**RC23785 (W0511-057) November 7, 2005**
**Computer Science**

# IBM Research Report

## Middleware and Performance Issues for Financial Risk Analytics on Blue Gene/L

**Thomas Phan**
IBM Research Division
Almaden Research Center
650 Harry Road
San Jose, CA 95120-6099

**Satoki Mitsumori**
NIWS Co., Ltd.

**Ramesh Natarajan, Hao Yu**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

# Middleware and Performance Issues for Financial Risk Analytics on Blue Gene/L

**Thomas Phan[1], Satoki Mitsumori[2], Ramesh Natarajan[3], and Hao Yu[3]**

**[1] IBM Almaden Research Center**
**[2] NIWS Co., Ltd.**
**[3] IBM T.J. Watson Research Center**

## Abstract

We describe a set of middleware components for supporting the execution of financial risk analytics applications on Blue Gene/L. Although, motivated by the requirements of a specific proprietary application involving a Value-at-Risk (VaR) computation for an asset portfolio using Monte Carlo simulation, this middleware is equally relevant to any application where the input and output data are stored in externally in SQL databases for example, where an automatic pre-staging and post-staging of the data between these external data sources and the computational platform is required. More generally, in order to support such applications on Blue Gene/L, the middleware provides a number of core features including an automated data extraction and staging gateway, a standardized high-level job specification schema, a well-defined web services (SOAP) API for interoperability with other applications, and a secure HTML/JSP web-based interface suitable for general users (i.e., non-developers). Finally, continuing with the focus on the data movement requirements, we describe generic I/O performance optimizations that were used for this application on the Blue Gene/L platform.

## 1. Introduction

The financial industry has been impacted in recent years by the increasing competitive pressure for profits, the emergence of new financial products, and the tighter regulatory requirements being imposed for capital risk management. A number of quantitative applications for financial risk analytics have been developed and deployed by banks, insurance companies and corporations, and these applications are computationally intensive, and require careful attention to the discovery and aggregation of the relevant financial data used in the analysis.

This paper considers the data management and computational issues that were encountered while executing a specific proprietary financial risk analysis application on Blue Gene/L (henceforth BG/L), with the goal of broadly identifying the requirements of middleware components for the deployment of a large class of financial risk applications on BG/L in a production setting.

In general, the requirements of financial risk applications differ from other scientific/engineering applications in many ways, as listed here and discussed in detail subsequently:

- Financial risk applications may require external data sources such as SQL databases, remote files, spreadsheets, and web services or streaming data feeds, in addition to the usual pre-staged or pre-existing flat files on the file system of the computing platform.
- These applications often interact with larger intra- or inter-company business workflows such as trading desk, portfolio tracking and optimization, and business regulatory monitoring applications.
- High-level services specifications must be separated from low-level service and resource provisioning for these applications, since there is a frequent requirement to provide dynamic resource provisioning based on of quality-of-service or time-to-completion requirements.
- The computationally-intensive parts of some applications are often independent and "embarrassingly parallel," and after an initial distribution of financial data to the compute nodes, each node performs independent computations with very little inter-processor communication and synchronization.

The specific proprietary Value-at-Risk (VaR) application in our study has all these characteristics listed above. The relevant input data for this application consists of historic market data for the risk factors, simulation data and asset portfolio details, which is initially extracted from

an SQL database. The required computational analysis is floating-point intensive and easily parallelizable. The resulting output data consists of empirical profit-loss distribution data, which is stored back into an SQL database for post-processing and archiving.

Although the data management and computing requirements of financial risk applications appear straightforward, there are some inhibitors to porting and deploying these applications on high performance computing (HPC) platforms. A major difficulty is that many HPC platforms, including BG/L, are not well-suited for accessing data stored in high-latency, external data sources outside of their local area network. In desktop computing platforms, the network and transaction latencies for remote data access can be overlapped with other useful work by using multithreaded programming in the application. However, the individual BG/L compute nodes do not allow multithreaded user execution, and as a result, the long and unreliable latencies of remote data access cannot easily be optimized away. Another issue is that, like most other distributed-memory HPC platforms, applications run on BG/L in a space-sharing rather than a time-slicing mode; i.e., each application uses a distinct, physically-partitioned set of nodes which is reserved for the entire duration of the application. Lastly, for performance reasons, it is desirable and in some cases mandatory in many HPC systems to have the program data staged to and from a specialized parallel file system, such as GPFS, that is tightly coupled to the HPC platform. These data staging requirements lead to the application deployment on HPC platforms being quite *ad hoc* in nature, with specialized scripts dealing with the data specification, data migration, and job scheduling for each individual application or problem instance.

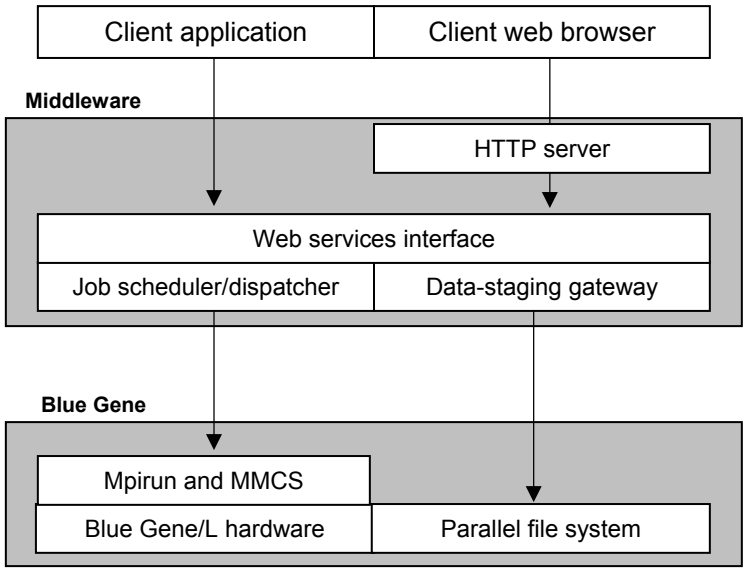


**Figure 1: Logical view of middleware components between the users and the BG/L core. The middleware has two interfaces, via web services and HTML/JSP, to its job scheduling/dispatching and data staging components.**

To address these limitations, we developed a middleware layer, shown in Figure 1, which takes a client job specification and completely automates the tasks of marshaling the data between a set of distributed storage repositories and the specialized parallel file system. In addition, the middleware also handles the synchronization and scheduling of the computing tasks on the HPC platform.

More specifically, our middleware layer provides the following features:

- It supports inter-operability with external business workflows via a Web services (SOAP) API, thereby allowing any standards-compliant programs with proper authentication to submit and monitor application requests.
- It enables users to customize job requirements for specific applications by standardizing the job submission parameters in an XML job specification schema.

- A data-staging gateway is used to prepare and stage data between external heterogeneous data servers and the BG/L file system, with support for data compression (and future support for encryption).
- In order to facilitate the use of the application by a wider set of non-privileged users, it provides a job submission wizard using a secure HTML/JSP interface, as well as a Linux/Unix command-line toolkit for the same purpose.

Finally, in this paper, we also focus on the data movement requirements of financial risk applications, and describe certain generic I/O optimizations that take advantage of the BG/L processor and communication architecture for these applications.

The overall outline is as follows. Section 2 provides a brief overview of the BG/L hardware and software, along with a general description of Value-at-Risk (VaR) as well as the specific details of the motivating proprietary application. Section 3 describes the design of the middleware layer for executing the application workflow. Section 4 describes the communication and node performance optimizations which can improve the I/O performance on BG/L. Section 5 concludes with a summary of the work.

# 2. Background

## 2.1. Blue Gene/L overview

The BG/L supercomputer [1] is a new family of supercomputers that is designed for highly scalable operation, with thousands of processors, a low-latency, high-bandwidth internal communication network, low power consumption, and minimal floor space requirements compared to equivalent systems. BG/L is organized in racks, with each rack consisting of 1024 compute

nodes with a peak performance of 5.6 Teraflops per rack, and the largest configuration has 64 racks consisting of 65,536 compute nodes with a theoretical peak performance of nearly 360 Teraflops.

Each compute node of BG/L has two PowerPC 440 processors operating at 700 MHz, where each processor has dual floating point units with fused multiply-add instructions, 32 KB each of L1 data and instruction cache, and 512MB of overall memory per compute node. The inter-node communication is through multiple, independent networks, which include a 3-D torus network used for point-to-point and multicast messaging (with a bidirectional hardware bandwidth of 308 MB/sec on each link), and a tree network used for broadcast and combining operations (with a network hardware bandwidth of 337 MB/sec).

Each compute node has a lightweight, single-user operating system which supports the execution of a single dual-threaded application process, with each thread being bound to one of the two processors on the compute node. A runtime library supports inter-processor communication via the torus and tree networks to the other compute nodes, as well as to a separate smaller set of I/O nodes. These I/O nodes implement a full-featured operating system and are not used for running application code but to augment the compute nodes by providing support services via function shipping for tasks such as file I/O, socket connections to external processes, job monitoring and control, and debugging support.

Parallel applications for BG/L can be developed using an optimized version of the well-known Message Passing Interface (MPI) communications library [2, 3] which provides basic support for point-to-point and global communication. The node programs that link to the MPI library can be written in FORTRAN, C++ and C. Generally speaking, applications running on BG/L use the compute nodes in one of two modes. In the *Communications Co-processor* mode, one of the compute node processors is used exclusively for messaging protocol tasks and is therefore not

available for computation (so that the peak performance per rack in this mode is 2.8 Teraflops). In the *Virtual Node* mode, both compute node processors can be used for the computation with separate MPI ranks being assigned to them, and all node resources such as memory and communication links are evenly split between the two processors

## 2.2. Value-at-Risk overview

Value-at-Risk (VaR) [4] is an important metric used by financial institutions to characterize the market risk exposure of their asset portfolio. A number of ISV's and financial data providers have developed data sets and services for computing VaR, and many financial services companies have their own proprietary methodologies and tools as well.

Specifically, VaR is based on an estimate of a certain lower $\alpha$ quantile of the profit-loss distribution of the portfolio over a certain time horizon $T$, given that the portfolio is dependent on certain underlying risk factors that are subject to typical market variations. The VaR is then the difference between the expected value of the portfolio and the $\alpha$ quantile (where $\alpha$ might typically be chosen as 5%) over the time horizon $T$ under these market variations. The portfolios being evaluated may consist of assets such as equities, bonds, currencies, commodities, mortgages, as well as other more complex derivative or securitized assets. The underlying risk factors may include variations in stock prices and equity indices, interest rates, currency exchange rates, swap rates, money market rates, government bond rates, commodity prices, and economic factors, among others.

The time horizon $T$ for VaR depends on the dynamics of the changes in the portfolio assets and risk factors, and can therefore vary from a few hours at an active trading desk, to a few weeks for intermediate-term managed investment portfolios, to several months for long-term enterprise-level regulatory reporting requirements.

The VaR metric has been used by financial institutions in the following ways. First, it serves as a standard reference for setting policy, monitoring and managing internal risk. Second, it is used for fine-tuning and adapting business objectives to the institutional tolerance to risk. Third, it is used for ensuring compliance with external regulatory requirements for set-aside capital to cover potential portfolio losses.

The most widely used approach for estimating VaR is Monte Carlo simulation, which can most effectively handle the case where the underlying risk factors have non-normal distributions, and where certain portfolio assets (such as options) have complicated nonlinear dependencies on the underlying risk factors. Although the Monte Carlo approach is computationally intensive compared to some other analytical approaches, the portfolio pricing computations for each simulated market condition are independent, and therefore this approach is well-suited for massively parallel computing. In summary, the Monte Carlo simulation method relies on randomly sampling the multivariate risk-factor distribution to generate as set of independent risk scenarios. The asset portfolio is then priced for each risk scenario and the results aggregated to obtain the empirical profit-loss distribution for the portfolio. Finally, the VaR (at say the 5% level) is estimated from the appropriate quantile of the empirical profit-loss distribution. An example of a parallel Monte Carlo VaR using idle cycle scavenging on a grid of office computers is described in [5].

### 2.3. Financial Risk application

The proprietary financial risk application implemented on BG/L is an example of a Monte Carlo calculation for estimating VaR as described in Section 2.2. The code provided by the customer is an evaluation prototype for preliminary porting and benchmarking studies. Therefore our primary objective was to use this prototype to understand the deployment and performance of the generic class of financial risk applications on BG/L in a future production setting.

The end-to-end execution of this prototype application proceeds in three phases, namely data pre-staging, computation, and data post-staging, as described here. The input data for this application consists of all the necessary data on portfolio holdings, the simulation data for generating scenarios, and the various algorithm parameters (e.g., the number of scenarios to be evaluated), and is roughly 300 MB distributed across 44 files. These files were extracted from a 4 GB database using standard SQL queries and procedures in a pre-staging phase. We note that this pre-staging phase would not be necessary if the HPC platform could directly and efficiently access the external database via a programmatic interface. As noted earlier in Section 1, the long latencies and completion uncertainties of remote communication makes it very inefficient for a space-partitioned, distributed-memory HPC platform like BG/L to have direct database connectivity. In the parallel computation phase, the 300 MB of input data is copied to each compute node and independent Monte Carlo simulations are performed. These independent simulations use samples from the risk factor distributions to generate market scenarios, which are then used to price the instruments in the portfolio for each scenario. The output results from the individual compute nodes are written to disk, and in the final post-staging phase, these results are saved to the SQL database for archiving and further post-processing and analysis.

This prototype application is typical of the intra-day market risk calculations that are routinely performed in many large banks. The input data for this application changes between successive calculations only for those variables that are based on market conditions, such as equity prices, exchange rates, yield curves and forward curves. We estimate that in production settings, a typical large bank might hold about 250,000 instruments in its portfolio, of which 20% may need to be priced by Monte Carlo, while the remaining 80% may be priced by closed-form approximations. In addition, roughly 100,000 scenarios are required in the Monte Carlo simulations to obtain

empirical profit-loss distributions for estimating the relevant VaR quantiles with the required statistical confidence.

# 3. Middleware Architecture

## 3.1. Requirements

The design of the middleware layer is motivated by requirements that are likely to be encountered across a broad range of financial computing applications. First, this layer simplifies and automates the application workflow and provides a framework for application code organization. Second, it separates the data extraction and data migration from the computational steps on the HPC platform, so that the overall performance in a multi-application environment can be optimized by co-scheduling these different steps, in conjunction with the relevant platform-specific resource schedulers, reservation systems, and administrative policies on the data and computing platforms. Third, it provides support for accessing a rich variety of data sources, including databases, spreadsheets, flat files and potentially web services data, so that client applications can be customized to the different data requirements that might be required, say for risk computations with specific trading profiles or for faster response to changing market conditions maintained in heterogeneous data sources. Fourth, the use of the fast parallel file systems for intermediate data staging ensures the best I/O performance during the computational phase, so that valuable computer time is not tied up in high-latency, unreliable I/O operations to remote servers. Fifth, it provides the capability to invoke the application via a web service interface, thereby allowing BG/L to participate in external business workflows as well as insulating the end-user from the specifics of the data storage and operational platforms. Sixth and finally, it ensures and enhances the mechanisms in the data management platforms for data validation, security, privacy, and audit-trail logging by

extending these to the case when the data is used by an external HPC application in an on-demand mode.

These requirements help to simplify the migration and the routine deployment of financial applications on BG/L. The primary focus in the current work is on the data migration aspects of the middleware layer, which to our knowledge is an aspect that is not addressed in other platform-specific schedulers and reservation systems.

### 3.2. Overall design

The middleware component for job submission and monitoring is currently implemented on the BG/L front-end node, which is physically and logically separated from the compute nodes, and serves as the login point for users to compile and submit jobs to BG/L (thereby being the natural place for locating the middleware layer).

At a high level, the middleware contains the following key elements:

- Jobs are specified using the Job Submission Description Language (JSDL) XML schema, which is maintained by the JSDL-WG working group in the Global Grid Forum [6]. We have extended the JSDL schema to provide a richer set of XML elements for specifying job parameters such as the number of compute nodes, data pre-staging and post-staging descriptions, MPI parameters, and authentication and password information. The JSDL file itself is intended to remain opaque to most users, and an HTML/JSP forms-based job submission wizard allows users to compose and save new JSDL documents (which can be modified and customized as necessary). Alternatively, users can compose the JSDL file directly using an external XML editor of their choice, in which case, a specific password generator is provided using common

public key encryption, which must be used for entering the encrypted values for all password fields in this JSDL file.

- The middleware provides alternate job management interfaces, including an HTML-based set of forms (as mentioned above) and a web services API via SOAP. The web services API can be used by external programs to directly submit and manage jobs, thereby enabling BG/L computation to be part of larger set of web service workflows between collaborating entities, e.g., using orchestration mechanisms such as the Business Process Execution Language (BPEL) [7]. The command-line toolkit that we have implemented for BG/L job management is itself a particular application of this web services API.
- Internally, the middleware contains three major subcomponents:
    - A job meta-scheduler, which queues incoming job requests for execution, and can also serve as a front-end to other schedulers, including the proposed co-scheduler described below. Currently, this meta-scheduler only implements a simple FCFS queue.
    - A job dispatcher, which interacts directly with the MPI *mpirun* command (which in turn interacts with the BG/L node management and control system) to allocate a partition and launch programs on the BG/L nodes.
    - A data-staging gateway, which automates the data migration between the external data sources and the attached file system on BG/L. This component allows users to specify rich data sources and obviates the need to manually stage program data between the original data source and the BG/L file system. Currently, the data-staging gateway supports the scp (secure copy) mode for

remote flat files, and SQL query mode for database extracts. The data sources are specified at job submission time in the JSDL document for the job. The data stage-in module retrieves data from external sources and places them on the specified file system before the computational job is scheduled. Similarly, the data stage-out module places the results data in the specified external stores after the job completion.

These middleware subcomponents are examined in greater detail in the next few subsections.

## 3.3. Job Specification and Submission

The current approach for submitting jobs to BG/L is through the MPI *mpirun* command, which has parameters for specifying the executable filename, the number of processors, the processor partition, and numerous other runtime options. Like most UNIX/Linux command-line programs, *mpirun* is usually invoked via shell scripts, but this approach is problem-specific and *ad hoc* in nature. The JSDL XML schema, which establishes the syntax of the job submission parameters, helps to normalize job submission specifications and facilitates the use of cross-platform interaction between BG/L and external clients using web services.

Like any XML schema, the baseline JSDL establishes a proper syntax, including namespaces and elements (tags). We extended this schema to include tags for *mpirun*-specific information and for the automated data stage-in and stage-out; in the future the schema can be further extended to support other high-level quality-of-service specifications that can be appropriately provisioned. A schematic JSDL file for a typical job submission is shown in Figure 2.

Users can create this JSDL file using a text editor or an XML toolkit, and upload the file through web services to the middleware for immediate job submission. Alternatively, a JSDL file construction wizard is provided as part of the HTML/JSP based user interface, through which the

various parameters of the job submission can be specified. At the end of the process, the user can either save the resulting JSDL file for future modification and/or immediately submit the job to the scheduler.

```
<?xml version="1.0" encoding="UTF-8"?>
<jsdl:JobDefinition sid="xxxxxxx"
xmlns:jsdl="http://schemas.ggf.org/jsdl/2005/04/jsdl">

  <jsdl:JobDescription>

    <jsdl:JobIdentification>
      <jsdl:JobName>HelloWorld</jsdl:JobName>
      <jsdl:Description>This program salutes the world</jsdl:Description>
    </jsdl:JobIdentification>

    <!-- parameters for the executable -->
    <jsdl:Application>
      <jsdl:ApplicationName>HelloWorld</jsdl:ApplicationName>
      <jsdl:BG_MpirunApplication>
        <jsdl:BG_Mpirun_Exe>/bgl/jdoe/bin/helloworld</jsdl:BG_Mpirun_Exe>
        <jsdl:BG_Mpirun_Cwd>/bgl/jdoe/bin</jsdl:BG_Mpirun_Cwd>
      </jsdl:BG_MpirunApplication>
    </jsdl:Application>

    <!-- parameters for BG/L resources used by the executable -->
    <jsdl:Resource>
      <jsdl:StdoutFileName>/tmp/stdout.txt</jsdl:StdoutFileName>
      <jsdl:StderrFileName>/tmp/stderr.txt</jsdl:StderrFileName>
      <jsdl:BG_MpirunResource>
        <jsdl:BG_Mpirun_Partition>M011_32_NE</jsdl:BG_Mpirun_Partition>
        <jsdl:BG_Mpirun_Np>32</jsdl:BG_Mpirun_Np>
        <jsdl:BG_Mpirun_Verbose>1</jsdl:BG_Mpirun_Verbose>
      </jsdl:BG_MpirunResource>
    </jsdl:Resource>

   <!-- Do data stage-in. The stage-out is similar. -->
    <jsdl:PreDataStaging>
      <jsdl:SCPDataStaging>
        <jsdl:ArrangingOrder>1</jsdl:ArrangingOrder>
        <jsdl:Direction>transmit</jsdl:Direction>
        <jsdl:RemoteHostName>127.0.0.1</jsdl:RemoteHostName>
        <jsdl:RemoteUserId>doej</jsdl:RemoteUserId>
        <jsdl:RemotePassword>iluv1bm</jsdl:RemotePassword>
        <jsdl:SourceFileName>/users/doej/data/00input.blob</jsdl:SourceFileName>
        <jsdl:TargetFileName>/home/jdoe/temp/00temp.file</jsdl:TargetFileName>
      </jsdl:SCPDataStaging>
    </jsdl:PreDataStaging>

  </jsdl:JobDescription>
```

**Figure 2: Simplified job submission file based on JSDL syntax**

Since the JSDL specification may contain user and encrypted password information that is vulnerable to hacking, a secure channel is required between the user and the middleware. For example, in the case of middleware access through the HTML pages, the HTTP server must implement SSL encryption. If the interaction is through the web services API, the security requirements as per WS-Security guidelines should be enabled.

The web services API provides a set of remote methods for job submission and monitoring via the SOAP protocol. After the user submits a complete JSDL file to the middleware, a 32-byte job identifier token is returned to the user, which can be used to query the job status, delete the job from the queue, or to kill a running job. When jobs are submitted to the middleware, they are first placed in a meta-scheduler that pre-stages the data, invokes *mpirun* to load and execute code on the BG/L nodes, and finally post-stages the results. Our current implementation uses a simple FCFS scheduling algorithm for these steps, but we plan to look into optimized heuristics for co-scheduling and synchronizing the data movement and job dispatching in conjunction with the existing platform-specific scheduling and calendaring systems. Schedulers for BG/L described in [8] already go beyond FCFS to include job priority and node fragmentation issues; the new co-scheduling algorithms will further extend these to deal with the data transfer and storage issues [9].

### 3.4. Data-Staging Gateway

The data-staging gateway automates the data transfer between the external data sources and the BG/L-attached file system, based on the JSDL specifications in the job submission file, thereby replacing the current practice of performing these data transfers in a manual or *ad hoc* fashion. The design supports the requirement for running the same application repeatedly in response to changing market or portfolio data that is stored and updated in SQL databases.

```
<jsdl:SQLQueryDataStaging>
        <jsdl:ExecutionOrderGroup>11</jsdl:ExecutionOrderGroup>
        <jsdl:SQLStatement>
          <![CDATA[
            SELECT DISTINCT A.INOFC_CD, (
                  (DECIMAL(SUBSTR(A.BAS_YMD,1,4)) - DECIMAL(SUBSTR(B.DEALBASDAY ,1,4))) - 0.0 +
                  (DECIMAL(SUBSTR(A.BAS_YMD,5,2)) - DECIMAL(SUBSTR(B.DEALBASDAY ,5,2))) / 12.0 +
                  (DECIMAL(SUBSTR(A.BAS_YMD,7,2)) - DECIMAL(SUBSTR(B.DEALBASDAY ,7,2))) / 365.0
                ), CASE '02' WHEN ? THEN A.REDEMPTION_RATE
                  ELSE A.REDEMPTION_RATE + A.INTEREST_RATE END
              FROM GRID.R02X A, GRID.R10B B WHERE (A.INOFC_CD = ?) AND
                ((DECIMAL(SUBSTR(A.BAS_YMD,1,4)) - DECIMAL(SUBSTR(B.DEALBASDAY ,1,4))) - 0.0 +
                (DECIMAL(SUBSTR(A.BAS_YMD,5,2)) - DECIMAL(SUBSTR(B.DEALBASDAY ,5,2))) / 12.0 +
                (DECIMAL(SUBSTR(A.BAS_YMD,7,2)) - DECIMAL(SUBSTR(B.DEALBASDAY ,7,2))) / 365.0) >
               0.0
          ]]>
        </jsdl:SQLStatement>
        <jsdl:SQLOutputFileName>/bgl/phant/datadir/data/DATA041</jsdl:SQLOutputFileName>
        <jsdl:SQLInputFileName>/bgl/phant/datadir/data/DATA041_01</jsdl:SQLInputFileName>
        <jsdl:SQLOutputFileCreationFlag>overwrite</jsdl:SQLOutputFileCreationFlag>
        <jsdl:SQLInputFileCompression>yes</jsdl:SQLInputFileCompression>
        <jsdl:SQLOutputFileCompression>yes</jsdl:SQLOutputFileCompression>
</jsdl:SQLQueryDataStaging>
```

**Figure 3: Example of JSDL syntax for data extraction from SQL databases**

The middleware currently supports data extraction from databases using SQL queries and scp data transfers for remote flat files. In the case of scp data-staging in Figure 2, the information following the SCPDataStaging tag includes the remote site, account name, password, and the filenames of the remote and copied files. In the case of SQL queries, as shown in Figure 3, the information following the SQLQueryDataStaging tag includes the database connection parameters, the SQL query statement, and the name of the file on the HPC file system to which the data is extracted (this file is stored in a standard CSV format). The specification is complicated by the fact that multiple SQL queries may have to be executed in a certain sequence because of inter-query data dependencies. The ExecutionOrderGroup tag indicates the group in which the specific SQL statement can be executed. All SQL statements with the same value for this parameter can be executed in independent parallel threads, but only after the SQL statements in the previous stage have been completed. The SQLInputFileName parameter gives the input file for a specific query, which contains the data from a

previous stage, which are used to assign the values of the wild card parameters in the SQL query, as specified using the PreparedStatement interface of the JDBC driver. Data stage-out is performed analogously, for example, with SQL "update" instead of "select" queries.

The data-staging gateway also provides the option for applying streaming transformations to the data during the data-staging operations. For example, the data files may be block-compressed on the fly to save disk storage, or even to optimize the subsequent I/O performance on BG/L (as described below).

In the current design, the data-staging gateway is integrated with the other components of the middleware for simplicity of deployment. However, when the data server is only accessible over a wide-area network, it is preferable especially for large files, to optimize the long-haul data transfer by implementing the data extraction and compression modules as stored procedures on the data server, with the compressed files being directly transferred to the BG/L file system. Other potential server-side data transformations, such as encryption or statistical data scrambling, can also be implemented to protect data privacy on the external network and HPC file system.

### 3.5. Middleware components and schematic workflow

The end-to-end workflow of the financial risk application using our middleware is shown in Figure 4, which illustrates the sequence of steps starting with the client risk application invoking the job submission procedure via web services in step 1, and ending with the archival of the final results in the database in step 11. The database and file sizes shown in Figure 4 were the specific values for the prototype application and are given here for illustrative purposes only.

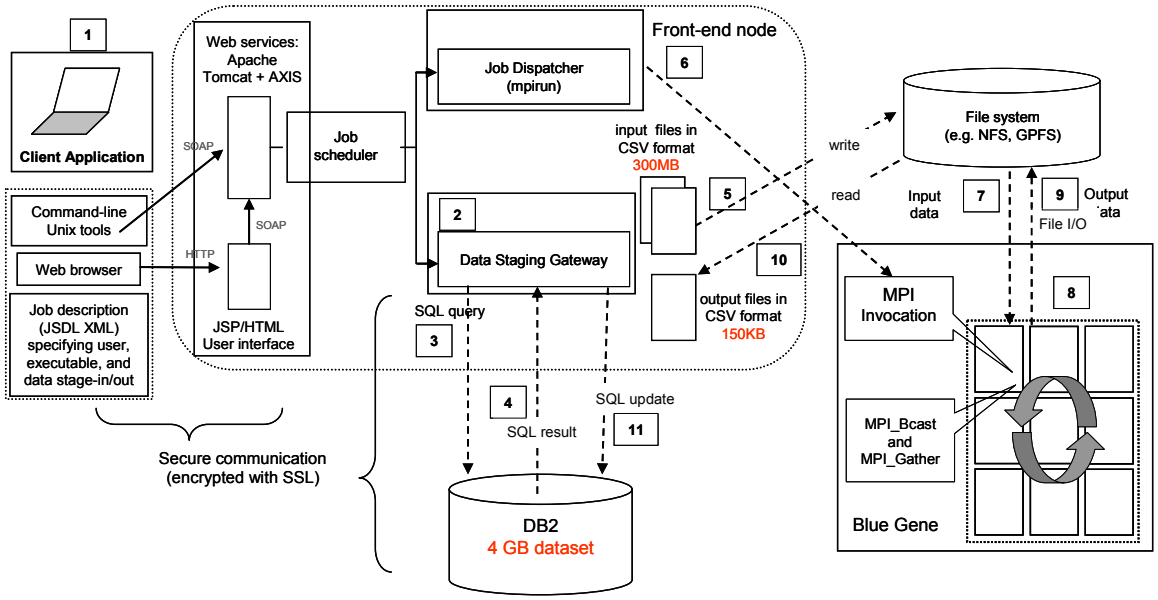


**Figure 4: Schematic of the steps in the workflow as automated by the middleware**

# 4. Parallel Performance Issues

Since the node computations in some financial risk applications are independent and require little explicit inter-processor communication, it may appear that these applications do not derive much advantage from the high-speed interconnection network on BG/L. However, an important and often overlooked aspect in the running time for the application is the I/O time for transferring data between the HPC file-system and the individual processor nodes. A significant part of this data transfer is in fact a collective communication operation for which scalable network architecture of BG/L is especially advantageous (with special hardware features like the deposit bit that can be enabled for broadcast-like operations on the torus network, see [1]), when compared to the equivalent switching networks used in cluster systems or blade servers.

We have therefore considered the I/O performance in greater detail, focusing in particular on the sequential blocking reads of disk-resident files by the individual BG/L compute nodes, with imilar considerations apply to sequential disk writes as well. In the proprietary risk application being benchmarked, the amount of disk reads is large, while the disk writes is relatively small.

A naïve and baseline implementation of sequential disk reads is for each compute node to independently read the file data using the POSIX-compliant *open* and *read* calls to the file system. This baseline implementation can be improved by having a single compute node perform the POSIX *open/read* sequence followed by a broadcast to the other nodes using the MPI_Bcast call from the MPI library. This eliminates redundant data transfer on the file-system access network, and uses the optimized MPI_Bcast function from the customized MPI collective communications library [10].

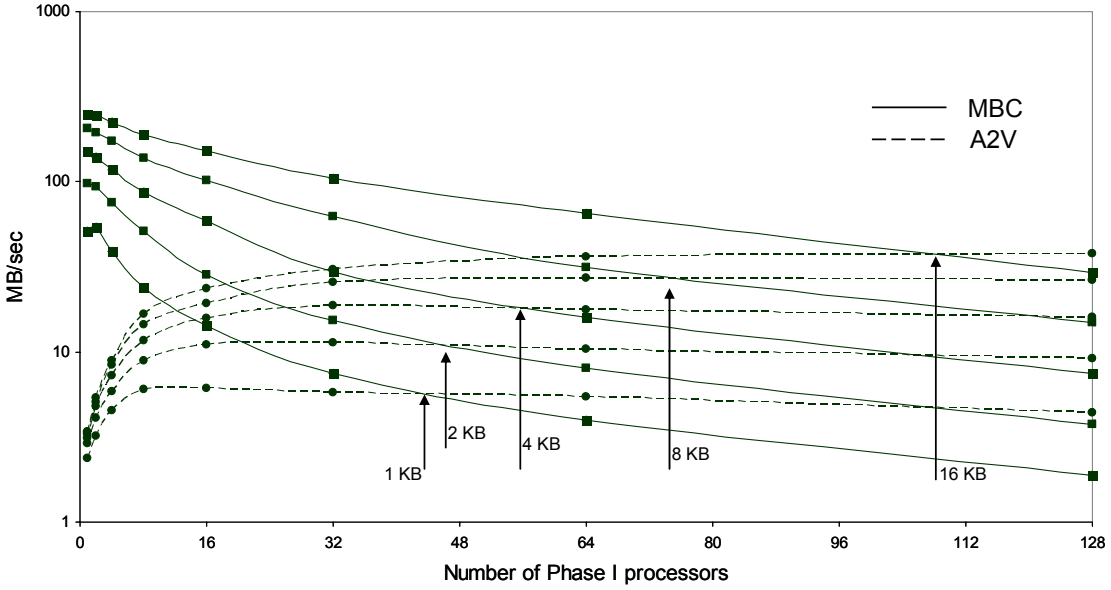


**Figure 5: I/O bandwidth in phase 2 for a 128 node BG/L as a function of number of phase 1 access nodes for the MBC (red) and A2V (blue) algorithms respectively. The sets of smoothed curves are for different segment lengths, respectively 1KB, 2KB, 4KB, 8KB and 16KB from bottom to top. The arrows indicate the cross-over points for the optimal algorithm.**

An even better approach is to use the MPI I/O standard interface [11, Chapter 9], specifically the MPI_File_Read_All function call for performing the blocking broadcast reads (for broadcast writes, the counterpart MPI_File_Write_All function call would be used), Furthermore, as in the

case of the MPI collective communications library the default public-domain implementations [12] can be specifically tuned to take advantage of special features of the BG/L architecture and the associated file I/O subsystem, including processor set (*pset*) groupings of BG/L compute nodes that share an associated dedicated I/O node [1], the use of processor groupings and customized collective communication functions optimized to the BG/L networks [10], and the parallel and buffered I/O capabilities provided by special file systems such as GPFS [13].

The specifically tuned version of the MPI_File_Read_All developed for our financial applications consists of two phases. In phase 1, a small subset of nodes, termed the *access nodes*, perform parallel reads of distinct, non-overlapping sections of the required file segment. In the phase 2, an MPI collective communication distributes the individual sections from the access nodes to the entire partition. The only optimization required in the phase 1 is to distribute the access nodes equally amongst the different *pset* groupings in the partition. The I/O performance tends to increase linearly with the number of access nodes in phase 1, particularly for the so-called "I/O-rich" BG/L configurations (which have the best 1:8 ratio of I/O nodes to compute nodes), although this performance may level off due to bandwidth saturation on the external file access network. The phase 2 implementation will depend on the file segment length and the number of access nodes used in phase 1. For example, the two best algorithms for phase 2 are the MBC algorithm in which each of the nodes in the first phase take turns to broadcast their data to all the remaining nodes via a sequence of MPI_Bcast calls, and the ATV algorithm in which the MPI_All2all function is used to directly effect this transfer in one global exchange of data. As shown in Figure 5 for results on a 128 node partition, each of these algorithms is better over a range of parameters, with the MBC algorithm performing better than ATV for longer segments and fewer phase 1 access nodes. The

MPI_File_Read_All implementation is parameterized to pick the best algorithm dynamically when it is invoked.

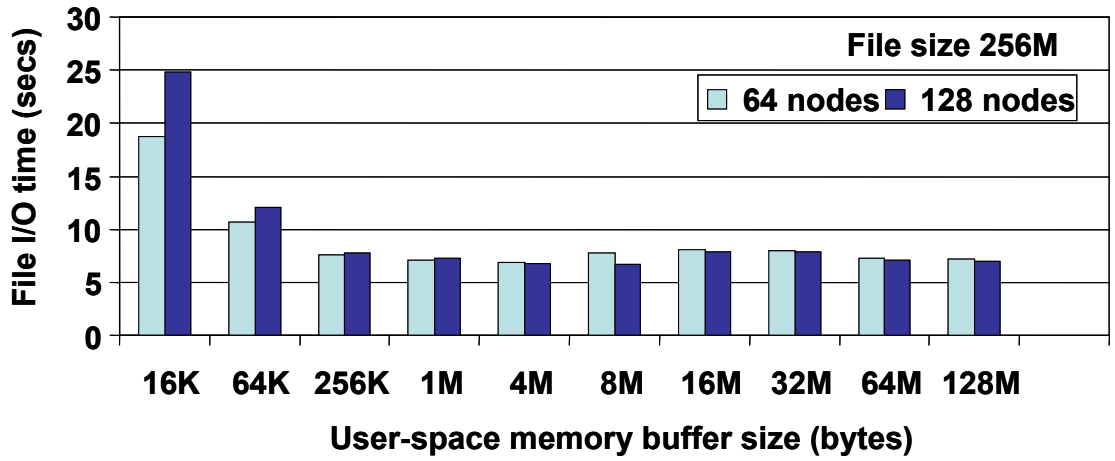


**Figure 6: Time taken for a sequential broadcast read of a 256M file in 256 byte line increments, for varying user-space memory buffers ranging in size from 16K to 128M.**

The use of optimized MPI I/O functions, such as MPI_File_Read_All, can be augmented with other user-level performance optimizations, and two of these are of particular interest, user-space memory buffers and data compression, both of which were implemented as part of a user-level I/O library. For example, in the individual node programs of the risk application, the disk files are read sequentially line by line on the compute nodes. In order to reduce the number of I/O operations and to maximize the data transfer in each such operation, a user-space memory buffer can be used to hold several input lines, and the actual disk read/write operations are carried out only when this buffer space needs to be overwritten and reused. This explicit buffering is useful even when there is underlying support in the disk I/O and communication sub-systems for read-ahead or write-behind operations. Figure 6 describes results for a situation in which a 256 MB file on the GPFS file system is read sequentially by all the nodes in 64 and 128 node partitions, in 256 byte line chunks. The

results indicate that a 1MB buffer gives good performance without imposing an excessive user-space memory overhead on the node program.

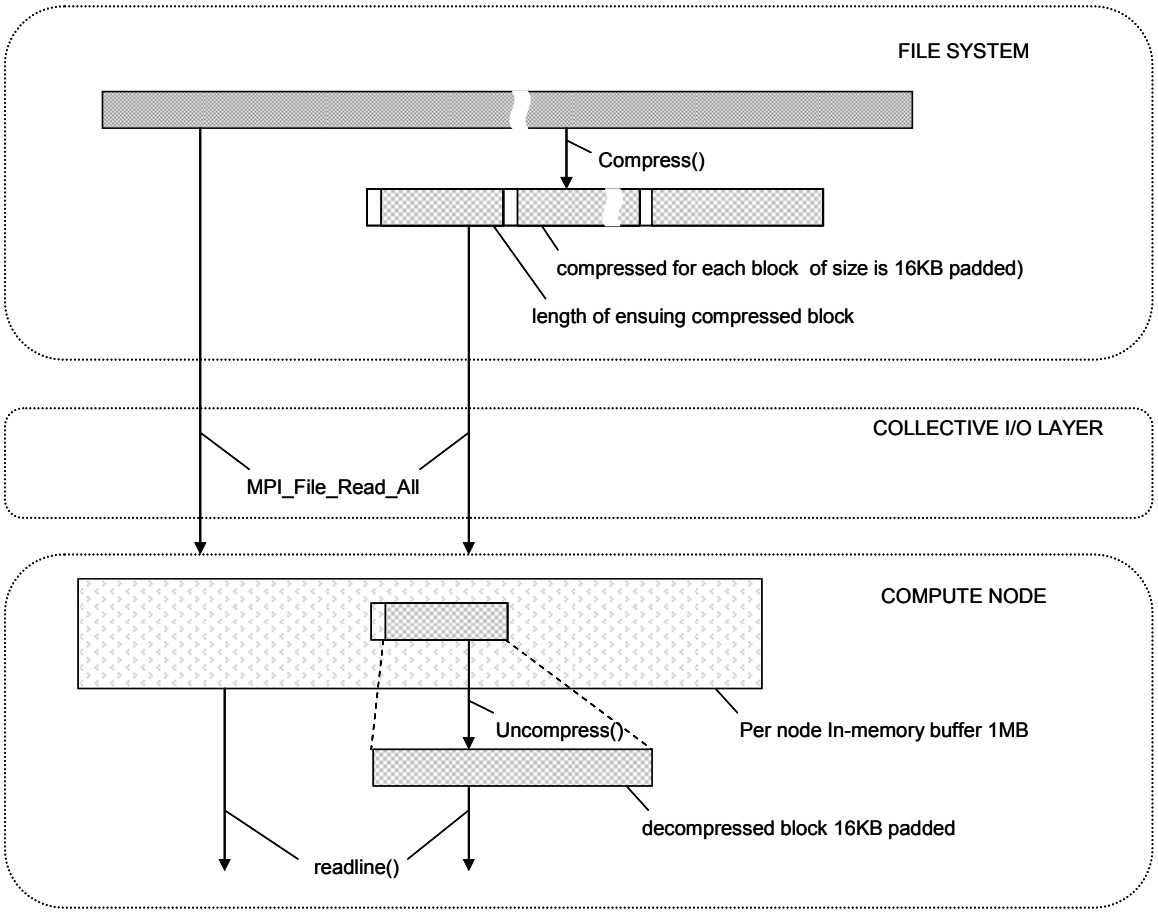


**Figure 7: Schematic of sequential file reads using buffering and (optionally) compression**

Similarly, it is often advantageous to store the data extracted from the database repository in a compressed form on the BG/L file system, and further savings are possible if this compressed data can be directly transferred with the required uncompress (for reads) or compress (for writes) operations is performed on the BG/L compute nodes itself. This approach is schematically shown in Figure 7 for the specific case of sequential disk reads. The BG/L compute nodes use the in-memory routines from the zlib library [14] for the uncompress/compress operations, which are invoked on-

the-fly as the data is transferred to and from the user-space I/O buffers. The block size (16KB) in this library should be consistent with that used by the equivalent compression routines in the data-staging gateway (Section 3.4). The results of using these user-level I/O optimizations in conjunction with the generic (i.e., non-optimized) MPI_File_Read_All function and the regular file system are shown in Table 1, where the 2 data files considered were taken from the proprietary application are considered. These results show the significant improvements over the baseline performance that can be obtained using these optimizations. The efficacy of using data compression depends on the trade-off between the faster communication due to the data reduction and the overhead of the decompression operations on the compute nodes, but the overall approach is always advantageous for disk reads when the data is already stored on the file system in the compressed format.

| | File Size (MB) | | Baseline Read Time (secs) | User-level Optimized Read Time(secs) | |
|---|---|---|---|---|---|
| | Uncompressed | Compressed | | Uncompressed | Compressed |
| DATA043 | 3.4 | 0.27 | 0.186 | 0.124 | 0.074 |
| DATA009 | 286 | 46 | 15.65 | 8.662 | 6.704 |

**Table 1: Performance comparison of sequential broadcast read times on a 128 node partition, using the user-level I/O library and generic MPI_File_Read_All for 2 different files (with and without compression) from the prototype application.**

# 5. Summary

A middleware for deploying financial risk applications on BG/L is described in which the data marshalling and migration is decoupled from the computation, thereby enabling a richer set of external data sources to be used in the application. Furthermore, this decoupling makes it possible to optimize the overall workload in a multi-application environment by taking advantage of the fact that the sequential but otherwise independent steps of data movement and computation can be

flexibly scheduled for each individual application so as to avoid "dead-spots" in usage of the computing resources. This middleware contains components for automating the elements of this application workflow, along with various supporting tools and interfaces. Finally, we also described the I/O optimizations for the data transfer between the file system and the BG/L compute nodes using the collective communication capabilities of the BG/L inter-processor network, which when used in conjunction with special-purpose math libraries described in [16], [17] can significantly improve the performance of the node programs of financial risk applications on BG/L.

# 6. References


1. N. R. Adiga et al, "An Overview of the Blue Gene Computer," IBM Research Report, RC22570, September 2002.
2. G. Almasi et al, "MPI on Blue Gene/L: Designing an Efficient General Purpose Messaging Solution for a Large Cellular System," IBM Research Report RC22851, July 2003.
3. G. Almasi et al, "Architecture and Performance of the Blue Gene/L Message Layer," IBM Research Report RC23236, July 2004.
4. D. Duffie and J. Pan, "An overview of value at risk," Journal of Derivatives, Vol. 4, 1997, p. 7.
5. S. Tezuka et al, "Monte Carlo Grid for financial risk management," Future Generation Computer Systems, Vol. 21, 2005, p. 811.
6. The Job Submission Description Language Specification, Version 1.0, /projects/jsdl-wg/document/draft-ggf-jsdl-spec/en/21.
7. Business Process Execution Language for Web Services, http://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.
8. E. Krevat, J. G. Castanos and J. E. Moreira, "Job Scheduling for the Blue Gene/L System," Job Scheduling Strategies for Parallel Processing, 8th International Workshop, Edinburgh, Scotland, UK, 2002, p. 38.
9. Thomas Phan, Kavitha Ranganathan, and Radu Sion, "Evolving Toward the Perfect Schedule: Co-scheduling Job Assignments and Data Replication in Wide-Area Systems Using a Genetic Algorithm," 11th Workshop on Job Scheduling Strategies for Parallel Processing, Cambridge MA, June 2005.
10. G. Almasi et al, "Optimization of MPI collective communication on Blue Gene/L Systems", 19th ACM International Conference on Supercomputing, Boston MA, June 2005.
11. MPI-2 Extensions to the Message Passing Interface, http://www.mpi-forum.org/docs/mpi2-report.pdf.
12. MPICH2 implementation of MPI, http://www-unix.mcs.anl.gov/mpi/mpich.
13. General Parallel File System, http://www-03.ibm.com/servers/eserver/clusters/software/gpfs.html.

14. H. You et al, “High Performance File I/O for the Blue Gene/L Supercomputer”, under review, 2005.
15. ZLIB compression library, http://www.zlib.net.
16. Mathematics Acceleration Subsystem for Blue Gene/L, http://www-306.ibm.com/software/awdtools/mass/bgl/mass-bgl.html.
17. R. F. Enekel et al, “Custom Math functions for Molecular Dynamics”, IBM Journal of Research and Development, V. 49, No. 2, 2005, p. 465.