SAND2009-2773C

Access to External Resources Using Service-Node Proxies

Ron A. Oldfield, Andrew Wilson, George Davidson, and Craig Ulmer
Sandia National Laboratories[*]

Todd Kordenbrock
Hewlett Packard

ABSTRACT: Partitioning massively parallel supercomputers into service nodes running a full-fledged OS and compute nodes running a lightweight kernel has many well-known advantages but renders it difficult to access externally located resources such as high-performance databases that may only communicate via TCP. We describe an implementation of a proxy service that allows service nodes to act as a relay for SQL requests issued by processes running on the compute nodes. This implementation allows us to move toward using HPC systems for scalable informatics on large data sets that simply cannot be processed on smaller machines.

KEYWORDS: SQL, Proxy, Informatics, Cray, Application Services

1. Introduction

High-performance computing (HPC) systems, particularly large-scale parallel supercomputers, have traditionally been used to address complex scientific problems; however, the recent interest in informatics, especially related to cyber security and social networking, has motivated the informatics community to consider these platforms as a viable solution for informatics problems. Informatics applications that require substantial numerical operations, such as latent semantic analysis (LSA) [5, 6] and eigenvalue decomposition, are particularly well suited for HPC. On an HPC system, these applications can take advantange of existing software like Sandia's Trilinos numerical solvers [8] that were designed and tuned for large-scale parallel systems. In addition to the algorithms requirements, the data requirements for emerging informatics problems are immense. While cluster solutions require substantial culling of the data before analysis, a supercomputer can analyze much larger data sets, perhaps identifying global data relationships among the data that could not be found at a smaller scale.

While HPC platforms provide effective capability to address numerical-computing requirements of informatics applications, they are severely lacking in other respects. For example, informatics applications often require access to large databases, but HPC systems only provide storage to a local parallel file system. In addition, the HPC system architecture, while great for computational physics, lacks functionality provided by Linux clusters that enable remote access to databases. In particular, proprietary network interfaces and protocols do not support TCP which is required by ODBC drivers for remote databases.

In this paper, we describe the design and implementation of a proxy service that allows service nodes to act as a relay for SQL requests issued by an application running on Sandia's Cray XT3 system, Red Storm. This technology represents a major step in the ability to leverage the elite class of HPC systems for informatics applications and provides a valuable template for how to access other remote resources from an HPC system.

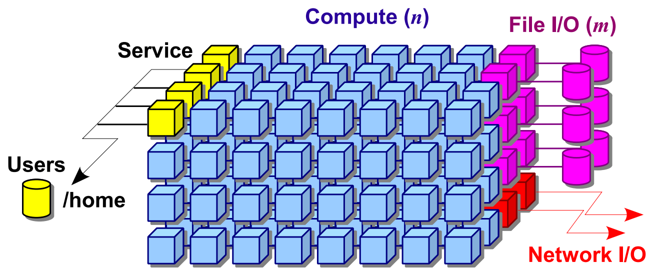

Figure 1: Compute nodes in a partitioned architecture use a "lightweight" operating system with no support for threading, multitasking, or memory management. Service, Network, and I/O nodes use a more "heavyweight" operating system (e.g., Linux) to provide shared services.

2. Background

This section describes the system architecture of the Cray XT3 at Sandia and its limitations with respect to support for informatics applications. We also describe application-level services, and the Titan toolkit – two technologies that, when combined, enable remote database access from a parallel application.

2.2. HPC System Architecture

Figure 1 illustrates the partitioned architecture used by the Cray XT3 Red Storm system at Sandia National Laboratories. This type of architecture is typical for high-end parallel supercomputers and is also the model for the IBM BlueGene series as well as the rest of the Cray XT and XMT systems. A partitioned architecture system [7] is comprised of compute

[*]Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under Contract DE-AC04-94AL85000.

nodes that use a lightweight kernel [11, 19] operating system and service nodes running a full-fledged operating system, typically some variant of Linux. By design, lightweight kernels provide a very limited set of functionality to the application. In the case of Catamount [9] and Compute Node Kernel [19], the operating systems used by Cray and IBM, there is no support for threading, multi-tasking, or memory management. Compute Node Linux [20], the operating system used by the Cray XT4 compute nodes, provides some of these services, but is still somewhat restricted when compared to standard Linux distributions.

In addition to the specialized operating systems, high-end parallel supercomputers also use proprietary networks with specialized protocols. For example, the Cray XT3 and XT4 use the SeaStar network interface [2] that provides extremely high bandwidth, but is only accessible through the Portals programming interface [3]. Since there is no support for TCP on the compute nodes, there is also no support for ODBC, the standard API used to access remote databases. Providing ODBC for a Red Storm application requires an implementation of ODBC or TCP over Portals, an option that incites memories of a bad burrito I once had in downtown El Paso. Instead, we chose to create an application-level service that provides database functionality through an extension of the Titan vtkSQLDatabase class.

2.2. Application-Level Services on the XT3

Application-level services are jobs (either serial or parallel) that process service requests from a parallel application. Unlike a traditional shared service, an application-level service is dedicated to the application and is typically instantiated with the application as part of the launch task.

To enable application-level services, we use a remote-procedure call (RPC) library developed for the lightweight file system project [13, 14]. The LWFS-RPC is based on the Sun RPC Interface [12, 18], but includes modifications specifically designed for efficient data-movement on HPC platforms. For example, in contrast to the Sun RPC, the LWFS RPC is completely asynchronous. This allows clients to overlap computation and I/O — a feature particularly important given that I/O operations are remote for most MPP architectures.

Another key optimization for the LWFS RPC library is the use of separate communication channels for control and data messages. A *control message* is typically small. It identifies the operation to perform, where to get arguments, the structure of the arguments, and so forth. In contrast, a *data message* is typically large and consists of "raw" bytes that, in most cases, do not need to be encoded/decoded by the server. The LWFS-RPC client uses the RPC-like interface to push control messages to the servers, but the server uses a different, one-sided API to push or pull data to/from the client. This protocol allows interactions with heterogeneous servers, but also benefits from allowing the server to control the transport of bulk data [10, 17]. The server can thus manage large volumes of requests with minimal resource requirements.

On the Cray XT3 Red Storm system at Sandia National Laboratories, the LWFS RPC is layered on top of the Portals Message Passing Interface [1]. Portals is a particularly good choice for for MPP systems because it is connectionless, it has one-sided communication APIs that enable the exploitation of remote direct-memory access (RDMA) and operating-system bypass to avoid memory copies in the kernel-managed protocol stack, and it is designed for lightweight operating systems such as the Catamount OS for the Cray XT3 [4].

When a remote service starts, before it can accept requests, it allocates a buffer for incoming requests and creates all the necessary Portals data structures required to direct an incoming request to the right location in the buffer. The Portals data structures, shown in Figure 2 include a memory descriptor to describe the server's buffer for incoming requests, a match list used to identify appropriate messages, a Portal table to index match lists in the Portals library, and an event queue that contains a log of successfully matched messages.

To illustrate how the RPC protocols work, Figure 2 shows the network protocol and Portals data structures used for the `lwfs_write()` function. The client initiates the protocol by encoding/marshaling an RPC request buffer and putting the request on the server's incoming-request buffer. The request buffer includes the memory descriptor of the source data buffer, the memory descriptor of the result buffer, the operation code of the request, and the arguments required for that operation. When the `put()` completes, the server gets a notification (i.e., Portals event) that a new request has arrived. The server then decodes the request, identifies the request as a `write`, and calls the local write function with the decoded arguments. The write function then begins a sequence of one-sided `get()` calls to pull the data from the client into preallocated buffers on the server. While the data is being pulled from the client, the server is writing data from the data buffer to the back-end storage, overlapping the I/O to disk with the network I/O. When the server receives all the data from the client, it puts a message on the client's result buffer notifying the client of completion.

It is important to note that `lwfs_write()` is an asynchronous operation. The client does not have to sit idle waiting for the remote operation to complete. When the client is ready for the result, the `lwfs_wait()` function blocks the client until the specified remote request is either complete (possibly with an error) or has timed out. When the `lwfs_wait()` function returns, the client may release or reuse all buffers reserved for the remote operation.

2.2. Titan Components for Informatics

Our informatics applications are built using components from Titan [21], a parallel informatics processing framework within Kitware's Visualization Toolkit (VTK) [16]. Titan and VTK implement a pipeline architecture where data sets are drawn from sources, processed by chains of filters, and then directed to output sinks which can render to a display or write results back to storage.

We target specifically the Titan classes that allow an application to read from and write to a database. These classes are structured according to Figure 3. The `vtkSQLDatabase` class defines common functions such as opening and clos-

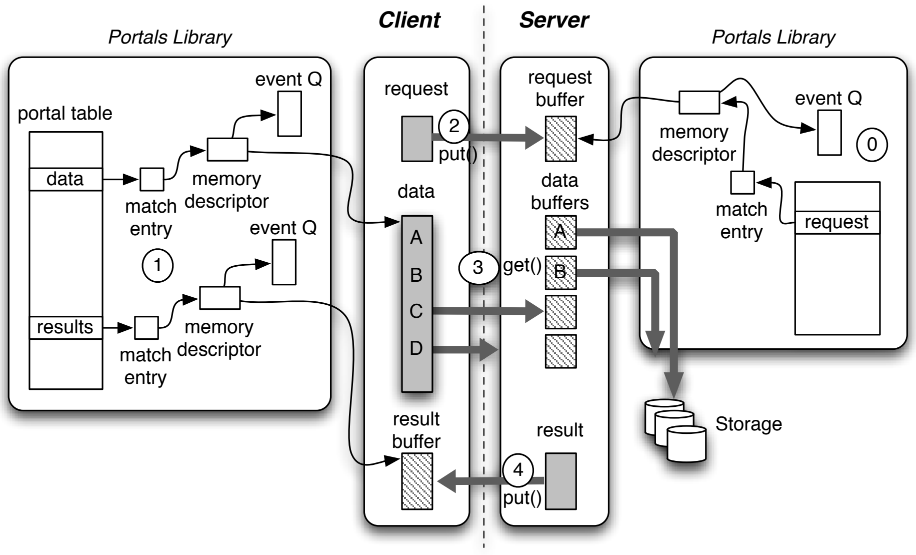

Figure 2: The figure illustrates the required Portals data structures and network protocol used by the `lwfs_write()` function.

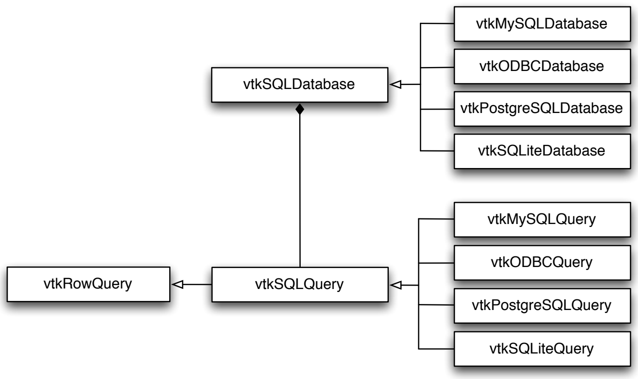

Figure 3: The Titan toolkit implements database connectivity through two abstract classes (vtkSQLDatabase and vtkSQLQuery) that define all operations and several concrete implementations that adapt those operations to third-party, database-specific client libraries. These third-party libraries (and thus the Titan drivers) typically assume that the underlying OS supports POSIX I/O including files and sockets.

ing a database connection. Operations such as specifying and executing an SQL query, checking for error and retrieving results are defined in the abstract `vtkRowQuery` and `vtkSQLQuery` classes. The implementation- and vendor-specific details of executing these operations on various databases are handled in concrete subclasses of `vtkSQLDatabase` and `vtkSQLQuery`. As of May 2009 Titan supports MySQL, PostgreSQL, SQLite and the cross-platform, database-agnostic ODBC protocol.

Titan components are designed primarily to run in a regular desktop or cluster environment. That is, they assume that the underlying operating system has a full set of POSIX I/O calls and sockets and that third-party database interfaces (e.g. the MySQL client library or an ODBC driver manager) can be linked in as shared or static libraries. Our main challenge in this work is to bridge from the compute nodes, where these assumptions do not hold, to the service nodes where a full-featured "heavyweight" operating system is available.

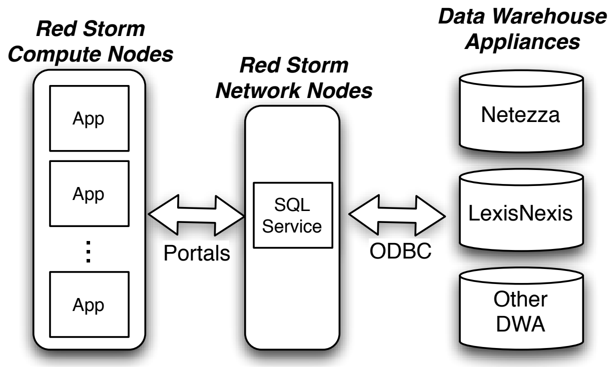

Figure 4: The SQL service is a proxy for SQL requests to a remote database.

3. A Service for Remote Database Access

To enable compute-node informatics applications to access remote databases, we implemented an *SQL Service* that runs on the network nodes of Red Storm and acts as a proxy for SQL requests from compute-node applications. The SQL service, illustrated in Figure 4, uses the LWFS-RPC library to communicate with running applications through Portals, and it uses the Titan database classes to access the remote database using the ODBC API over TCP/IP.

Listing 1: Example code that queries a remote database for all people under the age of 20.

```
// Open a Netezza database (requires appropriate .odbc.ini setup)
vtkSQLDatabase *db = vtkRemoteSQLDatabase::CreateFromURL("odbc://atw_tezz/");
db->Open(passwd);

// Query the database
vtkSQLQuery* query = db->GetQueryInstance();
query->SetQuery("SELECT name,age,weight FROM people WHERE age < 20");
query->Execute();

// Output results
vtkVariantArray *va = vtkVariantArray::New();
while ( query->NextRow( va )) {
    for ( int field=0; field<va->GetNumberOfValues(); field++) {
        if (field > 0) cout << ", ";
        cout << va->GetValue(field).ToString().c_str();
    }
    cout << endl;
}

// Tear down vtk pipeline
va->Delete();
query->Delete();
db->Delete();
```

To understand how the SQL service works, consider the example code shown in Listing 1 that queries a remote database for all people equal to or under the age of 20. The only difference between the example and the equivalent non-remote version is the use of `vtkRemoteSQLDatabase` instead of `vtkSQLDatabase` for the the call to the static method `CreateFromURL()`. The `CreateFromURL` method uses the LWFS RPC API to send a "create" request to the SQL service. The service then instantiates the appropriate subclass of `vtkSQLDatabase` on the network node (in this case, it's a `vtkODBCDatabase` class that connects to a Netezza database), assigns it a unique identifier, and manages the database object in a table of open databases. The `vtkRemoteDatabase`, on the client side, uses id generated by the SQL service when it forwards database requests.

Once connected to a remote database, the client needs to get a query instance from the database object. The `GetQueryInstance()` method returns a `vtkRemoteSQLQuery`, which is a subclass of the `vtkSQLQuery` class. Like the `vtkRemoteSQLDatabase`, the remote query object uses the LWFS-RPC API to send requests to a "partner" query class that exists on the SQL Service. Listings 2 and 3 show the client and server stubs for the `vtkSQLQuery->Execute()` method.

The client-side code simply copies the method parameters into data structures that the LWFS RPC API can serialize and send to the SQL service. The parameters include the ID of the remote query structure and the SQL code to be executed by the remote query object. The asynchronous function `lwfs_call_rpc` passes the handle to the remote LWFS service, an op-code that identifies the Execute op, the function arguments, and structures for result. It then returns a handle to the pending request.

The server-side stub uses a standard set of parameters defined for the LWFS RPC services. The parameters include the process ID of the calling process, the argument structure for the method, and two remote-memory addresses: one for bulk data (not used in this example), and one for the result. To properly execute the query, the server stub has to find the right `vtkSQLQuery` object, referenced by the query ID, set the query string, and call the `Execute()` method. The result, which signals success or failure is sent back with the result structure when stub calls the `lwfs_send_result` function.

Similar stubs exist for the remaining methods of the `vtkSQLQuery` and `vtkSQLDatabase` classes. The details of encoding, decoding, and transferring application-specific data structures are all handled by the LWFS RPC library, making the task of developing proxy-based services relatively straight forward.

4. A Statistics Code Demonstration

To demonstrate the functionality of the SQL service, we implemented a sample parallel statistics code [15] using the remote database classes. Given one or more data series, the statistics code calculates mean, variance, skewness, kurtosis, the covariance matrix and its Cholesky decomposition.

Our initial implementation uses the SQL service to pull the data from a remote Netezza system to node 0 of the parallel application using an SQL `SELECT` statement. The application then uses MPI to evenly distribute the rows of the table to the parallel nodes, calculates statistics on the data in parallel, gathers the results on node 0, and finally uses the SQL service to store the results in a new table back on the remote Netezza system using SQL `INSERT` statements.

Listing 2: Client-side stub for the `vtkRemoteSQLQuery::Execute()` method.

```
bool vtkRemoteSQLQuery::Execute()
{
    // handle to the RPC request
    lwfs_request req;

    // XDR data structure for args and results (these get serialized by LWFS RPC)
    vtk_sql_query_execute_args args;
    vtk_sql_query_execute_res res;

    // Set the arguments for the remote Execute function.
    args.qid = this->GetRemoteQueryID();
    args.qstr = this->GetQuery();

    // Marshal and send the request to the SQL Service
    lwfs_call_rpc(this->GetRemoteService(),VTK_SQL_QUERY_EXECUTE_OP,&args,NULL,0,&res,&req);

    // Wait for async request to complete (no timeout used)
    lwfs_wait(&req,LWFS_INFINITY);

    return res.status;
}
```

Listing 3: Server-side stub for the `vtkRemoteSQLQuery::Execute()` method.

```
int vtk_sql_query_execute_stub(
        const lwfs_remote_pid *caller,
        const vtk_sql_query_execute_args *args,
        const lwfs_rma *data_addr,    // not used
        const lwfs_rma *res_addr)
{
    // A data structure for the result
    vtk_sql_query_execute_res res;

    // Lookup the partner query object (stored in an STL map)
    query = query_map[args->qid];

    // Execute the query
    if (query) {
        query->SetQuery(args->qstr);
        status = query->Execute();

        res.status = status;
    }

    // Send the result of the Execute back to the client
    return lwfs_send_result(VTK_SQL_QUERY_EXECUTE_OP, rc, &res, res_addr);
}
```

We successfully ran the statistics code on 100 nodes of the Red Storm system at Sandia National Laboratories. The input query read a dataset of 100,000 rows and 4 columns (3.2 MB) and dumped back approximately 8KB of results.

The focus of this first demonstration was functionality, not performance. At the time, we were in a race to complete the demonstration before the system switched to a mode that no longer allowed unclassified research. To accelerate our progress, we implemented the minimal set of functionality required to ensure success. In our haste, we skipped of a number of methods and features (some discussed in Section 5) that could have improved performance dramatically. In our next version, we will address these issues and explore other means to improve performance of remote access.

5. Conclusions and Future Work

This paper describes an early prototype of an application-level SQL-proxy service that enables compute-node applications running on a Cray XT3 to interface with a remote database. Preliminary results demonstrate functionality, but would not be acceptable in a production environment do to extremely poor access rates. The poor performance stems from a combination of factors including no support for the bulk I/O features in the ODBC API, minimal effort to optimize data movement for the service, and poor connectivity between the network nodes and the remote database (in this case the Netezza system).

Our first, and current, implementation of the `vtkSQLQuery` class only fetches a single data value at a time and does not take advantage of the bulk I/O features of the LWFS RPC library. To address this, we plan to implement the `GetRow` function, which for large rows, could have a dramatic impact on performance. We are also considering implementing a (non-standard) `GetTable` function that reads and distributes an entire table to nodes in a parallel application. For statistics and graph-analysis codes that plan to analyze the entire result set, this type of functionality could prove critical. Finally, we plan to implement the `Bind` method that reduces database-server latency by binding variables to previously generated execution plans. In many cases, this optimization can significantly reduce query latency, leading to an effective improvement in access rates.

Another critical issue for remote access to data-warehouse appliances is networking. While both the Netezza and Red Storm are designed for high throughput within their machine, their external connectivity is often lacking. For example, since all traffic from the Netezza has to go through a single head node, we expect the limiting factor for remote access to large datasets to be the network interface on that head node. In the case of our Netezza system, we have a single 1 Gb/s interface to rest of the network, and performance is substantially slower when the systems are not co-located.

Although the SQL service still requires a fair amount of work to be complete, it successfully demonstrated the utility of application-level services. Over the next year, we plan to develop a number of new services, including a service for interactive remote visualization and control, a service for real-time data analysis, a service for I/O caching in the compute-node fabric, and a service for in-situ particle detection for a shock physics code.

The SQL service also demonstrated the ability to interface running applications with remote resources. This new capability represents a first-step toward expanding the usage model and customer base for high-end capability systems and is particularly appealing for compute- and data-intensive informatics and cyber-security applications. We now have a clear path towards a usage model that includes integrated use of data-warehouse appliances, interactive visualization and analytic tools, and high-end computing resources.

6. About the Authors

Ron A. Oldfield is a senior member of the technical staff at Sandia National Laboratories in Albuquerque, NM. He received the B.Sc. in computer science from the University of New Mexico in 1993. From 1993 to 1997, he worked in the computational sciences department of Sandia National Laboratories, where he specialized in seismic research and parallel I/O. From 1997 to 2003 he attended graduate school at Dartmouth college and received his Ph.D. in June, 2003. In September of 2003, he returned to Sandia to work in the Scalable Computing Systems and Scalable Architectures departments. He currently leads a number of I/O, resilience, and systems architecture projects. His research interests include parallel and distributed computing, parallel I/O, resilience, and performance modeling. He can be reached at Sandia National Laboratories, P.O. Box 5800, Albuquerque, NM 87185-1110. Email: raoldfi@sandia.gov.

Andrew T. Wilson a senior member of the technical staff at Sandia National Laboratories in Albuquerque, NM. He received the B.Sc. in computer science from Northwestern University in 1993 and his Ph.D. in computer science in 2002 from the University of North Carolina where he specialized in interactive rendering of large data sets. Since then he has worked in the visualization and data analysis group at Sandia. His research interests include informatics algorithms on novel architectures, the visualization and manipulation of uncertain data, and on-line algorithms for tracking trends in streaming data. He can be reached at Sandia National Laboratories, P.O. Box 5800, MS 1323, Albuquerque, NM 87185-1323, USA. Email: atwilso@sandia.gov.

[1] Ron Brightwell, Tramm Hudson, Arthur B. Maccabe, and Rolf Riesen. The Portals 3.0 message passing interface. Technical Report SAND99-2959, Sandia National Laboratories, November 1999.

[2] Ron Brightwell, Kevin Pedretti, Keith Underwood, and Trammell Hudson. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3):41–57, 2006.

[3] Ron Brightwell, Rolf Riesen, Bill Lawry, and Arther B. Maccabe. Portals 3.0: protocol building blocks for low overhead communication. In *Proceedings of the International Parallel and Distributed Processing Symposium*. IEEE Computer Society Press, April 2002.

[4] William J. Camp and James L. Tomkins. The red storm computer architecture and its implementation. In *The Conference on High-Speed Computing: LANL/LLNL/SNL*, Salishan Lodge, Glenedon Beach, Oregon, April 2003.

[5] Peter A. Chew, Brett W. Bader, and Ahmed Abdelali. Latent Morpho-Semantic Analysis: Multilingual information retrieval with character n-grams and mutual information. In *Proceedings of the 22nd International Conference on Computational Linguistics (Coling 2008)*, pages 129–136, Manchester, UK, August 2008.

[6] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society for Information Science*, 41(6):391–407, 1990.

[7] David S. Greenberg, Ron Brightwell, Lee Ann Fisk, Arthur B. Maccabe, and Rolf Riesen. A system software architecture for high-end computing. In *Proceedings of SC97: High Performance Networking and Computing*, pages 1–15, San Jose, California, November 1997. ACM Press.

[8] Michael Heroux, Roscoe Bartlett, Vicki Howle Robert Hoekstra, Jonathan Hu, Tamara Kolda, Richard Lehoucq, Kevin Long, Roger Pawlowski, Eric Phipps, Andrew Salinger, Heidi

Thornquist, Ray Tuminaro, James Willenbring, and Alan Williams. An overview of trilinos. Technical Report SAND2003-2927, Sandia National Laboratories, 2003.

[9] Suzanne M. Kelly and Ron Brightwell. Software architecture of the Light Weight Kernel, Catamount. In *Proceedings of the Cray User Group Meeting*, Albuquerque, NM, May 2005.

[10] David Kotz. Disk-directed I/O for MIMD multiprocessors. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 35, pages 513–535. IEEE Computer Society Press and John Wiley & Sons, 2001.

[11] Arthur B. Maccabe and Stephen R. Wheat. Message passing in PUMA. Technical Report SAND-93-0935C, Sandia National Labs, 1993.

[12] Sun Microsystems. RPC: remote procedure call protocol specification, version 2. Technical Report RFC 1057, Sun Microsystems, Inc., June 1988.

[13] Ron A. Oldfield, Arthur B. Maccabe, Sarala Arunagiri, Todd Kordenbrock, Rolf Riesen, Lee Ward, and Patrick Widener. Lightweight I/O for scientific applications. In *Proceedings of the IEEE International Conference on Cluster Computing*, Barcelona, Spain, September 2006.

[14] Ron A. Oldfield, Patrick Widener, Arthur B. Maccabe, Lee Ward, and Todd Kordenbrock. Efficient data-movement for lightweight I/O. In *Proceedings of the 2006 International Workshop on High Performance I/O Techniques and Deployment of Very Large Scale I/O Systems*, Barcelona, Spain, September 2006.

[15] Philippe Pebay and David Thompson. Scalable descriptive and correlative statistics with Titan. Technical Report SAND2008-8260, Sandia National abs, 2008.

[16] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit An Object-Oriented Approach To 3D Graphics*. Kitware, Inc., 4th edition, December 2006.

[17] K. E. Seamons, Y. Chen, P. Jones, J. Jozwiak, and M. Winslett. Server-directed collective I/O in Panda. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995. IEEE Computer Society Press.

[18] Robert J. Souza and Steven P. Miller. UNIX and remote procedure calls: A peaceful coexistence? In *In proceedings of the 6th International Conference on Distributed Computing Systems*, pages 268–277, Cambridge, Massachusetts, May 1986.

[19] The BlueGene/L Team. An overview of the BlueGene/L supercomputer. In *Proceedings of SC2003: High Performance Networking and Computing*, Baltimore, MD, November 2002.

[20] David Wallace. Compute Node Linux: Overview, progress to date & roadmap. In *Proceedings of the Cray User Group Meeting*, Helsinki Finland, May 2007.

[21] Brian Wylie and Jeffrey Baumes. A unified toolkit for information and scientific visualization. In Katy Börner and Jinah Park, editors, *Proceedings of the SPIE Conference on Visualization and Data Analysis*, volume 7243, San Jose, CA, USA, January 2009. SPIE.