Distributed Query Processing on the Grid

Jim Smith[1], Anastasios Gounaris[2], Paul Watson[1], Norman W. Paton[2], Alvaro A.A. Fernandes[2], and Rizos Sakellariou[2]

[1] Department of Computing Science, University of Newcastle upon Tyne, Newcastle, NE1 7RU, UK
(Jim.Smith, Paul.Watson)@newcastle.ac.uk
[2] Department of Computer Science, University of Manchester, Manchester, M13 9PL, UK
(gounaris, norm, alvaro, rizos)@cs.man.ac.uk

Abstract. Distributed query processing (DQP) has been widely used in data intensive applications where data of relevance to users is stored at multiple locations. This paper argues: (i) that DQP can be important in the Grid, as a means of providing high-level, declarative languages for integrating data access and analysis; and (ii) that the Grid provides resource management facilities that are useful to developers of DQP systems. As well as discussing and illustrating how DQP technologies can be deployed within the Grid, the paper describes Polar*, a prototype implementation of a DQP system running over Globus. Polar* can handle complex data by adopting the ODMG object model and its query language OQL, which supports the invocation of user-defined operations. The Globus components are accessed through the MPICH-G interface rather than in a lower level way. A case study from bioinformatics is used throughout the paper, to show the benefits of the approach.

1 Introduction

To date, most work on data storage, access and transfer on the Grid has focused on files. We do not take issue with this – files are clearly central to many applications, and it is reasonable for Grid middleware developers to seek to put in place effective facilities for file management and archiving. However, database management systems provide many facilities that are recognised as being important to Grid environments, both for managing Grid metadata (e.g., [9]) and for supporting the storage and analysis of application data (e.g., [27]).

In any distributed environment there are inevitably multiple related data resources, which, for example, provide complementary or alternative capabilities. Where there is more than one database supported within a distributed environment, it is straightforward to envisage higher-level services that assist users in making use of several databases within a single application. For example, in bioinformatics, it is commonly the case that different kinds of data (e.g., DNA sequence, protein sequence, protein structure, transcriptome) are stored in different, specialist repositories, even though they are often inter-related in analyses.

There are perhaps two principal functionalities associated with distributed database access and use – distributed transaction management and distributed query processing (DQP) [20]. This paper is concerned with DQP on the Grid, and both: (i) discusses the role that DQP might play within the Grid; and (ii) describes a prototype infrastructure for supporting distributed query optimisation and evaluation within a Grid setting.

There is no universally accepted classification of DQP systems. However, with a view to categorising previous work, we note that DQP is found in several contexts: in distributed database systems, where an infrastructure supports the deliberate distribution of a database with some measure of central control [21]; in federated database systems, which allow multiple autonomous databases to be integrated for use within an application [19]; and in query-based middlewares, where a query language is used as the programming mechanism for expressing requests over multiple wrapped data sources (e.g., [16, 13, 7]). This paper is most closely related to the third category, in that we consider the use of DQP for integrating various Grid resources, including (but not exclusively) database systems.

Another important class of system in which queries run over data that is distributed over a number of physical resources is parallel databases. In a parallel database, the most common pattern is that data from a centrally controlled database is distributed over the nodes of a parallel machine. Parallel databases are now a mature technology, and experience shows that parallel query processing techniques are able to provide cost-effective scaleability for data-intensive applications (e.g., [8, 15, 17, 24]). This paper, as well as advocating DQP as a data integration mechanism for the Grid, also shows that techniques from parallel database systems can support data access and analysis for Grid applications.

The claims of this paper with respect to DQP and the Grid are as follows:

1. In providing integrated access to multiple data resources, DQP in and of itself is an important functionality for data intensive Grid applications.
2. The fact that certain database languages can integrate operation calls with data access and combination operations means that DQP can provide a mechanism for integrating data and computational Grid services.
3. Given (1) and (2), DQP can be seen to provide a generic, declarative, high-level language interface for the Grid.
4. By extending technologies from parallel databases, implicit parallelism can be provided within DQP environments on the Grid.

The paper makes concrete how these claims can be supported in practice by describing a prototype DQP system, Polar*, which runs over the Globus toolkit, and illustrates the prototype using an application from bioinformatics.

The remainder of this paper is structured as follows. Section 2 presents the principal components of a DQP system for the Grid, in particular indicating how this relates to other Grid services. Sections 3 and 4 describe, respectively, how queries are planned and evaluated within the architecture. Section 5 describes some measurements of an example query in a grid environment. Finally, Section 6 presents some conclusions and pointers to future work.

2 Architecture

The two key functions of any DQP system are query compilation and query execution. In the Polar* system described in this paper, both these components are based on those designed for the Polar project [25]. Polar is a parallel object database server that runs on a shared-nothing parallel machine. Polar* exploits Polar software components where possible, but as Polar* must provide DQP over data repositories distributed across a Grid, there are a number of key differences between query processing in the two systems. These include:

1. The data dictionary must describe remote data storage and analysis resources available on the Grid – queries act over a diverse collection of application stores and analysis programs.
2. The scheduler must take account of the computational facilities available on the Grid, along with their variable capabilities – queries are evaluated over a diverse collection of computational resources.
3. The data stores and analysis tools over which queries are expressed must be wrapped so that they look consistent to the query evaluator.
4. The query evaluator must use Grid authentication, resource allocation and communication protocols – Polar* runs over Globus, using MPICH-G [11].

A representative query over bioinformatics resources is used as a running example throughout the paper. The query accesses two databases: the Gene Ontology Database *GO* (`www.geneontology.org`) [2] stored in a MySQL (`www.mysql.com`) RDBMS; and *GIMS* [6], a genome database running on a Polar parallel object database server, based on the *Shore* system [4]. The query also calls a local installation of the BLAST sequence similarity program (`www.ncbi.nlm.nih.gov/BLAST/`) [1] which, given a protein sequence, returns a set of structs containing protein IDs and similarity scores. For that, the BLAST program accesses a third biological database, which in the example is *SWISS-PROT* [3], in a way transparent to the user. Both the BLAST program (and more specifically its *blastall* operation) and its associated database comprise a BLAST server. The query identifies proteins that are similar to human proteins with the GO term *8372*[1]:

```
select p.proteinId, Blast(p.sequence)
from   p in proteins, t in proteinTerms
where  t.termID='8372' and p.proteinId=t.proteinId
```

In the query, *proteins* is a class extent in *GIMS*, while *proteinTerms* is a table view in *GO*. Before submitting the query, a global database schema has been constructed to describe and combine the relevant views of the participating databases, along with the BLAST program. The ODL [5] definition of that schema is as follows:

[1] For what it is worth, the GO term GO:0008372 is used for "the annotation of gene products whose localization is not known or cannot be inferred." (`www.godatabase.org`)

```
forward class Protein;
forward class ProteinTerm;

struct Entry {
    string BLASTproteinID;
    long score;
};

static set<Entry> blast(in string protein);

class Protein //from GIMS
(extent proteins) {
    attribute string proteinID;
    attribute string sequence;
};

class ProteinTerm //from GO
(extent proteinTerms) {
    attribute string proteinID;
    attribute string term;
};
```

Therefore, as illustrated in Figure 1, the infrastructure initiates two sub-queries: one on GIMS, and the other on GO. The results of these sub-queries are then joined in a computation running on the Grid. Finally, each protein in the result is used as a parameter to the call to BLAST.

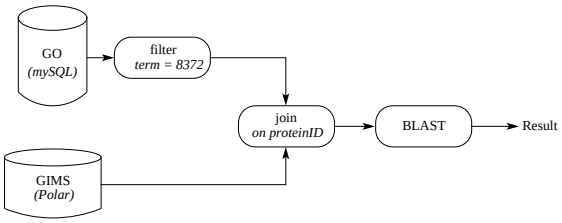

Fig. 1. Evaluating the example query.

One key opportunity created by the Grid is in the flexibility it offers on resource allocation decisions. In the example in Figure 1, machines need to be found to run both the join operator, and the operation call. If there is a danger that the join will be the bottleneck in the query, then it could be allocated to a system with large amounts of main memory so as to reduce IO costs associated with the management of intermediate results. Further, a parallel algorithm could be used to implement the join, and so a set of machines acquired on the Grid could each contribute to its execution. Similarly, the BLAST calls could be speeded-up by allocating a set of machines, each of which can run BLAST on a subset of the proteins.

The information needed to make these resource allocation decisions comes from two sources. Firstly, the query optimiser estimates the cost of executing each part of a query and so identifies performance-critical operations. Secondly, the Globus Grid infrastructure provides information on available resources. Once a mapping of operations to resources has been chosen, the single sign-on capabilities of the Grid Security Infrastructure simplify the task of gaining access to these resources. Figure 2 illustrates the high level architecture of the system. The metadata repository, which is needed to compile a query, is populated by both data from the database schema (e.g. types of attributes), and by information about the Grid nodes. Each Grid node has specific computational capacities and data stores. The coordinator node is a usual Grid node with the additional characteristic that it has the Polar* query compiler installed.

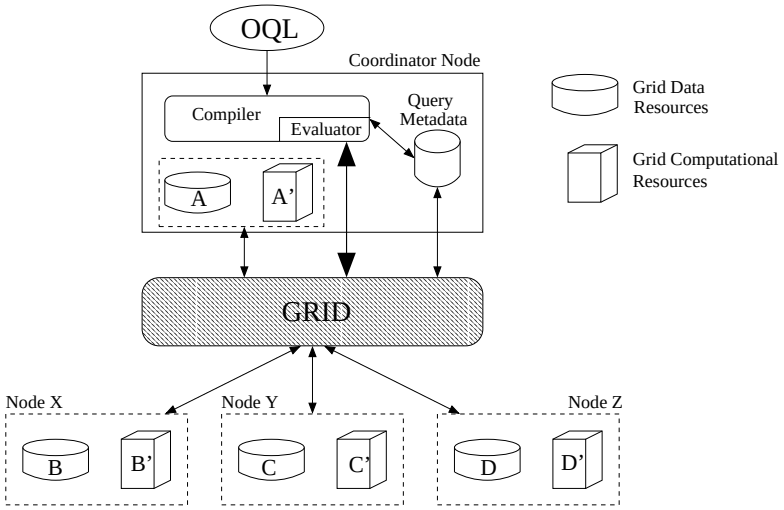

Fig. 2. The Polar* architecture.

3 Query Planning

Polar* adopts the model and query language of the ODMG object database standard [5]. As such, all resource wrappers must return data using structures that are consistent with the ODMG model. Queries are written using the ODMG standard query language, OQL. The reason for choosing the object data model rather than the relational one, is that it provides a richer model for source wrapping and for representing intermediate data.

The main components of the query compiler are shown in Figure 3. The Polar* optimiser has responsibility for generating an efficient execution plan for the declarative OQL query which may access data and operations stored on

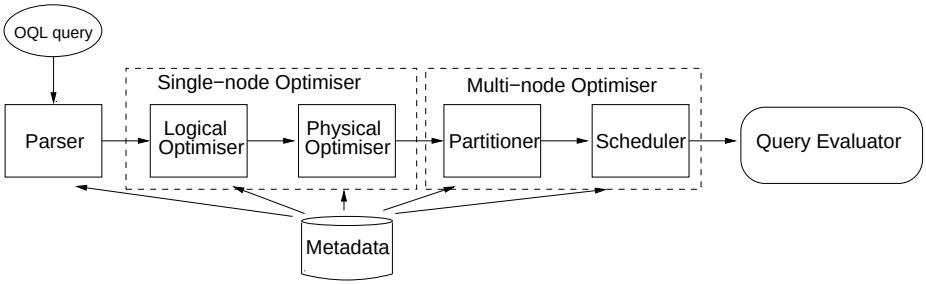

Fig. 3. The components of Polar* query compiler.

many nodes. To do this, it follows the two-step optimisation paradigm, which is popular for both parallel and distributed database systems [20]. In the first phase, the single node optimiser produces a query plan as if it was to run on one processor. In the second phase, the sequential query plan is divided into several partitions or subplans which are allocated machine resources by the scheduler.

3.1 Construction of the single-node plan

Figure 4(a) depicts a plan for the example query expressed in the logical algebra of Fegaras and Maier [10], which is the basis for query optimisation and evaluation in Polar*. The logical optimiser performs various transformations on the query, such as fusion of multiple selection operations and pushing *project*s (called *reduce* in [10] and in the figures) as close to *scan*s as possible.

The physical optimiser transforms the optimised logical expressions into physical plans by selecting algorithms that implement each of the operations in the logical plan (Figure 4(b)). For example, in the presence of indices, the optimiser prefers *index_scans* to *seq_scans*. Operation calls, like the call to BLAST, are encapsulated by the *operation_call* physical operator. For each OQL query, many physical plans are produced, and the physical optimiser ranks these according to a cost metric based on the predicted size of the intermediate results produced during the query execution. Changing the order of the operators results in plans with different costs.

3.2 Construction of the multi-node plan

A single-node plan is transformed into a multi-node one by inserting parallelisation operators into the query plan, i.e., Polar* follows the operator model of parallelisation [14]. The *exchange* operator encapsulates flow control, data distribution and inter-process communication. The partitioner firstly identifies whether an operator requires its input data to be partitioned by a specific attribute when executed on multiple processors (for example, so that the potentially matching tuples from the operands of a join can be compared). These

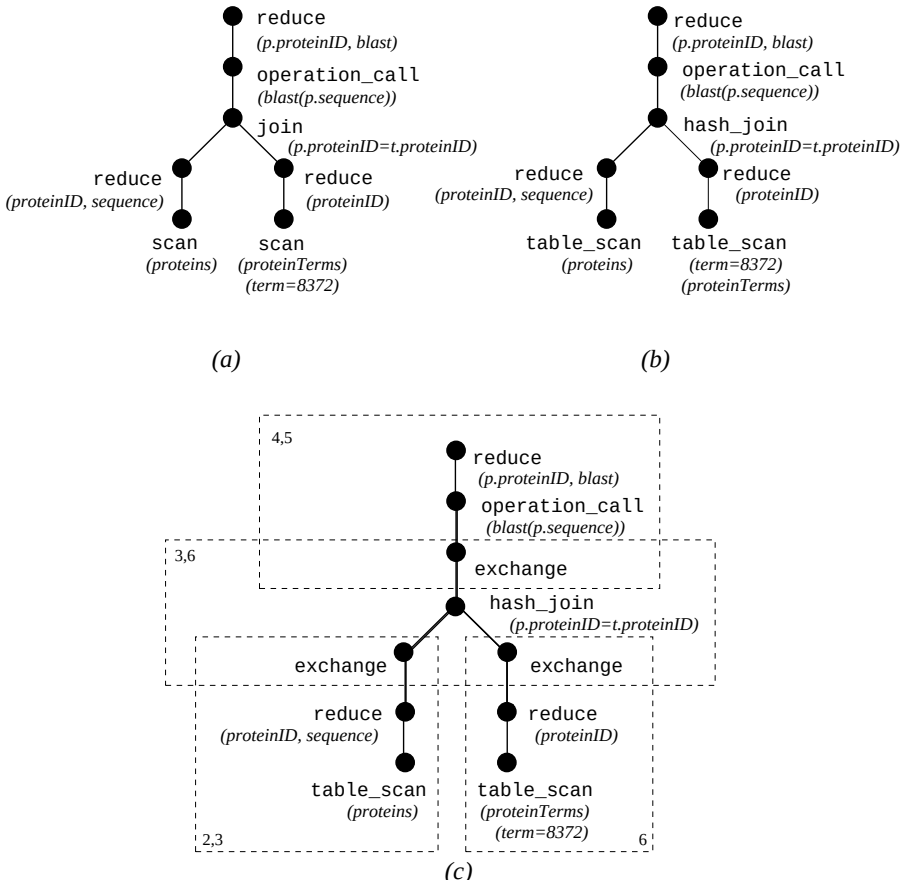

Fig. 4. Example query: (a) single-node logical plan, (b) single-node physical plan, (c) multi-node physical plan.

operators are called *attribute sensitive* operators [18], and [23] presents the classification of the parallel operators as attribute sensitive or attribute insensitive. Secondly, the partitioner checks whether data repartitioning is required, i.e., whether data needs to be exchanged among the processors, for example for joining or for submitting to an *operation_call* on a specific machine. If the children of an attribute sensitive operator in a query plan are partitioned by an attribute other than its partitioning attribute, or there is no data partitioning defined, then data repartitioning needs to take place. Another case for data repartitioning is when not all the candidate nodes for evaluating parts of the query can evaluate a physical operator, which is the usual case for *operation_calls*.

The physical algebra extended with *exchange* constitutes the parallel algebra used by Polar*. The *exchange*s are placed immediately below the operators that require the data to be repartitioned. For each *exchange* operator, a data

distribution policy needs to be defined. Currently, the policies Polar* supports include *round_robin*, *hash_distribution* and *range_partitioning*. The last two policies provide support for non-uniform data distribution among instances of the same physical operator, which is desirable for heterogeneous environments. A multi-node query plan is shown in Figure 4(c), where the *exchange*s partition the initial plan into many subplans (delimited by dashed lines in the figure). In the example query, there is one attribute sensitive operator, the *hash_join*, which needs to receive tuples partitioned by the *proteinID* attribute. Since the children of the *hash_join* are not partitioned in this way, two *exchanges* are inserted at the top of each subplan rooted by the join. *Operation_call* is attribute insensitive, but since it can be called only from specific nodes, an *exchange* is inserted as its child. The optimiser checks if the *exchanges* transmit redundant data. A *reduce* operator ensures that only data that are required by operators higher in the query plan are transferred through the network. If there are no such *reduces* placed by the single-node optimiser, the parallel optimiser inserts them.

The final phase of query optimisation is to allocate machine resources to each of the subplans derived from the partitioner, a task carried out by the scheduler in Figure 3 using an algorithm based on that of Rahm and Marek [22]. For running the example query, suppose that six machines are available, and that three of the machines host databases (numbers *2* and *3* for the GIMS database, and *6* for the GO database). The *table_scan* operators are placed on these machines in order to save communication cost. For the *hash_join*, the scheduler tries to ensure that the relation used to construct the hash table can fit into main memory, for example, by allocating more nodes to the join until predicted memory requirements are satisfied or all available memory is exhausted. In the example, nodes *3* and *6* are allocated to run the *hash_join*. As some of the data is already on these nodes, this helps to reduce the total network traffic.

The data dictionary records which nodes support BLAST, and thus the scheduler is able to place the *operation_call* for BLAST on suitable nodes (*4* and *5* in Figure 4(c)). The cost of a call to BLAST is much higher than the cost to send a tuple to another node over the network. In any case, the whole set of the results of the *hash_join* needs to be moved from the nodes *3* and *6*. For these two reasons, the optimiser has chosen the maximal degree of parallelism for the BLAST operation. The scheduler uses a heuristic that may choose not to use an available evaluator if the reduction in computation time would be less than the increase in the time required to transfer data (e.g., it has decided not to use machine *1* in the example).

4 Query Evaluation

4.1 Evaluating the Parallel Algebra

The Polar* evaluator uses the *iterator* model of Graefe [15], which is widely seen as the model of choice for parallel query processing [20]. In this model, each operator in the physical algebra implements an interface comprising three operations: *open()*, *next()* and *close()*. These operations form the glue between

```
class HashJoin: public Operator {
private:
    Tuple *t; HashTable h; Predicate predicate;
    Operator *left; set<Attribute> hash_atts_left;
    Operator *right; set<Attribute> hash_atts_right;
public:
    virtual void open() {
        left->open(); t = left->next();
        while (! t->is_eof()) {
            h.insert(t, hash_atts_left); t = left->next();
        }
        left->close(); right->open(); t = right->next();
    }
    virtual Tuple *next() {
        while (! t->is_eof()) {
            if (h.probe(t, hash_atts_right) && t->satisfies(predicate))
                return t;
            delete t; t = right->next();
        }
        return t;
    }
    virtual void close() {
        right->close(); h.clear();
    }
};
```

Fig. 5. Implementing *hash-join* as an iterator.

the operators of a query plan. An individual operator calls *open()* on each of its input operators to prompt them to begin generating their result collections. Successive calls to *next()* retrieve every tuple from that result collection. A special *eof* tuple marks the end of a result collection. After receiving an *eof* from an input, an operator calls *close()* on that input to prompt it to shut itself down.

We note that although the term *tuple* is used to describe the result of a call to *next()*, a tuple in this case is not a flat structure, but rather a recursive structure whose attributes can themselves be structured and/or collection valued.

To illustrate the iterator model, Figure 5 sketches an iterator-based implementation of a *hash_join* operator. *open()* retrieves the whole of the left hand operand of the join and builds a hash table, by hashing on the attributes for which equality is tested in the join condition. In *next()*, tuples are received from the right input collection and used to probe the hash table until a match is found and all predicates applying to the join result are satisfied, whereupon the tuple is returned as a result.

The iterator model can support a high degree of parallelism. Sequences of operators can support *pipeline parallelism*, i.e., when two operators in the same query plan are independent, they can execute concurrently. Furthermore, when

invocations of an operation on separate tuples in a collection are independent, the operation can be partitioned over multiple machines (i.e., *partitioned* or *intra-operator parallelism*).

Whereas data manipulation operators tend to run in a request-response mode, *exchange* differs in that, once *open()* has been called on it, the producers can run independently of the consumers. The tuples, which may be complex structures, are flattened for communication into buffers whose size can be configured. Underlying an instance of *exchange* is a collection of threads managing pools of such buffers so as to constrain flow to a lagging consumer, but to permit flow to a quicker consumer, within the constraints of the buffer pools. This policy is very conveniently implemented in MPI [26] where the tightly defined message completion semantics permit the actual feedback to be hidden within its layer. The *send* and *recv* calls issued at the MPI level in non-blocking synchronous mode, are managed by multiple user level threads encapsulated in the *exchange* operator. Each Polar* machine contains a global buffer pool, whose size is controlled via the global metadata. During evaluation of a given query, flow control is achieved by restricting each instance of the exchange operator to an allocation of buffers from the global pool; it can only send or receive buffers within the capacity of that allocation. This use of MPI has enabled the Polar *exchange* to port easily to MPICH-G [11], for use with Globus.

Since MPICH-G is layered above Globus, a parallel query can be run as a parallel MPI program over a collection of wide area distributed machines, oblivious of the difficulties inherent in such meta-computing, which are handled by the underlying Globus services. A parallel program running over such a Grid environment has to find suitable computational resources, achieve concurrent login, transfer of executables and other required files, and startup of processes on the separate resources. In the Polar* prototype, concurrent login to separate accounts is achieved through GSI (the Grid Security Infrastructure for enabling secure authentication and communication over an open network), and executable staging across wide area connections through GASS (Global Access to Secondary Storage, which is used for remotely accessing data), but all these features are accessed through the MPICH-G interface rather than in a lower level way.

Because the phyical operators in Polar*, like Polar, are implemented to manipulate generic tuple structures, the implementation of each operator can be present as a C++ object in each installation of the evaluator code. A subquery thus only needs to contain a reference to the specific operator IDs (i.e. *scan, hash_join, exchange*, etc) and the parameters for those instances. These parameters include the index of each input operator and the operator specific parameters, such as the arbitrartor policy and list of consumers for an *exchange*, the join attributes and predicate for a *hash_join*, etc. While input operators are contained in the same subquery, exchange consumers can be identified by a machine number (i.e. rank number) and index number pair. Thus the description of a query subplan is quite simple, and can be expressed concisely in textual format. By contrast, the dynamically loadable module required by an operation

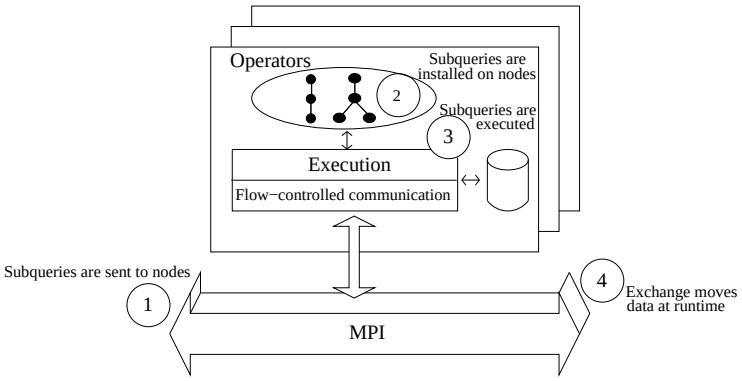

Fig. 6. The different phases involved in planting and executing a query on Grid nodes. Phases 1 and 2 are executed once for each query at the beginning, while phases 3 and 4 are executed concurrently many times.

call, containing the user defined code, is installed as part of the procedure of registering that operation call as available on the particular machine.

The collection of machines included in a Polar* distributed system is recorded in the global metadata of that system, located on the coordinator machine. Polar* generates files required by the underlying MPICH-G system from this metadata. Assuming the user has built the coordinator on his own machine, the OQL compiler/optimizer can be called as a local process by the *polar-query* program to generate a collection of subqueries, and a corresponding list of database nodes. The *polar-query* program can thereby construct a single communicator MPI program with one process, rank 0, running on the local machine and the remainder running on those machines that are required to participate in the query, and supply the subqueries to that MPI program via a text file.

Figure 6 shows the four main phases of query execution. The evaluator receives a set of subqueries from the query compiler as shown in Figure 3. First, the subqueries are sent to the Grid nodes through the network and second, they are installed on them via the MPI interface. The next phase involves the execution of the operators comprising the query plan as iterators. This operator execution may in turn lead to the moving of data between nodes, using a flow-controlled communications infrastructure which itself uses the MPI interface.

4.2 Accessing Computational and Data Resources During Query Evaluation

From the point of view of the evaluator, both database and analysis operations referred to within a query are external resources, which must be wrapped to enable consistent passing of parameters and returning of results during query evaluation.

To support access to external tools such as BLAST, Polar* implements, in *iterator* style, the *operation_call* operator in Figure 7. While concerns regarding

```
class PhysicalOperationCall: public PhysicalOperator {
private:
    string signature;              // operation to call
    string opslib_name;            // name of library
    list<Expression> expression;   // select params from tuple
    Predicate predicate;           // predicate on output tuple
    int key;                       // from cross ref in opslib
    vector<concrete_object*> arg;  // sized appropriately
    class operation_library_stub;  // functions etc in shared library
    operation_library_stub *stub;
    Tuple *t;
public:
    virtual void open() {
         input->open();      // input is pointer to operator
         stub = load_operation_library(opslib_name);
         key = stub->xref(signature);
    }
    virtual Tuple *next() {
         while (! t->eof()) {
            t = input->next();
            for (list<Expression>::iterator it = expression.begin();
                 it != expression.end(); it++)
                 arg[i+1] = t->evaluate(*it); // leave arg[0] for result
            stub->call_operation(key, arg);
            t->insert(arg[0]);
            if (t->evaluate(predicate))
                 return t;
            delete t;
         }
         return t;
    }
    virtual void close() {
         unload_operation_library(stub);
         input->close();
    }
}
```

Fig. 7. The iterator-based implementation of external *operation calls*.

the integrity of the evaluator lead to the isolation of such an operation call within a separate user context in some databases, the view taken so far in Polar* is that higher performance is achievable when a user can in good faith code an operation to be executed in the server's own context. Thus, such operations are linked into dynamically loadable modules, with stub code generated by the Polar* system to perform the translation of types between tuple format and the format of the programming language used to implement the operation. At runtime, *operation_call* loads the appropriate module in its *open()* call and unloads it in *close()*. Within *next()*, the operator passes attributes from the current tuple to the stub code which assembles required parameters, makes the call and translates the result, which may be collection valued and/or of structured type, back to tuple format before inserting it into the output tuple which is passed back as the result of the operator.

By making such an application available for use in a Polar* schema, the owner does not provide unrestricted, and thereby unmanageable, access on the internet. By contrast, since Polar* sits above Globus, a resource provider must have granted access rights to a user, in the form of a local login, which they can revoke. Subsequent accesses, while convenient to the user through the single sign-on support of Globus, are authenticated through the GSI.

A specific case that requires access to external operations is the provision of access to external, i.e. non Polar*, repositories.

For example, the runtime interface to a repository includes an *external_scan* operator, which exports an *iterator* style interface in common with other operators (Figure 8). However, below the level of *external_scan* the interface to an arbitrary external repository requires special coding. When an external collection is defined in the schema, the system generates the definition of a class, which has the three operations of the *iterator* interface plus other operations to gather statistics and set attributes of the interface such as a query language and the syntax of results generated by the external repository.

The generated class is the template within which a user can implement an *iterator* style interface to the external repository. The mechanisms used are at the discretion of the user, but the pattern in which the operations of this system-defined class are called is defined by the Polar* system. Fixed attributes, such as the syntax of result tuples returned, are set at the time the completed access class is registered with the Polar* system. The *open()*, *next()* and *close()* operations are simply called by *external_scan* in the usual iterator-based style. However, the results returned by the *next()* operation are translated from the selected syntax into Polar* tuples.

In the running example, the *GO* database is implemented using MySQL, so access to it is through such a system-specified user-defined class of operations. A delegated sub-query such as the access to *proteinTerm* tuples is expressed in SQL, and the results are formatted within the *MySQLAccess* class in Object Interchange Format (OIF)[2]. The *next()* operation of the *external_scan* operator parses each OIF instance to construct its result tuple.

[2] OIF is a standard textual representation for ODMG objects.

```
class PhysicalExternalScan: public PhysicalOperator {
private:
    string opslib_name;            // name of library
    string query_text;             // to be passed to external DB
    enum RESULT_FORMAT;            // e.g. OIF, SQL
    RESULT_FORMAT fmt;
    ObjectType otype;              // given object type and input format
    ObjectWalker *walker;          // walker supports map to/from tuple
    string open_signature;
    string next_signature;
    string close_signature;
    Predicate predicate;           // predicate on output tuple
    int open_key, next_key, close_key;
    vector<concrete_object*> open_arg;   // only has string parameter
    vector<concrete_object*> next_arg;   // only has result
    vector<concrete_object*> close_arg;   // no parameters
    Tuple *t;
public:
    virtual void open() {
         stub = load_operation_library(opslib_name);
         open_key = stub->xref(open_signature);
         next_key = stub->xref(next_signature);
         close_key = stub->xref(close_signature);
         open_arg[1] = query_text;
         stub->call_operation(open_key, open_arg);
         walker = new ObjectWalker(otype, fmt);
    }
    virtual Tuple *next() {
         while (! t->eof()) {
            stub->call_operation(next_key, next_arg);
            t = walker->map(next_arg[0]);
            if (t->evaluate(predicate))
                 return t;
            delete t;
         }
         return t;
    }
    virtual void close() {
         stub->call_operation(close_key, close_arg);
    }
}
```

Fig. 8. The iterator-based implementation of *external scans*.

5 Case Study

The experiments described in this section illustrate the behaviour of the Polar* system when evaluating the example query introduced in Section 2. The query is run in various different configurations, and in cases where parameters of the query are varied, to indicate how these influence scheduling decisions. For this purpose, it is only query response time that is of concern here.

The case study has been conducted on the heterogeneous collection of machines described below.

– At Newcastle upon Tyne.

Machine	CPU (MHz)	Memory (Mbyte)	OS	Globus
ncl-pc1	500	128	RedHat Linux 8.0	GT 2.0
ncl-pc2	700	256	RedHat Linux 8.0	GT 2.2
ncl-pc3	2 × 800	256	RedHat Linux 7.3	GT 2.0

– At Manchester.

Machine	CPU (MHz)	Memory (Mbyte)	OS	Globus
man-pc1	N/A	N/A	N/A	no
man-pc2	1100	512	RedHat Linux 7.1	GT 2.0

Man-pc1 is dedicated as a *MySQL* server for a reference *GO* database, with no direct user access. The wrapper for this database is managed by an *external scan* operator located on the most local machine available, *man-pc2*, and accesses the GO data via a *MySQL* client. The machines in Newcastle are inter-connected via a 100Mbps network. The bandwidth for the connection between Manchester and Newcastle is constrained by a slow connection between Leeds and Newcastle of 34Mbps (`http://www.norman.net.uk/old/janet/index.html`).

The GIMS data (3 Mbytes in total) is located in a Polar parallel object database distributed over 2 volumes located on *ncl-pc3*. The subquery submitted to the GO database retrieves 31006 tuples, from which 2518 are joined with the GIMS tuples. 35 tuples in the resulting dataset satisfy the join criterion.

The cost of a call to the *BLAST* program is several orders higher than a join between two tuples. Consequently, the query cost is expected to be dominated by the *BLAST* calls, so this is where parallelisation is attempted. Under different circumstances, parallelising other operators as well (like *hash_joins*) has significant impact, as explained in Section 3. In this case study, all machines can support a BLAST server apart from *man-pc1*, and the SMP *ncl-pc3* can support two. The *SWISS-PROT* dataset referenced in the *BLAST* calls contains 111825 sequences and occupies a total of 70 Mbytes. In itself, then, this data can be accommodated in memory on any of the machines, so that the execution time of a *BLAST* call should be dominated by CPU processing. The CPU cost of

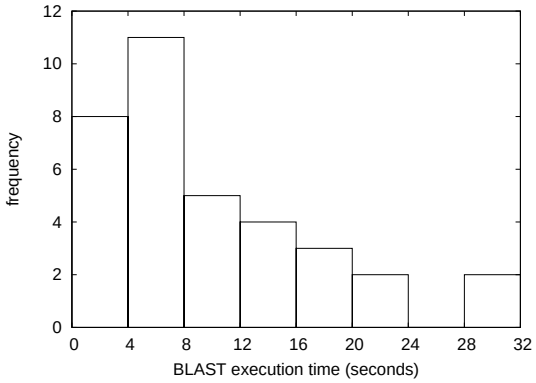

Fig. 9. The variation of the costs of the *BLAST* calls for the *ncl-pc1*.

a *BLAST* call for a given stored dataset varies significantly depending on the input sequence. For the 35 sequences resulting from the join, this variation is shown in Figure 9. Since the CPU speeds vary, there is a range of options for increasing the parallelism in the *operation call* operator.

Two policies for parallelising *operation calls* are considered here. In the first, *fastest-processor-first*, the parallelisation of the *operation call* operator is increased by allocating an extra instance to the fastest processor not currently allocated one. In the second case, *slowest-processor-first*, each extra instance is allocated to the slowest processor not currently allocated one. Such a strategy may be adopted if the user is charged for using Grid resources, and the cost of a resource depends on its power.

Orthogonally to the two processor allocation policies, the data distribution may or may not take into account the different machine characteristics. Static load balancing is improved by skewing the *round-robin* distribution of tuples to the multiple *operation_call* operators, according to the speed of the relevant machine.

Figure 10 shows measurements of the performance of the query in Figure 11. The four plots correspond to the four different combinations of processor allocation and data distribution policies. In the *fastest-processor-first*, *man-pc2* is the first node allocated to evaluate the *BLAST*, *ncl-pc3-1* is the second, *ncl-pc3-2* is the third, *ncl-pc2* is the fourth, and *ncl-pc1* is added last. In the *slowest-processor-first* the order is reversed. Each measurement is taken five times, and the average of the last three has been used. Also, the measurements are taken during quiet periods, at night and weekend, when activity on the network as well as the machines used is low.

The cost of the non-parallelised part of the query, i.e. the cost to retrieve the remote data and perform the join, is 17.6 seconds. This is relatively high for such volumes of data, and it is mainly due to the cost of sending the GIMS data from *ncl-pc3* to *man-pc2*. Some interesting observations can be made from

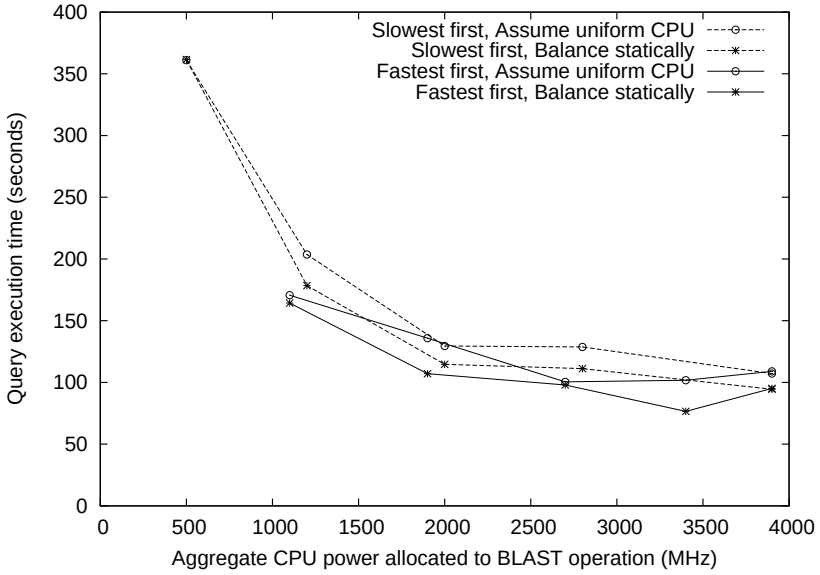

Fig. 10. Measured performance of the example query when the *BLAST* call is parallelised according to *fastest-processor-first* and *slowest-processor-first*.

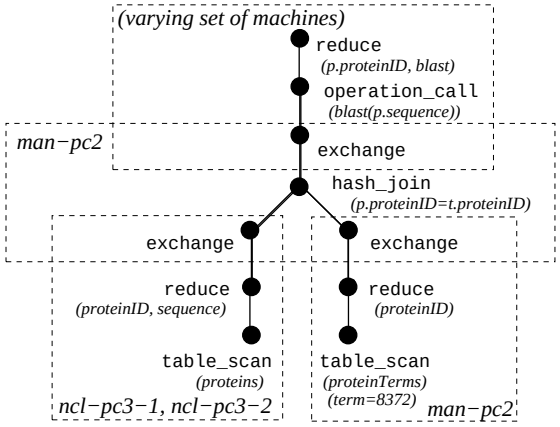

Fig. 11. The query plan used in the measurements. The degree of parallelism of the *operation call* is varying.

these measurements. First, the benefits of the resource sharing outweigh the Grid overheads and the communication cost significantly. In the case in which the fastest processor is chosen first, the first choice is *man-pc2*, which means that the *hash_join* and the *BLAST* call reside on the same processor. Further increases in the parallelism entail movement of data from Manchester to Newcastle. Even in that case, such increases yield better results. Also, for four processors, the aggregate CPU power is 3400 MHz. Assuming that the time of the parallelised part is inversely proportional to the total CPU power available, the performance of an ideal parallel computation would be $17.6+(164.3-17.6)(1100/3400) = 65.1$. The fastest measured execution time is 76.6, suggesting the query processor has done a reasonable job at this point. The most powerful machine, *man-pc2*, would evaluate all the *BLAST* calls with no other processes running in no less than 147 seconds. It is the responsibility of the scheduler component of the query compiler to choose the best combination of machines and data distribution policy.

A second observation is that even totally static load balancing policies over highly skewed resources can improve the performance of the system. In both cases of processor allocation, assuming uniform CPU capacities leads to worse performance than taking into consideration a very simple resource characteristic like the maximum CPU power. The difference between the Grid and other distributed systems is that the Grid provides the mechanisms to publish and advertise such characteristics, both static and dynamic.

These measurements barely scratch the surface of investigations into the performance of query processing in a Grid context, but they do hint at the complexity inherent in planning wide area distributed computations over heterogeneous resources, and demonstrate that even simple policies based on static measurements can be followed to arrive at sensible configurations. It is easy to see how the problem of planning this type of computation becomes more difficult in the presence of other usage, or if for instance the BLAST dataset were much larger, so that memory availability needs to be considered. In local and distributed database query processing, such considerations have driven the development of sophisticated optimization techniques [20]. The argument here is that such techniques can be beneficial in a Grid setting and should be extended to take account of the metrics generated by a grid infrastructure, including both static metrics such as CPU speed and memory size used here and dynamic metrics such as CPU and memory availability, in order to implement a manageable programming model for query style computations over a Grid.

6 Conclusions

One of the main hopes for the Grid is that it will encourage the publication of scientific and other data in a more open manner than is currently the case. If this occurs then it is likely that some of the greatest advances will be made by combining data from separate, distributed sources to produce new results. The data that applications wish to combine will have been created by different researchers or organisations that will often have made local, independent decisions

about both the best database paradigm and design for their data. The role of DQP in such a setting is to provide high-level, declarative facilities for describing requests over multiple data stores and analysis facilities.

The ease with which DQP allows such requests to be phrased has been illustrated through the example query in Section 2. This query is straightforward, but has illustrated how DQP can be used to provide optimisation and evaluation of declarative requests over resources on the Grid. We note that the alternative of writing such a request using lower-level programming models, such as MPICH-G or a COG kit [28] could be quite time consuming. We note also that as the complexity of a request increases, it becomes increasingly difficult for a programmer to make decisions as to the most efficient way to express a request. Developing efficient execution plans for such tasks using existing Grid programming environments would take a skilled developer a significant time. We believe that DQP can serve an important role in Grid environments by:

1. increasing the variety of people who can form requests over multiple Grid resources;
2. reducing development times for certain categories of Grid programming task; and
3. enabling typical requests to be evaluated efficiently as a result of system-supported query optimisation and support for implicit parallelism.

The Polar* prototype DQP system described in this paper is the result of our ongoing work on query processors for the Grid. The prototype has been implemented over Globus middleware using MPICH-G, and experiments have been conducted over bioinformatics databases and analysis tools at the authors' geographically remote sites. Future work will: (i) extend the range of physical operators in the algebra; (ii) increase the amount of system information used by the scheduler in query planning; (iii) explore the development of more powerful scheduling algorithms; and (iv) conduct performance evaluations over more and larger databases. We also work to evolve the Polar* system to be compliant with the emerging Open Grid Services Architecture [12], and to make use of standard service interfaces to databases [29] to reduce the cost of wrapper development.

Acknowledgements: This work was funded by the EPSRC Distributed Information Management Initiative and the UK e-Science Programme, whose support we are pleased to acknowledge. We are grateful for the contribution of Sandra Sampaio, whose work on Polar has been built upon extensively in Polar*. Also, we want to thank Dr. Mike Cornell and Dr. Phil Lord for helping with the bioinformatics.

References

1. S. F. Altschul, W. Gish, W. Miller, E.W Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.
2. M. Ashburner, C. Ball, J. Blake, D. Botstein, H. Butler, J. Cherry, A. Davis, Dolinski K, S. Dwight, J. Eppig, M. Harris, D. Hill, L. Issel-Tarver, A. Kasarskis,

S. Lewis, J. Matese, J. Richardson, M. Ringwald, G. Rubin, and G. Sherlock. Gene ontology: tool for the unification of biology. *Nature Genetics*, 25(1):25–29, 2000.
3. A. Bairoch and R. Apweiler. The swiss-prot protein sequence database and its supplement trembl in 2000. *Nucleic Acids Res.*, 28:45–48, 2000.
4. M. Carey, D.J. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, and M. Zwilling. Shoring up persistent applications. In R. Snodgrass and M. Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data, Minneapolis, Minnesota, May 24-27, 1994*, pages 383–394. ACM Press, 1994.
5. R.G.G. Cattell and D.K. Barry. *The Object Database Standard: ODMG 3.0*. Morgan Kaufmann, 2000.
6. M. Cornell, N.W. Paton, S. Wu, C.A. Goble, C.J. Miller, P. Kirby, K. Eilbeck, A. Brass, A. Hayes, and S.G. Oliver. GIMS – A Data Warehouse for Storage and Analysis of Genome Sequence and Functional Data. In *Proc. 2nd IEEE Symposium on Bioinformatics and Bioengineering (BIBE)*, pages 15–22. IEEE Press, 2001.
7. S.B. Davidson, J. Crabtree, B.P. Brunk, J. Schug, V. Tannen, G.C. Overton, and C.J. Stoeckert. K2/kleisli and gus: Experiments in integrated access to genomic data sources. *IBM Systems Journal*, 40(2):512–531, 2001.
8. D. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communication of ACM (CACM)*, 35(6):85–98, June 1992.
9. P. Dinda and B. Plale. A unified relational approach to grid information services. Technical Report GWD-GIS-012-1, Global Grid Forum, 2001.
10. L. Fegaras and D. Maier. Optimizing object queries using an effective calculus. *ACM Transactions on Database Systems*, 24(4):457–516, December 2000.
11. I Foster and N. T. Karonis. A Grid-Enabled MPI: Message Passing in Heterogeneous Distributed Computing Systems. In *Proc. Supercomputing (SC)*. IEEE Computer Society, 1998. Online at: http://www.supercomp.org/sc98/proceedings/.
12. I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *IEEE Computer*, 35:37–46, 2002.
13. H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajararnan, Y. Sagiv, J. Ullman, V. Vassalos, and J. Widom. The TSIMMIS Approach to Mediation: Data Models and Languages. *J. Intelligent Information Systems*, 8(2):117–132, 1997.
14. G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In *ACM SIGMOD*, pages 102–111, 1990.
15. G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, June 1993.
16. L. Haas, D. Kossmann, E.L. Wimmers, and J. Yang. Optimizing Queries Across Diverse Data Sources. In *Proc. VLDB*, pages 276–285. Morgan-Kaufmann, 1997.
17. W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 25(3):28–33, 1996.
18. W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB'95, Proceedings of 21th International Conference on Very Large Data Bases, September 11-15, 1995, Zurich, Switzerland*, pages 239–250. Morgan Kaufmann, 1995.
19. D. Hsiao. Tutorial on Federated Databases and Systems. *The VLDB Journal*, 1(1):127–179, 1992.
20. D. Kossmann. The State of the Art in Distributed Query Processing. *ACM Computing Surveys*, 32(4):422–469, 2000.
21. M.T. Ozsu and P. Valduriez, editors. *Principles of Distributed Database Systems (Second Edition)*. Prentice-Hall, 1999.

22. E. Rahm and R. Marek. Dynamic multi-resource load balancing in parallel database systems. In *Proc. 21st VLDB Conf.*, pages 395–406, 1995.
23. S. F. M. Sampaio, N. W. Paton, P. Watson, and J. Smith. A parallel algebra for object databases. In *Proceedings of the Tenth International Workshop on Database and Expert Systems Applications*, pages 56–60, Florence, Italy, September 1999. IEEE Computer Society.
24. S.F.M. Sampaio, J. Smith, N.W. Paton, and P. Watson. An Experimental Performance Evaluation of Join Algorithms for Parallel Object Databases. In R. Sakellariou et al., editors, *Proc. 7th Intl. Euro-Par Conference*, pages 280–291. Springer-Verlag, 2001.
25. J. Smith, S. F. M. Sampaio, P. Watson, and N. W. Paton. Polar: An architecture for a parallel ODMG compliant object database. In *Conference on Information and Knowledge Management (CIKM)*, pages 352–359. ACM press, 2000.
26. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI - The Complete Reference*. The MIT Press, Cambridge, Massachusetts, 1998. ISBN: 0-262-69215-5.
27. A. Szalay, P. Z. Kunszt, A. Thakar, J. Gray, and D. R. Slut. Designing and mining multi-terabyte astronomy archives: The sloan digital sky survey. In *Proc. ACM SIGMOD*, pages 451–462. ACM Press, 2000.
28. G. von Laszewski, I. Foster, J. Gawor, and P. Lane. A Java Commodity Grid Kit. *Concurrency and Computation: Practice and Experience*, 13(8-9):643–662, 2001.
29. P. Watson. Databases and the Grid. Technical Report CS-TR-755, University of Newcastle, 2001.