Middleware and Performance Issues for Computational Finance Applications on Blue Gene/L

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

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

Abstract

We discuss real-world case studies involving the implementation of a web services middleware tier for the IBM Blue Gene/L supercomputer to support financial business applications. These programs that are representative of a class of modern financial analytics that take part in distributed business workflows and are heavily database-centric with input and output data stored in external SQL data warehouses. We describe the design issues related to the development of our middleware tier that provides a number of core features, including an automated SQL 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. Further, we provide observations on performance optimizations to support the relevant data movement requirements.

1. Introduction

Although high-performance computing (HPC) has been used successfully for scientific applications, there have been few reports on the applicability of HPC for the financial business industry. In this paper we provide real-world case studies and discuss key design issues that arose during the development of a middleware tier that we used to support our customers' computational finance applications on the IBM Blue Gene/L (BG/L) supercomputer [2]. These applications fully utilize the higher degree of parallelism and faster interprocessor communication network that BG/L provides over other architectures, such as computational clusters. The main observations we made from this project are the importance of web services interoperability, SQL data access integration, and performance optimizations for critical data movement.

The financial industry is continuously challenged by increasingly competitive pressure for profits, the emergence of new financial products, and tighter regulatory requirements imposed for capital risk management. The latter concerns have led banks, insurance companies, and other financial corporations to develop and deploy computationally-intensive quantitative applications for financial analytics [10]. From a computing perspective, a principal challenge of these applications is the discovery, aggregation, and staging of the relevant financial data in the analysis.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

This paper describes our experience with data management and performance issues encountered in case studies involving proprietary financial applications on BG/L. Although BG/L has been used extensively in scientific computing [6], ours is one of the first attempts to fully leverage this architecture for computational finance. A larger goal of this work was to also identify the middleware functionality for deploying a broader class of similar business and scientific applications onto BG/L in a production setting.

In general, the characteristics of modern financial applications differ from other scientific and engineering applications in several key ways:

- Financial applications usually require data from external and independent data sources such as SQL databases, remote files, spreadsheets, web services, or streaming data feeds. In contrast, scientific applications usually use pre-staged 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, or regulatory business monitoring applications.
- High-level services specifications must be separated from low-level resource allocation so that dynamic resource provisioning based on quality-of-service or time-to-completion metrics must be considered.

In this paper we provide our observations from our work with two proprietary financial applications that both exhibit the characteristics described above. Unfortunately, the applications and our customers must remain anonymized due to nondisclosure agreements. To make our discussion concrete, we describe one of our customers' applications in sufficient detail in order to explore relevant issues. This application calculates Value-at-Risk (VaR), a key metric in making investment decisions for large corporations. (The second application, while it does not calculate VaR, has the same computation and communication characteristics.) The VaR application was a prototype to help determine the feasibility of running computational finance applications on BG/L, so we have omitted final application performance details in order to respect our disclosure agreement obligations; in the future we look to provide more complete measurements once the applications have been deployed in a production environment.

The input data for the VaR application (consisting of historic market data for risk factors, simulation data, and asset portfolio details) is initially extracted from an SQL database. The finance analytics are floating-point intensive and run in parallel on BG/L while taking advantage of the fast interprocessor communication network for distributing data and collecting results. The output consists of empirical profit-loss data, which is then stored back into an SQL database for post-processing and archiving. In Section 2 we discuss the VaR calculation and its data requirements.

Although the database access and the computing requirements of this application appear straightforward, there are significant inhibitors to its deployment onto existing HPC platforms. A major difficulty is that many HPC platforms, including BG/L, are not well-suited for accessing data in external high-latency data sources. In traditional desktops, the network and transaction latencies for remote access can be overlapped with other work by using multithreaded programming. In contrast, the BG/L compute nodes do not allow multithreading, and as a result, the remote access latencies cannot easily be hidden. Also, applications on BG/L run in a space-sharing rather than a time-slicing mode; i.e., each application uses a distinct, physically-partitioned set of nodes reserved for the application's duration. Lastly, for performance reasons on many HPC systems, it is desirable to stage the program data to and from a specialized file system, such as the General Parallel File System (GPFS) [16]. In a typical use case, these data staging requirements lead to ad hoc solutions with specialized shell scripts dealing with the data extraction and job submission for individual applications; any resulting performance optimizations for the workflow are thus reinvented on a case-by-case basis.

As described in Section 3, we addressed these issues by developing a middleware layer that executes on BG/L's front-end node. This middleware, shown in Figure 1, provides: (1) a data-staging gateway that stages data from external heterogeneous data servers, such as SQL databases, to the BG/L file system; (2) a web services (SOAP) API; (3) a standardized schema for specifying job submission parameters; and (4) a job-submission wizard using a secure HTML/JSP interface and a Unix command-line tool.

We found that data movement was particularly important: because these programs rely on SQL data extraction, the pathway between the database server, the BG/L file system, and the compute nodes becomes critical. This issue is particularly important to the customer, who is spending money for the HPC platform to do computation, not data transfer. In Section 4 we describe the last of our case study observations: the data movement characteristics of these applications and general I/O optimizations.

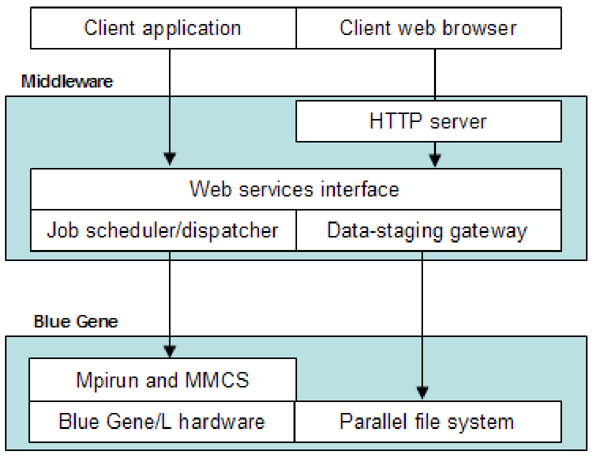

Figure 1. Logical view of middleware components between the users and the BG/L core.

2. Background

2.1. Blue Gene/L overview

The BG/L supercomputer [1] [2] is a family of supercomputers 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. We performed our work at the 2048-node BG/L installed at Almaden Research Center in San Jose, California. Parallel message-passing applications can be developed using an optimized version of the well-known Message Passing Interface (MPI) communications library [3] [5].

2.2. Value-at-Risk overview

Value-at-Risk (VaR) [11] [9] is an important metric used by financial institutions to characterize the market risk exposure of their asset portfolio. The details of VaR are beyond the scope of this paper, but here we briefly describe the most salient and relevant points.

A number of independent software and financial data providers have developed services for computing VaR, and many financial services companies have their own proprietary methodologies. VaR is based on an estimate of a lower quantile of a portfolio's profit-loss distribution over a time horizon given that the portfolio is dependent on underlying risk factors that are subject to market variations. The VaR is then the difference between the expected value of the portfolio and the quantile over the time horizon under these variations. The portfolios being evaluated may consist of assets such as equities, bonds, and currencies, as well as other more complex derivative or assets. The underlying risk factors may include variations in stock prices and equity indices, interest rates, and currency exchange rates.

From a computational perspective, the most widely used approach for estimating VaR is Monte Carlo simulation, which relies on randomly sampling the multivariate risk-factor distribution to generate as set of independent risk scenarios [13]. The asset portfolio is then priced for each risk scenario, and the results are aggregated to obtain the empirical profit-loss distribution for the portfolio. An example of a parallel Monte Carlo VaR using idle cycle scavenging on a grid of computers is described in [21].

From a data engineering perspective, the dynamic nature of the portfolios is critical. The time horizon for VaR depends on 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. During the time horizon the underlying portfolio will be updated repeatedly, *but the computation must be performed with the latest data from the warehouse when the computation runs.* This approach is unlike the common situation in scientific computing, where a large data set is uploaded once and computation is executed by varying parameters [8].

2.3. Application domain of our work

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

End-to-end execution proceeds in three phases: data pre-staging, computation, and data post-staging. Application input consists of data on portfolio holdings, simulation data for generating scenarios, and various algorithm parameters (e.g., the number of scenarios to be evaluated). This input data is roughly 300 MB across 44 files extracted from a 4 GB database using SQL queries during pre-staging.

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, 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 compute phase, the 300 MB of input data is distributed to each compute node, and independent Monte Carlo simulations are run. These simulations use samples from the risk factor distributions to generate market scenarios, which are then used to price the instruments in each scenario's portfolio. The output is written to disk, and in the final post-staging, these results are saved to the SQL database for archiving and further post-processing and analysis.

This prototype is typical of the intra-day market risk calculations that are routinely performed in large banks. The application input changes between successive calculations for variables that are based on market conditions, such as equity prices, exchange rates, and yield 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. 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.

Finally, we note that although some of the issues outlined above may seem specific to financial and business computing applications, they are increasingly becoming true for scientific computing applications as well. For example, in [14] Jim Gray advocates using SQL databases for storing data from scientific instruments and computer simulations due to the advantages of general-purpose databases, such as optimized data layout, metadata organization, application independence, parallelism, scalability, and reliability. An example of using a commercial DBMS for storing data and organizing the workflow of a finite element analysis application is described in [15]. The use of a DBMS enables all the steps to be performed without requiring application-specific data transformation scripts for data selection, transformation, and exchange between different workflow steps.

3. Middleware Architecture

3.1. Specifications

The design of the middleware layer was motivated by requirements that are likely to be encountered across a broad range of financial computing applications. The requirements included the following (the semantics of "must" and "should" follow the guidelines set in IETF RFC 2119):

1. The middleware must automate the data and dispatching workflow and provide a framework for application code organization.

2. The middleware must provide support for accessing a rich variety of external data sources, including SQL databases, spreadsheets, flat files, and potentially web

services, so that the client applications can be customized to the different data requirements (e.g., for risk computations with specific trading profiles, or in response to changing market conditions).

3. The data extraction and migration from external data sources must be separated 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.

4. The middleware must provide 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.

5. The middleware should facilitate the use of a fast parallel file systems like GPFS for intermediate data staging to ensure the best I/O performance without tying up valuable compute time in high-latency, unreliable I/O operations to remote servers.

6. The middleware should ensure and enhance 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.

Our primary focus is on the data migration aspects of the middleware layer, which to our knowledge is not fully addressed in other platform-specific scheduler systems.

3.2. Overall design

The middleware component for job submission and monitoring is implemented on the BG/L front-end node, which is a Linux box physically and logically separated from the compute nodes and serves as the login point for users to compile and submit jobs The middleware is written in Java and runs as a web service, and in our work we have used both the IBM Websphere and Apache Tomcat/AXIS application servers. To parse and handle XML, we used the XMLBeans toolkit.

The end-to-end workflow of the financial application using our middleware is shown in Figure 2; the sequence of steps starts with the client invoking the job submission procedure in step 1 and ends with the archival of the results into the database in step 11. The database and file sizes shown were the values for the prototype application; they are only for illustrative purposes.

- 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 [17]. This schema and our extensions are described in Section 3.3.

- The middleware provides two job management interfaces: HTML-based forms and a web services SOAP API. The web services can be used by external programs, thereby enabling BG/L computation to be part of larger workflows between collaborating entities, e.g., using orchestration mechanisms such as the Business Process Execution Language (BPEL) [7]. Our command-line tool for BG/L job management itself uses the web services API.

- Internally, there are 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.
  - 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 BG/L's file system. This component allows users to specify rich data sources (such as SQL databases) and obviates the need to manually stage data; it is described in Section 3.4.

3.3. Job specification and submission

The current approach for submitting jobs to BG/L is through the MPI *mpirun* Unix command-line program, which has parameters for specifying the executable filename, the number of processors, the processor partition, and numerous other runtime options. Like most command-line programs, *mpirun* can be invoked via shell scripts, but this approach is problem-specific and ad hoc in nature. We complement the scripting approach by using the JSDL XML schema, which normalizes parameter syntax and facilitates the use of cross-platform interaction between BG/L and external clients using web services.

The JSDL file itself is intended to remain opaque to most users, and an HTML/JSP forms-based wizard allows users to compose and save new JSDL documents. Alternatively, users can compose the JSDL file directly using an external editor of their choice, in which case a password generator is provided for entering encrypted values for all password fields into the file. 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.

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

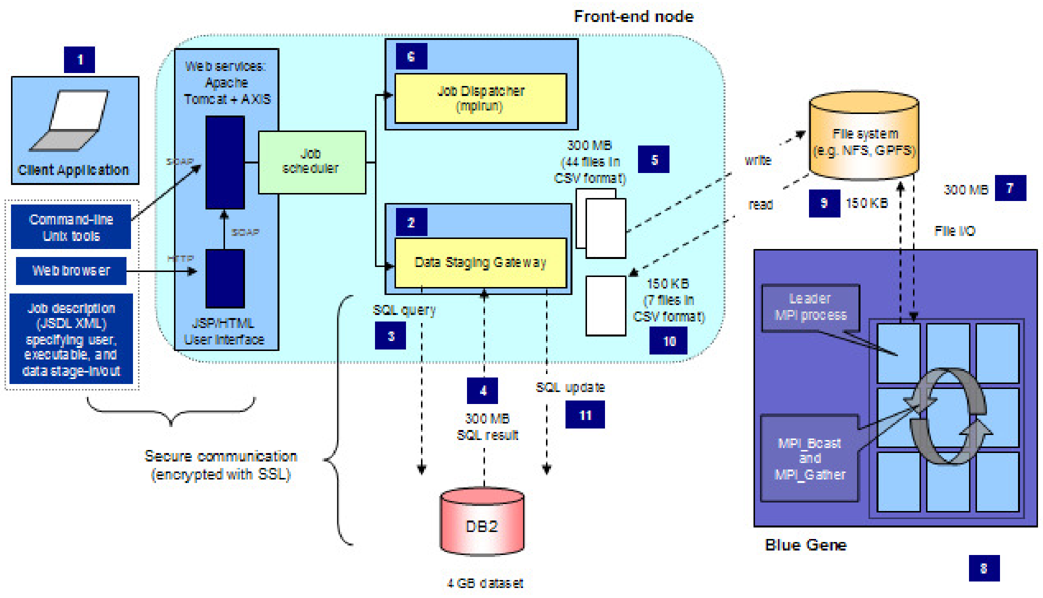

Figure 2. High-level workflow steps automated by the middleware.

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. An example JSDL file is shown in Figure 3.

Since the JSDL specification may contain user and encrypted password information, 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 provide 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 queued job, or kill a running job. When jobs are submitted, 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.

Schedulers for job queueing described in [18] include job priority and node fragmentation issues. Our implementation uses a FCFS scheduling algorithm, but we will look into optimized heuristics for simultaneously co-scheduling the data movement and job dispatching [20] in conjunction with the existing platform-specific scheduling systems.

3.4. Data-staging gateway

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

The middleware can execute data extraction from databases using SQL queries and scp file transfers. In the case of SQL queries, as shown in Figure 4, the information following the SQLQueryDataStaging tag includes the database connection parameters, the SQL query statement, and the name of the file to which the data is extracted (written in a standard CSV format). The ExecutionOrderGroup tag indicates the statement's execution thread. The SQLInputFileName parameter gives the input file for a specific query, which contains the data from a previous stage used to assign the values to wild card parameters in in prepared statements. Data stage-out is performed analogously, for example, with SQL "update" instead of "select" queries.

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 to optimize the long-haul data transfer by implementing extraction and compression as stored procedures in the data server itself, 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.

```
<?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>
</jsdl:JobDefinition>
```

Figure 3. A job submission file using JSDL syntax. Included in this file are sections for job identification, parameters for the executable when run on BG/L (including the executable name and working directory), parameters for parallel job execution (including the number of compute nodes, the name of which BG/L node partition to use, and the Unix standard-out and standard-error file names), and data pre-staging with secure file transfer (including which direction to send the data, the user ID and password for the server from which data is being staged, and the files being transferred). Not shown is the analogous information for doing data post-staging after the computing is done.

4. Parallel Performance Issues

In this section we discuss the last of our observations from our case studies, the data movement performance. The database server's performance is a critical portion of the data pathway, but unfortunately, there is tremendous variability during deployment. The database is under the authority of the customer, who has final say in the DBMS software and the server's hardware, OS, and location (for example, it could be co-located on BG/L's LAN or on the other side of the country). This approach is typical in customer engagement scenarios. Because the database-side is out of our control and cannot be reasonably generalized, here we instead focus on the data pathway that is under our control, namely the path between the BG/L file system and the compute nodes. In the future we will provide more detailed system measurements once the middleware has been deployed into production.

The proprietary financial applications were written in C using the MPI message-passing library and made use of the MPI communication calls we discuss here. We further implemented customized versions of the MPI-IO communication calls in order to improve the data movement performance as described in this section.

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 the scalable network architecture of BG/L is especially advantageous when compared to the equivalent switching networks used in cluster systems or blade servers. (For example, BG/L has special hardware features such as the deposit bit that enables efficient broadcast-like operations on the torus network [5].)

We 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 similar considerations applicable to sequential disk writes as well. In the proprietary risk application being benchmarked, we note that the number of disk reads is large, while the number of disk writes is relatively small.

A naive 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

```
<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 4. An example of JSDL syntax for data extraction from SQL database. The SQL query is embedded directly inside of the XML. The ExecutionOrderGroup tag specifies which thread this query runs in. Other tags provide information on the name of the input file to fill in the wild cards, the name of the output file the results into which the results are written, and compression flags.

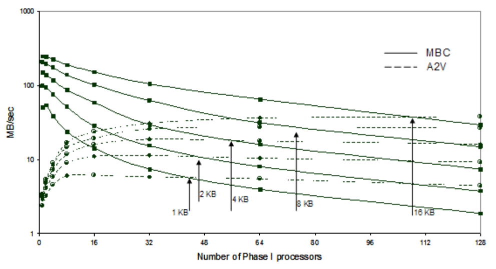

Figure 5. I/O bandwidth in phase 2 for a 128 node BG/L as a function of number of phase 1 access nodes for the MBC (red) and ATV (blue) algorithms respectively. The sets of smoothed curves are for different segment lengths, respectively 1KB, 2KB, 4KB, 8KB and 16KB from bottom to top. The arrows indicate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

eliminates redundant data transfer on the file-system access network and uses the optimized MPI_Bcast function from the MPI collective communications library.

An even better approach is to use the MPI I/O standard interface [19], 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, just as the default public domain implementation of MPI can be optimized, the default MPI-IO library can also be specifically tuned to take advantage of the BG/L architecture. We implemented these optimizations while taking into consideration the associated file I/O subsystem, including processor set (pset) groupings of BG/L compute nodes that share an associated dedicated I/O node [4], the use of processor groupings and customized collective communication functions optimized to the BG/L networks, and the parallel and buffered I/O capabilities provided by GPFS [16].

For example, our specifically tuned version of the MPI_File_Read_All consists of two phases. In phase 1, a small subset of nodes, termed as access nodes, perform parallel reads of distinct, non-overlapping sections of the required file segment. In phase 2, an MPI collective communication distributes the individual sections from the access nodes to the entire partition. The only optimization required in phase 1 is to distribute the access nodes equally among 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 invoked.

The use of optimized MPI I/O functions, such as MPI_File_Read_All, can be augmented with other user-level performance optimizations, of which two, user-space memory buffers and data compression, 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

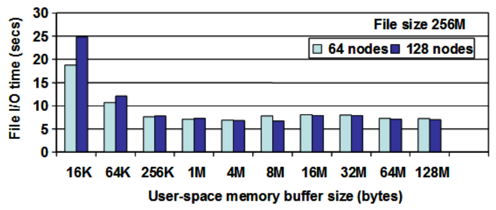

Figure 6. Time taken for a sequential broadcast read of a 256M file in 256 byte line increments, for varying user-space memory buffers ranging in size from 16K to 128M.

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.

5. Conclusion

In this paper we described our real-world observations from deploying a middleware architecture on our BG/L installation to support financial applications. This middleware decouples the data extraction from the computation, thereby enabling the use of a richer set of external data sources, such as SQL databases. 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 in order to avoid underutilization of the compute nodes while waiting for data. This middleware contains components for automating the elements of this application workflow, along with various supporting tools and interfaces.

In the future, we will provide a more detailed end-to-end analysis of the applications' performance. Since the work in this paper centered on one prototype VaR application, full end-to-end performance results will be obtained when the applications have been finally deployed into production.

References

[1] N. Adiga, et al. "An Overview of the Blue Gene Computer," *IBM Research Report, RC22570*, September 2002.

[2] F. Allen, et al. "Blue Gene: A Vision for Protein Science Using a Petaflop Supercomputer," *IBM Systems Journal*, 40(2), 2001.

[3] 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.

[4] G. Almasi, et al. "An Overview of the Blue Gene/L System Software Organization," In *Proceedings of Euro-Par*, 2003.

[5] G. Almasi, et al. "Architecture and Performance of the Blue Gene/L Message Layer," *IBM Research Report RC23236*, July 2004.

[6] G. Almasi, et al. "Early experience with scientific applications on the Blue Gene/L supercomputer," In *Proceedings of Euro-Par Parallel Processing Conference*, 2005.

[7] "Business Process Execution Language for Web Services," `www6.software.ibm.com/software/developer/library/ws-bpel.pdf`

[8] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. "Heuristics for Scheduling Parameter Sweep Applications in Grid Environments," In *Proc. of the Heterogeneous Computing Workshop*, 2000.

[9] M. Crouhy, D. Galai, and R. Mark. "A Comparative Analyis of Current Credit Risk Models," *Journal of Banking and Finance*, vol. 24, 2000.

[10] M. Crouhy, D. Galai, and R. Mark *Risk Management*, McGraw-Hill, New York, 2001.

[11] D. Duffie and J. Pan. "An overview of value at risk," *Journal of Derivatives*, vol. 4, 1997.

[12] R. Enekel et al. "Custom Math functions for Molecular Dynamics," *IBM Journal of Research and Development*, vol. 49, no. 2, 2005.

[13] P. Glasserman, P. Heidelberger, and P. Shahabuddin. "Efficient Monte Carlo Methods for Value-at-Risk," *IBM Research Technical Report RC21812*, 2000.

[14] J. Gray, et al. "Scientific Data Management in the Coming Decade," *SIGMOD Record*, vol. 34, 2005.

[15] G. Heber and J. Gray. "Supporting Finite Element Analysis with a Relational Database Backend; Part I: There is Life beyond Files," *Microsoft Technical Report MSR-TR-2005-49*, April 2005.

[16] IBM. *The General Parallel File System*, `www-03.ibm.com/servers/eserver/clusters/software/gpfs.html`.

[17] "The Job Submission Description Language Specification," `https://forge.gridforum.org/projects/jsdl-wg/`

[18] E. Krevat, J. G. Castanos and J. E. Moreira "Job Scheduling for the Blue Gene/L System," In *Proceedings of Job Scheduling Strategies for Parallel Processing, 8th International Workshop*, 2002.

[19] "MPI-2 Extensions to the Message Passing Interface," `www.mpi-forum.org/docs/mpi2-report.pdf`

[20] T. Phan, K. Ranganathan, and R. Sion. "Evolving Toward the Perfect Schedule: Co-scheduling Job Assignments and Data Replication in Wide-Area Systems Using a Genetic Algorithm," In *Proc. of the Workshop on Job Scheduling Strategies for Parallel Processing*, 2005.

[21] S. Tezuka, et al. "Monte Carlo Grid for financial risk management," *Future Generation Computer Systems*, vol. 21, 2005.

[22] "The Top500 Supercomputer Sites, June 2006," `www.top500.org/lists/2006/06`