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

Michael A. Kochte
Institut fuer Technische Informatik, Universitaet Stuttgart
Pfaffenwaldring 47, 70569 Stuttgart, Germany
kochte@iti.uni-stuttgart.de

Ramesh Natarajan
IBM T. 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. This framework is advantageous for many DBMS workloads in the following two aspects. First, the long-running user-defined programs can be speeded up by taking advantage of the greater hardware parallelism available 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 of long-running user-defined programs to the attached HPC platform. Our framework provides a new approach for integrating high-performance computing into the workflow of query-oriented, computationally-intensive applications.

Categories and Subject Descriptors
C.1.4 [**Distributed Architectures**]: Parallel Architectures – *distributed architectures.*

General Terms
Performance, Design.

Keywords
High-performance computing, database accelerators, parallel user-defined database programs.

1. 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, including images, documents, multimedia, raw event streams from scientific instruments, and unprocessed results of computational modeling and simulations [1]. The relevant database processing for these complex data types is typically more than just simple database archival or retrieval, and it includes the 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 and computer-aided design.

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, along with 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 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 benefits, the processing requirements for executing 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 DBMS servers.

Large-scale DBMS are typically hosted on shared-memory multiprocessors or on high-availability network-clustered computer platforms. On these 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 platforms rarely provide a programming interface for external applications or embedded user-defined programs to directly take advantage of this 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,

CF'08, May 5–7, 2008, Ischia, Italy.

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, 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 [2] provides a detailed discussion of these default restrictions for one specific commercial database).

In certain cases (depending on whether the "safe" serial semantics need to be preserved in the implicit parallel execution of the user-defined program) the default restrictions on the implicit parallel execution of user-defined programs can be explicitly overridden by the applications programmer. Even then, the overall degree of parallelism that can be used for executing these programs is often fixed or restricted by pre-configured parameters in the database platform (for example, these parameters include the maximum number of parallel threads on a shared memory platform or the maximum number of data partitions or processors in a distributed cluster platform), while the user-defined program may be capable of exploiting a much higher degree of parallelism. Furthermore, while these database configuration parameters can be nominally set to their maximum values supported by the underlying hardware platform, it is often the case that within this range of parameter values, each individual database application has its own optimal parallel granularity that is determined by a complex interplay of factors involving the level of inherent parallel coordination, synchronization and data movement in the application. It is therefore unlikely that there is a single global optimal setting for all the applications running 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 and data parallelism in most 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, nor do they provide 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.

In this paper we propose a configuration consisting of a DBMS and an attached HPC platform and the required software framework for deployment, scheduling and execution of compute-intensive parallel user-defined programs on this attached HPC platform.

The remainder of this paper is organized as follows. Section 2 considers previous work on speeding up query execution using parallel hardware-based accelerators for commercial databases, and motivates the need for the proposed configuration. Section 3 gives a full description of the software framework used for deploying, scheduling and executing compute-intensive queries on the attached HPC platform. Section 4 describes our experience with this configuration for applications in bio-informatics and life sciences. Section 5 contains our concluding remarks.

2. RELATED WORK AND COMPARISON

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, the use of database query workload profiling to identify the most time-consuming operations, and the use of custom VLSI hardware filters in the data path between the disk storage interface and the CPU for these specific operations is proposed in [3] (see also the similar ideas in [4] and [5]). A similar approach using custom hardware accelerators for string and pattern matching operations in text-oriented database applications is described in [6].

A more recent approach is "active-disk" technology [7], 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 database systems. For many database queries, the execution at the disk controller interface achieves a substantial pre-filtering and reduction in the data volume to be 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. 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 comparably small data transfer requirements, or for multiple queries on the same target data, the achievable computational performance gains on the HPC platform significantly offsets this data transfer overhead. Furthermore, the overall approach obviates the need for database application developers and users to be familiar with any specialized parallel programming and parallel execution expertise in order to schedule data movement and computational processing on the external HPC platform. In fact, 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. 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.

3. DESCRIPTION OF THE FRAMEWORK

Figure 1 gives a schematic overview 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.

Specifically, the compute-intensive parts of the query workload, such as embedded parallel user-defined programs, are scheduled and executed on the HPC platform. The results of the remote execution are then transmitted back to the database server for any further query processing before final incorporation into the eventual result set returned to the client application.

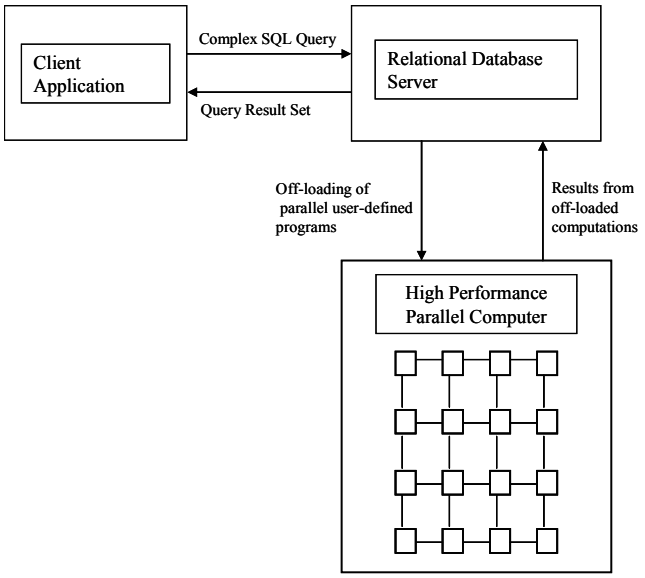

Figure 1. Schematic of HPC accelerator for Database analytics

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.

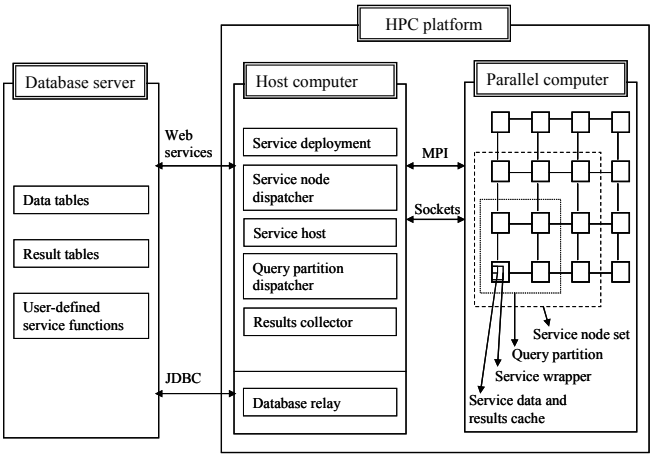

Figure 2. Schematic of Components used in the Database Accelerator Framework

On the parallel computer, 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 node application service using a simple programming interface to retrieve these table rows.

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

1) A service deployment module that is responsible for loading the application service on the required subset of the nodes of the HPC platform.

2) A service node dispatcher component that maintains the state of the individual nodes of the HPC platform.

3) 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).

4) 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 prespecified temporary tables on the database server.

5) 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 currently support any of 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 of the parallel computer. For example, MPI- or UNIX socket-based communication may be used with

the HPC platform, while the standard database access protocols like JDBC may be used with the database server.

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 parallel tasks and the return of the results to the database server.

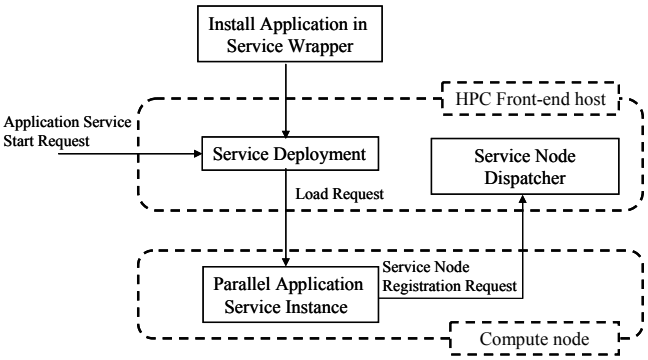

Figure 3. Description of Phase I for the remote service initialization

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 nodes 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 platform-specific compiler and 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 also usually platform-specific, such as MPIRUN loader used for MPI-based application binaries [8]). 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 for the subsequent query execution is transferred from the database server to a specified subset of the compute nodes that have been initialized in Phase I (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 III will be run. After this request from the database server, the query partition dispatcher first checks if an active query partition for the target table already exists and is ready to process new queries. If no such partition exists, the query partition dispatcher creates a new partition 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. This data transfer may be routed through the database relay component, i.e., the application service wrapper forwards the data transfer request to the database relay component, which executes the query to retrieve data from the database. The relay then forwards the result set to the compute nodes in the appropriate representation for storage in the local data cache.

The individual data partitions, which are typically stored in in-memory data caches allocated in the application service wrapper for best performance, can be accessed during subsequent query execution by the application service using a simple and standard programming interface at each node.

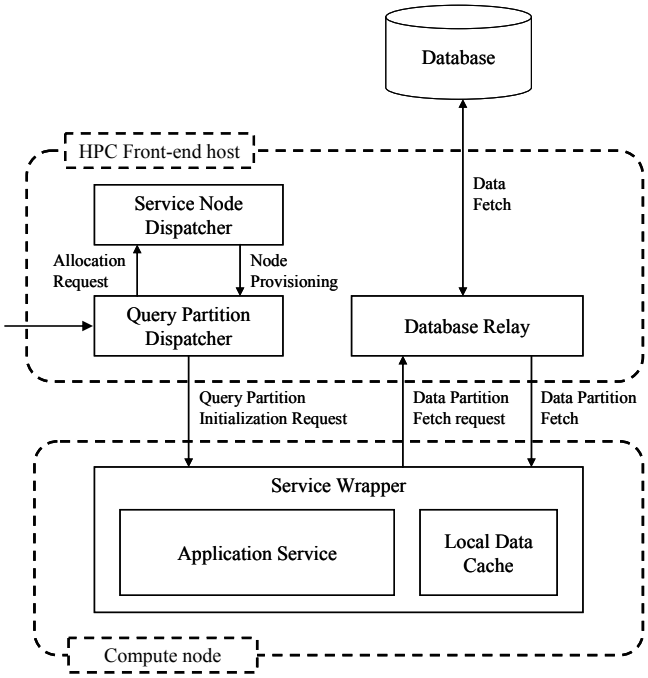

Figure 4. Description of Phase II involving the creation of Active Query Partitions

If a query partition for a particular target table already exists and has been initialized but is either reserved or otherwise unsuitable for processing a new query, then the query partition dispatcher may opt to clone this partition. This is done by allocating a set of application service nodes and having the relevant data copied over to it from the existing query partition. In this way the data transfer takes place within the HPC platform itself, 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. The query request is initiated by a user-defined function stub executed on the database server and 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.

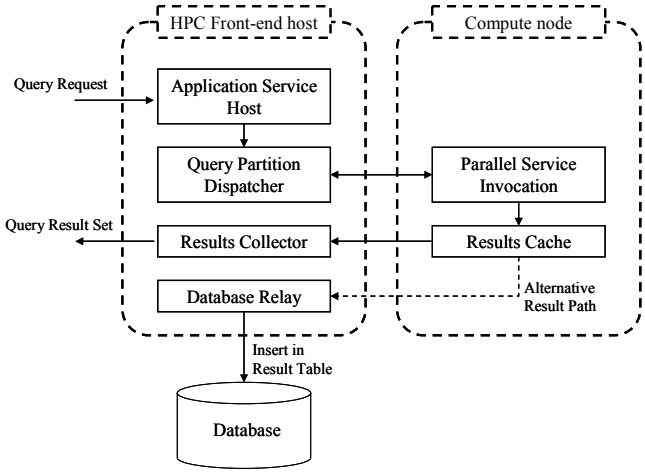

Figure 5. Description of Query Execution on a remote Active Query Partition

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 job completion.

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 either returned directly to the originating database function or is inserted in a results table in the database server using the database relay. Since the originating user-defined function 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, by SQL operations like ORDER BY or GROUP BY based on the result column values). Similarly, the results table can be joined to other database tables as required by the overall query workflow.

4. APPLICATION PERFORMANCE

A specific prototype implementation of this framework, along with the empirical performance results and deployment details for a bio-informatics application is described in this section.

4.1 DBMS and HPC platform setup

The commercial database server platform used in our application enablement experiments is IBM DB2 Version 9.1 [9] running on a dual-processor, Xeon 2.4 GHz CPU with 2GB of RAM storage and a 100 MBit/s 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 [10] with 1024 compute nodes. Each compute node comprises 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.

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 database commands relayed from the compute nodes on these socket connections, as well as for communicating the result sets or query status codes of these database commands back to the compute nodes initiating database query requests.

4.2 Bio-informatics application

An essential task in DNA or protein bio-informatics is the comparison of a new genome or protein sequence or sequence fragment against a subset of sequences from an existing sequence repository, in order to detect sequence similarities or homologies [11]. 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. 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. Furthermore, many of the steps in the information collation for sequence matching require data integration and aggregation, so that the overall workflow for this task is greatly facilitated if the sequence data and sequence metadata, as well as any results of the sequence matching algorithms, are all accessible from a standardized 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 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 [12], 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 database server. A set of embedded wrapper functions on this database server provides the necessary mapping of the input queries and query results that need to be exchanged between the database server and the back-end heterogeneous data sources.

The use of an SQL-based query interface to invoke various biological sequence matching algorithms has previously been considered in two different ways in the literature. In the first approach, these algorithms have been implemented as embedded user-defined programs, as described specifically for the BLAST algorithm in [13]. In the second approach, again specifically for the BLAST algorithm [14], 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 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, the two approaches discussed above have not been implemented in terms of a general-purpose, parallel computation framework for application deployment, as is the case for the application 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 [15]). 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 available 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 [16], with extensions for optimizing the parallel disk I/O performance in [17].

The second approach, termed query segmentation, considers 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 [18]. This approach is limited by the memory on the individual nodes, which may not be sufficient for storing the entire target sequence library. 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 having thousands of processors [19].

To our knowledge, none of the parallel implementations of BLAST, or the other sequence matching algorithms, has considered the issue of accessing these algorithms from a standardized SQL interface, in order to facilitate the easy integration of these algorithms into the processing requirements of a larger query workflow. Furthermore, it is difficult to directly incorporate any of these specific parallel implementations as embedded user-defined programs in a database extender, since these parallel implementations make extensive use of message-passing libraries and other parallel programming constructs that are generally not supported in the 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 computational complexity of the order of the product of the sizes of the two input sequence strings [11]. Since these algorithms have greater computing requirements than the BLAST algorithm, their corresponding data transfer overheads to the attached HPC platform are a smaller fraction of the overall execution time, and therefore, they exhibit a greater performance benefit from the proposed 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 have the potential to even further reduce the overall execution time.

```
select * from table(ssearch_call('drosoph', 'query sequence',
'MPMILGYWNVRGLTHPIRMLLEYTDSSYDEKRYTMGDAPDFDRSQWLNEKFKLGL...',
6)) as RESULT_TABLE
```

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 tag for the query sequence.
- '*MPMIL.....'* 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

Figure 6 illustrates an example of an SQL query request for executing the SSEARCH algorithm [20], which performs 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 in the result. 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 returned for this specific query after the remote execution are also shown in the figure.

Table 1 shows the results of the query performance for deploying this application on our prototype framework implementation. Three different protein and genome databases of varying sizes, swissprot [21] (230k sequences), sts [22] (930k sequences) and est_human [23] (7895k sequences) were used as the target tables for the queries. The table shows runtimes in seconds for query execution on the different databases using different numbers of compute nodes of the parallel computer. The cases studied include where the top 10, 100 or 500 matching sequences are returned as the result set to the query, and ranked according to the z-score criterion of the SSEARCH implementation of the Smith-Waterman algorithm. The timings include (A) the target library data transfer times in Phase II for creating the active query partitions, (B) the overall query processing times in

Phase III for queries similar to that shown in Figure 6, and (C) the computation processing times on the compute nodes in Phase III alone, with the query and result transport times excluded.

The data transfer times reported for copying the target library are consistent with the I/O capabilities of the database server and the LAN hardware specifications used in the prototype setup.

The query processing runtimes in column B of Table 1 show the expected near-linear speed-up in this phase with increasing number of compute nodes. We note that subsequent queries for the matching of new input sequences on the same target library will not incur the data transfer overheads from Phase II (column A). The near-linear speedups for node counts from 4 through 128 nodes reported here are also depicted graphically for the sts database in Figure 7. Similar speedup results are obtained with the swissprot and human_est databases.

For comparison, a similar query against the largest database est_human takes more than 5500 seconds when executed entirely on the database server itself (this measurement used a standalone database server comprising of a single-core AMD Athlon 64 3200+, 2GHz processor, 2GB RAM). The off-loading of this computation to an attached Blue Gene/L platform therefore reduces the query execution time considerably (by a factor of over 20 using 128 Blue Gene/L nodes in our framework, see Table 1), even when taking the communication overheads into consideration.

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.

Num. nodes	A. Data transfer	B. Total query processing time			C. Sequence scan time
		Top 10	Top 100	Top 500	
swissprot, 230k sequences					
4	44.0	134.0	134.0	134.2	133.5
8	37.8	68.3	68.3	68.9	67.8
16	37.6	36.5	36.7	37.8	36.1
32	37.9	19.1	19.5	22.1	18.7
64	38.5	10.7	11.8	16.8	10.3
128	44.0	5.7	8.9	10.6	5.2
sts, 930k sequences					
4	63.1	680.6	680.6	680.9	680.0
8	71.6	383.8	384.0	384.3	382.5
16	84.5	193.8	193.9	195.2	193.3
32	71.5	98.7	99.3	101.5	98.1
64	74.9	50.9	52.0	56.8	50.5
128	81.7	27.7	29.9	39.6	26.4
est_human, 7895k sequences					
32	803.7	812.0	812.2	814.7	811.1
64	899.0	466.7	468.1	472.4	466.1
128	1090.5	246.6	249.0	258.8	246.0

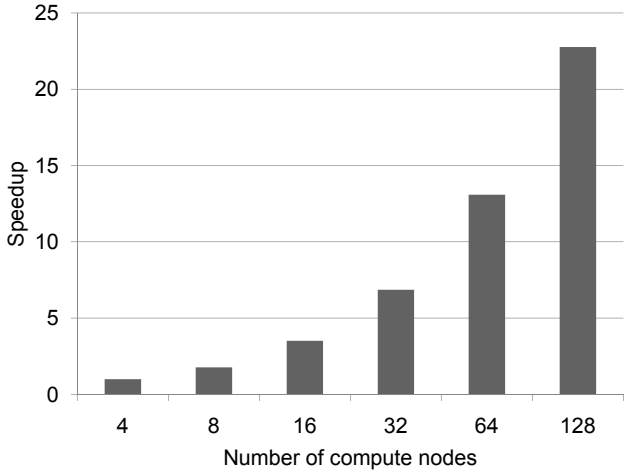

Figure 7. Speedup (relative to 4 nodes) for the entire query execution for the sts database from Table 1.

The timing results in column C in Table 1 denote the sequence comparison and processing alone excluding the overheads of the data transfer. Taken in conjunction with the timings in column B, those results 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 needs to be aggregated or returned as the number of matches and the number of processors is increased. The second is due to 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 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. The current prototype implementation is however fairly generic and relies on standard components and protocols that are supported on most other database servers and HPC platforms.

5. 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 specific parallelized user-defined programs on the database server to be ported and scaled without having to upgrade the entire database hardware platform. The performance overhead of moving the relevant query data and results between the database platform and the high-performance computing platform is amortized in our framework in several ways; for example, (i) by exploiting the fine-grained parallelism and superior hardware performance on the parallel computing platform for speeding up compute-intensive calculations; (ii) 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; (iii) by replicating data within the parallel computing platform so that multiple independent queries on the same target data set can be simultaneously processed using inde-

pendent parallel partitions of the high-performance computing platform.

A prototype of this framework, comprising of an IBM DB2 database server attached to an IBM Blue Gene/L HPC platform on a 100 MBit Ethernet LAN, has been used for deploying applications and performance benchmarking.

In closing, this framework provides a new approach towards integrating parallel HPC programs into database applications and data-oriented workflows, with potential applications in diverse areas such as multimedia databases, life-sciences, financial computing, scientific computing, and general-purpose applications in search, ranking and aggregation. The structure of the specific parallel user-defined programs in our framework is similar to the scan-aggregation operations used in distributed data analytics frameworks such as MapReduce [24] and the PML toolkit [25]. The framework, therefore, additionally supports the relational processing of the input data and output results, so that these parallel and distributed analytic constructs can be incorporated into complex query workflows.

6. ACKNOWLEDGMENTS

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

7. REFERENCES

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

[2] Chamberlin, D. 1998 A Complete Guide to DB2 Universal Database, Morgan-Kaufman, San Francisco.

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

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

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

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

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

[8] The message passing interface (MPI) standard, http://www-unix.mcs.anl.gov/mpi 2007.

[9] IBM DB2 Version 9.1, http://www.ibm.com/software/data/db2 2007.

[10] Gara, A. et al. 2005 Overview of the Blue Gene/L system architecture, IBM Journal of Research and Development, vol. 49, no. 2/3, pp. 195-212.

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

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

[13] Stephens, S. M., Chen, J. Y., Davidson, M. G., Thomas, S. and Trute, B. M. 2005 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.

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

[15] Chen, Y., Mak, J., Skawratananond, C. and Tzeng, T-H. K. 2004 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.

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

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

[18] Braun, R. C., Pedretti, K.T., Casavant, T.L., Scheetz, T.E, Birkett, C.L., and. Roberts, C.A. 1999 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.

[19] Rangwala, H., Lant, E., Musselman, R., Pinnow, K., Smith, B. and Wallenfelt, B. 2005 Massively Parallel BLAST for the Blue Gene/L, High Availability and Performance Computing Workshop, Santa Fe NM.

[20] SSEARCH documentation, http://helix.nih.gov/docs/gcg/ssearch.html 2007.

[21] Bairoch, A. 2000 Serendipity in bioinformatics, the tribulations of a Swiss bioinformatician through exciting times! Bioinformatics vol. 16, no. 1, pp. 48-64.

[22] sts database, ftp://ftp.ncbi.nih.gov/blast/db 2008.

[23] Boguski, M. S. 1995 The turning point in genome research, Trends in Biochemical Sciences, vol. 20, no. 8, pp. 295-296.

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

[25] Yom-Tov, E., Aharoni, U., Ghoting, A., Pednault, E., Pelleg, D., Toledano, H. and Natarajan, R. An Introduction to the IBM Parallel Mining Toolkit, http://www.ibm.com/developerworks/grid/library/gr-ipmlt/index.html 2007.