Parallel Computing for Computational Finance Applications: A Case Study Parallelizing NAG with Zircon Software

Copyright © 2010 Goethe University Frankfurt, Zircon Computing LLC, NAG

Abstract

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. This paper describes how we applied the Zircon adaptive high-performance computing software platform and tools with the NAG C library to substantially improve the performance of a representative complex computational finance application via distribution and parallelization, thereby reducing the total computation time from several hours to several minutes.

1. Introduction

Parallel application programming is becoming increasingly important to leverage hardware advances and perform complex calculations quickly and scalably in competitive domains, such as computational financial. Prior work has focused on high-performance computing and grid computing platforms to support the development of parallel applications. These technologies, however, are hard to program and manage due to the need to use explicit concurrency mechanisms and manual lifecycle management techniques. What is needed, therefore, are software technologies and tools that are both easy to use *and* highly optimized for the new generation of parallel computing hardware.

This paper presents the results of a case study that calibrates Heston [1] 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 [2], 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.

We selected the NAG C Library since it provides many useful and efficient functions to perform numerical analysis. For example, the `nag_opt_bounds_no_deriv()` 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 [2] 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 [2] used only five underlying securities for calibrations, whereas there are ∼6,000 individual securities in OptionMetrics available for such analysis.

To speedup Heston calibrations, we combine the NAG C library with the Zircon [3] adaptive high-performance computing software platform and tools. Zircon software allows developers to design applications as if they are programming for a single computer by automatically distributing and parallelizing task executions scalably, reliably, and resource-efficiently. The result is a straightforward programming model that can improve the performance of complex computational finance applications by many orders of magnitude. In particular, the experiments in this paper show how integrating NAG and Zircon achieved near-linear throughput speedup as a function of resources (*e.g.*, local and remote cores) used, which reduced the amount of time required to perform Heston calibrations from hours to minutes on multi-core machines.

The remainder of this paper is organized as follows: Section 2 explains the Heston model; Section 3 discusses the limitations of applying traditional parallel computing

approaches to Heston calibration; Section 4 describes the structure and functionality of Zircon; Section 5 shows how Zircon Software was used to reimplement a parallel and distributed version of the Heston calibration application; Section 6 analyzes the results of experiments that quantify the performance improvements after parallelizing a sequential Heston calibration application using Zircon Software; Section 7 presents concluding remarks and lessons learned; and Appendix 8 shows key portions of the code for Heston calibration, with and without Zircon colocated and distributed parallelization support.

2. Overview of the Heston Model

The Heston [1] model assumes the following risk-neutral dynamics for the underlying stock S and its local variance v:

$$dS_t = rS_t dt + \sqrt{v_t} S_t dW_t^S$$
$$dv_t = \kappa(\theta - v_t)dt + \sigma_v \sqrt{v_t} dW_t^v$$

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_v 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 minimization function `nag_opt_bounds_no_deriv()` .

We calibrated the model to observed option prices 1,065 times, where each task takes several seconds to several minutes. The total time to complete this computation using a conventional sequential algorithm implementation was ~9 hours. Conversely, the Zircon-based version of this algorithm ran several orders of magnitude faster in parallel in a cloud of 32 multi-core machines, requiring only ~18 minutes to complete.

3. Challenges of Parallel and Distributed Application Development

This section discusses the challenges involved in developing high-performance computing applications and the limitations of existing parallel and distributed programming techniques. We encountered these challenges when porting a sequential implementation of the Heston calibration application to a multi-core machine. Figure 1 shows how the sequential Heston calibration application only used one core on a multi-core machine. Likewise, Figure 2 shows that the sequential Heston calibration application calibrated 1,065 models on a multi-core machine in ~32,400 seconds, irrespective of the number of cores available.

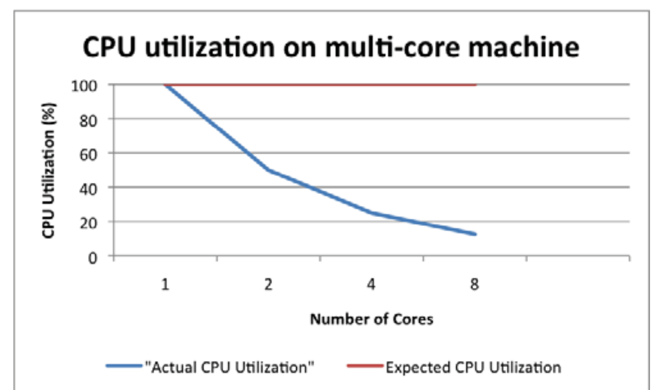

Figure 1. CPU Utilization of the Sequential Heston Model Implementation on a Multi-core Machine

We initially considered parallelizing the Heston calibration application using conventional operating system threading libraries and/or deploying the calculations on a distributed cloud or grid. We found this approach had the following limitations, however:

- **Non-portable.** Native operating system threading libraries are not portable across different hardware/software platforms, which makes it hard to write portable applications and run them on multi-core processors or clouds/grids.

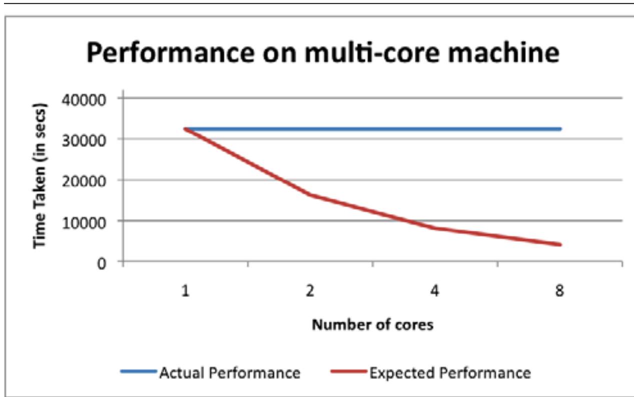

Figure 2. Performance of the Sequential Heston Model Implementation on a Multi-core Machine

- **Tedious and error-prone parallel programming.** Developers of conventional multithreaded Heston applications must manually program tedious and error-prone concurrency mechanisms, such as synchronization and atomic operations, when developing their multithreaded applications.
- **Tedious and error-prone distributed programming.** Developers of distributed Heston applications must also handle tedious and error-prone concerns, such as node-discovery, deployment, flow-control, load balancing, and fault-tolerance.

What we needed, therefore, was an easy-to-program/use/-manage high-performance computing software platform that alleviated these limitations.

4. Solution Approach: the Zircon Adaptive High-Performance Computing Platform

The Zircon Software Product Suite from Zircon Computing provides an adaptive high-performance computing platform that addresses the limitations described in Section 3. Zircon 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 software supports three computing and communication models required by many mission-critical applications that need high performance, as shown in Figure 3 and described below:

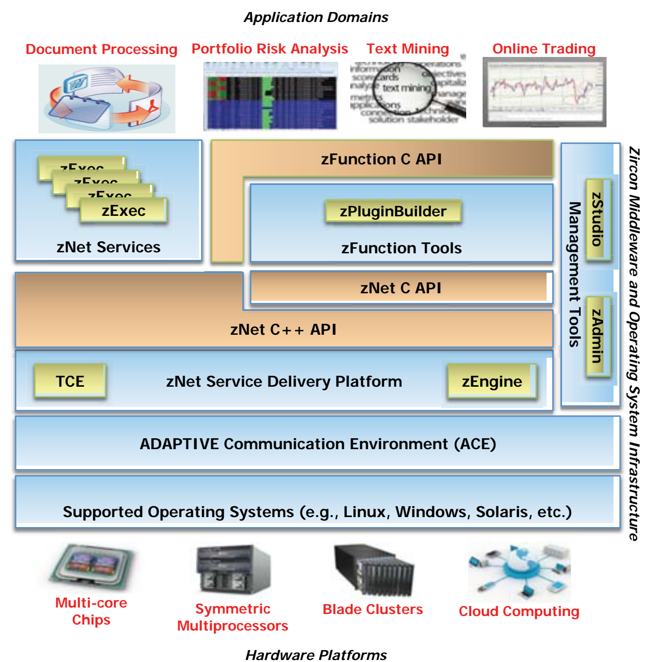

Figure 3. Zircon Software Architecture

- **Application function parallelism**, such as the capabilities provided by computation grids to run application operations 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 parallelism service runs any executables 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 [4]. 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) [5, 6], which is portable C++ host infrastructure middleware that shields Zircon software from operating system dependencies.

5. Implementing the Heston Calibration Application Using NAG and Zircon Software

This section describes how we developed a high-performance Heston calibration application by combining the NAG C Library's minimization function `nag_opt_bounds_no_deriv()` with the Zircon zNet service delivery platform. As described in Section 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 objective function is calculated as the mean squared pricing error between the observed price and the calculated price for each combination of parameters. The optimization routine is implemented by using the NAG `nag_opt_bounds_no_deriv()` 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 ∼9 hours by invoking the `calibrate_heston()` function 1,065 times in a loop. The `calibrate_heston()` function internally invokes the NAG `nag_opt_bounds_no_deriv()` minimization function to perform model calibration.

We observed that all `calibrate_heston()` invocations ran independently of each other, so application performance can be improved significantly by processing multiple invocations in parallel. We therefore used the Zircon zNet API described in Section 4 to convert the sequential Heston calibration application into a high-performance computing implementation. The remainder of this section describes how we significantly enhanced the performance of Heston calibrations by running the NAG `nag_opt_bounds_no_deriv()` minimization function concurrently using zNet's colocated parallelization and distributed parallelization models.

5.1. Colocated Parallelization Application

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 each worker thread invokes the NAG `nag_opt_bounds_no_deriv()` minimization function for model calibration. In general, we found that the number of worker threads running on a machine should correspond to the number of cores on the machine, *e.g.*, on a quad-core machine, 4 worker threads can run in parallel and 4 models can be calibrated in parallel, as shown in Figure 4.

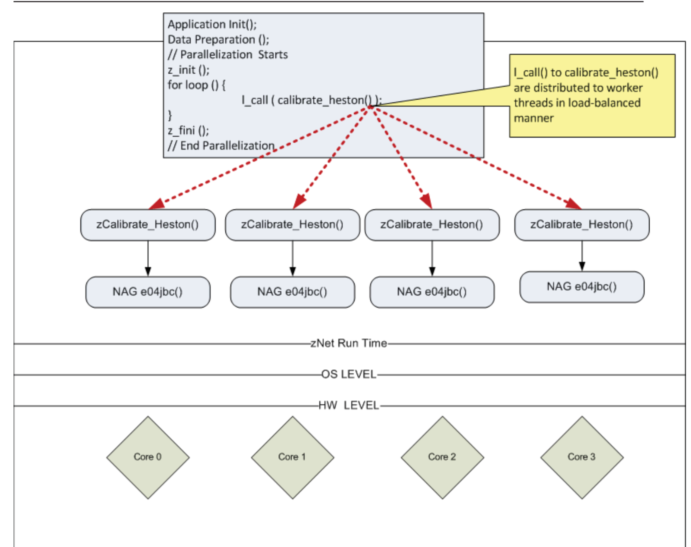

Figure 4. Colocated Parallelization of the Heston Model on a Multi-core Machine

In the colocated implementation, 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 4 shows how the `l_call()` functions are distributed to the worker threads

in our colocated Heston calibration application implementation.

5.2. Distributed Parallelization Application

Distributed parallelization enables applications to run in multiple processes and handle multiple requests in parallel in multiple worker threads on multiple machines, which is best suited for applications that require a larger number of processor cores than a standalone machine can provide. In this implementation of the Heston calibration application zNet distributes and equalizes the requests from a client to the servers dynamically, thereby speeding up its processing by several orders of magnitude.

The Heston client application initializes the zNet runtime and passes all calibration request to the runtime by invoking the zNet `z_call()` function in a loop. The `z_call()` method forwards calibration requests to remote worker threads that process the requests in parallel on multiple machines. Likewise, the Heston server applications also initialize the zNet runtime, register themselves with the Zircon license server on startup, and listen for incoming client requests. Figure 5 shows how remote calls are distributed and load balanced on zNet servers running in a cluster or cloud controlled by Zircon software.

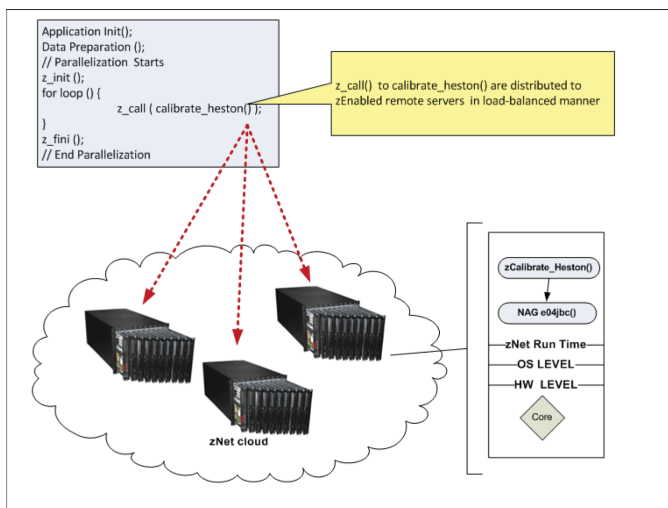

Figure 5. Distributed Parallelization of the Heston Model in a Cluster or Cloud

6. Analysis of Empirical Results

This section presents the results of experiments that quantify the benefits of parallelizing and distributing the Heston calibration application using Zircon software. These results show that the performance gain for the parallelized Heston application is near linear to the number of servers/cores used for computation. 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.

6.1. Colocated Parallelization Results

We first ran the colocated parallelization version of the Heston calibration application described in Section 5.1 using dual-core[1] and quad-core machines to compare its performance with the baseline sequential implementation described in Section 3. The results of the tests shown in Figure 6 demonstrate that the colocated parallelization application fully utilized all cores on a machine, which improved application performance near linearly as the number of cores increased. Likewise, the CPU utilization of

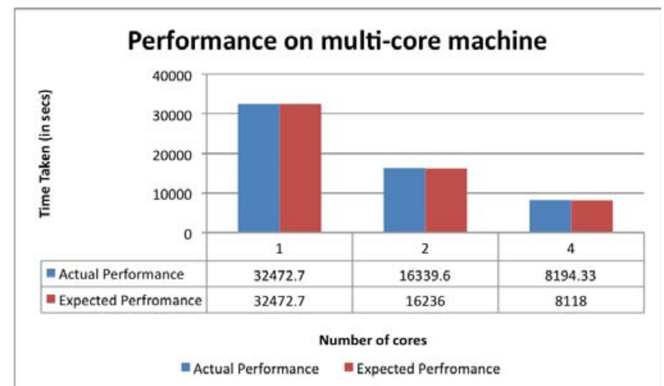

Figure 6. Performance of zNet Version of Heston Model on a Multi-core Machine

the multi-core machine for this experiment matched the expected CPU utilization shown in Figure 1.

6.2. Distributed Parallelization Results

We then ran the distributed parallelization version of the Heston calibration application described in Section 5.2 on one, two, four, and eight multi-core machine cluster configurations to compare their performance with the baseline sequential implementation described in Section 3. Each machine in the cluster contained four cores, so we started four

1 One processor was disabled to simulate dual-core machine behavior for some experiments.

instances of the Heston calibration server on each machine to leverage all four cores. The results of these tests in Figure 7 show that the performance gain is nearly linear to the number of cores used for computations. In particular, it took

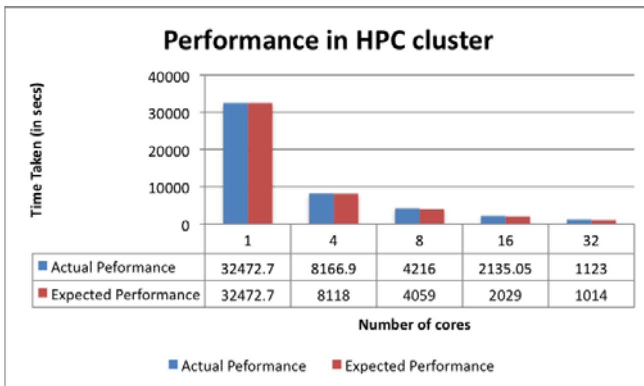

Figure 7. Performance of zNet Version of Heston Model on a Cluster

~9 hours for the sequential Heston calibration application to process 1,065 models. Conversely, it took ~18 minutes for the distributed parallelism version of Heston calibration application to process 1,065 models, which is several orders of magnitude faster.

The performance speed-up provided by the colocated parallelism solution is limited by the number of cores available on a single multi-core machine as it provides speedup by running worker threads in a single-process. The distributed parallelism version of the Heston calibration application, however, can be deployed on a cluster of machines to gain significant performance improvements by leveraging a larger number of cores available in the cluster. The distributed parallelization can therefore be scaled transparently to use any number of cores available to users. Moreover, these cores could be allocated dynamically, *e.g.*, using the elastic provisioning capabilities of a public or private cloud computing environment.

7. Concluding Remarks

This paper showed how combining the NAG C library with the Zircon adaptive high-performance computing platform can benefit CPU-intensive and time-consuming computational finance applications. Zircon software handles all the distributed and parallel development concerns via easy-to-program/use/manage C/C++ interfaces that utilize available computing resources scalably and efficiently within an organization or cloud. The results of our experiments showed how Zircon software accelerated the performance of a NAG-based Heston calibration application by many orders of magnitude on a cluster of multi-core machines.

The following are the advantages we observed based on our experience developing and evaluating several implementations of the Heston calibration application using the combination of NAG and Zircon software:

- **The NAG C Library integrated easily with Zircon software** and together provide a powerful and reliable platform for computational finance applications. NAG's minimization function `nag_opt_no_derive()` facilitated Heston model calibration within user-defined upper and lower bounds for a 5-dimensional objective function.
- **Zircon Software is easy to learn and use** since it requires no tedious and error-prone low-level programming related to network protocols or managing the lifecycle of distributed processes. We migrated the sequential version of the Heston calibration application to use the zNet distributed and parallel computing middleware in less than one day.
- **Zircon Software maximizes hardware utilization** by creating threads to distribute requests to different servers and synchronizes those threads using messages and locks. By overlapping I/O and CPU computations, performance scales linearly as more processors and/or cores are assigned to server tasks.
- **Zircon is extremely scalable** with predictable performance and built-in real-time load equalization. Servers can be added or upgraded without disrupting operations since Zircon software automatically recognizes new processors and equalizes the workload.

Readers can find information about Zircon software at *www.zircomp.com* and the NAG C Library at *www.nag.co.uk/numeric/CL/cldescription.asp*. Developer documentation for zNet API is available at *www.zircomp.com/downloads/docs/html_znet/index.html* and usage manual for NAG function `nag_opt_bounds_no_deriv()` can be found at www.nag.co.uk/numeric/CL/nagdoc_cl09/pdf/E04/e04jbc.pdf.

8. Participants

Goethe University

Goethe University Frankfurt is one of the major institutions of higher education in Germany, committed to providing a wide spectrum of disciplines in research and teaching, to generating outstanding achievements, and to breaking new ground through the targeted utilization of the

advantages and synergies of interdisciplinary work in research and teaching. The University has more than 15 academic departments with about 37,000 students and more than 2,500 researchers. The newest part of the University, hosting the researchers involved in the current study - the House of Finance - opened in spring 2008, and it incorporates the university's interdisciplinary research on finance, monetary economics, and corporate and financial law under one umbrella.
For more information, please contact Grigory Vilkov (`vilkov@finance.uni-frankfurt.de`).

Zircon Computing, LLC

Zircon Computing, LLC, is an international software and services company based in Wayne, New Jersey USA. Founded in 2005 by senior technologists from the financial services industry, Zircon Computing develops and markets the Zircon Software Product Suite of adaptive middleware and high performance software and services worldwide, both directly to enterprise clients and through an international network of partners. Zircon Computing is privately held. For more information, please visit `http://www.zircomp.com`.

NAG

NAG The Numerical Algorithms Group (NAG) is dedicated to making world-class cross-platform mathematical, statistical, data mining components and tools for developers as well as 3D visualization application development environments. NAG serves it customers from offices in Oxford, Chicago and Tokyo supporting over 10,000 customer sites worldwide in finance, engineering, and scientific research. NAG software is the choice of over 25 independent software vendors including Oracle, IBM, DemandTec and many others. For more information, please visit `http://www.nag.co.uk.`

References

[1] S. Heston, "A closed-form solution for options with stochastic volatility with applications to bond and currency options," *Review of Financial Studies*, vol. 6, pp. 327–343, 1993.

[2] D. Horn, E. Schneider, and G. Vilkov, "Hedging options in the presence of microstructural noise," *SSRN eLibrary*, 2007.

[3] J. Balasubramanian, A. Mintz, A. Kaplan, G. Vilkov, A. Gleyzer, A. Kaplan, R. Guida, P. Varshneya, and D. 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.

[4] F. Buschmann, K. Henney, and D. C. Schmidt, *Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, Volume 4*. New York, USA: Wiley and Sons, 2007.

[5] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Reading, Massachusetts, USA: Addison-Wesley, 2001.

[6] ——, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Reading, Massachusetts, USA: Addison-Wesley, 2002.

A. Heston Calibration Function

```
void calibrate_heston
  (const std::vector<optionsdata> &v_optdata,
   double &v, double &kappa, double &theta,
   double &sigmaV, double &rho,
   const option_range &r,
   const int count_day, const int index)
{
  ....
  // Initialize parameters for NAG function.
  Nag_BoundType bound;
  static NagError fail;
  Nag_E04_Opt options;
  Nag_Comm param;
  fail.print = Nag_FALSE;
  e04xxc (&options);
  options.list = Nag_FALSE;
  options.print_level = Nag_NoPrint;
  bound = Nag_Bounds;
  ...
  // Invoke NAG function
  e04jbc (n, objfun_nag_heston, bound,
  bl, bu, x, &objf, g,
  &options, &param, &fail);
  ...
}
```

B. Code for Collocated Parallelization

```
// zNet call handler that redirects
// local calls to calibrate_heston().
static void call_handler (parameter_SV* r)
{
  calibrate_heston
    (r->options_data_vector_ptr->_data,
     r->v, r->kappa, r->theta,
     r->sigmaV, r->rho,
     r->option_range_vector_ptr->
        _data[r->count_day],
     r->count_day, r->m);
}

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

```
  ...
  // Initialize the zNet computation environment
  Z::init (argc, argv);

  // Register the local function with
  // zNet computation environment.
  ZBroker::register_local_function (&call_handle

  for (int m = 0; m < cal_range; ++m)
  {
    ...
    // local call to heston_calibration function
    ZBroker::l_call (&results[m]);
  }

  // Wait to obtain replies for the requests sen
  ZBroker::process_all ();

  // Finalize the zNet computation environment
  // to clean up all resources used for
  // computations.
  Z::fini ();

  // Exit the application.
  return 0;
}
```

C. Distributed Parallelization Server Code

```
// zNet server.
class Heston_Calibration_server_Component
  : public ZUserComponent
{
  // call handler that processes incoming
  // requests for heston calibration.
  bool call_handler (long client_id_,
                     parameter_SV *d_)
  {
    ...
    calibrate_heston
      (r->options_data_vector_ptr->_data,
       r->v, r->kappa, r->theta,
       r->sigmaV, r->rho,
       r->option_range_vector_ptr->
         _data[r->count_day],
       r->count_day, r->m);
    return true;
  }
};

// Entry point into the example
// application server task.
int
main (int argc, char **argv)
{
  ...
  // Instantiate the Heston server class.
  Heston_Calibration_server_Component comp;

  // Initialize the zNet computation
  // environment with the user written
  // class as an input so the service
  // can be activated or deactivated
  // automatically.
  Z::init (argc, argv, &comp);

  // Run the main thread continuously,
  // allowing zNet to dispatch data,
  // events, and invocations to this server.
  Z::wait ();

  // Finalize the zNet computation
  // environment to clean up all
  // resources used for computations.
  Z::fini ();

  // Exit the application.
  return 0;
}
```

D. Distributed Parallelization Client Code

```
// Entry point into the example
// application client task.
int main (int argc, char** argv)
{
  ...
  // Initialize the zNet
  // computation environment.
  Z::init (argc, argv);

  // Create zNet client
  ZBroker::Client heston_client (1);

  for (int m = 0; m < cal_range; ++m)
  {
    // remote call to
    // heston_calibration() function.
    heston_client.z_call
      (&results[request_count]);
  }

  // Wait to obtain replies
  // for the z_call() methods sent.
  heston_client.process_all ();

  // Finalize the zNet computation
  // environment to clean up all
  // resources used for computations.
  Z::fini ();

  // Exit the application.
  return 0;
}
```