RC24402 (W0712-070) December 20, 2007
Computer Science

IBM Research Report

A Framework for Scheduling Parallel DBMS User-defined Programs on an Attached High-Performance Computer

Michael A. Kochte
Institute fuer Technische Informatik
Universitaet Stuttgart
Postfach 10 60 37, 70049
Germany

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

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

A Framework for Scheduling Parallel DBMS User-defined Programs on an Attached High-Performance Computer

Michael A. Kochte
Institut fuer Technische Informatik
Universitaet Stuttgart
Postfach 10 60 37, 70049, Germany
kochte@iti.uni-stuttgart.de

and

Ramesh Natarajan
IBM Thomas J. Watson Research Center
Yorktown Heights, NY, 10570, USA
nramesh@us.ibm.com

Abstract

We describe a software framework for deploying, scheduling and executing parallel DBMS user-defined programs on an attached high-performance computer (HPC) platform, which is advantageous for many DBMS workloads in the following two aspects. First, those queries which invoke long-running user-defined programs can be speeded up by taking advantage of the greater hardware parallelism on the attached HPC platform. Second, the interactive response time of the remaining applications on the database server platform is improved by the off-loading these long-running user-defined programs to the attached HPC platform. This framework provides a new approach for integrating high-performance computing into the workflow of query-oriented, computationally-intensive applications.

I Introduction

Commercial database management systems (DBMS) have been widely used for applications in transactional processing, online analytics and data warehousing. However, many emerging DBMS applications require the ability to store, query and analyze a wide variety of complex data types, such as images, documents, multimedia, raw event streams from scientific instruments, and unprocessed result sets of high-performance computer simulations [BC05]. The relevant database processing for these complex data types is typically more than just simple database archival or retrieval, and includes compute-intensive processing of the raw data before its use by external client applications. Specific examples of such compute-intensive processing include operations such as

high-level semantic query and search, content-based indexing, sophisticated data modeling, data mining analytics, computer-aided design etc.

Since many of these data and analytical transformations are broadly useful, they are often implemented as embedded DBMS user-defined programs, thereby encapsulating this generic functionality for use in a variety of external client applications. For example, "Database extenders," which are a collection of related user-defined complex data types, and concomitant user-defined stored procedures or user-defined functions defined over these data types, often provide the intrinsic database functionality, performance and modularity in support of specific classes of external applications. From a functionality perspective, external application developers can invoke these embedded user-defined programs using the familiar set-oriented or SQL-based syntax and query interface. From a performance perspective, the use of embedded user-defined programs often reduces the overhead of moving the raw data across the network from the database server to the client application, either by virtue of transforming the raw data to a more compressed representation, or by substantially pre-filtering the raw data on the database server itself before transmission to the external client application. Finally from a software perspective, the use of embedded user-defined programs makes it easier to ensure the privacy, integrity and coherence of the raw data within the database, by providing an "object-like" interface to the raw data (whose contents and representation can be kept private, and need not be explicitly copied or shared with the external applications).

Notwithstanding these advantages, the processing requirements for executing these embedded user-defined programs on the database server can be extremely large, and to our knowledge, this performance aspect has rarely been addressed in the conventional database performance benchmarks, or in the design and sizing of hardware platforms for general-purpose database server systems.

Large-scale, commercial database management systems (DBMS) are typically hosted on shared-memory multiprocessors or on network-clustered computer platforms. On these hardware platforms, the database controller software (which is responsible for coordinating the execution of the parallel query plan generated by the database query optimizer), is able to take advantage of this underlying hardware parallelism for speeding up query execution. However, commercial DBMS systems rarely provide any programming interfaces that would allow any external applications or embedded user-defined programs to directly take advantage of the underlying hardware parallelism. In some cases, the database controller software can implicitly parallelize the execution of certain embedded user-defined functions within a parallel query plan during query execution. Nevertheless, most commercial database

systems impose severe restrictions on the user-defined programs that can be implicitly parallelized in this fashion. For example, these restrictions often apply to user-defined functions that use scratchpad memory for storing information between repeated function invocations, that perform external actions such as file input-output operations, or that involve non-deterministic execution (i.e., where different function outputs may be returned for the same inputs, such as with parallel asynchronous calls to a random number generator), or for user-defined table functions that return multiple rows of values for each function invocation (Chapter 6 in [C98] provides a detailed discussion of these default restrictions for one specific commercial database). Furthermore, although these default restrictions can be over-ridden by the programmer (e.g., based on whether the "safe" serial semantics are important and need to be preserved in the implicit parallel execution of the user-defined program), the overall degree of parallelism that can be used for executing these user-defined programs is often fixed or restricted by pre-configured parameters in the database platform that specify the allowable maximum number of threads on a shared memory platform, or the maximum number of data partitions or processors in a distributed cluster platform, even though these user-defined programs may possibly be capable of exploiting a much higher degree of parallelism. In terms of flexibility, even when these database configuration parameters are set to the maximum values supported by the underlying hardware platform, it is often the case that within this range of parallelism, each individual database application has its own optimal parallel granularity that is determined by a complex interplay of factors involving the level of parallel co-ordination, synchronization and data movement in the application, and there is unlikely to be single global optimal setting for all the applications that run on the database server. Finally, in this scenario, improving the parallel performance of even a single embedded user-defined program beyond the limitations of the existing hardware parallelism requires an overall and expensive upgrade of the entire database platform.

In summary, therefore, the underlying hardware control or data parallelism in existing commercial database systems is typically only exposed to the query processing engine and database controller. These database systems do not provide application programming interfaces (API's) for writing general-purpose, parallel, user-defined stored procedures and user-defined functions, or the flexibility to be able to tune the performance of implicitly parallelized embedded applications on an individualized basis beyond the range of the pre-configured limitations of the database platform.

The outline of this paper is as follows. Section II considers the previous work on speeding up query execution using parallel hardware-based accelerators for commercial databases, and motivates the need for the

proposed configuration consisting of an attached HPC platform and associated software framework as described in this paper. Section III gives a full description of software framework that is used for deploying, scheduling and executing compute-intensive queries on this attached HPC platform. Section IV describes our experience with this configuration for applications in bio-informatics and life sciences. Section V contains our conclusions and future recommendations.

II Previous Work

There have been several proposals for improving the performance of database query processing for specific compute-intensive applications using special-purpose hardware accelerators within the database server platform itself. For example, [LHMK91] profiled some database query workloads to identify the most time-consuming operations, and proposed using custom VLSI hardware filters in the data path between the disk storage interface and the CPU for these specific operations (see also the similar ideas in [FM91] and [AS9]). A similar approach using custom hardware accelerators for string and pattern matching operations in text-oriented database applications is also described in [MLF91].

A more recent approach is "active-disk" technology [RFGN01], where a portion of the query processing that would normally be performed entirely on the main CPU of the database server itself, is instead scheduled to run on the general-purpose microprocessor units that are increasingly being used at the disk controller interfaces of individual storage disk drives. This approach takes advantage of the much higher degree of parallelism found at the storage interfaces of multi-disk commercial database systems. For many database queries, the execution at the disk controller interface achieves a substantial pre-filtering and reduction in the data volume that is transmitted to the main database server CPU via the storage system network. However, there are limitations on the nature of the workload that can be off-loaded in this way. Particularly since the individual disk controllers do not directly communicate with each other, these off-loaded tasks are limited to simple filtering and transformation operations on the respective independent data streams. In summary, while this technology is very effective for simple stream-oriented operations on the raw data from disk, the overall approach does not yet have the flexibility and programmability for more complex operations that require parallel synchronization and communication between these independent data streams.

The framework described in this paper, in contrast to these previous approaches, schedules the execution of compute-intensive, DBMS parallel user-defined programs on a separate general-purpose HPC platform. The major performance limitation in our framework is the overhead of data movement between the database server and attached HPC platform, but for long-running computations with small data transfer requirements, or for multiple queries on the same target data, the achievable computational performance gains on the HPC platform significantly offsets these data transfer overheads. Furthermore, our overall approach obviates the need for database application developers and users to be familiar with any specialized parallel programming and parallel execution expertise on the HPC platform, or with the need to explicitly schedule the data movement and computational processing on the external HPC platform. In fact, any rather than being ad hoc and non-automated, our approach makes it possible to compose complex database queries, with the desired remote computation of parallel user-defined functions taking place entirely within the SQL query framework itself. Overall, therefore, the software framework described below provides a flexible, reliable and automated approach for scheduling and accelerating parallel DBMS user-defined functions on an attached HPC platform.

III Description of Framework

Figure 1 is a schematic of the proposed framework consisting of a database server and an attached high-performance parallel computer (HPC) platform. A client application issues one or more SQL queries to the database server, and parts of the query workload are dispatched and executed on the parallel computer, with the results being transmitted back to the database for final incorporation into the result set for the client application. Specifically, the compute-intensive parts of the query workload, such as any embedded parallel user-defined programs, are scheduled and executed on the HPC platform, and the results are then transmitted back to the database server for any further query processing before final integration into the eventual result set returned to the client application.

The two important aspects of this proposed framework are as follows. First, the off-loading of the compute-intensive workload to the attached parallel computer can improve the query performance and query response time on the database server for either a single query invocation, or for multiple and related query invocations on the same target database table. Second, the entire process by which this performance improvement is obtained does not require any significant reworking of the client application, since the execution of the user-defined program on the back-end HPC platform takes place with the same semantics, results and reliability as if executed on

the database server itself. The framework also provides the client application with the ability to customize and optimize certain aspects of this off-loaded, remote execution using the familiar SQL interface on the database server.

Figure 2 illustrates the various software components of the framework in greater detail, with specific components for initializing the services for executing future off-loaded computations, for scheduling these computations when requested, and for collecting and transmitting the results back to the database server. Typically these individual components are deployed either on the HPC platform, or on one or more of its front-end host computers. A different set of components are deployed within the database server itself, and consist of specific user-defined program stubs that invoke the corresponding services on the back-end HPC platform using standard protocols such as web services or JDBC (Java Database Connectivity). In addition, the database server allocates a set of temporary tables for storing any intermediate or final result sets as required by the given query workflow.

On the HPC platform itself, the main component is a service wrapper which runs on each parallel compute node and encapsulates the actual application service on that node for executing the parallel tasks. This service wrapper is responsible for communication with the other components on the front-end host for the overall scheduling and synchronization. It is also responsible for storing a distinct sub-partition of the appropriate target database table or materialized view in a form that can be efficiently accessed by the underlying node application service using a simple programming interface to retrieve these table rows.

As described here, the front-end host computer contains many of the important components of the framework including:

- A service deployment module that is responsible for loading the application service on the required subset of the nodes of the HPC platform.
- A service node dispatcher component that maintains the state of the individual nodes of the HPC platform.
- A query partition dispatcher component that works in conjunction with the service node dispatcher to requisition and set up a subset of nodes on the HPC platform for a specific service invocation, and to execute a distributed query on this query partition. Future queries are also dispatched to the same query partition if the underlying target database table or materialized view is unchanged between the invocations (so as to avoid the overhead of recopying the target table data from the database).

— A results collector component that aggregates the results from the individual compute nodes on the parallel machine, with these results being returned to the invoking service function on the database server, or alternatively, being directly inserted into pre-specified temporary tables on the database server.

A database relay component may also be required in specific implementations of this framework, including the prototype configuration described in Section IV, since many parallel HPC platforms do not support the standard protocols or programming API's for interactive database access. The database relay component manages the data transport between the database server and the parallel computer nodes, mediating between the data transfer protocols used for the database server, and the I/O protocols for the individual nodes on the HPC platform.

Figures 3 through 5 show the sequence of phases in the off-loaded parallel query execution on the HPC platform. Here Phase I refers to the deployment of the application, Phase II to the data initialization, and Phase III to the execution of the off-loaded tasks and the return of the results to the database server.

Figure 3 illustrates the steps involved in the Phase I of the query execution where the application service that is responsible for executing the required off-loaded database queries is installed on a set of compute nodes in the parallel computer (these are termed the application service nodes). We assume that the software implementation of the desired database user-defined function is provided as an application service, and is embedded within the service wrapper. This application service (along with the encapsulating service wrapper) is compiled and linked into binaries for the individual compute nodes on the HPC platform using the appropriate parallel libraries.

When a specific request is received from the database server as part of its application workflow execution, the deployment service on the front-end host invokes the program loader to start up the application service on a given collection of compute nodes (this program loader is usually platform-specific, such as MPIRUN loader used for MPI-based application binaries [MPI]). As the application service is loaded on these compute nodes, control is transferred to the service wrapper which initiates a message to register the node with the service node dispatcher component running on the front-end host. The service node dispatcher maintains a registry of all the compute nodes that are available with each specific application service deployed in this fashion.

Figure 4 illustrates the steps involved in the Phase II of the query execution where the target database table required in the subsequent query execution is transferred from the database server to a given subset of the compute nodes that were initialized in Phase 1 (this subset of nodes is termed an active query partition). The data

initialization phase is triggered by a request from the database server to the query partition dispatcher to prepare an active query partition for a target table against which future queries in the ensuing Phase 3 will be run. Upon the request from the database server the query partition dispatcher first checks if an active query partition for the target table already exists and is in ready state for handling new queries. If no such partition exists, the query partition dispatcher will create one as outlined in Figure 4. To obtain compute resources for the new partition, the query partition dispatcher negotiates with the service node dispatcher to allocate a subset of the available compute nodes on which the relevant application service has been initialized. The service wrappers on these individual application service nodes then initiate separate data transfers to copy mutually-exclusive but collectively-exhaustive row partitions of the required data from the database server (and this data transfer may be routed through the database relay component, as described below). The individual data partitions are stored in in-memory data caches allocated in the application service wrapper, from which this data can be accessed during subsequent query execution by the application service using a simple programming interface.

The database relay component, which runs either on the compute nodes or on the HPC front-end host, is responsible for mediating the different data transfer and communication protocol requirements, for example by converting between MPI- or UNIX socket-based communication on the HPC platform, and the standard database access protocols like JDBC on the database server. The application service wrapper on the compute nodes "function ships" the relevant SQL query to the database relay component, which completes the query and transmits the result set back to compute nodes in the appropriate representation for storage in the local data cache.

The query partition dispatcher may be unable to find a suitable query partition for a particular target table already loaded on another partition that is either reserved or in use. In this case, the relevant data can be copied over to clone another active query partition, so that this data transfer takes place within the HPC platform using its high-speed internal network, rather than reverting to the original database which entails much higher communication costs.

Figure 5 illustrates the steps involved in the Phase III of the query execution, in which the relevant query parameters are transmitted to the appropriate active query partition previously set up in Phase II, and the results are collected and either returned directly to the invoking database function or inserted in a results table in the database server. The query request is initiated by a user-defined function stub executed on the database server, and it encapsulates all the input parameters required for the application service on the nodes of the HPC platform,

including the name of the particular target table against which the query is executed. The endpoint for this query request is the application service host component running on the HPC front-end host, which in turn inserts this query request into a set of queues maintained in the query partition dispatcher. Separate queues are maintained for each query partition that has been allocated and assigned to a specific target table in Phase II above.

The query partition dispatcher eventually submits this query request to the application service wrappers running on the nodes of a suitable query partition and then waits for the job completion, with the job status code being returned to the user-defined function in the database server issuing the application execution request. When an application service wrapper receives a query request, it extracts the parameter values from the request message and invokes the application service. The results of the query, which are temporarily stored in the results cache in the application service wrapper for each node in this query partition, are eventually aggregated within the results collector component on the front-end host. Finally, this aggregated result data set is returned to the originating user-defined function on the database server. Since the originating user-defined function that invoked the remote execution is either a user-defined table function or is embedded in a user-defined table function, these results can be further processed as part of a complex SQL query workflow (for example, SQL operations like ORDER BY or GROUP BY can based on the result column values). Similarly, the results table can be joined to other database tables as required by the overall query workflow. An alternative mechanism, in which the application service wrapper can use the database relay as in Phase II to directly insert the results into a specified results table on the database, is also supported in the framework.

IV Application Enablement and Performance Results

The commercial database server platform used in our application enablement experiments is IBM DB2 Version 9.1 [DB2] running on a dual-processor, Xeon 2.4 GHz CPU with 2GB of RAM storage with a 1000 Mbit Ethernet interface.

The attached HPC platform that was used for remote execution of the parallel user-defined programs is a single rack of an IBM Blue Gene/L e-server platform [BG] with 1024 compute nodes, with each compute node comprising of two PowerPC 440 processors operating at 700 MHz with 512 MB of RAM storage per node. Although our specific use of the Blue Gene/L platform here does not require the MPI message-passing

communication libraries, the use of these libraries for the application service programming is not precluded in the framework.

There are some specific technical issues in configuring each database server/HPC platform combination for this software framework. For instance, for the combination of the IBM DB2 database and the IBM Blue Gene/L parallel computer (and in fact for most other equivalent database and parallel computer platforms) there is as yet no parallel programming API or any operating systems support for direct access to the database server from the individual compute nodes on the parallel computer. Therefore a separate IBM P-series server connected over the local area network to the Blue Gene/L system is used for hosting various components of the framework, including

1) The scheduler component which contains a registry of the Blue Gene/L compute-node partitions available for the query processing application;

2) The web service component that supports SOAP-based web services calls initiated from the database server to execute various components of the query workflow;

3) The job-submission interface component to reserve and start up applications on the compute nodes of the Blue Gene/L computer;

4) The database relay component that maintains one or more socket connections to the individual Blue Gene/L compute-nodes, and is responsible for executing various database commands relayed from the compute nodes on these socket connections, as well as for communicating the result sets or completion codes of these database commands back to the compute nodes initiating database query requests.

The applicability of this framework to some life sciences applications in the field of bio-informatics and computational chemistry is considered in greater detail in this section.

IV a. Bio-Informatics Application

This application is in the area of bio-informatics algorithms for sequence similarity and alignment in DNA and protein sequence databases. In recent years, the amount of gene and protein sequence data has been growing rapidly, and this data is now being stored in a variety of repositories including commercial relational databases as well as numerous proprietary, non-relational database formats. An essential task in bio-informatics is the comparison of a new sequence or sequence fragment against a subset of sequences from an existing sequence repository, in order to detect sequence similarities or homologies. The resulting matches are then collated with other scientific data and metadata on the closely matching sequences (such as conformation and structural details,

experimental data, functional annotations etc.) in order to provide information for further biological or genomic investigation on this new sequence. Since many of the steps in this information collation require data integration and aggregation, the workflow for this task is greatly facilitated if the sequence data and sequence metadata, as well as the results of the sequence matching algorithms, are all accessible from an SQL query interface.

One approach to achieving this capability, often termed as the extract/transform/load (ETL) approach, requires the relevant sequence libraries to be imported into a commercial relational database from their original data formats, using custom loader scripts for each proprietary data format in which the original sequence libraries and metadata are stored. An alternative approach described in [HSK+01], retains the sequence data in its original data repositories, but layers an abstract or federated view of this heterogeneous set of data sources on the primary front-end database server, using a set of embedded wrapper functions on this primary front-end database to provide the necessary mapping of the input queries and query results that need to be exchanged between the primary database and the back-end heterogeneous data sources.

The use of an SQL-based query interface to invoke various biological sequence matching algorithms has been considered in two different ways in the previous literature. In the first approach, these algorithms have been implemented as embedded user-defined programs, as described specifically for the BLAST algorithm in [SCD+05]. In the second approach, again specifically for the BLAST algorithm [ED04], the database wrapper approach has been extended by transferring the required calculations to a separate BLAST server, and then mapping the results back into tables on the database server. These two approaches differ quite substantially in the implementation details, but they both provide some important capabilities, viz., the ability to use the database SQL query interface for accessing and querying one or more data sources containing biological sequence data and metadata, and the ability to invoke sequence matching algorithms such as BLAST directly from these database queries. These capabilities allow application developers to generate complex queries, which for example can incorporate filtering of the initial search space of sequences using predicates based on the sequence metadata, and post-processing of the results by indexing and joining the top-ranked sequences returned from the matching algorithms with other related data repositories that contain further information on them. In this way, these database-enabled implementations of sequence matching algorithms provide the capability to automate, enhance and accelerate the process of new scientific discovery from the sequence data. However, neither of the two approaches discussed above has been implemented in terms of a general-purpose, parallel computation framework as described in the present paper.

There is considerable previous work in the development of parallel algorithms for biological sequence matching and alignment on a variety of HPC platforms ranging from special-purpose accelerators, multi-threaded symmetric multiprocessing systems, and distributed-memory computers (see for example, [CMS+04]). From the point of view of scalability, the distributed memory platforms are the most interesting, and two main approaches have been pursued here for implementing parallel biological sequence matching algorithms.

In the first approach, termed database segmentation, the target library of sequences is partitioned across a set of compute nodes (preferably using sufficient nodes so that each individual partition fits within the node memory). The parallel scalability of this approach is eventually limited by the data movement overhead for distributing the library sequence data and collecting the results from a large set of compute nodes. A study of the performance optimizations required for implementing this distributed memory parallel approach can be found in [DCF03], with extensions for optimizing the parallel disk I/O performance in [LMC+05].

In the second approach, termed query segmentation, there is a batch of similar but independent queries, and each individual query in this batch is simultaneously executed in parallel against the target sequence library. The target sequence library is therefore replicated across multiple nodes on the distributed memory platform, as described in [BPC+99]. This approach is limited by the memory on the individual nodes, which may not be sufficient for storing the entire target sequence library, but this particular difficulty can be overcome by using a combination of database and query segmentation, which is the most effective and scalable approach for distributed-memory parallel computers that have thousands of processor (see [RLM+05]).

To our knowledge, none of these parallel implementations of BLAST or any other sequence matching algorithms has considered the issue of accessing these algorithms from an SQL query interface for easy integration into the processing requirements of a larger query workflow. Furthermore, as mentioned earlier, it is difficult to directly incorporate these parallel implementations as embedded user-defined programs in a commercial relational database, due to their extensive use of message-passing libraries and other parallel programming constructs that are generally not supported in database programming and runtime environments.

The BLAST algorithm has a computational complexity that is roughly linear in the size of the two input sequence strings to be matched. Other search and matching algorithms in bioinformatics, such as the Needleman-Wunsch algorithm, Smith-Waterman algorithm, Maximum-Likelihood matching, and Phylogenetic matching, have a higher-order computational complexity of the order of the product of the sizes of the two input sequence strings

[P01]. Since these algorithms have greater computing requirements than the BLAST algorithm, the corresponding data transfer overheads to the attached HPC platform will be a smaller fraction of the overall execution time, and these algorithms are therefore even better suited for our present framework. In addition, specific optimizations such as in-memory data structures and fine-grained parallelism are more easily implemented on the HPC platform than on the database server, and these optimizations can significantly reduce the overall execution time.

Figure 6 illustrates an example of an SQL query request for executing the SSEARCH algorithm [SSEARCH], which does a Smith-Waterman similarity match of a given input sequence against a specific target library of sequences stored in the database server. This query initiates the DB2 user-defined table function ssearch_call, whose parameter list includes the target sequence library, a descriptor string for the input sequence to be matched, the input sequence itself, and the number of top-ranked matches that are desired. The parallel user-defined program implementing the Smith-Waterman algorithm is executed on an active query partition on the remote parallel computer to which the target sequence is copied, and the results of the comparison returned after the remote execution are also shown.

Table 1 shows the results of the query performance using the prototype implementation of this framework. Three different protein and genome databases are used as the target tables for the queries as shown. The top 10, 50, 100 or 500 matching sequences ranked according to the z-score criterion in the SSEARCH implementation of the Smith-Waterman algorithm is returned as the result set to the query. The timings include (A) the target library data transfer times in Phase II for creating the active query partitions, (B) the overall query execution times in Phase III for queries similar to that shown in Figure 6, and (C) the computation processing times on the HPC nodes in Phase III excluding the query and result transport times. The results for the target library data transfer time in (A) which is the major performance overhead, is quite consistent with the database server and LAN hardware specifications. The query processing timing results in (B) shows the expected near-linear speed-up in this phase with increasing Blue Gene/L nodes. We note that subsequent queries for matching new input sequences on the same target library will not incur the data transfer overheads in (A). The timing results in (C) are for the computation processing alone, and considered in conjunction with the timings in (B), indicate that the overhead of returning the ranking results is generally small, but increases as the number of desired top matches is increased.

Our results show two main sources of performance degradation during the query execution phase. The first is the increase in the data volume of results that is being returned as the number of matches and the number of

processors is increased. The second is from the processing that is required in the results aggregator module to combine the ranking of the individual results from each of the compute node partitions. In specific platform implementations, these overheads can be further reduced, for example, by taking advantage of the fast internal data network of the Blue Gene/L platform for the results collection, and by using MPI collective communication calls to perform an on-the-fly ranking aggregation of the top matches. The customization and performance tuning of our framework for such specific database and HPC platform combinations is an important future practical consideration. Our current prototype implementation is however completely generic and can be easily ported to any other combination of database server and HPC platform.

V Summary

We have described a framework for deploying, scheduling and executing computationally-intensive parallel DBMS user-defined programs on an attached HPC platform. This framework allows the parallel performance of select applications on the database server to be scaled without having to upgrade the entire database hardware platform. Prospective applications are able to amortize the performance overhead for moving the required data and results between the database platform and the high-performance computing platform in several ways, for example,

- by exploiting the fine-grained parallelism and superior hardware performance on the parallel computing platform for speeding up compute-intensive calculations;
- by using in-memory data structures on the parallel computing platform to cache data sets between a sequence of time-lagged queries on the same data, so that these queries can be processed without further data transfer overheads;
- by replicating data within the parallel computing platform so that multiple independent queries on the same target data set can be simultaneously processed using independent parallel partitions of the high-performance computing platform.

An implementation of this framework was carried out using an IBM DB2 database server attached to an IBM Blue Gene/L HPC platform on a 1000 Mbit Ethernet LAN, and is being used for deploying prototype applications in bio-informatics and life sciences.

In closing, this framework provides a different approach towards integrating parallel HPC programs into database applications and data-oriented workflows, with possible applications in diverse areas such as multimedia

databases, life-sciences, financial computing, scientific computing, and general-purpose applications in search, ranking and aggregation. Our framework requires specific implementations to be structured in the form of scan-aggregation operations on the HPC platform, which is similar to other distributed data analytics frameworks such as MapReduce [DG04] and the PML toolkit [YAG+07]. Our extension of these ideas therefore supports the relational processing of the input datasets and output results for composing more interesting query workflows involving such distributed analytics on a HPC platform.

Acknowledgements

We thank Jim Sexton, Amanda Peters and Carlos Sosa of IBM for valuable discussions.

References

[BW05] J. Becla and D. L. Wong, "Lessons Learned From Managing a Petabyte," Second Biennal Conference On Innovative Data Systems Research, pp. 70-83, Asilomar CA (2005).

[C98] D. Chamberlin, "A Complete Guide to DB2 Universal Database, Morgan-Kaufman, San Francisco (1998).

[LHM91] K. C. Lee, T. M. Hickey and V. W. Mak, "VLSI Accelerators for Large Database Systems," IEEE Micro, vol. 11, no. 6, pp. 8-20 (1991).

[FM91] P. Faudemay and M. Mhiri, "An Associative Accelerator for Large Databases," IEEE Micro, vol. 11, no. 6, pp. 22-34 (1991).

[AS91] M. Abdelguerfi and A. K. Sood, "A Fine-Grain Architecture for Relational Database Aggregation Operations," IEEE Micro, vol. 11, no. 6, pp. 35-43 (1991).

[MLF91] V. W. Mak, K. C. Lee, and O. Frieder, "Exploiting Parallelism in Pattern Matching: An Information Retrieval Application," ACM Transactions on Information Systems, vol. 9, no. 1, pp. 52-74 (1991).

[EFG+01] E. Riedel, C. Faloutsos, G. A. Gibson and D. Nagle, "Active Disks for Large-Scale Data Processing," IEEE Computer, vol. 34, no. 6, pp. 68-74 (2001).

[MPI] http://www-unix.mcs.anl.gov/mpi

[HSK+01] L. M. Haas, P. M. Schwarz, P. Kodali, E. Kotler, J. E. Rice, and W. C. Swope," DiscoveryLink: A System for Integrated Accesss to Life Sciences Data Services," IBM Systems Journal, Vol. 40, no. 2, pp. 489-511 (2001).

[SCD+05] S. M. Stephens, J. Y. Chen, M. G. Davidson, S. Thomas and B. M. Trute, "Oracle Database 10g: a platform for BLAST search and Regular Expression pattern matching in life sciences," Nucleic Acids Research, Vol. 33, Database issue, pp. 675-679 (2005).

[ED04] B. Eckman and D. Del Prete, "Efficient Access to BLAST Using IBM DB2 Information Integrator," IBM Healthcare and Life Science Publication (2004).

[DCF03] A. E. Darling, L. Carey, W. Feng,"The Design, Implementation and Evaluation of mpiBLAST," Proceedings of the 2003 Clusterworld conference, San Jose CA, (2003).

[LMC+05] H. Lin, X. Ma, P. Chandramohan, A. Geist and N. Samatova, "Efficient data access for parallel blast," Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium, pp. 72-74, Denver CO, (2005).

[BPC+99] R. C. Braun, K.T. Pedretti, T.L. Casavant, T.E. Scheetz, C.L. Birkett, and C.A. Roberts, "Three Complememtary Approaches to Parallelization of Local BLAST Service on Workstation Clusters," Proceedings of the 5th International Conference on Parallel Computing Technologies (PACT), Lecture Notes in Computer Science (LNCS), vol.1662, pp. 271-282 (1999).

[RLM+05] H. Rangwala, E. Lantz, R. Musselman, K. Pinnow, B. Smith and B. Wallenfelt, "Massively Parallel BLAST for the Blue Gene/L," High Availability and Performance Computing Workshop, Santa Fe NM (2005).

[P00] R. Pearson, "Protein Sequence comparison and Protein evolution," Tutorial of Third International Conference on Intelligent Systems in Molecular Biology, La Jolla CA (2000).

[EB06] B. A. Eckman and P. G. Brown, "Graph data management for molecular and cell biology,", IBM Journal of Research and Development, vol. 50, no. 6, pp. 545-560 (2006).

[SW81] T. F. Smith and M. S. Waterman, "Comparison of Bio-sequences," Advances in Applied Mathematics vol. 2, pp. 482-489 (1981).

[P95] W. R. Pearson, "Comparison of methods for searching protein sequence databases," Protein Science, vol. 4, no. 6, pp. 1145-1160 (1995).

[DB2] IBM DB2 Version 9.1, http://www.ibm.com/software/data/db2.

[BG] IBM Blue Gene, http://www.research.ibm.com/bluegene.

[CMS+04] Y. Chen, J. Mak, C. Skawratananond and T-H. K. Tzeng, "Scalability Comparison of Bioinformatics for Applications on AIX and Linux on IBM e-server pSeries 690," IBM Redbook http://www.redbooks.ibm.com/abstracts/redp3803.html, (2004).

[SSEARCH] SSEARCH documentation, http://helix.nih.gov/docs/gcg/ssearch.html.

[DG04] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," Proceedinsg of Operating System Design and Implementation, pp. 137-150, San Francisco CA (2004).

[YAG+07] E. Yom-Tov, U. Aharoni, A. Ghoting, E. Pednault, D. Pelleg, H. Toledano and R. Natarajan "An Introduction to the IBM Parallel Mining Toolkit", http://www.ibm.com/developerworks/grid/library/gr-ipmlt/index.html.

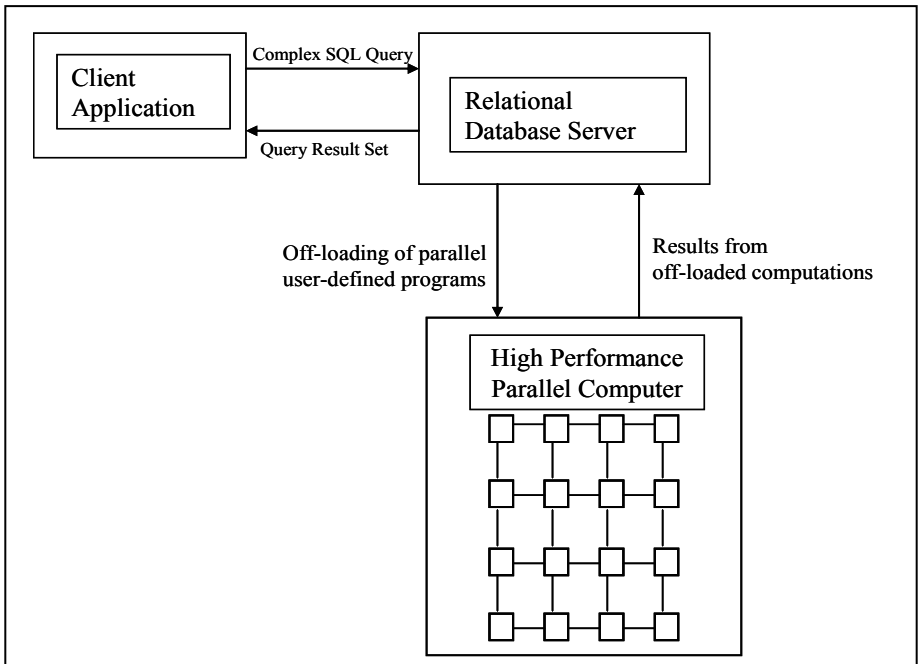

Figure 1: Schematic of HPC Accelerator for Database Analytics

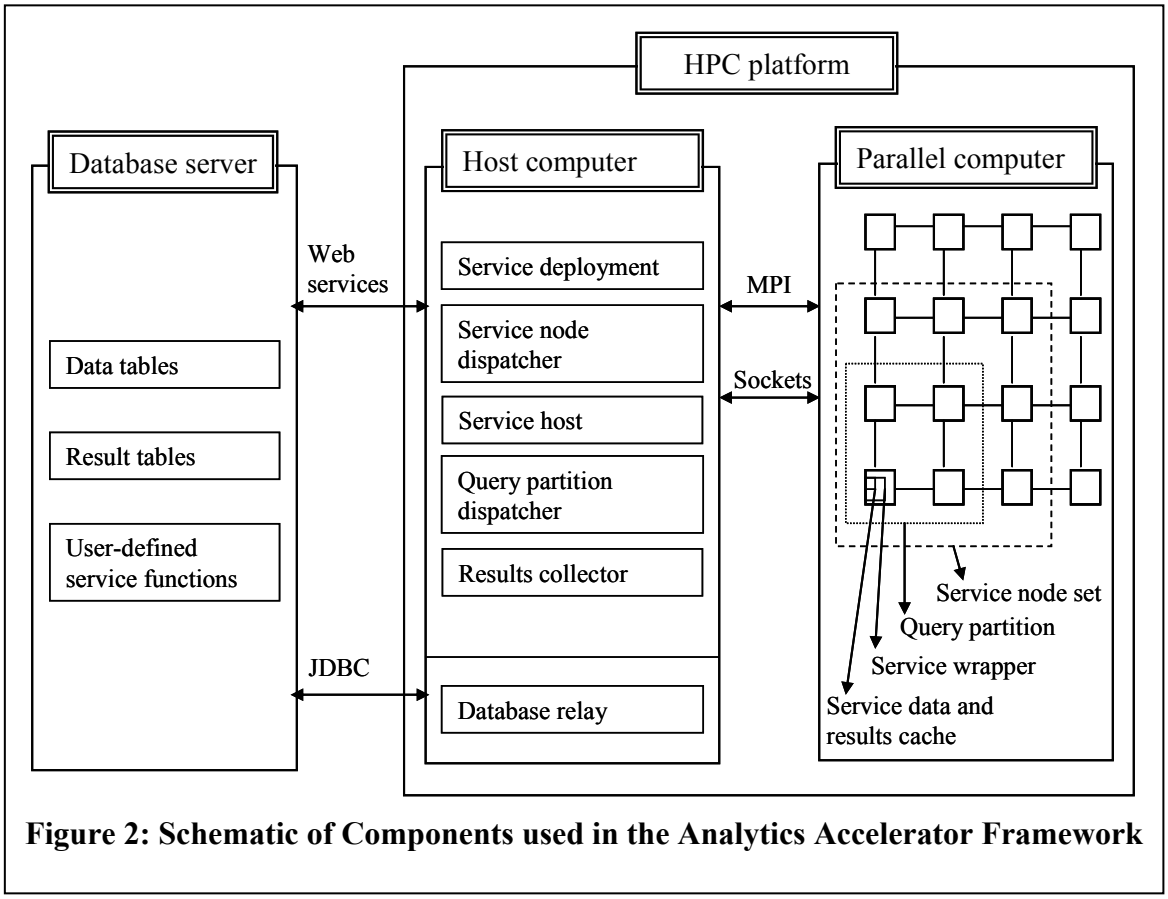

Figure 2: Schematic of Components used in the Analytics Accelerator Framework

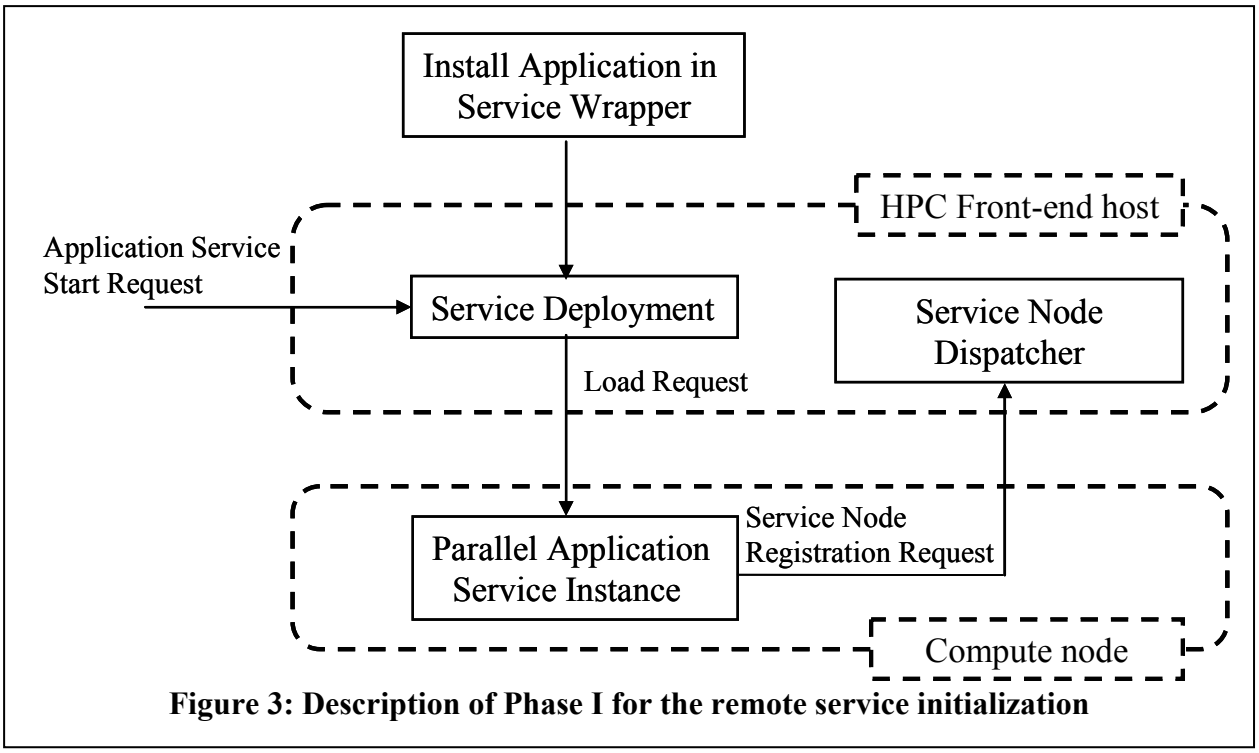

Figure 3: Description of Phase I for the remote service initialization

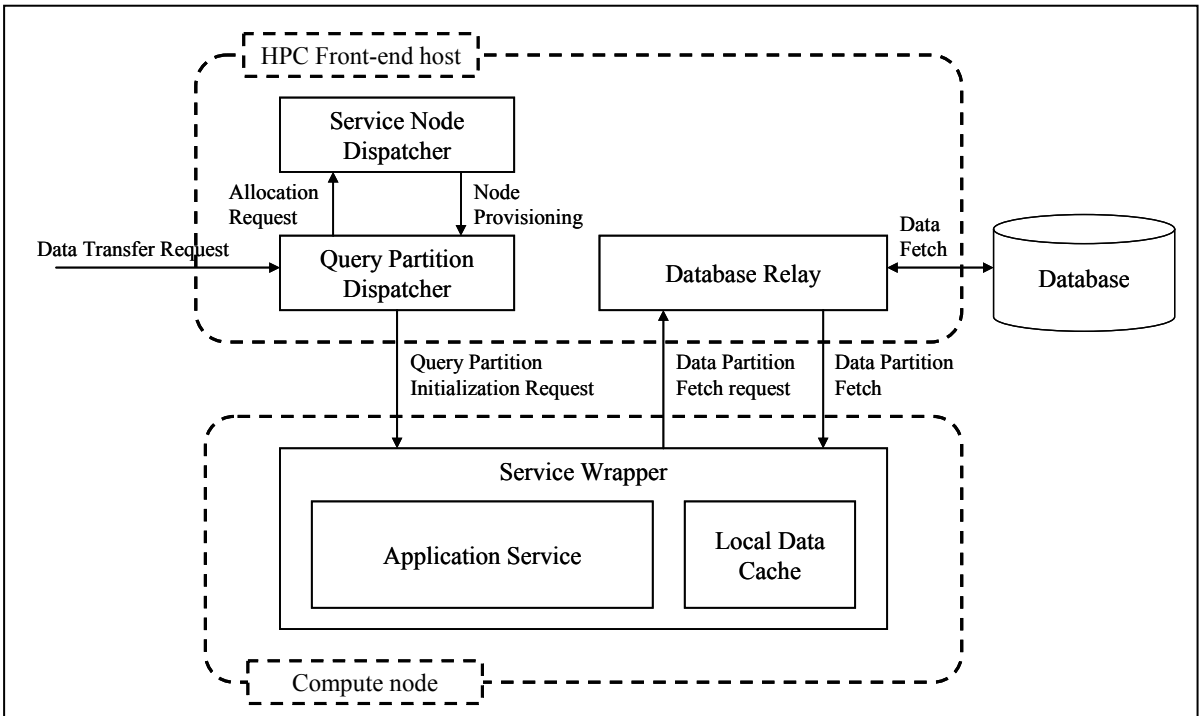

Figure 4: Description of Phase II involving the creation of Active Query Partitions

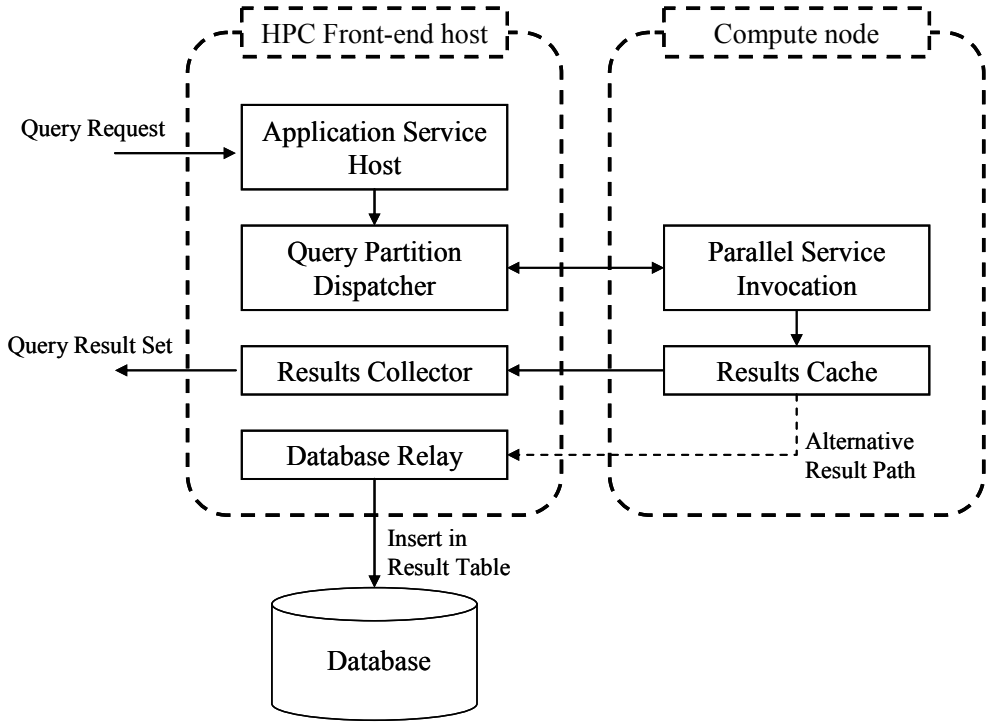

Figure 5: Description of Query Execution on a remote Active Query Partition

```
select * from table(ssearch_call('drosoph', 'query sequence',
'MPMILGYWNVRGLTHPIRMLLEYTDSSYDEKRYTMGDAPDFDRSQWLNEKFKLGLDFPNLPYLI',
6)) as A
```

Description of Arguments to ssearch_call in Query:

'drosoph' is the library/partition name against which the matching query is executed
'*query sequence'* is the name/description of the query sequence, this parameter may be deprecated.
'*MP .....'* is the query sequence for which the match is desired.
'6' is a request to show the top 6 hits in the result set of the query.

This query returns the result table below, where ID is a database internal integer id for sequences.

ID	SW	E	Z	BIT
1280	0	+8.22683627922823E-001	+1.06285301282655E+002	+2.72131589312291E+001
1071	0	+1.08072106272098E+000	+1.04158194576657E+002	+2.68195743620312E+001
1191	0	+1.41969279048097E+000	+1.02031087870658E+002	+2.64259897928334E+001
296	0	+3.21837510633985E+000	+9.56497677526638E+001	+2.52452360852397E+001
927	0	+5.55390162202516E+000	+9.13955543406675E+001	+2.44580669468439E+001
127	0	+5.55390162202516E+000	+9.13955543406675E+001	+2.44580669468439E+001

Figure 6: Description of an example SQL query for invoking the SSEARCH sequence matching algorithm and the result set after execution

nodes	A. Data transfer time from DB to HPC	B. Total Query Processing Time				C. Sequence scan time on HPC	# sequences per node
		top 10	top 50	top 100	top 500		
swissprot: 230k sequences							
1	116.2	511.8	511.8	511.9	512.4	511.4	230000
4	44.0	134.0	134.0	134.0	134.2	133.5	57500
8	37.8	68.3	68.3	68.3	68.9	67.8	28750
16	37.6	36.5	36.6	36.7	37.8	36.1	14375
32	37.9	19.1	19.4	19.5	22.1	18.7	7188
64	38.5	10.7	11.3	11.8	16.8	10.3	3594
128	44.0	5.7	6.8	8.9	10.6	5.2	1797
sts: 930K sequences							
4	63.1	680.6	680.6	680.6	680.9	680.0	232500
8	71.6	383.8	383.9	384.0	384.3	382.5	116250
16	84.5	193.8	193.8	193.9	195.2	193.3	58125
32	71.5	98.7	99.0	99.3	101.5	98.1	29063
64	74.9	50.9	51.4	52.0	56.8	50.5	14531
128	81.7	27.7	28.6	29.9	39.6	26.4	7266
est_human: 7895K sequences							
32	803.7	812.0	812.0	812.2	814.7	811.1	246719
64	899.0	466.7	467.5	468.1	472.4	466.1	123359
128	1090.5	246.6	247.7	249.0	258.8	246.0	61680

Table 1. The elapsed time in seconds for the execution of a query of the form shown in Figure 6 on the prototype system for the analytics accelerator framework on IBM DB2/IBM Blue Gene/L configuration. The target table consists of three different DNA and protein data sets of varying sizes. The queries are written to return the top 10, 50, 100 and 500 matches according to the z-score ranking criterion from the SSEARCH implementation of the Smith-Waterman algorithm. Timings are shown for (A) Initial data transfer of the table data from DB2 to the Blue Gene/L nodes, (B) Overall Query execution times, and (C) Computation times on the Blue Gene/L nodes alone, that is excluding the time for query and results transport. The last column contains the number of database sequences assigned to each parallel node in the query partition.