Adaptive Parallel Computing for Large-scale Distributed and Parallel Applications

Jaiganesh Balasubramanian[†], Alexander Mintz[†], Andrew Kaplan[†],
Grigory Vilkov[†], Artem Gleyzer[†], Antony Kaplan[†], Ron Guida[†],
Pooja Varshneya[‡], Douglas C. Schmidt[‡]
[†]Zircon Computing LLC, Wayne, NJ, USA
[‡]Vanderbilt University, Nashville, TN, USA

Abstract

This paper presents the structure and functionality of zFunction, which is an adaptive distributed computing platform that supports a user-friendly programming model for developing parallel processing applications. It allows developers to design software as if they are programming for a single computer and then it automatically takes care of data distribution and task parallelization activities on different cluster nodes or multiple CPU cores. zFunction thus substantially improves the performance of complex distributed applications that process a large amount of data in real time, mission critical systems. This paper uses a representative case study from the financial services domain to show how these types of applications can benefit from zFunction.

1. INTRODUCTION

Parallel programming is increasingly becoming important for researchers and developers of large-scale distributed and parallel applications in a number of domains, including financial risk assessment and modeling (*e.g.*, Value-At-Risk and historical calculations), real-time decision-making based on algorithmic feedback (e.g. market making, electronic strategy arbitrage, and high-frequency trading), and processing, archiving, storing and searching content repositories for enterprise content management systems (*e.g.*, news websites and web encyclopedias). With the advent of commodity multi-core processors and cloud computing systems, researchers and developers also need newer parallel programming techniques that can maximize the utilization of such systems.

Traditional parallel programming techniques, such as message passing [8] and shared memory grid computing middleware [15], have been applied by researchers in universities and national labs to develop and deploy enterprise-scale distributed and parallel applications. Parallel application development remains a challenging problem, however, in the domain of large-scale development of distributed and parallel 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 technologies are not platform agnostic.
- There is a steep learning curve involved in mastering these parallel programming paradigms.

To address these limitations, we have developed an adaptive distributed computing middleware called *zFunction* that enhances large-scale distributed and parallel applications by creating adaptive, real-time, and distributed computing on demand. zFunction 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.

Paper organization. The remainder of this paper is organized as follows: Section 2 describes a case study from the financial services domain to showcase the challenges of developing distributed and parallel applications; Section 3 explores the capabilities zFunction provides to simplify the development of large-scale distributed and parallel applications; Section 4 describes how zFunction provides solutions for the challenges described in the case study; Section 5 compares zFunction with related work; and Section 6 presents concluding remarks.

2. A CASE STUDY IN FINANCIAL ANALYSIS

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. Since markets are increasing dominated by electronic trading systems, however, real-time performance becomes an increasingly critical factor in making timely trading decisions.

DD4LCCI'2010, April 27, Valencia, Spain

A common problem encountered in the financial services industry is risk estimation. *Monte Carlo methods* [9] relying on simulations based on hypothetical market behavior scenarios have proven quite useful in risk calculations, especially for portfolios involving derivatives. 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.

The remainder of this section presents a case study in financial analysis based on a *value-at-risk* (VaR) [12] calculation using Monte Carlo simulation to showcase key design challenges of developing parallel computational finance applications.

2.1 The Definition and Applications of VaR Analysis

In financial mathematics and financial risk management, Value at Risk (VaR) is a widely used measure of the risk of loss on a specific portfolio of financial assets. For a given portfolio, probability and time horizon, VaR is defined as a threshold value such that the probability that the mark-to-market loss on the portfolio over the given time horizon exceeds this value (assuming normal markets and no trading in the portfolio) is the given probability level.

For portfolios involving traditional instruments like stocks, effective and computationally parsimonious analytical methods (such as the variance-covariance method [11, 7]) for calculating the value-at-risk have been devised. Crucially, such methods rely on important assumptions about the nature of the loss distribution, including the stipulation that it is a normal distribution. Hence, such algorithms cannot be applied to portfolios that contain exotic instruments (such as options and other derivatives) and risk managers must resort to more generalized techniques.

Due to their generality, *Monte Carlo* methods are often utilized for VaR calculations for portfolios with options. Instead of modeling future portfolio performance on purely theoretical considerations, such methods simulate a large, representative set of possible performance scenarios and then base the VaR measurements on tallies of the results. One variant of such methods—design for portfolios with options—is to use the historical performance of the options' underlying securities at different randomly-selected times to generate plausible scenarios of future underlying securities' values. Since powerful algorithms (such as the Black-Scholes model [5] and the binomial tree model [6]) exist to predict option prices based on underlying prices, such scenarios can be extended to generate predictions about the performance of all positions in the portfolio, and thus the portfolio's overall performance.

Although such *Monte Carlo* (we use *Monte Carlo* methods based on historical simulations) methods are general and versatile, they are extremely computationally intensive. In particular, the algorithms for deriving options prices from those of their underlying securities are quite involved, and the computational cost is compounded by the fact that many such predictions must be calculated to generate a sufficiently large number of scenarios for reliable statistical analysis. Indeed, the computational cost of such calculations is often the factor limiting their broader use in the financial industry.

2.2 A Typical Serial Implementation of VaR Calculation

Inputs and results for VaR calculations are often hosted in applications such as *Microsoft Excel spreadsheets* and calculations are often controlled by a *Visual Basic for Applications* script. The actual calculations may be performed either within the Excel process or delegated to another process in a client/server configuration.

This paper focuses on a representative VaR calculation where the computational task is to estimate the 1-day value at risk for a portfolio with positions in stocks, an index, and a number of American options on these securities. Figure 1 shows an architecture of such a serial VaR calculation, where a *Visual Basic* client applications computes the VaR for a set of portfolios (whose input data are stored in a *Microsoft Excel* spreadsheet) by invoking remote requests on a VaR evaluation library hosted in a *Linux* server. The *Monte Carlo* simulation achieves this VaR calculation by (1)

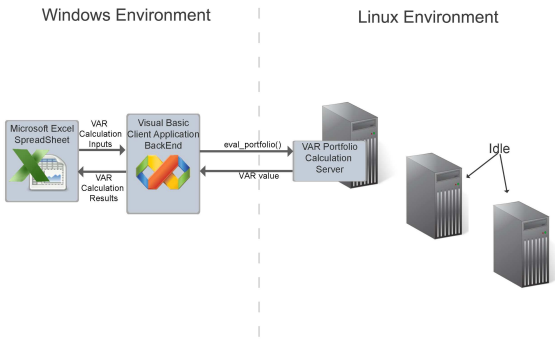

Figure 1: A Serial VaR Calculation Example

computing the value of the portfolio at the end of the time horizon under a large number of market behavior scenarios and (2) quantifying the maximum loss expected with a given probability (*i.e.*, the simulation's confidence interval).

In our case study, 381 options, 30 stocks, and one index tracking fund qualify for our portfolio. We collect the history of daily stock returns for two years preceding the VaR calculation date in an *Microsoft Excel* spreadsheet. The next step is creating 1,000 distinct scenarios by randomly drawing 1,000 sets of returns and calculating the underlying 1-day VaR value. As shown in Figure 1, the VaR calculation function is embedded in a library hosted on a Linux machine. Likewise, the client for this computation is an *Excel* spreadsheet and a *Visual Basic* client executable assists the *Excel* spreadsheet to make remote invocations on the Linux library. In the next section, we will describe the design challenges for parallelizing such VaR calculations.

2.3 Design Challenges of Parallelizing the VaR Calculation

In our case study, we have American options with discrete dividends on individual stocks, as there is no closed-form solution available for them and one has to use time-consuming binomial trees to calculate the price. Hence, the following computational steps are involved in calculating 1-day VaR: (1) randomly picking N historical dates, (2) applying the returns for each historical date to the underlying prices on the initial date to obtain a scenario for the underlying prices at the end of the time horizon, (3) for each scenario for the prices of the underlying securities, evaluating the prices of all the options in the portfolio, and (4) after evaluating all scenarios, forming the simulated distribution of the portfolio values and computing VaR.

Since the options price model calculations are logically independent and quite numerous within the scope of a single portfolio's analysis, this VaR calculation has great exploitable concurrency and is easy to parallelize. Although parallelization provides a realistic and economical way to improve the performance of such analysis, conventional implementations of this parallel calculation face a number of design challenges in offloading serial calculation

to run in parallel on hundreds or thousands of distributed computation servers. The remainder of this section describes some of these key design challenges.

Challenge 1: Discovery and addressing of remote computation servers for distributed computation. If application developers write source code manually to perform parallel programming, they would have to identify the IP addresses of the client and server machines, determine multicast addresses, and also handle the variabilities associated with the underlying network stack to transfer requests and replies across the network. Moreover, this process would be repeated whenever the underlying platforms change, *e.g.*, the input data could be moved from the *Excel* spreadsheet to a database, or the VaR calculation server could be moved from the *Linux* host to a *Solaris* host. Irrespective of these changes in the underlying network, hardware, and platform topologies, the VaR calculation data must be distributed and distributed computations must be performed. As described above, manually modifying source code to handle such sophisticated use cases is hard.

Challenge 2: Data Dissemination for remote distributed computation. Distributed computations involves transforming the internal state of a program (*e.g.*, the input for the VaR calculation stored in the Excel spreadsheet) in an external format that be can transferred via the network to remote computation servers. The programming technique used to accomplish this transformation is called as *marshaling* and the reverse process of converting the external data format to internal data format is called *demarshaling*. Historically, application developers have manually written (de)marshaling code to meet the distributed computing requirements of VaR calculations. This (de)marshaling code is highly dependent on the format of the data being sent and the platforms hosting the client-server processes, which complicates manual source code development activities.

Challenge 3: Efficient distribution of remote computation requests for effective resource management across the network. After application developers devise solutions to challenges 1 and 2 above, intelligent request scheduling and distribution algorithms are needed to disseminate requests across the various computation servers. Efficient request dissemination 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.

Challenge 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 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. 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 finance applications.

Challenge 5: Concurrency management. Computational finance applications, such as the VaR 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 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.

The remainder of this paper uses the VaR case study to motivate how the zFunction middleware can address the above described distributed and parallel application development challenges associated with large-scale computational finance applications.

3. STRUCTURE AND FUNCTIONALITY OF ZFUNCTION

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

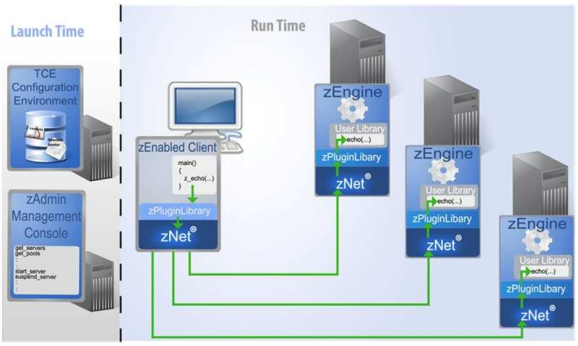

Figure 2: Overview of the zFunction Architecture

Figure 2 shows the following key elements in the zFunction middleware:

• **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.

The remainder of this section outlines the types of applications

that can benefit from zFunction and describes how its components in Figure 2 address the parallelization challenges described in Section 2.3.

3.1 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 zFunction 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 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 3). The output is a library

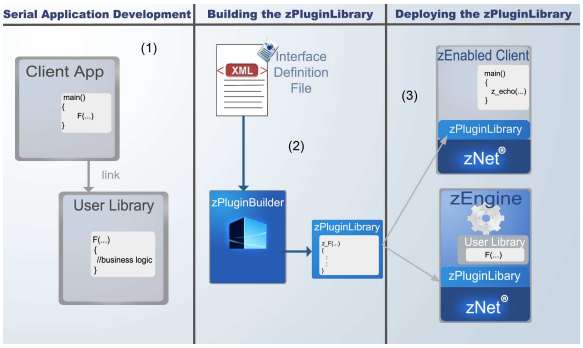

Figure 3: zEnabling a Serial Application with zFunction

(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 3). 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.

3.2 Resolving Distributed and Parallel Application Design Challenges with zFunction

We now describe how the zFunction components shown in Figure 2 address the key distributed and parallelize application design challenges summarized in Section 2.3.

Resolving challenge 1: Providing an information service for discovery and addressing of remote computation servers. *The Configuration Environment* (TCE) acts as an information service for zFunction and bootstraps all the applications in the network. All other components in a zFunction 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 zFunction components, so that applications can communicate with each other at runtime without collaborating with TCE.

Resolving challenge 2: Providing transparent management of data distribution for remote communications. zFunction allows application developers to optimize the system performance by providing flexible data-dissemination mechanisms. zFunction clients do not send data with every request; instead, data is sent only once, and with every request, zFunction sends a reference to each server on where the data could be found. Moreover, if new data needs to be updated midway through the computations, zFunction also provides a mechanism to signal all the servers and allow them to reach a common snapshot or checkpoint, receive the new input data from the client, and then resume computations. zFunction provides a utility called the zPluginBuilder that automatically generates zPluginLibraries that serve as adapters between the generic zFunction middleware and specific client/server applications. These adapters emit efficient (de)marshaling code that enables zFunction middleware to transparently support remote communication across heterogeneous platforms and networks.

Resolving challenge 3: Providing effective resource management of remote computation servers. When zEnabled client requests are sent to a server pool, zFunction middleware's intelligent load-balancer is used to evenly distribute work amongst existing computation servers in real-time, as shown in Figure 4. 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.

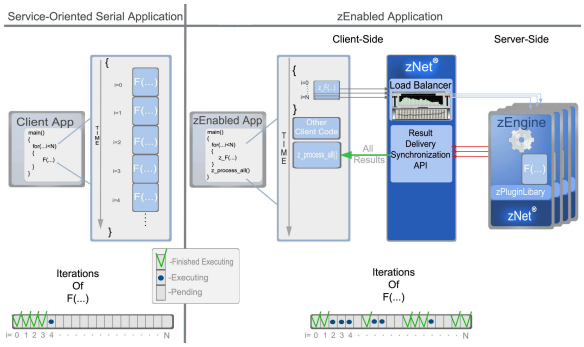

Figure 4: Parallel Application Development with zFunction

Resolving challenge 4: Providing application-transparent multi-layer fault tolerance. zFunction also ensures application execution irrespective of hardware failures, and transparently provides fault recovery and failover by re-executing requests at servers that are still operational. zFunction keeps track of the execution history of each request and to which zEngine the request has been sent to. When zFunction detects that a zEngine has failed, it automatically resends the request to a new or a rejuvenated Engine and ensures that the computations are performed irrespective of hardware failures.

Resolving challenge 5: Providing implicit scalability using core-aware multi-threading. zFunction attains parallelization by

executing multiple instances of an application's parallelizable function simultaneously in zEngine processes running on different machines on a network. zFunction provides implicit concurrency support and automatically creates threads for distributing requests to different servers and also synchronizes those threads using messages and locks.

4. APPLYING ZFUNCTION TO THE VAR CASE STUDY

This section presents an updated VaR application that uses zFunction to parallelize calculations on a portfolio of stocks and options represented as a *Microsoft Excel* spreadsheet. As discussed in Section 3, the zFunction middleware permits the effective decoupling of the client code and the parallelizable function implementation so that they can run on different platforms, *e.g.*, Windows for the client and Linux for the servers. Moreover, different parts of the VaR application can also be written in different languages, *e.g.*, Visual Basic for the client and C/C++ for the servers.

The interface between the client and the parallelizable function is specified concisely in an XML-based *Interface Description File*. The zPluginBuilder tool is then used to generate plug-in libraries appropriate for both the client application and the zEngine server implementation. For this case study, the client is a *Microsoft Excel* spreadsheet that uses a COM interface (via *Visual Basic*) to integrate with the zFunction middleware. Conversely, servers are deployed in a pool of zEngines deploying Linux C/C++ libraries for VaR calculation.

4.1 zFunction-based Client Implementation

The client code for this application resides in a *Microsoft Excel* workbook that contains data about the market behavior scenarios (see Figure 1), which is a natural and typical medium for data-intensive financial calculations. As shown in Figure 5, the client

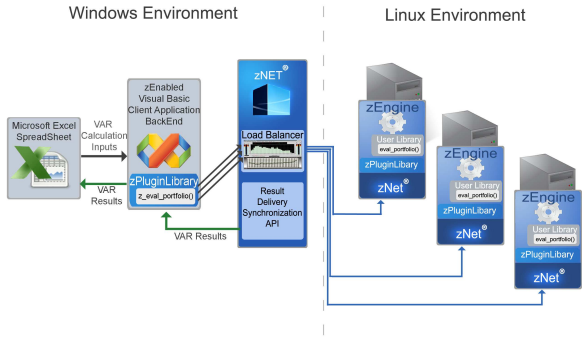

Figure 5: A zEnabled VaR Calculation Example

can therefore easily support a legacy serial implementation of the calculation written in *Visual Basic* for applications. This figure also shows how the *zEnabled* application requires two superficial transformations of the client-side code embedded in the spreadsheet:

- The calculations are dispatched asynchronously, and a separate callback function receives responses from remote zEngines and populates target cells in the resulting spreadsheet with these responses.
- The zPluginBuilder tool is used to generate a client-side COM interface that allows the invocation of the appropriate function from within the *Visual Basic* client code.

4.2 zFunction-based Server Implementation

When developing the server-side code of a *zEnabled* application, it is only necessary to (1) create a user library containing the parallelizable function and (2) describe the function's interface using an interface description file (in XML). The zPluginBuilder tool is then used to generate a plugin library that can be dynamically loaded into zEngines running on any supported platform, as shown in Figure 3.

The server implementation consists of an `eval_portfolio()` function, whose input parameters include a portfolio definition and a stock price scenario and uses the binomial options pricing model to evaluate all the options in the portfolio. The final result of the computation for a single scenario is a portfolio value. Each scenario (which is defined by its distinct set of hypothetical stock prices at the end of the simulation's time horizon) thus yields an independent and parallelizable portfolio value calculation. As shown in Figure 5, the zEngines devoted to the calculation can complete all these independent scenario calculations efficiently, with the client-side load balancer integrated into the zNet middleware distributing the work automatically.

4.3 Benefits of Applying zFunction to the VaR Case Study

The automated zEnabling process addresses all the challenges from Section 2.3 that are faced by developers of computational finance applications. As shown in Figure 5, to resolve challenges 1, 3, 4, and 5, the zFunction middleware automatically provides discovery, addressing, load balancing, fault detection and recovery mechanisms, ensuring that all client requests handed to it will eventually run, irrespective of communication or server failures. This fault-tolerance is provided by zFunction components on both sides of the network, requiring no application developer effort. The application-specific zFunction generated by the zPluginBuilder also encapsulates the concerns related to robust distributed computing behind an interface similar to that of the synchronous parallelizable function, thereby raising the level of programming abstraction experienced by application developers.

5. RELATED WORK

This section compares and contrasts our work on zFunction with related work on parallel application development and deployment.

Aspect-Oriented Programming (AOP). Recent work has focused on using AOP [13] to separate parallelization concerns from application specific source code [10, 17, 14]. However, in order to provide real-time capabilities like fault-tolerance, load-balancing and data dissemination using AOP, newer technologies need to be used that support composition of aspects. zFunction provides all these benefits with minimum modification to existing applications.

Grid computing middleware. Many projects have explored the idea of utilizing distributed computing architectures to accelerate complex calculations over cluster of computers. Some well-known examples include the SETI@Home [3] and BOINC [2] 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, create performance bottlenecks, and yields a single point of failure. In contrast, zFunction provides a highly optimized and decentrallized middleware infrastructure for application parallelization, interprocess communication and data distribution.

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, [16] discusses practical experiences associated with data management and performance issues encountered in developing financial services applications in the IBM Bluegene supercomputer [1]. Likewise, PicsouGrid [4] 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. zFunction 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.

6. CONCLUDING REMARKS

This paper showcased the capabilities of the zFunction middleware that can parallelize complex computation and data intensive distributed applications by using a simplified programming model and creating an adaptive, real-time, fault-tolerant distributed computing environment on-demand. zFunction is well-suited for domains where complex real-time calculations are needed quickly and predictably, and which can benefit from distributed workload processing across a (potentially heterogeneous) network. It can substantially improve the performance of such systems at a low cost by enabling the applications to run parallely on COTS hardware, desktops, clusters and the cloud.

7. REFERENCES

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

[2] D. 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.

[3] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. Seti@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[4] S. Bezzine, V. Galtier, S. Vialle, F. Baude, M. Bossy, V. D. Doan, and L. 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.

[5] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, 81(3), May 1973.

[6] J. C. Cox, S. A. Ross, and M. Rubinstein. Option Pricing: A Simplified Approach. *Journal of Financial Economics*, 4, 1979.

[7] D. Duffie and J. Pan. An Overview of Value At Risk. *The Journal of Derivatives*, 4(3), Apr. 1997.

[8] M. Forum. Message Passing Interface Forum. www.mpi-forum.org.

[9] P. Glasserman. *Monte Carlo Methods in Financial Engineering (Stochastic Modeling and Applied Probability)*. Springer Verlag, 2003.

[10] B. Harbulot and J. 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.

[11] J. C. Hull. *Risk Management and Financial Institutions*. Prentice Hall, Upper Saddle River, NJ, 2006.

[12] P. Jorion. *Value at Risk: The New Benchmark for Managing Financial Risk*. McGraw-Hill, New York, NY, third edition, 2006.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, pages 220–242, June 1997.

[14] M. E. F. Maia, P. H. M. Maia, N. C. Mendonca, and R. 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.

[15] OpenMP. OpenMP Home Page. www.openmp.org.

[16] T. Phan, R. Natarajan, S. Mitsumori, and H. Yu. Middleware and performance issues for computational finance applications on blue gene/l. *Parallel and Distributed Processing Symposium, International*, 0:371, 2007.

[17] J. Sobral. Incrementally developing parallel applications with aspectj. *Parallel and Distributed Processing Symposium, International*, 0:95, 2006.