DISTRIBUTED AND ADAPTIVE PARALLEL COMPUTING FOR COMPUTATIONAL FINANCE APPLICATIONS

By

Pooja Varshneya

Thesis

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

August, 2010

Nashville, Tennessee

Approved:

Dr. Douglas C. Schmidt

Dr. Aniruddha Gokhale

To my Dad and Akshat
for giving me strength to pursue my dreams.

I saw a man pursuing the horizon;
Round and round they sped.
I was disturbed at this;
I accosted the man.
"It is futile," I said,
"You can never -"

"You lie,"he cried,
And ran on.

- Stephen Crane

ACKNOWLEDGMENTS

I would like to express my sincere gratitude to my advisor Dr. Douglas C. Schmidt for his constant encouragement, guidance and support during this work and my graduate studies at Vanderbilt. He gave me the opportunity to work with Distributed Object Computing(DOC) group and helped me hone my programming and writing skills. I am also grateful to Dr. Aniruddha Gokhale for his time and support in reviewing this thesis.

I am greatly thankful to Zircon Computing LLC's founders: Alexander Mintz and Andrew Kaplan for providing me with the opportunity to work on this project. I am greatly indebted to Dr. Jai Balasubramanian, a former DOC group member and my mentor at Zircon Computing for helping me to formulate my ideas by providing valuable feedback and by sharing his domain-knowledge in several brain-storming sessions.

I would like to thank Dr. Larry Dowdy, Dr. Jerry Roth, Dr. Yuan Xue and Dr. Jules White for their guidance during my graduate coursework.

On a personal note, i would like to thank my friends Vikash, Deepti, Tripti, Aditya, Samyuktha, Kamya and Poojitha who made Nashville a second home for me.

I thank my mother and my sister, Swati for their unconditional love and support. A special thanks to Akshat for imbibing discipline and focus in me and for always being there.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

I.1 Classification of Parallel Computing

Analysts, scientist, engineers, and multimedia professionals require massive processing power to analyze financial trends, create test simulations, model climate, compile code, render video, decode genomes and other complex tasks. Although these groups could use specialized super computers, the custom development time and the hardware costs are prohibitive. In order to overcome these problems, current trends focuses on using commodity hardware and public clouds for large scale parallel and distributed applications.

In Sections I.1.1, I.1.2 and I.1.3 we describe three commonly used techniques for parallel and distributed computing.

I.1.1 Multicore/Multiprocessor Computing

With evolution of chip-manufacturing technologies, multicore processors have become a norm. Multicore processors can drastically improve application performance by running multiple tasks (threads) at the same time to increase performance for heavy workload scenarios, such as data mining, financial computations, mathematical analysis, graphical simulations and web services. Muticore/multiprocessor machines use concurrent programming to boost application performance and throughput.

I.1.2 Cluster Computing

With the growing availability of multi-core/multi-processor machines, it is also becoming increasingly easier to create a cluster of nodes using cheap and readily available common off-the-shelf (COTS) hardware. These clusters usually comprise of heterogenous hardware, connected using GBit ethernet and can be quickly expanded or reduced in size

by adding or removing nodes at run-time. Such clusters can be very easily created using personal desktops and workstations and therefore, reduce the cost of computation by use of readily available hardware and software resources.

I.1.3 Cloud computing

Clouds provide on-demand access to large pools of computational resources, system software and storage on a datacenter that can be used by the users for their computing requirements. The datacenter hardware and software resources form a *Cloud*. Various cloud providers *e.g.* Amazon EC2, Google AppEngine, Microsoft Azure and Eucalyptus provide their resources to the users in a "pay-as-you-go manner", *i.e.* users pay for the hardware resources and the storage space only for the duration of time for which the resources are utilized. This form of cloud computing is known as Infrastructure as a Service (IaaS) and it is best suited for computation-intensive batch-processing and business analytics jobs that takes hours to finish.

With the advent of commodity multi-core processors, HPC clusters and cloud computing systems, researchers and developers also need newer parallel programming techniques that can maximize the utilization of such systems and enable the users to transparently port applications across different parallel computing platforms.

I.2 Motivation

Traditional parallel programming techniques, such as message passing [15] and shared memory grid computing middleware [3], have been applied by researchers in universities and national labs to develop and deploy enterprise-scale distributed and parallel applications. OpenMPI is one of the most widely used implementations of Message Passing Interface (MPI) library used for cluster computing.

However, parallel application development remains a challenging problem in the domain of large-scale development of distributed applications, where traditional grid computing technologies cannot be applied due to the following limitations:

- Complex programming models that do not have inherent support for features like node-discovery, data dissemination, load-balancing and concurrency control. Applications written using such techniques, do not scale well for complex mission-critical systems.

- Traditional grid computing and cloud-computing technologies are not platform agnostic. Every cloud provider uses a different API for application development and deployment which makes it hard for application developers to port the applications from one cloud provider to another or onto a private HPC cluster.

- There is a steep learning curve involved in mastering these parallel programming paradigms, which increases the cost of parallel and distributed application development.

These limitations are addressed by Zircon middleware software, which is an adaptive distributed computing middleware that enhances large-scale distributed and parallel applications by creating adaptive, real-time, and distributed computing on demand. Zircon middleware software provides following capabilities to researchers and developers:

- Configurable middleware whose pluggable services automate many tedious and error-prone activities related to network programming, including handling different network protocols, (de)marshaling, fault-tolerance, thread creation and management, and advanced load balancing across a network of computation servers.

- A decentralized software architecture that has no single point of failure.

- A straightforward parallel programming model that allows developers of complex,

large-scale applications (*e.g.*, computational finance and data processing applications) to design software that runs in a cluster of computers as if they are programming for a single computer.

In this thesis, we benchmark the performance of OpenMPI and Zircon middleware software by parallelizing two CPU-intensive financial computation applications using both the technologies and find that Zircon middelware software is much easier to use and can be quickly and effectively used to parallelize existing serial applications, in comparison to OpenMPI that requires complete rewriting of existing applications for parallelization.

The remainder of this thesis is organized as follows: Chapter II gives a background of financial computation applications used for benchmarking. Chapter III gives an overview of MPI standard and OpenMPI. Chapter IV discusses various features of Zircon middleware software and explains its advantages over OpenMPI framework. Chapter V describes the benchmarking experiments conducted on OpenMPI and Zircon middleware software and evaluates the results of these experiments. Chapter VI compares Zircon software with other parallel computing technologies. Chapter VII summarizes the accomplishments of this work.

CHAPTER II

BACKGROUND

Financial industry is one of the fastest growing areas of scientific computing. Computational finance applications involving massive simulations, are well suited for distribution and parallelization. Unfortunately, the prohibitive effort that is needed to parallelize these applications using traditional mechanisms has restricted the financial industry's movement in this direction. Option pricing using model calibrations and risk assessment are important techniques that are increasingly becoming critical in making timely trading decisions. The computational intensity of such methods, however, generally limits the frequency with which they can be used. Hence, there is a significant benefit from boosting the performance of such computations.

II.1 Overview of the Binomial Option Pricing Model

In finance, an option [22] is defined as a contract that gives the buyer the right, but not the obligation, to buy or sell an underlying asset at a specific price on or before a certain date. There are primarily two kind of options: *american options*, that can be exercised at any time between the date of purchase and the expiration date and *european options* that can only be exercised on the expiration date.

In this case study, we use binomial option pricing model for evaluating option prices for american options. The binomial tree model was proposed by Cox, Ross and Rubinstein [13] and it is a very popular technique used for risk-neutral option valuation.The binomial pricing model traces the evolution of the option's key underlying variables in discrete-time. This is done by means of a binomial lattice (tree), for a number of time steps between the valuation and expiration dates. Each node in the lattice, represents a possible price of the underlying at a given point in time. Valuation is performed iteratively, starting at each of

the final nodes (those that may be reached at the time of expiration), and then working backwards through the tree towards the first node (valuation date). The value computed at each stage is the value of the option at that point in time. Option valuation using this method is a three-step process: i) price tree generation, ii) calculation of option value at each final node, and iii) sequential calculation of the option value at each preceding node.

The expected value for an option is calculated at each node using the option values from the later two nodes (Option up and Option down) weighted by their respective probabilities – "probability" p of an up move in the underlying, and "probability" $(1-p)$ of a down move. The expected value is then discounted at r, the risk free rate corresponding to the life of the option.

The following formula to compute the expectation value is applied at each node:

$$C_{t-\Delta t,i} = e^{-r\Delta t}(pC_{t,i+1} + (1-p)C_{t,i-1})$$

The parameters in the above equations represent the following:

- $C_{t,i}$ is the option's value for the i^{th} node at time t
- $$p = \frac{e^{(r-q)\Delta t} - d}{u - d}$$

  is chosen such that the related Binomial distribution simulates the geometric Brownian motion of the underlying stock with parameters r and σ.
- q is the dividend yield of the underlying corresponding to the life of the option. It follows that in a risk-neutral world futures price should have an expected growth rate of zero and therefore we can consider $q = r$ for futures.

We implemented a parallel computing application to calculate option prices for 1000 american options, with heterogenous step sizes. Binomial tree option pricing model was implemented using both OpenMPI and Zircon's zNet API and the results show that both

implementations take ∼7 minutes for calculating option prices for 1000 options using 32 servers.

II.2 Overview of the Heston Model

In this case study, we calibrate Heston [20] stochastic volatility model with 5 free parameters under the risk-neutral probability measure. The case study is based on the work of Horn, Schneider, and Vilkov [21], who performed an extensive option pricing model calibration exercise to gauge the size and direction of the parameter misevaluation effect on hedging portfolio performance. As a base model for the analysis, we chose the Heston stochastic volatility model, implemented it using the NAG C library, and calibrated it on a daily basis to observed option prices for a period of several years. Similar calibrations of various asset-pricing models are common in finance, where speed and accuracy are essential factors in risk management and portfolio optimizations.

The Heston [20] model assumes the following risk-neutral dynamics for the underlying stock S and its local variance v:

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t^S$$

$$dv_t = \kappa(\theta - v_t)dt + \sigma\sqrt{v_t} dW_t^v$$

The parameters in the above equations represent the following:

- r is the risk free rate.
- θ is the long run variance mean; as $t \to \infty$, the expected value of $v_t \to \theta$.
- κ is the rate at which v_t reverts to θ, or speed of mean-reversion.
- σ is the volatility of the volatility, which determines the volatility of v_t.
- $E[dW_t^S dW_t^v] = \rho dt$, is the instantaneous correlation between the stock and the variance processes.

We use a non-linear least squares technique in our case study calibration to estimate five model parameters (starting variance value, long-run mean, speed of mean-reversion, correlation between the processes and the volatility of volatility) so that the theoretical prices get close (in terms of some norm) to the observed ones. We calibrate the Heston model using BID/ASK/MID prices of available call options for OEX, with maturities ranging from 14 to 180 days and with moneyness (strike/stock price) in the range [80,120]. The observed prices are taken from OptionMetrics (`www.optionmetrics.com`), with the usual data filters applied, *e.g.*, we removed options with missing implied volatility, zero bid prices, and zero open interest. The theoretical option prices are calculated using the Fourier transform technique and involve some numerical integration. We implemented the calibration in C++ using the Standard Template Library (STL) and the NAG C Library [1] minimization function *nag_opt_bounds_no_derive()* [2]. NAG C Library provides many useful and efficient functions to perform numerical analysis. For example, the *nag_opt_bounds_no_derive()* function is a comprehensive quasi-Newton algorithm for finding an unconstrained minimum of a function of several variables and a minimum of a function of several variables subject to fixed upper and/or lower bounds on the variables. Even with high-quality NAG functions, however, we still needed additional tools and platforms to speed-up Heston calibration computations to an acceptable level.

For example, the experiments reported in [21] processed thousands of individual Heston calibrations and took several days for the computations, which was far too long for typical research and practical purposes. It is essential to have calibration results for thousands of models within minutes or even seconds for industrial applications, such as risk management, hedging, or portfolio optimization. These calibrations run numerous times, especially for larger datasets, *e.g.*, the original project reported in [21] used only five underlying securities for calibrations, whereas there are $\sim$6,000 individual securities in OptionMetrics available for such analysis.

To speedup Heston calibrations, we created two parallelized versions of the NAG C

library based implementation of Heston calibration application, one using the OpenMPI high-performance computing library and another using Zircon adaptive high-performance computing software platform and tools.

We calibrated the model to observed option prices 1,065 times, where computation time for each task varied from several seconds upto several minutes. Both the OpeMPI and Zircon software based parallelized implementations of this algorithm, finished computation in $\sim$18 minutes using 32 servers.

CHAPTER III

MESSAGE PASSING INTERFACE

The Message Passing Interface Standard (MPI-2) [16] is a message passing library specification designed by MPI Forum (MPIF), which has over 40 participating organizations, including vendors, researchers, softwarelibrary developers, and users. The goal of the Message Passing Interface is to specify a portable interface for writing message-passing programs for distributed-memory multiprocessors, shared-memory processors, heterogenous network of workstations, and a combination of all these.The MPI defines language-independent semantics for an application programming interface that can be used for developing portable distributed, parallel computing applications. MPI library provides essential features like creating logical process topology, point-to-point communication, collective communication, synchronization, dynamic process management and support for derived datatypes.

III.1 OpenMPI

Open MPI [4] is an open source MPI implementation aiming to fully implement the MPI-2 [16] standard. OpenMPI is an all-new, production quality implementation of MPI-2 implementation, developed by a consortium of academic, research and industry partners. It aims to combine various features of previous MPIimplementations, namely FT-MPI [14], LA-MPI [18] and LAM/MPI [11] and become "the best MPI library available" [17]. Open MPI is designed to be portable to a large number of different platforms, ranging from small clusters to large supercomputers. It aims to support heterogeneous network environments in order to create multi-domain cluster systems. Its component-based architectureoffers flexibility, easy extendibility, run-time configuration and dynamic adaption to the environment.

Following section describes the architecture of OpenMPI.

III.1.1 OpenMPI Architecture

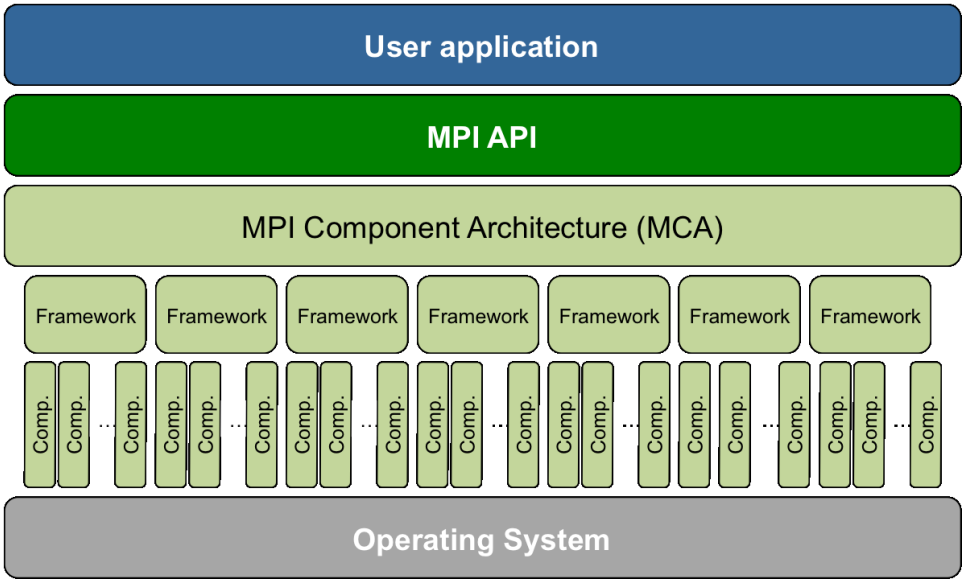

Figure III.1: OpenMPI Architecture

The OpenMPI design is centered around the MPI Component Architecture (MCA). OpenMPI is comprised of three main functional areas:

1. **MCA**: The backbone component architecture that provides management services for all other layers;

2. **Component frameworks** : Each major functional area in OpenMPI has a corresponding back-end component framework, which manages modules. Each framework provides a public interface that is used by external code, but it also own its internal services. An MCA framework uses the MCA services to find and load components at run time *i.e.* implementations of the framework's interface.

3. **Modules**: Self-contained software units that provide implementation of framework interfaces that can be deployed and composed with other modules at run-time.

Frameworks, components, and modules can be dynamic or static, *i.e.* they can be available as plugins or they may be compiled statically into libraries *e.g.*, libmpi.

III.1.2 OpenMPI Frameworks

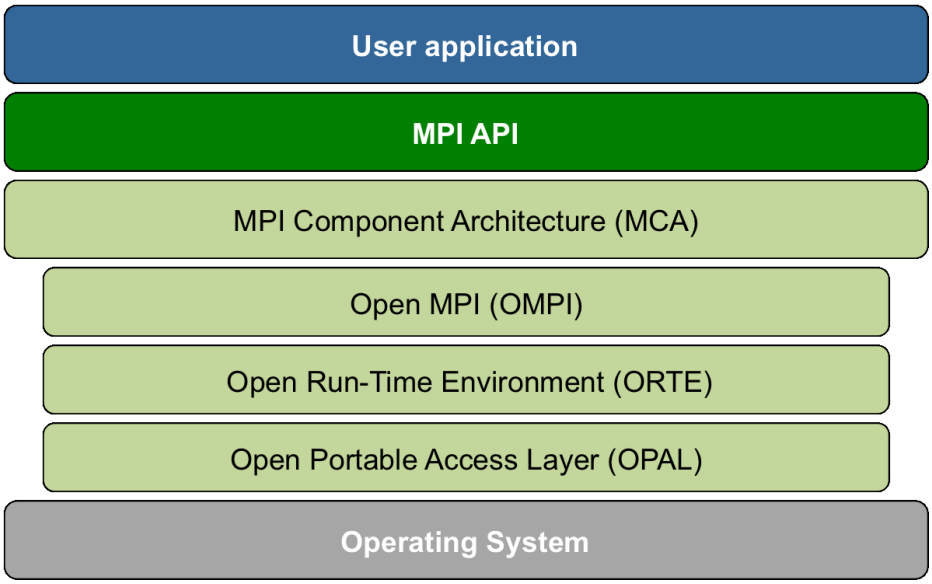

Figure III.2: OpenMPI Frameworks

Open MPI frameworks are broadly categorized as follows: The Open Portable Access Layer (OPAL) frameworks, the Open Run-Time Environment (ORTE) frameworks, and the Open MPI (OMPI) frameworks.

1. **OPAL framework** mainly provides utility and "glue" code used by OMPI and ORTE frameworks.

2. **ORTE framework** provides support for different back-end run-time systems. It enables high performance applications to run in heterogenous environments in a transparent and scalable fashion.

3. **OMPI framework** provides implementation of MPI API.

III.2 MPI Features

MPI-2 [16] specification focuses on specifying a message-passing parallel programming model, in which data can be moved from the address space of one process to that of another process through cooperative operations on each process. MPI library primarily facilitates parallel application development using these key features:

1. **Point-to-point communication** allows processes to send and receive data and messages to one another. Point-to-point operations are available in synchronous, asynchronous and buffered modes and can lead to buffer overflow errors if the length of the received message is greater than the size of receiver's buffer. Due to this reason, it is the responsibility of application developer to ensure that the receiver processes allocate sufficiently large buffer to receive data. Message sent by the sender process can be uniquely identified by the combination of its message tag, communicator group-id and receiver process rank in the communicator group. *MPI_Send()* and *MPI_Receive()* are the most commonly used point-to-point communication functions.

2. **Collective communication** allows processes to communicate with all other processes in a group. It involves functions such as *MPI_Bcast()* which allows broadcasting data from one member to all other members in a group, *MPI_Gather()* that gathers data from all members of a group to one member, *MPI_Scatter ()* that scatters data from one member to all members of a group and *MPI_Reduce()* that applies reduction operations such as sum, reduction, min or user-defined operations on the results collected from all other group members. The semantics of collective communication functions are similar to those of point-point communication.

3. **One sided communication** operations *MPI_Put(), MPI_Get(),* and *MPI_Accumulate()* have been recently added to MPI-2 specification in order to take advantage of fast communication offered by shared memory and special put/get operations available

in some hardware architectures. These functions allow remote writes, remote reads and remote update operations by making a window of a process's memory available for remote access to other processes in a group. However, the application developer needs to take care of synchronizing this communication using locks as the specification does not guarantee that these operations have taken place until a synchronization point.

4. **Communicators, groups and virtual topology** functions allows application developers to define ordered collection of processes and assigns unique ranks to each process in a communicator group. Communicators provide the appropriate scope for all communication operations in MPI and are divided into two kinds: intra-communicators for operations within a single group of processes and inter-communicators for operations between two groups of processes. Groups define a scope for process names in point-to-point and collective communication operations. A group is always associated with a communicator object. *MPI_Comm_create()* can be used to create new communicator, *MPI_Comm_group()* returns handle of global group from *MPI_COMM_WORLD*, *MPI_Group_incl()* forms a new group as a subset of global group and *MPI_Comm_free()* and *MPI_Group_free()* can be used to free up new communicator and group.

   A virtual topology provides a convenient naming mechanism for the processes of a group (within a communicator), and additionally may assists the runtime system in mapping the processes onto physical processors and thus may help to improve communication performance on a given machine. The communication pattern of a set of processes can be represented using graph and cartesian topologies. The functions *MPI_Graph_create()*, *MPI_Dims_create()* and *MPI_Cart_Create()* are used to create general (graph) virtual topologies and Cartesian topologies.

5. **Derived datatypes** allow users to define user-defined datatypes for sending data between processes in addition to predefined MPI datatypes such as MPI_INT, MPI_CHAR, MPI_DOUBLE, *etc*. MPI functions such as *MPI_Type_create_struct()*, *MPI_Type_contiguous()* and *MPI_Type_Vector()* can be used to create MPI datatypes for data-structures like structs, arrays and vectors.

6. **MPI environment management** functions are used for initializing and terminating the MPI environment and for getting and setting various attributes related to MPI implementation and execution environment (such as error handling). *MPI_Init()* and *MPI_Finalize()* are used to initialize and terminate MPI execution environment. Various functions like *MPI_Comm_size()*, *MPI_Comm_rank()*, *MPI_Wtime()*, *etc*. are used to query the runtime environment for its properties.

7. **Dynamic process management** is a feature of MPI-2 that allows for the creation and cooperative termination of processes after an MPI application has started. It provides a mechanism to establish communication between the newly created processes and the existing MPI application. It also provides a mechanism to establish communication between two existing MPI applications, even when one did not "start" the other. *MPI_Comm_spawn()* and *MPI_Comm_spawn_multiple()* functions can be used to start several different MPI processes and establish communication with them by placing them in the same MPI_COMM_WORLD and returning an intercommunicator.

8. **MPI I/O** provides a high-level interface that supports partitioning of file data among processes and allows complete transfers of global data structures between process memories and files by using the existing derived datatype functionality. It also supports features such as asynchronous I/O, strided accesses, and control over physical file layout on storage devices using functions such as *MPI_File_iread()*, *MPI_File_seek*, *MPI_File_sync()*, and so on.

Complete list of MPI functions implemented in OpenMPI can be found at www.open-mpi.org/doc/v1.4.

III.3 MPI Limitations

Although MPI-2 provides a rich library for developing efficient message passing applications, the specification does not support any abstractions for resource management, load balancing and fault-tolerance and leaves it for the application developers to customize these features for each application. These limitations makes it difficult for application developers to parallelize serial applications to run transparently on hundreds or thousands of distributed computation servers in a scalable fashion. MPI infrastructure typically allocates required number of processor nodes to an application on startup and distributes the task amongst them based on user's load-balancing design implementation. This results in resource wastage in case of typical applications that have a combination of both serial and parallel computation phases. During serial computation phase of such applications, multiple nodes allocated to this application remain under-utilized. MPI-2 specification have added dynamic process management functions to support dynamic spawning of new process at run-time, but it requires application developers to hard-code these error-prone mechanisms into the message-passing applications on a need-to-need basis. Due to the steep learning curve involved in learning MPI programming, the cost of parallel application development using MPI is quiet high. Therefore, there is a need for an easy-to-program/use/manage high-performance computing middleware platform that can alleviate these limitations and allow application developers to focus on business logic development for parallel applications.The remainder of this section describes some of these key design requirements for such a middleware framework:

Requirement 1: Automatic discovery and addressing of heterogenous remote computation servers for distributed computing. As described above, MPI does not support

automatic discovery of new computation servers added during the runtime of an application and also does not support interoperability between master and slave processes running on different operating systems such as Windows, Linux and Solaris. In MPI terminology, master process is the client component that distributes computation requests to slave/server processes for parallel computation. The task distribution and collection of results for multiple computation requests sent to heterogenous computation servers should be handled transparently by the middleware. The middleware should also support dynamic addition of computation servers at runtime for computation speedup and deletion of computation servers in case of resource restraint in a cluster of computers running multiple jobs in parallel.

Requirement 2: Easy to use programming frameworks for remote distributed computation. Distributed computations involves sending input data for request processing (*e.g.*, the input for Heston calibration application stored in an input file) in an external format that be can transferred via the network to remote computation servers.

```
struct optionsdata
{
  int day;
  int optid;
  double strike;
  double maturity;
  double price;
  double stock;
  double rate;
};
```

Figure III.3: Example of user-defined input data structure using OpenMPI

In message-passing programs, developers have to use MPI's error-prone and tedious data packing and unpacking functions or derived datatype declaration functions for sending and receiving input data in user-defined formats. This complicates source code development activities for parallel applications and highlights the need for simpler parallel

programming frameworks. For example, figure III.3 shows the definition of a simple user-defined data structure in C/C++ and figure III.4 shows the corresponding code for defining the same user-defined data-structure in MPI.

```
static void
create_options_datatype (MPI_Datatype &od_data_type)
{
  optionsdata test_options_data;
  int od_blocklengths[7] = {1, 1, 1, 1, 1, 1, 1};
  MPI_Datatype od_types[7] = {MPI_INT, MPI_INT, MPI_DOUBLE,
                              MPI_DOUBLE, MPI_DOUBLE,
                              MPI_DOUBLE, MPI_DOUBLE};

  MPI_Aint od_displacements[7];

  MPI_Aint od_start_address;
  MPI_Aint od_end_address;

  MPI_Get_address (&(test_options_data), &od_start_address);

  MPI_Get_address (&(test_options_data.day),
                   &od_end_address);
  od_displacements[0] = od_end_address - od_start_address;
  MPI_Get_address (&(test_options_data.optid),
                   &od_end_address);
  od_displacements[1] = od_end_address - od_start_address;
  MPI_Get_address (&(test_options_data.strike),
                   &od_end_address);
  od_displacements[2] = od_end_address - od_start_address;
  MPI_Get_address (&(test_options_data.maturity),
                   &od_end_address);
  od_displacements[3] = od_end_address - od_start_address;
  MPI_Get_address (&(test_options_data.price),
                   &od_end_address);
  od_displacements[4] = od_end_address - od_start_address;
  MPI_Get_address (&(test_options_data.stock),
                   &od_end_address);
  od_displacements[5] = od_end_address - od_start_address;
  MPI_Get_address (&(test_options_data.rate),
                   &od_end_address);
  od_displacements[6] = od_end_address - od_start_address;
  MPI_Type_create_struct (7, od_blocklengths, od_displacements, od_types,
                          &od_data_type);
  MPI_Type_commit (&od_data_type);
}
```

Figure III.4: Example of MPI derived datatype creation using OpenMPI

Requirement 3: Efficient distribution of remote computation requests for effective resource management across the network. In message passing programs, application developers have to repetitively implement different scheduling and request distribution algorithms for different applications. Parallel computing frameworks should support intelligent request scheduling and distribution algorithms for request dissemination across various computation servers. Efficient request dissemination should ensures that (1) all hardware resources are utilized efficiently, (2) remote computations are not impeded by load imbalance across computation servers, and (3) clients are shielded from heterogeneous hardware and software capabilities.

Requirement 4: Fault tolerance and application transparent fault detection and recovery. When remote computation servers execute complex application calculations, hardware failures can disrupt the calculations. These types of failures must be handled resiliently by the parallel computing framework since both the compute server(s) and communication links may be rendered unavailable. Developing source code for providing fault tolerance could involve writing code for detecting faults, identifying the requests that were being computed by the failed server, resending those requests to an alternate server, and taking rejuvenation actions such as restarting the failed servers using checkpoints. MPI provides the infrastructure for developing such applications, however, it is a tedious and error-prone process to write fault-tolerance infrastructure code for every application and makes it difficult for application developers to quickly parallelize existing applications.

Requirement 5: Concurrency management. Computational finance applications, such as the heston calibration and binomial option pricing calculation in our case study, are often highly computation intensive. These applications can therefore benefit greatly from proper concurrency management where all the cores in a multi-core processor are utilized efficiently for optimizing calculations. Programming these concerns requires application developers to manage concurrency explicitly, even in MPI based applications, by creating threads and synchronizing those threads with messages, and locks. This process must be

repeated for every platform since thread programming APIs differ from platform to platform, *e.g.*, differences in the thread API between Windows and Linux. Ideally, application developers should develop source code in a platform-agnostic manner so that application requests could be optimized depending on the availability of single-vs- multi-core processors.

CHAPTER IV

ZIRCON ADAPTIVE HIGH-PERFORMANCE COMPUTING MIDDLEWARE

The Zircon Middleware Software [9] from Zircon Computing [5] provides an adaptive high-performance computing middleware that addresses the limitations of OpenMPI as described in chapter III. Zircon middleware software automatically deploys a distributed computing infrastructure across (potentially) heterogeneous hardware platforms and operating systems, maps compute-intensive applications to a pool of processors, manages their execution, and dynamically equalizes the workload in real time to fit available resources. Application developers can thus exploit the processing power available to them, including newer technologies, such as multi-core processors and cloud computing systems, as well as traditional desktops and servers. Zircon software dramatically improves performance with little learning curve and configuration effort, and runs seamlessly over local-area networks; wide-area networks; public, private, or hybrid cloud deployments; and/or in dedicated data centers.

Zircon high-performance computing middleware supports three computing and communication models required by many mission-critical applications that need high performance, as shown in figure IV.1 and described below:

- **Application function parallelism**, such as the capabilities provided by computation grids to transparently run applications in a cluster of servers as if they are programmed for a single computer. The zFunction function parallelism API and supporting tools hide many low-level network programming concerns and unexpected complexities, simplifying fine-grained application parallelization.

- **Application executable parallelism**, such as the capabilities provided by data centers and clouds to launch applications on demand. The zExec application execution

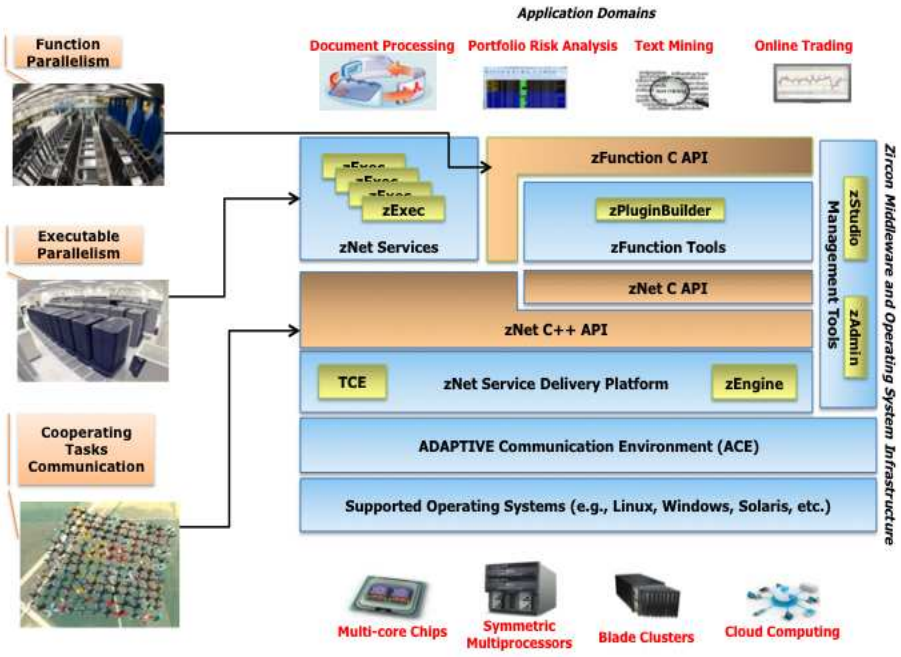

Figure IV.1: Zircon Middleware Architecture

parallelism service can runs any executable in a cluster of servers as a set of parallel jobs, thereby simplifying coarse-grained application parallelization.

- **Service delivery platforms**, such as the capabilities provided by distributed computing environments that support cooperating business tasks via distributed infrastructure patterns, such as Messaging, Broker, and Publisher/Subscriber [12]. The zNet API provides a C++ interface to the zNet service delivery platform that handles service discovery, reliable multicast communication, request workload equalization, and request dispatching.

Requests from applications that use these models can run on processors and cores in a collocated and/or distributed manner, with the choice of collocation or distribution largely transparent to application clients and servers. Zircon software runs on most general-purpose and real-time operating systems since it is implemented atop the open-source

ADAPTIVE Communication Environment (ACE) [26, 27], which is portable C++ host infrastructure middleware that shields Zircon software from operating system dependencies.

IV.1 Structure and Functionality of Zircon Parallel Computing Middleware Software

This section describes the structure and functionality of Zircon Software, which is adaptive distributed middleware for accelerating the performance of complex compute-intensive applications in a networked environment.

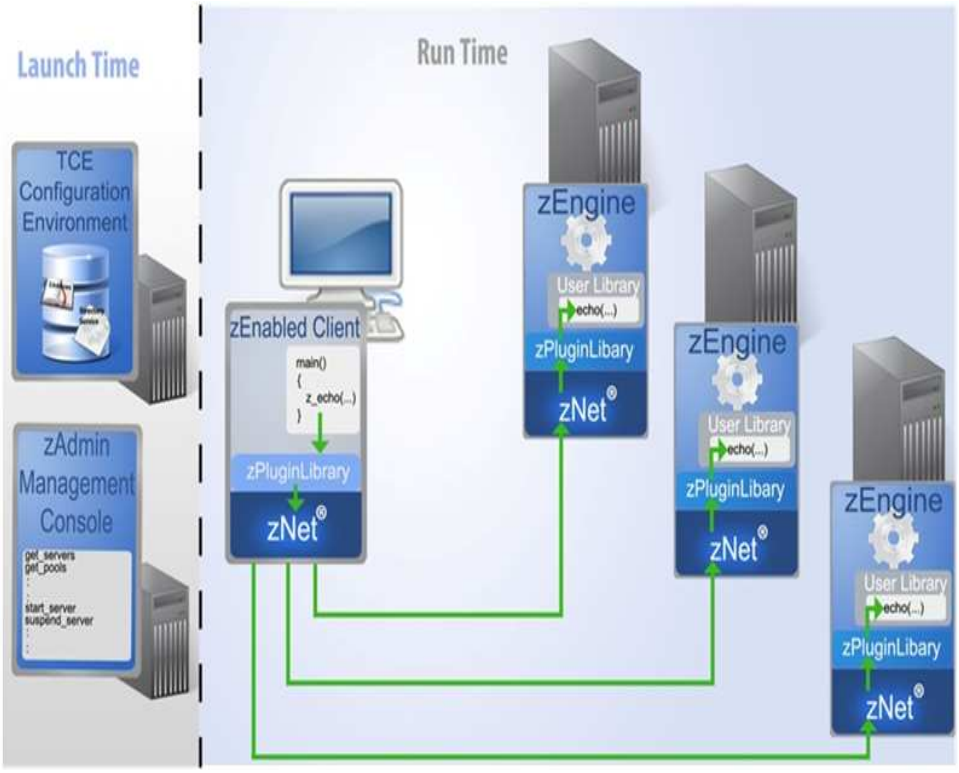

Figure IV.2: Zircon Parallel Computing Middleware Software Components

Figure IV.2 shows the following key components of Zircon middleware software:

• **Test Configuration Environment (TCE)**, is a application configuration utility that discovers, validates, and manages all applications in a deployment. It manages the compute

servers, clients, and monitoring utilities and provides IP addresses and multicast addresses for distributed execution environment.

• **zNet**, which is an optimized load balancing framework linked with the client applications and hence resides in the client address space. zNet automatically distributes computations to all the available servers, transparently parallelizes executions in a scalable, reliable, and resource-efficient fashion, and improves performance by orders of magnitude compared with conventional programming techniques.

• **zEngine**, which is a computational server container that is installed and launched on (potentially heterogeneous) target machines. This is the container in which parallelized computations actually run. A zEngine uses the underlying operating system scheduling mechanisms (*i.e.*, core-aware thread creation, synchronization, and management) to maximize processor utilization by executing an instance of a parallelized function on each core (a common practice is to start as many *zEngine* instances on each host as there are processor cores).

• **zPluginbuilder**, is a utility that is used to adapt serial client libraries into parallelizable plug-in libraries that can parallelize complex computations using zNet middleware.

• **zAdmin**, which is a utility for managing (*i.e.*, monitoring, installing, starting, and stopping) the resources, and applications in the system either graphically or via a command-line.

IV.2 zEnabling using zFunctionAdapters and zPluginLibraries

Any serial legacy application that performs complex calculations on large data-sets can be parallelized using zFunction. Parallelizing a serial application (which we call *zEnabling*) involves steps to link the application to Zircon middleware that transparently encapsulates the concerns of distributed and parallel processing from applications.

The *zEnabling* process shields application developers from low-level distribution concerns, such as discovery, addressing, (de)marshaling requests and replies, and deals with

variabilities in the underlying network protocol stack(s), so that applications can integrate with any platform and programming language seamlessly. *zEnabled* application contains an equivalent *zFunctionAdapter* z_F for every parallelizable function F. Client application developers only need to replace calls to F with calls to z_F for parallelization.

The zFunctionAdapter z_F is a client-side proxy that transparently dispatches asynchronous requests to the zEngines, thereby providing adaptive, distributed, and high performance computing on demand for client applications. zFunction makes use of the zPluginBuilder tool for *zEnabling* user libraries.

The input to the zPluginBuilder tool is an *XML* file as shown in figure IV.3, describing the function F, its input parameters, its output parameters, and the location of the library that contains the definition of the function F (shown in the middle section of figure IV.4).

The output is a library (called the zPluginLibrary) with zFunctionAdapter implementation z_F conforming to the same interface as the original function F. The generated zPluginLibrary is linked by both the client application as well as the zEngine (see the right side of the figure IV.4). On the server, the zPluginLibrary simply delegates the calls made from the client-side zPluginLibrary (on behalf of the client applications) to the function F defined in the library created by the service developers. With a minimal amount of development effort, therefore, zFunction users obtain a versatile, production-quality parallelized application that can be deployed in a network of parallel computing nodes.

IV.2.1 zFunction and zPluginLibraries

zFunction hides the low-level distributed computing concerns from application developers, encapsulate all these details and provides a zFunction *z_F* for every parallelizable function *F*. z_F's interface is very similar to that of the function F, and thus the client application developer can simply replace a call to F with a call to z_F. However, unlike F itself, z_F is asynchronous, meaning that it returns as soon as it has initiated a request, without waiting for the results.

```
<?xml version="1.0" standalone="yes"?>
<Module>
  <!-- Linux -->
  <Platform Name="Linux"
            OutIncludeDir="$(ZNET_ROOT)/plugin/include"
            OutLibDir="$(ZNET_ROOT)/plugin/lib"
            HeaderPath="$(ZNET_ROOT)/"
            HeaderFile="F.h"
            LibName="F"
            LibPath="$(ZNET_ROOT)/lib"
            LinkWithPlugin="1"
  />

  <zFunctionAdapter Name="F" Description="example function" >
    <Output Name="retval" DataType="double"
            Description="return value" />
    <Input Name="od" DataType="optionsdata"
           Description="optionsdata as input" />
  </zFunctionAdapter>

  <Struct Name="optionsdata">
    <Field Name="day" DataType="int"
           Description="output file name" />
    <Field Name="optid" DataType="int"
           Description="option id" />
    <Field Name="strike" DataType="double"
           Description="strike price" />
    <Field Name="maturity" DataType="double"
           Description="maturity" />
    <Field Name="price" DataType="double"
           Description="bid, ask and mid price" />
    <Field Name="stock" DataType="double"
           Description="number of stocks" />
    <Field Name="rate" DataType="double"
           Description="rate" />
  </Struct>

</Module>
```

Figure IV.3: Example XML input file for zPluginBuilder

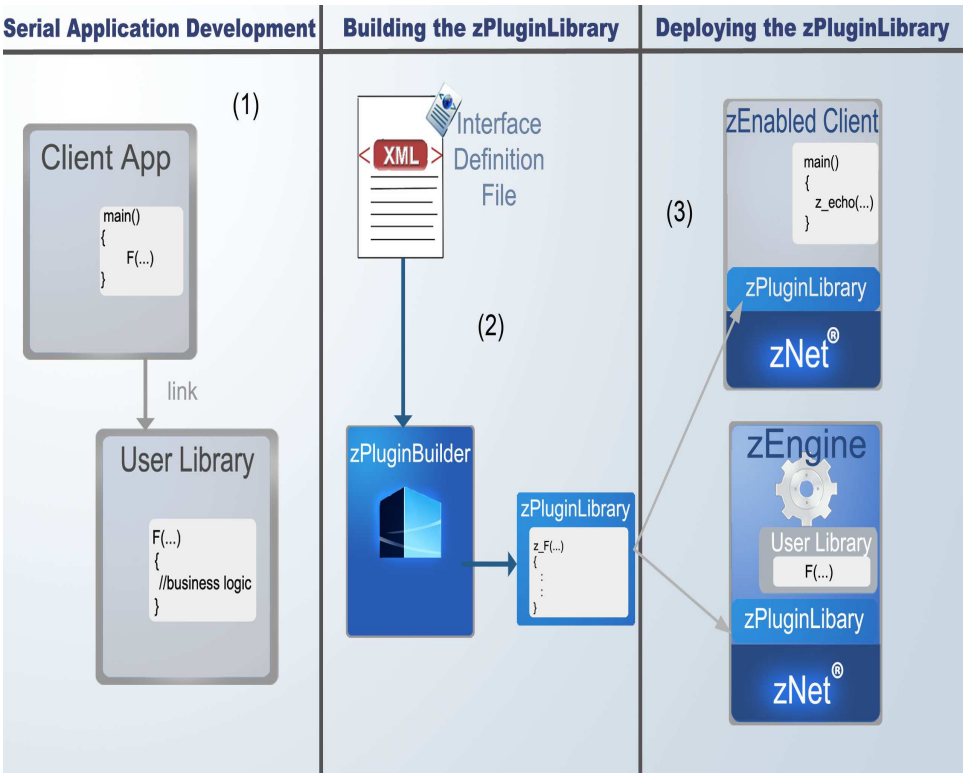

Figure IV.4: zEnabling a Serial Application with zFunction

The zFunction z_F acts as a client-side proxy and transparently dispatches asynchronous requests to the *zEngines*, thereby providing adaptive, distributed, and high performance computing on demand for client applications. Since a zFunction call z_F is meant to be a drop-in replacement for the invocation of its corresponding serial function call to F, the interface for the zFunction z_F must be straightforwardly derivable from that of the function F. This need necessitates the generation of custom zFunction code for each application, and zFunction provides the *zPluginBuilder* tool for this purpose.

The input to the zPluginBuilder tool is an *XML* file describing the function F, its input parameters, its output parameters, and the location of the library that contains the definition of the function F (shown in the middle section of figure IV.4). The output is a library (called the *zPluginLibrary*) with zFunction implementation z_F conforming to the same interface as the original function F. The generated library also references the original library defining the function F and is auto-loaded by *zEngine* (based on the location of the library specified

in the XML file). The generated *zPluginLibrary* is linked by both the client application as well as the *zEngine* (see the right side of figure IV.4).

The *zPluginLibrary* that is linked by both the client and the server applications serve two purposes. On the client side, the *zPluginLibrary* initiates the zFunction adaptive distributed computing middleware zNet to provide scalable, efficient, parallel, and highly available distributed communication between the clients and the servers in an application transparent manner. On the server side, the *zPluginLibrary* just delegates the calls made from the client-side *zPluginLibrary* (on behalf of the client applications) to the function F defined in the library created by the service developers.

Thus, with a minimal amount of development effort, zFunction users obtain a versatile, production-quality parallelized application ready to be deployed on any network.

IV.3 Parallel Application Development using zNet API

The zNet API provides a real-time, high-performance computing environment that enables rapid development of zNet-enabled distributed parallel computing applications and cooperating service tasks by simplifying and automating key distributed programming tasks, including service discovery, dynamic load balancing with real-time feedback, connection management, binary data transfer protocols, reliable multicast communication, flow control, parameter (de)marshaling, event/request demultiplexing, fault detection and recovery, service activation and management, concurrency and synchronization. zNet enables collaboration between distributed services in collocated and distributed HPC environments and is optimized for high-speed messaging, computation, and transactional services. These capabilities make zNet well-suited for mission-critical and time-sensitive applications. It is also well-suited for retrofitting legacy applications to exploit the power of multi-core processors.

IV.4 zNet Infrastructure Patterns

zNet API supports the following distribution infrastructure patterns for distributed parallel application development:

- **The Broker [12] pattern** which enables decoupled application components communication using twoway method invocations. zNet provides both synchronous and asynchronous twoway method invocation interfaces. Multiple client/server sessions can be created and started from within a single zNet-enabled application process. The participating components can reside in (1) the same process, (2) different processes on the same computer, or (3) remote computers.

- **The Messaging [12] pattern**, which enables services and applications to interact by exchanging oneway messages. Applications and services can use zNet to exchange messages with explicitly named receivers via reliable and/or "best effort" delivery semantics. zNet can also notify senders when reliable messages are dispatched to receivers. Moreover, zNet can exchange any native or custom data type on the network, as long as developers provide C++ insertion and extraction operators to encode/decode those data types using the OMG Common Data Representation (CDR) standard.

- **The Publisher-Subscriber [12] pattern**, which enables services and applications to interact by exchanging events asynchronously in a one-to-many configuration. Applications and services can use zNet to exchange events via reliable and/or "best effort" delivery semantics. zNet can also provide subscribers with the last published event upon subscription, thereby supporting operations in environments dominated by infrequent publishing and fleeting subscribers. Moreover, zNet can exchange any native or user-defined data type on the network, as long as developers provide C++ insertion and extraction operators to encode/decode those data types using the OMG Common Data Representation (CDR) standard.

IV.5 zNet API

zNet API contains several APIs that are based on the patterns discussed in section IV.4 and are used for distributed parallel application development.

IV.5.1 ZBroker Client API

This API facilitates the development of client applications using the Broker pattern. When participating tasks reside over remote computers, twoway method invocations between the ZBroker-enabled tasks are governed by the zNet Load Balancer, which zNet creates automatically when the client-side API is used by an application. The zNet Load Balancer optimizes workload across heterogeneous networks based on real-time feedback from processing services that are part of the zNet computing environment. *ZBroker* can run each twoway method invocation session between participating tasks concurrently without waiting or depending on any other session within the same process. For each session, *ZBroker* caches requests on the client-side network and resubmits them if associated server task becomes unavailable. *ZBroker* ensures that the client of each session receives replies, even when there are faults in its associated server task.

For distributed delivery, *ZBroker* transmits native and/or user-defined data types across the network, using C++ insertion and extraction operators to encode/decode those data types. Figure IV.5 shows an example of user-defined data structure declaration for use by *ZBroker API*. For collocated delivery, this encoding/decoding step is omitted and all data types are passed directly to other threads for processing. In both cases there is no need to inherit application tasks from any ZBroker-specific base classes since ZBroker handles the transfer of data in its native form. To define the C++ insertion and extraction operators to *ZBroker*, application developers simply declare the data type (which could be a struct or a class along with the C++ insertion and extraction operators), data type name, and an optional DLL that hosts the data type via the *TC_EVENTID_DECLARE_NAME(TYPE, NAME, DLL_NAME)* macro.

```
struct optionsdata
{
  int day;
  int optid;
  double strike;
  double maturity;
  double price;
  double stock;
  double rate;
};

inline bool operator<< (ACE_OutputCDR& strm, const optionsdata& s)
{
  return (strm << s.day) &&
         (strm << s.optid) &&
         (strm << s.strike) &&
         (strm << s.maturity) &&
         (strm << s.price) &&
         (strm << s.stock) &&
         (strm << s.rate);
}

inline bool operator>> (ACE_InputCDR& strm, optionsdata& s)
{
  return (strm >> s.day) &&
         (strm >> s.optid) &&
         (strm >> s.strike) &&
         (strm >> s.maturity) &&
         (strm >> s.price) &&
         (strm >> s.stock) &&
         (strm >> s.rate);
}
```

Figure IV.5: Example of user-defined datatype declaration using zNet API

Communications in *ZBroker* are based on types, which allows tasks to exchange instances of any native or user-defined type via the template-based *ZBroker::z_call()* API and by the *ZBroker::Client* class. Any such instances can be treated as an input/output (inout) type. When *ZBroker::z_call()* is invoked, its template data argument serves as an inout parameter, which is usually represented as a structure with some members designated as input data and some as storage for expected output replies.

The *ZBroker API* provides the following features:

- **Dynamic load balancing of asynchronous and synchronous invocations**, which routes calls to the least loaded servers.

- **Sticky engine**, which allows direct calls to desired servers that bypass zNet's Load Balancer and Routing algorithms.

- **Remote and local calls.** *ZBroker::z_call()* method can be used for asynchronous remote method invocations and *ZBroker::l_call()* method can be used for asynchronous local in-process invocations that optimizes in-process communication by avoiding marshaling and copying of parameter data. *ZBroker::z_sync_call()* and *ZBroker::l_sync_call()* methods are their synchronous counterparts.

- **Barrier and callback.** To process the results of asynchronous operations, zNet provides two mechanisms–barrier synchronizers *e.g. ZBroker::process_all()* method and asynchronous callbacks–that support a wide range of applications.

- Stream processing. ZBroker supports multi-threaded result processing callbacks, with high/low watermark that can be set using *ZBroker::hwm()* and *ZBroker::lwm()* methods, to process streams of high frequency data, with no predefined size of the request data set. Low and high watermark determine the maximum and minimum number of requests that can stay in the client/server request queue.

The *ZBroker API* consists of the following components:

1. **ZBroker Client API** that facilitates development of client applications using the Broker pattern.

2. **ZBroker Server API** that supports the binding of global functions and methods of user-defined components to the data types passed from the client side of zNet.

IV.5.2 ZMessaging API

This API facilitates the development of client and server applications using the Messaging pattern, which structures software systems whose services interact by exchanging oneway messages. Applications using zMessaging can exchange any native or user-defined data type with named receivers on the network, as long as developers provide C++ insertion and extraction operators to encode/decode those data types using the OMG Common Data Representation (CDR) standard.

ZMessaging::Sender locates *ZMessaging::Receiver* instances by their unique names and then sends oneway messages to them. To process oneway messages from the *ZMessaging::Sender*, the *ZMessaging::Receiver* provides mechanisms for registering message handler functions/methods that are dispatched automatically by *ZMessaging*. A *ZMessaging::Sender* uses an internal queue for outgoing messages, which are serviced by dedicated thread. This queue can be made persistent, thereby providing transactional support for messages. If a persistent *ZMessaging::Sender* instance fails, all messages currently stored in its queue will be recovered into the queue of the next *ZMessaging::Sender* instance initialized with the name used by the failed sender.

The *ZMessaging::Sender* and *ZMessaging::Receiver* can be started independently. If a *ZMessaging::Receiver* targeted by *ZMessaging::Sender* is not yet on-line, the *ZMessaging::Sender* can load messages in its queue (up to a high water mark) and attempt to connect and send them. This activity is conducted by a dedicated thread servicing an outbound queue. In case of failure of established connection, the *ZMessaging::Sender* will attempt

reconnection logic with constant interval of 5 sec and also provides disconnect callback hook for custom logic using *ZMessaging::Sender::register_disconnect_cbk()* method.

The ZMessaging API provides the following features:

- **Configurable delivery policies**, where messages can be sent via reliable and/or "best effort" delivery semantics using *ZMessaging::Sender::send()* method.
- **Automatic acknowledgements**, where senders can be notified when reliable messages are dispatched to their named receivers.
- **Selective subscription**, where receiver functions and/or methods can be registered to process specific data types using *ZMessaging::Receiver::register_handler()*
- **Dynamic service (re)configuration**, where services can be added and/or removed at runtime using *ZMessaging::Receiver::start()*, *ZMessaging::Receiver::stop()*, *ZMessaging::Sender::start()* and *ZMessaging::Sender::stop()*
- **Transparent failover**, with automatic reconnection if a receiver crashes.

The *ZMessaging API* consists of the following components:

1. **ZMessaging Sender API** that facilitates development of event publishing components using the Messaging pattern.
2. **ZMessaging Receiver API** that facilitates development of service objects that process one way messages sent by clients.

IV.5.3 ZPubSub API

This API facilitates the development of client/server applications using the Publisher-Subscriber pattern, which structures software systems whose components interact by exchanging events asynchronously in a one-to-many configuration. *ZPubSub* communication is based on types so that any object of native or user-defined type can be multicasted to

a set of services running in the pool. This functionality is supported on the client side by publishing an event using *ZPubSub::publish_event()* method that triggers an action or by publishing data using *ZPubSub::publish_cache()* method that enables the zNet distributed cache. *ZPubSub* enables client tasks to optimize the transmission of large and/or infrequently changing input data objects via a "send-once-use-across-multiple-calls" caching architecture. Instead of transmitting data with every request, the client publishes it to all services only when the value of the data changes, thereby eliminating redundant communication since the servers store the published data. Server components that subscribe to data cache using *ZPubSub::subscribe_cache()* method, can asynchronously access the cached data using *ZPubSub::get_cache()* method. Server components can also register custom event processing methods using *subscribe_event()* method, which gets dispatched automatically by zNet when an event of given type is published. Applications using *ZPubSub* can exchange any native or user-defined data type on the network, as long as developers provide C++ insertion and extraction operators to encode/decode those data types using the OMG Common Data Representation (CDR) standard.

The *ZPubSub API* provides the following features:

- **Data publishing**, where common data is broadcast across all services (and access is synchronized).
- **Event publishing**, where events are broadcast to all the services, which triggers function execution across all services.
- **Configurable delivery policies**, where events can be published via reliable and/or "best effort" delivery semantics using *ZPubSub::publish_event()* and *ZPubSub::publish_cache()* methods.
- **History-aware subscriptions**, where *ZPubSub* can provide any new subscribers with

the cached version of the last published event or data upon subscription using *ZPubSub::subscribe_event()* and *ZPubSub::subscribe_cache()* methods thereby supporting operations in environments dominated by infrequent publishing and fleeting subscribers.

ZPubSub API consists of the following sections:

1. **ZPubSub Client API** that facilitates development of publishing client applications using the Publisher-Subscriber pattern.

2. **ZPubSub Server API** that facilitates development of servers applications that processes asynchronous events sent by the client and use cached data for computations.

A complete listing of all zNet API methods can be found at `www.zircomp.com/downloads/docs/html_znet/index.html`.

IV.6 Resolving Distributed and Parallel Application Design Challenges with Zircon Middleware Software

We now describe how the zFunction components shown in figure IV.1 address the key distributed and parallelize application design requirements summarized in Chapter III.

Resolving requirement 1: Providing an information service for automatic discovery and addressing of remote computation servers for distributed computing. *The Configuration Environment* (TCE) acts as an information service for Zircon middleware framework and bootstraps all the applications in the network. All other components in the Zircon software deployment (including the clients and the zEngines that perform the remote computations) register with the TCE at startup. This process allows TCE to identify network settings such as the host IP addresses, network subnet identification, multicast addresses. TCE employs a handshaking protocol that provides network information to all Zircon middleware components, so that applications can communicate with each other at runtime without collaborating with TCE.

Resolving requirement 2: Providing easy to use programming frameworks for remote distributed computation. Zircon middleware framework provides an easy-to-use utility called the zPluginBuilder that automatically generates zPluginLibraries that serve as adapters between the generic Zircon middleware and specific client/server applications. These adapters emit efficient (de)marshaling code that enables Zircon middleware to transparently support remote communication across heterogeneous platforms and networks. Applications can be easily parallelized either by using zEnabling capabilities of Zircon middleware or by use of zNet API that requires minimal code changes for converting a serial application into a parallel application.

Resolving requirement 3: Providing effective resource management of remote computation servers. When zEnabled client requests are sent to a server pool, Zircon middleware software's intelligent load-balancer is used to evenly distribute work amongst existing computation servers in real-time, as shown in figure IV.6. By spreading computations evenly across all the available servers, zFunction maximizes resource allocation for critical applications and also ensures that hardware resources are utilized to their fullest.

Resolving requirement 4: Providing application-transparent multi-layer fault tolerance. Zircon middleware also ensures that application execute irrespective of hardware failures, and transparently provides fault recovery and failover by re-executing requests on servers that are still operational. As shown in figure IV.6, Zircon middleware keeps track of the execution history of each request and to which zEngine the request has been sent to. When a zEngine failure is detected, it automatically resends the request to a new or a rejuvenated zEngine and ensures that the computations are performed irrespective of hardware failures.

Resolving requirement 5: Providing implicit scalability using core-aware multithreading. Zircon middleware software performs parallelization by executing multiple instances of an application's parallelizable function simultaneously in zEngine processes running on different machines on a network. Zircon software provides implicit concurrency

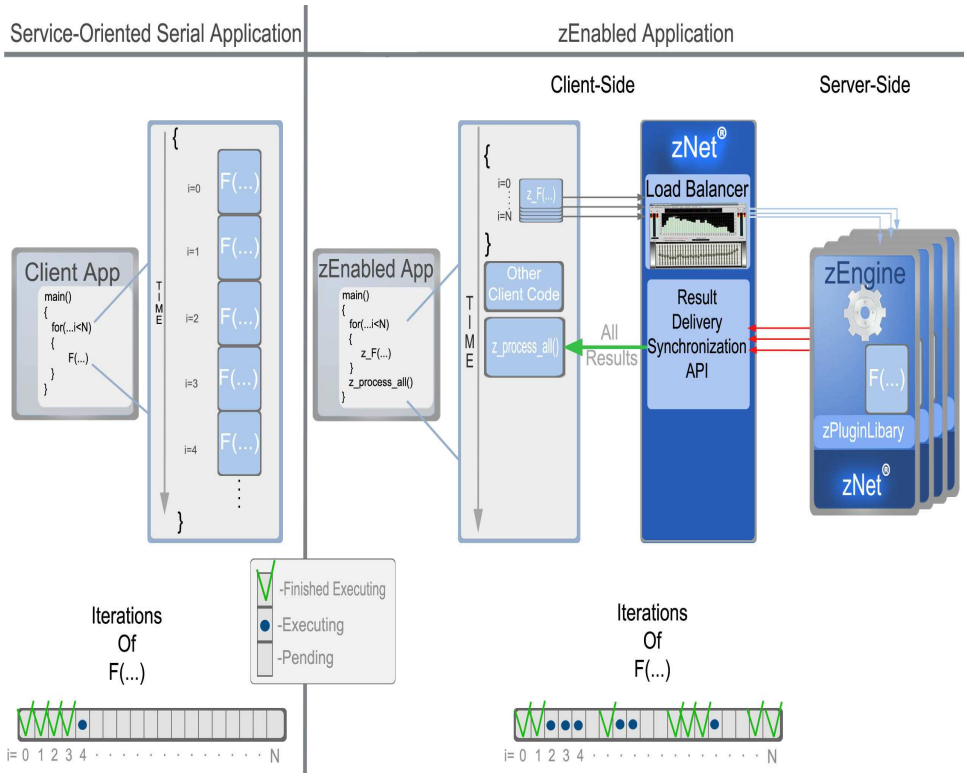

Figure IV.6: Parallel Application Development with zFunction

support and automatically creates threads for distributing requests to different servers and also synchronizes those threads using messages and locks. On multi-core machines, Zircon middleware software runs multiple instances of application's parallelizable function in multiple worker threads on a single zEngine process in a thread-safe manner and thus provides highly efficient utilization of multi-core machines with minimum overhead.

CHAPTER V

BENCHMARKING EXPERIMENTS AND RESULTS

We developed some financial computation applications, as described in chapter II, in order to benchmark the runtime performance of OpenMPI and Zircon parallel computing middleware software. We developed parallel implementations of binomial option pricing and Heston model calibration applications in order to test their performance on multicore machines and in high performance computing (HPC) clusters using multiple servers. In the rest of this chapter, we describe all the experiments and their results in detail.

V.1 Experiment Setup

All experiments were run on upto 8 Intel-Xeon 1520 series dual-processor/dual-core (for a total of upto 32 cores) 1.86 GHz machines running on 64-bit Red-Hat Enterprise Linux v2.6 and connected using Gigabit Ethernet.

V.2 Experiment 1

V.2.1 Objective

The objective of this experiment is to compare the performance of OpenMPI and zNet API in a HPC cluster environment and highlight the impact of efficient load-balancing algorithms for performance speed-up.

V.2.2 Experiment Description

For this experiment, we developed binomial option pricing application that evaluates option prices for 1000 american options.

As shown in figure V.1, the serial implementation of binomial option pricing application invokes the option pricing algorithm in a loop for n different options in a serial order and

```
// binomial option pricing algorithm
double option_price_call_american_binomial(...)
{
  ....
};

// functor that invokes option pricing algorithm
struct Invoke_OP_Call
{
  void operator () (Binomial_Option_Pricing_Request &iter)
  {

    iter.option_price =
      option_price_call_american_binomial (iter.cur_stock_price,
                                           iter.strike_price,
                                           iter.risk_free_rate,
                                           iter.volatility,
                                           iter.t,
                                           iter.n_steps);


  }
};

// main
int main(int argc, char** argv)
{

  // Read input
  ...

  // Calculate option prices
  std::for_each (requests.begin (),
                 requests.end (),
                 Invoke_OP_Call ());

  ...

  //process results
  ...

  return 0;
}
```

Figure V.1: Serial Implementation of Binomial Option Pricing Application

all function invocations are independent of each other. Therefore, this application can be parallelized by running multiple invocations of binomial option pricing algorithm in a loop.

We developed parallel implementation of this application using OpenMPI in which all requests to binomial option pricing algorithm were equally distributed asynchronously [1] amongst all the server processes using *MPI_Send()* and *MPI_Receive()* methods in a *for* loop and then all results were collected in a second *for* loop.

However, the processing time for each binomial option price computation request varied from a few milliseconds upto 25 seconds, which gave us a largely heterogenous request set. In order to optimize performance for heterogenous load, we also developed a load-balancing OpenMPI implementation of binomial option pricing application. In this load-balanced implementation, we used *first-in-first-out* scheduling for request allocation, so that more requests are sent to least-loaded servers for better CPU utilization.

In zNet based implementation, we parallelized the application by using *ZBroker API*. Instead of invoking *option_price_call_american_binomial()* function directly as shown in serial implementation in figure V.1, parallelized implementation uses *ZBroker::z_call()* to asynchronously distribute the calls to all the running servers in a load-balanced fashion using Broker pattern. The results are later automatically collected by the zNet's response processing threads.

V.2.3 Results

In this experiment, all the three above-mentioned parallel implementations of binomial option pricing application were run using four, eight, sixteen and thirty-two servers to compute prices for 1000 binomial options.

The results of this experiment are shown in figure V.2. The results show that the

[1] OpenMPI's implementation of MPI_Send() and MPI_Receive() functions uses buffering for small-sized messages i.e. MPI_Send() call do not need to block if a matching MPI_Receive call is not posted. It copies data into a buffer and returns control to the program. However for large-sized messages, it tries to send and receive data synchronously

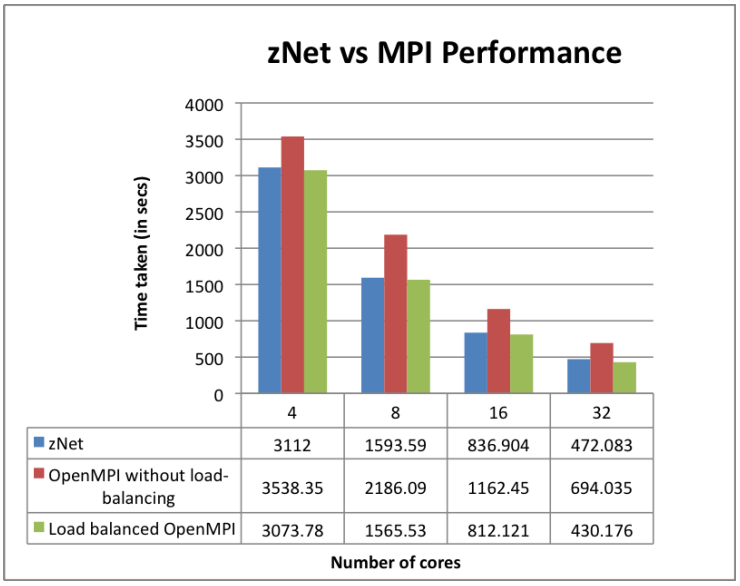

Figure V.2: Performance Results for OpenMPI and zNet based implementation of Binomial Option Pricing Application running on a HPC cluster

load-balanced implementations of zNet API and OpenMPI perform better than the non-load balanced implementation using OpenMPI. The results highlight the fact that load-balancing mechanisms are required to improve system performance, in case of largely heterogenous computation requests set. In the figure V.2, the results also show that both load-balanced OpenMPI implementation and zNet API based implementation have comparable performance. In zNet implementation, developers do not need to write any code for load-balancing the application and it is transparently handled by the zNet middleware. However, in case of OpenMPI, users need to write code for implementing load-balancing which requires good understanding of load-balancing algorithms and also increases the development time. The results highlight the ease-of-use/ease-of-programming of zNet API in comparison to OpenMPI.

V.3 Experiment 2

V.3.1 Objective

The objective of this experiment is to compare the performance of OpenMPI and Zircon's zFunction in a HPC cluster environment and highlight the benefits of Zircon's zEnabling feature for quick parallelization of serial applications.

V.3.2 Experiment Description

For this experiment, we developed a load-balanced, parallel implementation of Heston calibration application using OpenMPI and parallelized serial implementation of Heston calibration application using Zircon software's zFunction capability.

As described in section II.2, the Heston calibration application uses an optimization routine that minimizes a 5-dimensional objective function (one dimension for each parameter in the Heston model). The optimization routine is implemented by using the NAG C library's *nag_opt_bounds_no_derive()* minimization function to run 100 iterations of the objective function and find the minima. The maximum number of iterations is capped to 100, so the model calibration is considered to have failed if the procedure does not converge by that point. The differences among the calibration models' convergence properties contribute to significant fluctuations in the calculations' execution times, making the models *heterogeneous*, *e.g.*, model calibration time can vary from 1 millisecond up to 105 seconds. The sequential implementation of the Heston calibration application read input data and calibrated 1,065 models in $\sim$9 hours by invoking the *calibrate_heston()* function 1,065 times in a loop. All *calibrate_heston()* function invocations runs independent of each other, so application's performance can be improved significantly by processing multiple invocations in parallel. Heston calibration application uses historical option pricing data to calibrate the parameters. So, application performance can be further improved by broadcasting historical option pricing input data to all the servers at start-up, instead of repeatedly sending it to every request.

We developed load-balanced parallel implementation of Heston calibration application using OpenMPI that distributed all *calibrate_heston()* function invocations amongst all servers in a load-balanced manner. We used OpenMPI's collective communication function *MPI_Bcast ()* in order to broadcast input data vector to all the servers. Figure V.3 shows how the master process distributes all requests in a load-balanced manner to slave processes in OpenMPI based parallel implementation of Heston calibration application.

For zFunction based implementation of Heston calibration application, we used *zPluginBuilder* in order to generate function adaptors for *calibrate_heston()* function library. In this implementation, input data is broadcasted to the servers as *zCache* data and this information is provided in the XML file which is given as an input to *zPluginBuilder*. No code changes are required in user library for special handling of *zCache* data. Whenever *zCache* data is required by the user library, zNet middleware takes care of providing it to the user library from the cache. The client application is modified to publish *zCache* data at start-up and invoke *z_calibrate_heston()* function for request processing, instead of *calibrate_heston()* function. Figure V.4 shows the code for zFunction client application. We deployed user library in *zEngines* running on the HPC server using the zAdmin utility and started the *zClient* for sending remote computation requests.

V.3.3 Results

In this experiment, both the zFunction and OpenMPI based parallel implementations of Heston calibration application were run using four, eight, sixteen and thirty-two servers to calibrate 1065 models.

The results of the experiment are shown in figure V.5. The results show that both the OpenMPI implementation as well as zFuntion implementation have comparable performance. Minor differences in the total run-time of the application using OpenMPI and zFunction can be attributed to the heterogeneity of the requests load. ZFunction enables quick parallelization of serial applications, while OpenMPI requires application re-write

```
static int master (...)
{
  // Get communicator size, process rank
  MPI_Comm_size (MPI_COMM_WORLD, &n_tasks);
  MPI_Comm_rank (MPI_COMM_WORLD, &my_rank);

  // Read input
  ...

  // Broadcast size of input data
  MPI_Bcast (&v_optionsdata_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
  MPI_Bcast (&v_optionsdata.front (), v_optionsdata_size,
             od_data_type, 0, MPI_COMM_WORLD);

  for (int m = 0; m < cal_range; ++m)
    {
      ...
      MPI_Send (&call_data, 1, call_data_type, rank,
                CALL_DATA_TAG, MPI_COMM_WORLD);
    }

    // Send the next request to first available worker process
    while (count_day <= end_day)
      {
        // Wait for response
        MPI_Recv (&param_calib_SV, 1, sv_data_type,
                  MPI_ANY_SOURCE, SV_DATA_TAG, MPI_COMM_WORLD,
                  &status);

        // process result
        ...

        // Send the next request to worker process that sent response
        MPI_Send (&call_data, 1, call_data_type, status.MPI_SOURCE,
                  CALL_DATA_TAG, MPI_COMM_WORLD);
        ++count_day;
      }

    // Wait for all responses
    while (reply_count < num_requests)
      {
        MPI_Recv (&param_calib_SV_2, 1, sv_data_type,
                  MPI_ANY_SOURCE, SV_DATA_TAG, MPI_COMM_WORLD,
                  &status);

        // process result
        ...
      }
    }
  ...
  return 0;
}
```

Figure V.3: OpenMPI implementation code for Heston calibration application

```
int main(int argc, char** argv)
{
  std::vector<optionsdata> v_optionsdata;

  // Read input
  ...

  // Initialize the zFunction computation environment.
  if (z_init (argc, argv) < 0)
    return -1;

  // Broadcast input data
  z_cache_vector_optionsdata (v_optionsdata);

  for (int m = 0; m < cal_range; ++m)
  {
    z_calibrate_heston (...);
  }

  // Wait for all request to finish
  z_process_all ();

  // process results
  ...

  // Terminate zFunction computation environment.
  z_fini ();

  return 0;

}
```

Figure V.4: zFunction Client code for Heston calibration application

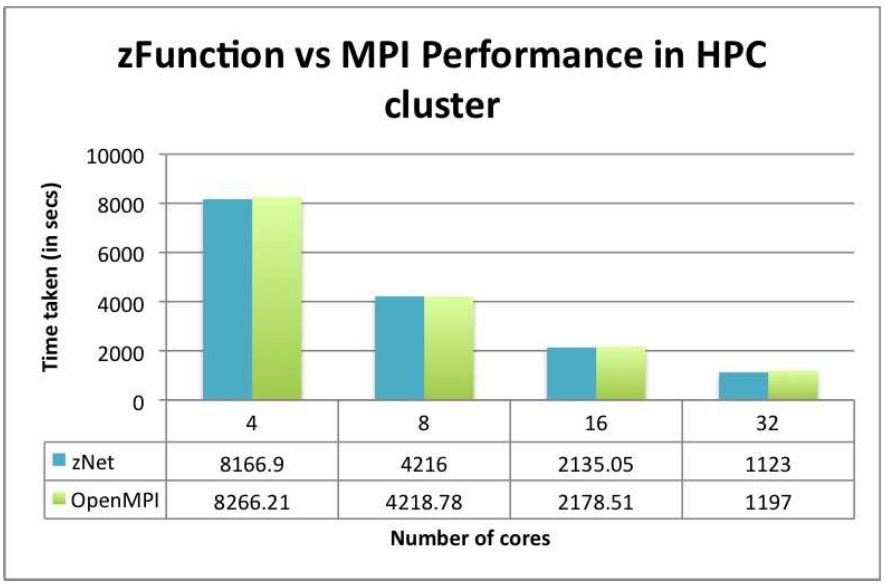

Figure V.5: Performance Results for OpenMPI and zFunction based implementation of Heston Model Application running on a HPC cluster

for parallelization. The results of this experiment show that the zFunction is a highly effective, efficient and easy-to-use technique for parallelizing serial applications with minimal code modifications, in contrast to OpenMPI which incurs a larger development cost for parallel application development.

V.4 Experiment 3

V.4.1 Objective

The objective of this experiment is to compare the performance of OpenMPI and zNet API in shared memory environment using multi-core machines.

V.4.2 Experiment Description

For an OpenMPI job/application, when we start *n* processes on a mutli-core machine with n cores, OpenMPI automatically starts using *smBTL* for communication. The *sm BTL* (shared-memory Byte Transfer Layer) is a low-latency, high-bandwidth mechanism for transferring data between two MPI processes via shared memory. For this experiment,

we used the same load-balanced OpenMPI implementation that we had developed for experiment 2 and ran it by starting multiple server processes on a single multi-core machine. In general, the number of slave processes is equal to the number of cores on a machine.

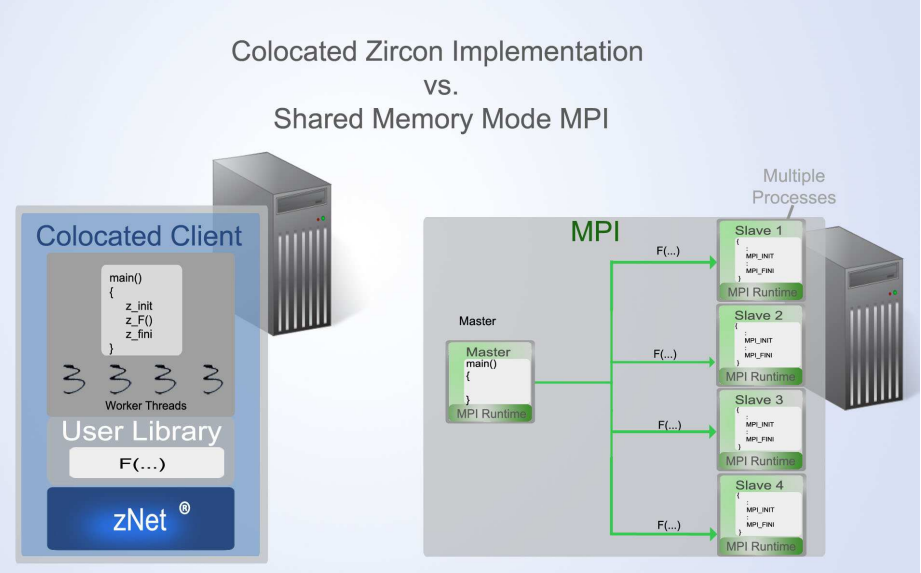

Figure V.6: Colocated zNet Application vs Shared Memory OpenMPI Application

We developed colocated implementation of Heston calibration application using Zircon's zNet API. Colocated parallelization enables applications to run in a single process and handle multiple requests in parallel in multiple worker threads in the same process, which is best suited for applications that run on a standalone multi-core machine. In this implementation of the Heston calibration application, the zNet *z_init()* method is invoked to initialize and start the zNet runtime, which internally spawns multiple worker threads that run the *calibrate_heston()* function in parallel. The zNet *l_call()* method forwards calibration requests to worker threads that process the requests in parallel on multiple cores on a standalone machine. Figure V.6 shows how the zNet based implementation distribute the work into multiple worker threads, while the OpenMPI based implementation distributes the requests to multiple slave processes.

V.4.3 Results

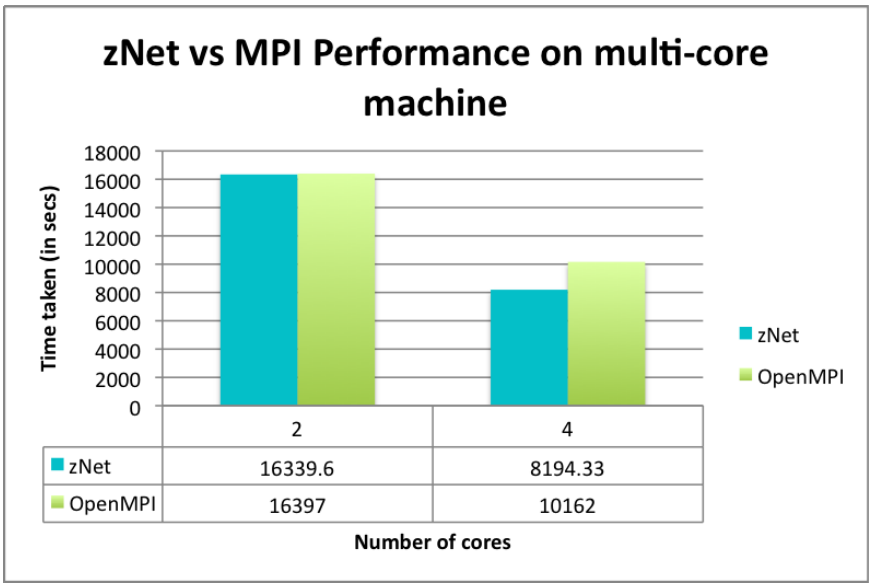

Figure V.7: Performance of zNet vs OpenMPI Implementation of Heston calibration application on a multi-core machine

In this experiment , both the colocated zNet implementation and OpenMPI implementation of Heston calibration application were run using dual-core[2] and quad-core machines to calibrate 1065 models. The results of the experiment as shown in figure V.7 demonstrate that the colocated zNet implementation of Heston calibration application is faster compared to the OpenMPI implementation. MPI implementation provides parallelization by starting multiple slaves/worker processes, while the zNet API based implementation provides parallelization by running multiple worker threads. When we ran the experiment on a quad-core machine, we started 4 MPI slaves process and 1 MPI master process that distributed the requests amongst all slave processes. While running the same experiment using zNet implementation, we started the application as a single process with 4 worker threads. The overhead of running multiple processes is higher than the overhead of running

[2]One processor was disabled to simulate dual-core machine behavior for some experiments.

multiple threads in the same process and due to this reason, zNet runtime is able to provide better utilization of computation resources in comparison to OpenMPI.

CHAPTER VI

RELATED WORK

This chapter compares and contrasts Zircon software with other related techniques available for parallel application development and deployment.

Aspect-Oriented Programming (AOP). Recent work has focused on using AOP [23] to separate parallelization concerns from application specific source code [19, 24, 28]. Such research provide a strong motivation for efforts that aim to make parallel programming more intuitive and less error-prone, as there is a strong decoupling in the roles played by domain experts (who write application specific code) and parallel programming experts (who write source code that deal with parallel programming concerns). However, the programmers are unnecessarily exposed to AOP technology. Further, if such research is used to provide the range of capabilities that Zircon middleware software offers (capabilities such as fault-tolerance, advanced load balancing, direct data transfer), newer technologies are required that support composition of aspects. In contrast, Zircon middleware software provides the benefits of parallel programming in a simple manner (that is easier to code); but is also highly sophisticated in the capabilities it provides.

Grid computing middleware. Many projects have explored the idea of utilizing distributed computing architectures to accelerate complex calculations on top of under-utilized network of processors or clusters. Some well-known examples include the SETI@Home [8] and BOINC [7] projects, which employ under-utilized networked processors to perform computational tasks. Likewise, Frontier (`www.frontier.com`) provides grid software for utilizing available processors to accelerate parallel applications. In general, in these approaches the client nodes communicate via a centralized master node to submit jobs, which can increase latency, incur contention that causes performance bottlenecks, and yields a

single point of failure. In contrast, Zircon software provides a highly optimized middleware infrastructure for communication, as well as a set of tools for rapid development, generation, and deployment of parallel software in decentralized networked environments.

Middleware for accelerating financial engineering applications. Prior work has also focused on developing and/or applying grid architectures and grid applications for financial services applications. For example, [25] discusses practical experiences associated with data management and performance issues encountered in developing financial services applications in the IBM Bluegene supercomputer [6]. Likewise, PicsouGrid [10] is a fault-tolerant and multi-paradigm grid software architecture for accelerating financial computations on a large scale grid. Other grid-based systems include Platform Computing (`www.platform.com`), DataSynapse, (`www.datasynapse.com`), and Microsoft HPC (`www.microsoft.com/hpc`), which provide distributed software environments for financial computations. Zircon middleware software differs from these technologies in its ease of use and integration, its real-time performance, its ability to handle both small as well as large scale computations, its support for portable architectures and platforms, and its advanced parallel programming features such as application-transparent fault-tolerance, load balancing, and implicit shared-memory thread programming.

CHAPTER VII

CONCLUSION

OpenMPI is the most commonly used standard API widely used by the parallel application development community for developing parallel computing applications that can efficiently utilize the hardware capabilities of multi-core machines, HPC clusters and clouds. However, in addition to OpenMPI, high-performance computing clusters have to use third party tools in order to perform cluster management activities such as job scheduling, resource monitoring, load-balancing and server deployments. It is not possible to quickly parallelize existing serial applications using OpenMPI because OpenMPI developers have to handle multiple issues related to distributed and network programming like synchronization, concurrency, load-balancing, fault-detection and recovery on a need-to-need basis for every application, which increases the cost of parallel application development.

This work compares the capabilities of OpenMPI framework with Zircon middleware software by developing some benchmark parallel computing applications and highlights the advantages of Zircon middleware software, which is better suited for parallel applications development due to its feature advantages as shown in Table VII.1, in comparison to OpenMPI.

Table VII.1: Features comparison between OpenMPI and Zircon Software

Features	**Zircon Software**	**OpenMPI**
Real-Time Scalability	Applications can be scaled to run on multiple nodes at run-time.	Node allocation for an application has to be done before startup and new nodes cannot be added at run-time.

Platform Independence	Zircon software is implemented atop ADAPTIVE Communication Environment(ACE) which is a portable C++ host infrastructure middleware and therefore can run on most general-purpose and real-time operating systems *e.g.* Windows XP, OS X, Linux and Solaris. It can run applications using mixed operating system environments.	OpenMPI is a low-level implementation of MPI standard. It is currently supported on Linux, OS X, Solaris and Windows. However, it does not have support for interoperability in mixed operating system environments

Programming Simplicity	Zircon Software can deploy binary versions of client libraries on a cloud/cluster of nodes and execute the computations in parallel. It automatically distributes the requests to different servers in a load-balanced manner and collects results from all the servers. It shields application developers from the complexity of distributing their applications and thus makes distributed computing easy and affordable for its users.	OpenMPI implementations require rewrite of existing implementations in a master-slave fashion, where master process distributes requests and data to slave processes and collects results from them. OpenMPI implementation does not have any support for load-balancing and fault-tolerance which makes it harder and tedious to develop applications using OpenMPI.
CPU Utilization	Zircon Software provides better CPU utilization when running applications in colocated mode as it processes computation requests in worker threads.	OpenMPI has no concept of threads and each application component runs as a different process, even on multicore machines, which has higher overhead in comparison to running multiple threads in a process.

Dynamic Load Balancing	Zircon software's load balancer transparently distributes the workload across all the servers and ensures that all available servers are fully utilized. Application developers do not need to implement any load balancing mechanisms within the application logic.	In OpenMPI, application developers need to implement load-balancing within the applications which requires rewriting existing applications.
Fault Tolerance	Zircon software automatically detects node failures and provides immediate failover and recovery by re-executing requests on active servers.	OpenMPI runtime environment has no inherent support for fault-tolerance and the active jobs/applications stops execution, in case of server failure. Developers can design fault-tolerant programs by catching error codes and implementing fault-recovery mechanisms in their applications, which an error-prone and tedious process.

Service Discovery	Zircon software can auto-detect the server nodes and distribute computation requests to active servers. Client application need not provide any server information on startup.	OpenMPI applications cannot auto detect servers. All servers for an OpenMPI job/application have to be allocated by resource managers on start-up. OpenMPI applications cannot detect any new servers added during the runtime of the application.
Monitoring Tools	Zircon software contains a utility called zAdmin that can be used for real-time monitoring of resources.	OpenMPI does not provide any such tools.

Zircon middleware software is best-suited for computation intensive financial applications that have highly heterogenous work-loads and have real-time scalability, load-balancing and fault-tolerance requirements. These applications can greatly benefit by the use of Zircon middleware software that has the plug-in capabilities to add computation resources at run-time to speed-up performance and is very easy to configure, program and use.

REFERENCES

[1] NAG C Library Description, . http://www.nag.co.uk/numeric/CL/cldescription.asp.

[2] Nag nag_opt_bounds_no_deriv (e04jbc) function documentation, . http://www.originlab.com/pdfs/nagcl07/manual/pdf/e04/e04jbc.pdf.

[3] OpenMP Home Page, . www.openmp.org.

[4] OpenMPI Home Page, . http://www.open-mpi.org.

[5] Zircon Home Page. http://www.zircomp.com.

[6] *et. al* Allen, F. Blue gene: a vision for protein science using a petaflop supercomputer. *IBM Syst. J.*, 40(2):310–327, 2001. ISSN 0018-8670.

[7] David P. Anderson. Boinc: A system for public-resource computing and storage. In *GRID '04: Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2256-4. doi: http://dx.doi.org/10.1109/GRID.2004.14.

[8] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11): 56–61, 2002. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/581571.581573.

[9] Jaiganesh Balasubramanian, Alexander Mintz, Andrew Kaplan, Grigory Vilkov, Artem Gleyzer, Antony Kaplan, Ron Guida, Pooja Varshneya, and Douglas Schmidt. Adaptive parallel computing for large-scale distributed and parallel applications. In *Proceedings of the Workshop on Data Dissemination for Large-scale Complex Critical Infrastructures (DD4LCCI)*. in conjunction with EDCC 2010, Valencia - Spain, 2010.

[10] Sebastien Bezzine, Virginie Galtier, Stephane Vialle, Francoise Baude, Mireille Bossy, Viet Dung Doan, and Ludovic Henrio. A fault tolerant and multi-paradigm grid architecture for time constrained problems. application to option pricing in finance. In *E-SCIENCE '06: Proceedings of the Second IEEE International Conference on e-Science and Grid Computing*, page 49, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2734-5. doi: http://dx.doi.org/10.1109/E-SCIENCE.2006.7.

[11] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994. URL http://www.lam-mpi.org/download/files/lam-papers.tar.gz.

[12] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4*. Wiley and Sons, New York, 2007.

[13] John C. Cox, Stephen A. Ross, and Mark Rubinstein. Option Pricing: A Simplified Approach. *Journal of Financial Economics*, 4, 1979.

[14] Graham E. Fagg, Edgar Gabriel, Zizhon Chen, Thara Angskun, George Bosilca, Antonin Bukovsky, and Jack J. Dongarra. Fault tolerant communication library and applications for high performance computing. In *In Los Alamos Computer Science Institute Symposium*, pages 27–29, 2003.

[15] MPI Forum. Message Passing Interface Forum. www.mpi-forum.org.

[16] MPI Forum. MPI Standard 2.0. www-unix.mcs.anl.gov/mpi/, 2000.

[17] Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, and Timothy S. Woodall. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *In Proceedings, 11th European PVM/MPI UsersÕ Group Meeting*, pages 97–104, 2004.

[18] Richard L. Graham, Sung eun Choi, David J. Daniel, Nehal N. Desai, Ronald G. Minnich, Craig E. Rasmussen, L. Dean Risinger, and Mitchel W. Sukalski Introduction. A network-failure-tolerant message-passing system for terascale clusters. In *International Journal of Parallel Programming*, pages 77–83, 2003.

[19] Bruno Harbulot and John R. Gurd. Using aspectj to separate concerns in parallel scientific java code. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 122–131, New York, NY, USA, 2004. ACM. ISBN 1-58113-842-3. doi: http://doi.acm.org/10.1145/976270.976286.

[20] Steven Heston. A closed-form solution for options with stochastic volatility with applications to bond and currency options. *Review of Financial Studies*, 6:327–343, 1993.

[21] David Horn, Eva Schneider, and Grigory Vilkov. Hedging options in the presence of microstructural noise. *SSRN eLibrary*, 2007.

[22] J. C. Hull. *Options, Futures and Other Derivatives*. Prentice-Hall, 1999.

[23] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[24] Marcio E. F. Maia, Paulo H. M. Maia, Nabor C. Mendonca, and Rossana M. C. Andrade. An aspect-oriented programming model for bag-of-tasks grid applications. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 789–794, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2833-3. doi: http://dx.doi.org/10.1109/CCGRID.2007.19.

[25] Thomas Phan, Ramesh Natarajan, Satoki Mitsumori, and Hao Yu. Middleware and performance issues for computational finance applications on blue gene/l. *Parallel and Distributed Processing Symposium, International*, 0:371, 2007. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2007.370561.

[26] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, Boston, 2002.

[27] Douglas C. Schmidt and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2002.

[28] J.L. Sobral. Incrementally developing parallel applications with aspectj. *Parallel and Distributed Processing Symposium, International*, 0:95, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2006.1639352.