the experiment programs used in this work are available on-line, at the following location:

`http://www.cs.umanitoba.ca/~ademko/thesis/`

To compile the applications on any platform, a developer needs (in addition to a C++ compiler) the following additional, readily available software packages:

- CMake for setting up the project or make files.
- PVM libraries (optional) for building the PVM test programs.
- A MPI library (optional) for building the MPI test programs.
- The Qt (optional) cross-platform application and UI framework is required for building the desktop application test program.

Appendix C: Experiment Protocol

This appendix will describe, in detail, how the experiments were conducted.

Performance Experiments

Exact performance results are specific to the hardware used in the experiments. The experiments in this work were carried out using the following hardware:

- Single-node: A Dell PowerEdge 1950 computer, with two 4-core Intel Xeon E5410 processors (running at 2.33 GHz), running Ubuntu Linux 8.10.
- Multi-node: A set of 10 IBM EServe 326 computers, each with two AMD Opteron 250 processors (running at 2.4 GHz), running Fedora Core 3 Linux, connected via a giga-bit network.

The boss-worker algorithm implementations are represented by the five executables: **`sfs_uni`**, **`sfs_thr`**, **`sfs_age`**, **`sfs_pvm`** and **`sfs_mpi`**. They are all command line programs that accept at variety of parameters that affect their runs. Upon completion, all the programs emit a one-line report that summarizes their performance in terms of algorithm iterations/second and iterations/second/processor.

All boss-work runs were run for 1,000 seconds (parameter T=1000), using the default dataset and parameters.

For the single-node runs, the four implementations (**`sfs_thr`**, **`sfs_age`**, **`sfs_pvm`** and **`sfs_mpi`**) were run using a varying number of processors (parameter P=1,2,4,6,8) and compared to the uniprocessor run (**`sfs_uni`**) for efficiency.

For multi-node runs, three implementations were tested (**`sfs_age`**, **`sfs_pvm`** and **`sfs_mpi`**) with a range of processors (parameter P=1,2,4,8,12,16,20).

The peer-to-peer algorithm implementations are represented by the five executables: **`conway_uni`**, **`conway_thr`**, **`conway_age`**, **`conway_pvm`** and **`conway_mpi`**. They are all command line programs that accept a variety of parameters that affect their runs. Upon completion, all the programs emit a one-line report that summarizes their performance in terms of algorithm iterations/second and iterations/second/processor.

All peer-to-peer runs were run for 500 seconds (parameter T=500) using the default initialization seed.

The single-node and multi-node runs were executed in the same manner as that of the boss-worker algorithm.

For the peer-to-peer algorithm, the job or image size must be specified with the parameter *N*. This parameter represents the image size, in cells (the actual width and height of the image are the square roots, approximately, of *N*). We used was *N*=1,000,000, *N*=10,000,000 and *N*=100,000,000. The numbers were abbreviated to *N*=1, *N*=10 and *N*=100 in the final reports.

The experiment results presented in Chapter 7 were the best efficiency values for multiple runs using all combinations of algorithms, libraries, and *P*. For a specific combination, *X*, of algorithm, library, and *P*, five runs were performed. Since each algorithm is deterministic in nature, it is expected that, apart from external factors such as extreme system loads, the efficiency values should be similar. If a run was unduly influenced by an external factor, it was ignored and *X* was run again. Since the standard deviations were small across all combinations, and in the interest of clarity of presentation, we decided not to include the error bars in the plots. Moreover, we extrapolated experiment results to larger *P* (e.g., *P*=64). The expositions relating to *P*=8 were not further elucidated with larger P. Therefore, in order to avoid redundancy, we decided not to include these results in Chapter 7.

Usability Experiments

For the objective code usability analysis, the source code was manually counted and itemized. A manual method was chosen for its simplicity and its feasibility, as the number of code lines was manageable.

For each relevant (that is, code involved in communication) code line, a C++ comment was embedded with the token count and (if any) dangerous operator counts. These special count comments remain in the source code for reference.

The simple utility program **`code_snip_all`** and **`code_snip`** inspect all the source code for these special count comments to produce an itemized report. The tables in the results chapter present the contents of this report verbatim.

Application Integration Experiments

The application integration demonstration was run on average hardware:

- A Dell workstation running Microsoft Windows 7,
- An Apple Mac Mini running OS X 10.6,
- A Dell workstation running Ubuntu Linux 9.10.

Bibliography

[1] Advanced Micro Devices, Inc.: http://www.amd.com/

[2] Alkalai L, Tai AT, *Long-life deep-space applications*, IEEE Computer, 31, 37-38 (1998)

[3] Amarasinghe SP, Anderson JM, Lam MS, Tseng CW, *The SUIF Compiler for Scalable Parallel Machines*, Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, Philadelphia: SIAM, 662-667 (1995)

[4] Amdahl G, *The Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities*, Proceedings of AFIPS Spring Join Computer Conference, Atlantic City: AFIPS Press, 483–485 (1967)

[5] Anderson D, Cobb J, Korpela E, Lebofsky M, Werthimer D, *SETI@home:*

An Experiment in Public-Resource Computing, Communications of the ACM, 45, 56-61 (2002)

[6] Anderson DP, *BOINC: A System for Public-Resource Computing and Storage*, 5th IEEE/ACM International Workshop on Grid Computing, 4-10 (2004)

[7] Anderson JC, Lehnardt J, Slater N, *CouchDB: The Definitive Guide*, Sebastopol: O'Reilly Media (2010)

[8] Apache CouchDB: http://couchdb.apache.org/

[9] Armstrong J, *The development of Erlang*, The Ninth Exhibition and Symposium on Industrial Applications of Prolog, October 16-18, Hino, Japan, 16-18 (1996)

[10] Armstrong J, *Programming Erlang: Software for a Concurrent World*, Pragmatic Bookshelf (2007)

[11] Barak A, La'adan O, *The MOSIX Multicomputer Operating System for High Performance Cluster Computing*, Journal of Future Generation Computer Systems, 13, 361-372 (1998)

[12] Berkeley Unified Parallel C: http://upc.lbl.gov/

[13] Bezdek J, Ehrlich R, Full W, *FCM: the fuzzy c-means clustering algorithm*, Computational Geosciences, 10, 191–203 (1984)

[14] Blanchette J, Summerfield M, *C++ GUI Programming with Qt 4*, Upper Saddle River: Prentice Hall (2008)

[15] Blitz++: C++ Library: http://www.oonumerics.org/blitz

[16] Boost C++ Libraries: http://www.boost.org

[17] Bowman KP, *An Introduction to Programming with IDL*, Burlington: Elsevier (2006)

[18] Brezinski ME, Tearney GJ, Boppart SA, et al, *Optical biopsy with optical coherence tomography: feasibility for surgical diagnostics*, Surg Res, 71, 32–40 (1997)

[19] Bryant R, Hawkes J, *Linux® Scalability for Large NUMA Systems*, Ottawa Linux Symposium (2003)

[20] Burns G, Daoud R, Vaigl J, *LAM: An open cluster environment for MPI*, Proceedings of Supercomputing Symposium, 379-386 (1994)

[21] Butenhof DR, *Programming with POSIX Threads*, Upper Saddle River: Addison-Wesley (1997)

[22] Caglar SG, Benson GD, Huang Q, Chu C, *USFMPI: A multi-threaded implementation of MPI for Linux clusters*, Proceedings of the International Conference on Parallel and Distributed Computing and Systems, November 3-5, Marina del Rey, USA, 392-104 (2003)

[23] Carlson W, Draper J, Culler D, Yelick K, Brooks E, Warren K, *Introduction to UPC and Language Specification*, Computing Sciences (1999)

[24] Chandra R, Menon R, Dagum L, Kohr D, Maydan D, McDonald J, *Parallel Programming in OpenMP*, San Francisco: Morgan Kaufmann (2000)

[25] Cray Inc.: http://www.cray.com/

[26] Dahl O, Nygaard K, *The development of the Simula language, History of Programming Languages (Wexelblat, R (ed))*, San Diego: Academic Press, 439-493 (1981)

[27] Dean J, Ghemawat S, *MapReduce: Simplified Data Processing on Large Clusters*, Proceedings of the Symposium on Operating System Design and Implementation, San Francisco, 137-149 (2004)

[28] Demko AB, Pizzi NJ, *Scopira: An open source C++ framework for biomedical data analysis applications*, Software—Practice and Experience, 39, 641–660 (2009)

[29] Demko AB, Pizzi NJ, Somorjai RL, *Scopira – A system for the analysis of biomedical data*, Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, 1093–1098 (2002)

[30] Demko AB, Vivanco RA, Pizzi NJ, *Scopira: An open source C++ framework for biomedical data analysis applications - a research project report*, Companion Proceedings of the ACM Conference Object-Oriented Programming, Systems, Languages, and Applications, 138-139 (2005)

[31] Duncan R, *A Survey of Parallel Computer Architectures*, IEEE Computer, 23, 5-16 (1990)

[32] Ellis MA, Stroustrup B, *The Annotated C++ Reference Manual*, Indianapolis: Addison-Wesley (1990)

[33] Erlang: http://www.erlang.org/

[34] Fagg G, Dongarra J, *PVMPI: An Integration of PVM and MPI systems*, Calculateurs Paralleles, 8(2), 151-166 (1996)

[35] Fenton W, Ramkumar B, Saletore VA, Sinha AB, Kale LV, *Supporting machine independent programming on diverse parallel architectures*, Proceedings of the International Conference on Parallel Processing, 193-201 (1991)

[36] Flynn M, *Some Computer Organizations and Their Effectiveness*, IEEE Transactions on Computers, 21(9), 948-960 (1972)

[37] Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam VS, *PVM: Parallel Virtual Machine*, Cambridge: MIT Press (1994)

[38] GNU Compiler Collection: http://gcc.gnu.org/

[39] GNU Unified Parallel C: http://www.gwu.edu/~upc/software/gnu-upc.html

[40] Google Inc.: http://www.google.com/

[41] Gough BJ, Stallman RM, *An Introduction to GCC*, Bristol: Network Theory Ltd. (2004)

[42] Gropp W, Lusk E, Ashton D, Buntinas D, Butler R, Chan A, Ross R, Thakur R, Toonen B, *MPICH2 User's Guide*, Argonne National Laboratory (2006)

[43] GTKGLExt: Main Page: http://www.k-3d.org/gtkglext/

[44] Hall MW, Anderson JM, Amarasinghe SP, Murphy BR, Liao SW, Bugnion E, Lam MS, *Maximizing Multiprocessor Performance with the SUIF*

Compiler, IEEE Computer, 29(12), 84-89 (1996)

[45] Hibbard PG, *Parallel Processing Facilities*, New Directions in Algorithmic Languages, (1976)

[46] Hill FS, Kelley SM, *Computer Graphics Using OpenGL*, Upper Saddle River: Prentice Hall (2006)

[47] Horton I, *Ivor Horton's Beginning Visual C++ 2008*, Indianapolis: Wrox (2008)

[48] Huang D, Swanson EA, Lin CP, et al, *Optical Coherence Tomography*, Science, 14, 1178–1181 (1991)

[49] Huettel SA, Song AW, McCarthy G, *Functional Magnetic Resonance Imaging*, Sinauer Associates: Sunderland (2004)

[50] Ibanez L, Schroeder W, *The ITK Software Guide 2.4*, Clifton Park: Kitware (2005)

[51] Infiniband Trade Association: http://www.infinibandta.org/

[52] Intel Compilers: http://software.intel.com/en-us/intel-compilers/

[53] Intel Corporation: http://www.intel.com/

[54] ITK: NLM Insight, Segmentation & Registration Toolkit: http://www.itk.org

[55] Java Programming Language: http://java.sun.com/

[56] Kale LV, *The Chare Kernel parallel programming language and system*, Proceedings of the International Conference on Parallel Processing, 99-108 (1990)

[57] Karlsson B, *Beyond the C++ Standard Library: An Introduction to Boost*, Reading: Addison-Wesley Professional (2005)

[58] Kernighan B, Richie D, *The C Programming Language*, Prentice Hall (1988)

[59] Khronos OpenCL: http://www.khronos.org/opencl/

[60] Korpela E, Werthimer D, Anderson D, Cobb J, *SETI@home - Massively Distributed Computing for SETI*, Computing in Science & Engineering, 3, 78-83 (2001)

[61] Koza J, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge: MIT Press (1992)

[62] Krause A, *Foundations of GTK+ Development*, New York: Springer-Verlag (2007)

[63] Kupferschmid M, *Classical Fortran: Programming for Engineering and Scientific Applications, Second Edition*, Boca Rotan: CRC Press (2009)

[64] Larson SM, Snow CD, Shirts M, Pande VS, *Folding@Home and Genome@Home: Using distributed computing to tackle previously intractable problems in computational biology*, Computational Genomics (2002)

[65] Laxmikant VK, Krishnan S, *CHARM++: A Portable Concurrent Object Oriented System Based On C++*, Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications, 91-108

(1993)

[66] Mandelzweig M, Demko AB, Dolenko B, Somorjai RL, Pizzi NL, *A projection method for the visualization of high-dimensional biomedical datasets*, Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering, 1453-1456 (2003)

[67] Maplesoft: http://www.maplesoft.com/

[68] Message Passing Interface Forum: http://www.mpi-forum.org

[69] Meyer B, *Object-Oriented Software Construction*, Upper Saddle River: Prentice Hall (1997)

[70] Moore GE, *Cramming More Components Onto Integrated Circuits*, Electronics Magazine, 38, 114-117 (1965)

[71] Moore GE, *Lithography and the Future of Moore's Law*, Optical/Laser Microlithography VIII: Proceedings of the SPIE, 2-17 (1995)

[72] MOSIX: Cluster and Multi-Cluster Management: http://www.mosix.org/

[73] MPI Forum, *MPI: A Message Passing Interface Standard*, University of Tennessee (1993)

[74] MPI Forum, *MPI-2: Extensions to the Message Passing Interface*, University of Tennessee (1997)

[75] Nwana HS, *Software Agents: An Overview*, Knowledge Engineering Review, 11, 1-40 (1996)

[76] Object Management Group: http://www.omg.org/

[77] OpenGL: The Industry's Foundation for High Performance Graphics: http://www.opengl.org

[78] OpenMP API Specification: http://openmp.org/

[79] Pizzi N, Vivanco R, Somorjai RL, *EvIdent: a functional magnetic resonance image analysis system*, Artificial Intelligence in Medicine, 21, 263–269 (2001)

[80] Pizzi NJ, *Classification of biomedical spectra using stochastic feature selection*, Neural Network World, 15(3), 257–268 (2005)

[81] Pizzi NJ, Demko A, Pedrycz W, *Classification using an adaptive fuzzy network*, Proceedings of the Annual Meeting of the North American Fuzzy Information Processing Society, July 12–14, 41–46 (2010)

[82] Pizzi NJ, Demko A, Pedrycz W, *Variance analysis and biomedical pattern classification*, Proceedings of the World Congress on Computational Intelligence, July 18–23, Barcelona, Spain, 3296–3303 (2010)

[83] Pizzi NJ, Demko A, Pedrycz W, *The analysis of software complexity using stochastic metric selection*, Journal of Pattern Recognition Research, 6(1), 19-31 (2011)

[84] Platform Computing Coporation: http://www.scali.com/

[85] PVM: Parallel Virtual Machine: http://www.csm.ornl.gov/pvm

[86] Python Programming Language: http://www.python.org/

[87] Qt - Cross-platform application framework: http://qt.nokia.com/

[88] QtConcurrent: http://labs.trolltech.com/page/Projects/Threads/QtConcurrent

[89] Reinders J, *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*, Sebastopol: O'Reilly Media (2007)

[90] Ridge D, Becker D, Merkey P, Sterling T, *Beowulf: Harnessing the Power of Parallelism in a Pile-of-PCs*, Proceedings, IEEE Aerospace, 79-91 (1997)

[91] Ruby Programming Language: http://www.ruby-lang.org/

[92] Sanders J, Kandrot E, *CUDA by Example: An Introduction to General-Purpose GPU Programming* , Reading: Addison-Wesley Professional (2010)

[93] Schroeder W, Martin K, Lorensen B, *Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, Clifton Park: Kitware (2006)

[94] Scopira Website: http://scopira.org/

[95] Seber G, *Multivariate Observations*, New York: Wiley (1984)

[96] Siegel J, *CORBA 3 Fundamentals and Programming*, Wiley Computer Books (2000)

[97] Sigmon K, Davis TA, *Matlab Primer*, Boca Raton: CRC Press (2004)

[98] Silicon Graphics International: http://www.sgi.com/

[99] Snir M, Gropp W, *MPI: The Complete Reference*, Cambridge: MIT Press (1998)

[100] Somorjai RL, Alexander M, Baumgartner R, Booth S, Bowman C, Demko A, Dolenko B, Mandelzweig M, Nikulin AE, Pizzi N, Pranckeviciene E,

Summers S, Zhilkin P, *A data-driven, flexible machine learning strategy for the classification of biomedical data*, In: Artificial Intelligence Methods and Tools for Systems Biology (Dubitzky W, Azuaje F (eds.)), 67-85 (2004)

[101] Somorjai RL, Demko A, Mandelzweig M, Dolenko B, Nikulin AE, Baumgartner R, Pizzi NJ, *Mapping high-dimensional data onto a relative distance plane — an exact method for visualizing and characterizing high-dimensional patterns*, Journal of Biomedical Informatics, 37, 366–379 (2004)

[102] Sowa MG, Friesen JR, Demko A, Schattka B, Stone T, McDonald DS, Sigurdson L, Buchel E, Hayakawa T, *Quantitative Analysis of ICG Fluorescence Angiography and Correlation with Flap Outcome: a Reverse McFarlane Skin Flap Model Study*, Plastic and Reconstructive Surgery (2010, submitted for review)

[103] Sterling T, Becker DJ, Savarese D, Dorband JE, Ranawake UA, Packer CV, *Beowulf: A Parallel Workstation For Scientific Computation*, Proceedings of the International Conference on Parallel Processing, 11-14 (1995)

[104] Stroustrup B, *The C++ Programming Language*, Indianapolis: Addison-Wesley (1986)

[105] Stroustrup B, *The Design and Evolution of C++*, Indianapolis: Addison-Wesley (1994)

[106] Stroustrup B, *The C++ Programming Language: Special Edition*, Reading:

Addison-Wesley Professional (2000)

[107] Surhone ML, Timpledon MT, Marseken SF, *Unified Parallel C*, Betascript Publishing (2010)

[108] Sutter H, *Exceptional C++*, Indianapolis: Addison-Wesley (1999)

[109] Sutter H, *More Exceptional C++*, Indianapolis: Addison-Wesley (2002)

[110] The GTK+ Project: http://www.gtk.org

[111] The MathWorks: http://www.mathworks.com

[112] The Portland Group: http://www.pgroup.com/

[113] Thread Building Blocks: http://www.threadingbuildingblocks.org/

[114] Tsuchiyama R, Nakamura T, Iizuka T, Asahara A, Miki S, Tagawa S, *The OpenCL Programming Book*, Fixstars Corporation (2010)

[115] Visual Information Solutions: http://rsinc.com/idl

[116] VTK: The Visualization Toolkit: http://www.vtk.org