Experience on Performance Evaluation with OGSA-DQP

M.Nedim Alpdemir, **Anastasios Gounaris[1]**, Arijit Mukherjee[2], Desmond Fitzgerald, Norman W. Paton[1], Paul Watson[2], Rizos Sakellariou[1], Alvaro A.A. Fernandes[1], Jim Smith[2]

[1]School of Computer Science, University of Manchester, Oxford Road
Manchester M13 9PL, United Kingdom
[2]School of Computing Science, University of Newcastle upon Tyne,
Newcastle upon Tyne NE1 7RU, United Kingdom

Abstract

OGSA-DQP is an open source service-based Distributed Query Processor; as such, it supports the evaluation of queries over collections of potentially remote data access and analysis services. As it operates over several layers of service-oriented infrastructure, one particular need (both among the developer team and the relevant community) has been to investigate the impact of the infrastructure layers, understand performance issues, identify bottlenecks and improve the response times of queries where possible. This paper conveys experiences gained in doing so, by describing the experiments carried out and presenting the results obtained.

1. Introduction

OGSA-DQP [3] is an open source service-based Distributed Query Processor; as such, it supports the evaluation of queries over collections of potentially remote data access and analysis services. OGSA-DQP uses Grid Data Services (GDSs) provided by OGSA-DAI [1] to hide data source heterogeneities and ensure consistent access to data and metadata. The current version of OGSA-DQP, OGSA-DQP 2.0, uses Globus Toolkit 3.2 for grid service creation and management. Thus OGSA-DQP builds upon an OGSA-DAI distribution that is based on the OGSI infrastructure [8]. In addition, both GT3.2 and OGSA-DAI require a web service container (e.g. Axis) and a web server (such as Apache Tomcat) below them (see Figure 1). A forthcoming release of OGSA-DQP, due in fall of 2005, will support the WS-I and WRRF platforms as well.

OGSA-DQP provides two additional types of services, Grid Distributed Query Services (GDQSs) and Grid Query Evaluation Services (GQESs). The former are visible to end users through a GUI client, accept queries from them, construct and optimise the corresponding query plans and coordinate the query execution. GQESs implement the query engine, interact with other services (such as GDSs, ordinary Web Services and other instances of GQESs), and are responsible for the execution of the query plans created by GDQSs.

Conducting a performance analysis and performing changes with a view to optimising performance in the light of the analysis results is a challenging task. This paper reports experience in doing so for OGSA-DQP 2.0 over time, and thus it is important to clarify that this implies that some of the figures presented do not describe the behaviour of the system any more and refer to pre-optimisation stages. Complementarily to this work, an effort is being made to benchmark the OGSA-DAI distributions [2][6], so that the users can have a more complete view of the performance of the data access and integration middleware services that are currently available.

Two main classes of experiments have been carried out:

1. Experiments that aim to identify the impact of the underlying infrastructure.
2. Experiments that aim to understand the behaviour of the OGSA-DQP framework and identify bottlenecks in its internal architecture.

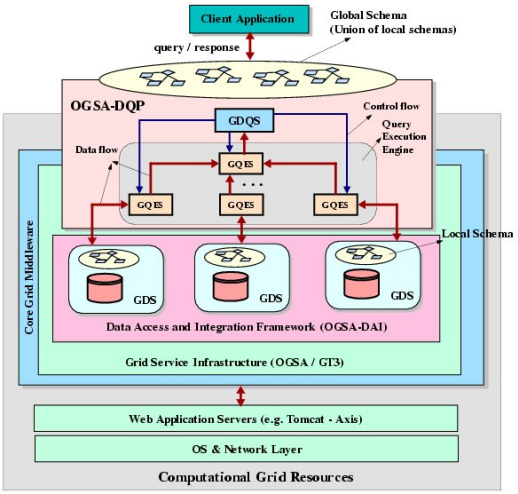

Figure 1. The architecture of OGSA-DQP.

The following sections briefly explain the approach taken in conducting those experiments and summarise the results in each case.

2. Exploring the impact of the service infrastructure

As mentioned before, OGSA-DQP, and more specifically the GQES service that it provides, relies on OGSA-DAI's GDSs to access underlying database management systems. GDSs provide a high level service interface for data access, delivery and transformation, but internally, they access databases via JDBC connections.

This first class of experiments measures the response time of a scan query (i.e. a query that returns the content of a complete table as its result) for:

1. a direct JDBC connection to a relational database,
2. a request sent via an OGSA-DAI 4.0 Grid Data Service (GDS) to the same relational database,
3. a request sent via OGSA-DQP 2.0 to the same relational database.

The results are then compared to see the impact of the layered service-based infrastructure. In addition, the overhead of XML encoding that causes the volume to increase significantly during data shipment has also been investigated. The experiments indicated that the overhead incurred by the service-based infrastructures, both in terms of increase in data volumes and in terms of increase in processing load, is significant. For example retrieving the same amount of data using a GDS took an order of magnitude more compared to a direct JDBC connection (i.e. 5.33 secs compared to 140 milisecs). It is therefore important to adopt policies to reduce the cost of using the infrastructure. For instance, in the case of asynchronous access to a GDS, it is essential to deliver data in buffers so that the message-sending overheads are shared across multiple tuples.

2.1 The Characteristics of the Data Sources and the Effect of XML

Two data sources are used, which are part of the demo application and databases included in the OGSA-DQP system:

1. *Protein_goterm* is a table with two columns, which contains proteins and their GeneOntology (www.geneontology.org) identifier. The table contains 16803 rows. Although the column sizes are variable, average row length is 24 Bytes, so the total size of the table is approximately 404188 bytes. A sample table with two rows of data is given below.

ORF [varchar (55)]	**GOTermIdentifier [varchar(32)]**
Q0010	GO:0000004
YAL037W	GO:0005554

Table 1. Sample protein_goterm data

2. *Protein_interaction* is a table with 6 columns: *ORF1*, *ORF2*, *baitProtein*, *interactionType*, *repeats*, *experimenter*. It contains experimental results involving proteins. The table contains 4716 rows with an average row length of 47 bytes, so the total size of the table is approximately 225688 bytes.

A Grid Data Service (GDS) delivers the query result as a WebRowSet [10], according to which each data tuple is wrapped inside XML tags. For instance each row in *protein_interaction* table (see Table 1) is represented in the form of an XML fragment of the following form:

```
<currentRow>
   <columnValue>YAL037W</columnValue>
   <columnValue>GO:0000004</columnValue>
</currentRow>
```

These tags add an extra 188 characters (of 2 bytes each) for each tuple in the *protein_interaction* table and 80 characters for each tuple in the *protein_goterm* table. As a result the amount of data on the wire grows significantly. To be more precise, for the *protein_interaction* table, there are 188*4716=886608 additional characters, plus the header section, which are transmitted over SOAP to the client. Considering that the original size of that table was ~226 KB, the total size becomes ~2 MB (assuming 2 bytes per character), resulting in an increase of more than eight times. Table 2 illustrates in more detail the impact of XML on data sizes of both tables used in the experiments.

Table name	**Original data size**	**XML overhead per row**	**Total XML overhead**	**Total Size**
protein_ goterm	404 KB	160 B	2.68 MB	~3 MB
protein_ interaction	226 KB	376 B	1.77 MB	~2 MB

Table 2.The impact of XML format on data sizes.

2.2 Experiments Description

During the experiments, seven main access methods were examined:

1. *Local JDBC access*. This indicates direct access to the data store, which is co-located with the JDBC client.
2. *Remote JDBC access*. This indicates that the data store is located on a separate machine, and is accessed via JDBC remotely.
3. *Local Synch-GDS*. This indicates that the data source is accessed via a GDS co-located with client, and that the GDS request is synchronous. In other words, the results are delivered to the client at once, as a single document (in XML WebRowSet format).
4. *Remote Synch-GDS*. This indicates that the data source is accessed via a GDS that is located on a different machine from the client, and that the GDS request is synchronous as in item 3 above.
5. *Local Asynch-GDS*. This indicates that the data source is accessed via a GDS co-located with the client, and that the GDS request is asynchronous with no block aggregation. In other words, the results are pulled by the client tuple by tuple, using the GDT (Grid Data Transport) port-type of GDS. Each tuple is wrapped in XML tags as discussed earlier.
6. *Remote Asynch-GDS.* This indicates that the data source is accessed via a GDS located at a different machine than the client machine, and that the GDS request is asynchronous as in item 5 above.
7. *OGSA-DQP Scan*. This indicates that the data store is accessed via OGSA-DQP. The aim is to compare this with the response time of the GDS. Note, however, that since leaf GQESs (which contain the SCAN operator) use asynchronous requests to stream data out of GDSs and since they are co-located with the GDSs, the comparison should be made against *Local Asynch-GDS*.

Three queries are used in the tests:

1. *Scan-1*, which is a full scan of the *protein_goterm* table: `select * from protein_goterm;`
2. *Scan-2*, which is a full scan of the *protein_interaction* table: `select * from protein_interaction;`
3. *Join*, which is an equi-join of the two tables:
```
select i.ORF2 from protein_goterm as p, protein_interaction as i where p.ORF=i.ORF;
```

The join query is significantly different in the case of OGSA-DQP because the two tables scanned are hosted by two separate machines, whereas in the case of GDSs they are hosted by the same machine and defined in the same database.

The data sources are hosted in mySQL databases, and duplicated on two different machines with similar computational characteristics. Each machine has an AMD Athlon 1.13 GHz processor and 512 MB RAM. Both machines are connected to the same 100Mps departmental network.

2.3 Experiments Results

The results show query execution times recorded for each data access mode. There is a bar chart for each of the three queries (Figures 2 – 4). Each query ran three times and the average is presented.

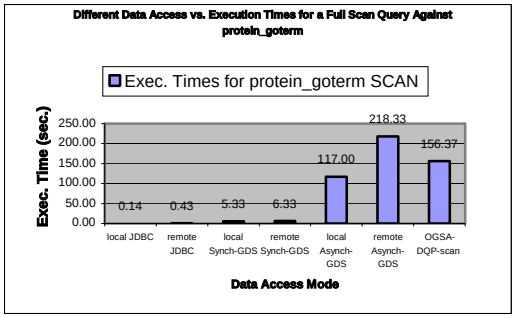

Figure 2. Execution times vs. different access modes for the first query Scan-1.

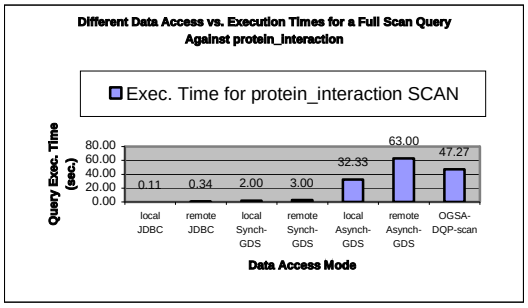

Figure 3. Execution times vs. different access modes for the second query Scan-2.

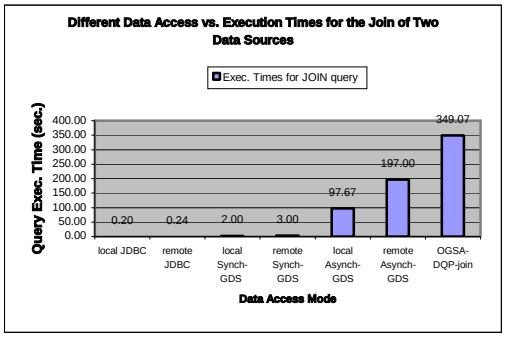

Figure 4. Execution times vs. different access modes for the third join query.

The figures above should be interpreted carefully, as the access modes examined refer to different layers of infrastructure (for example, GDS builds a service layer on top of simple JDBC access). Moreover, as already clearly documented, the performance of OGSA-DAI can vary significantly for different activities, delivery mechanisms, and result sizes [2][6].

In general, for non-small datasets, accessing databases through a GDS service synchronously, and retrieving the data tuple by tuple, incurs an overhead of an order of magnitude compared to simple JDBC access (from hundreds of miliseconds the response time becomes a few seconds, for the database sizes used in these experiments). Accessing GDSs asynchronously with no block aggregation incurs an additional overhead of an order of magnitude too (the response time is in tens of seconds).

Moreover, the overhead incurred by the GDS-GQES interaction is also significant. For *Scan-1*, the response time increases from 117 secs to 156.37 (33.6%), whereas for *Scan-2* the increase is from 32.33 secs to 47.27 (46.1%). Thus the average overhead is approximately 40%.

Intuitively, significant performance benefits are expected when tuples are retrieved by the GDS in blocks, which is a functionality that has been incorporated in OGSA-DAI. The next section investigates the impact of such blocks.

2.4 The Effect of Block Aggregation when Accessing the GDS.

This section investigates the effect of the block size (i.e. the number of tuples contained in a single block of data transport) when shipping data from the store. In this set of experiments, the ability to specify different block sizes when submitting a query request to a GDS is utilized.

Retrieving data from the GDS using the block aggregator activity improves the performance significantly. For this experiment we re-ran *Scan-1* (two iterations) with different block sizes. Table 3 and Figure 5 indicate the cost of accessing the *protein_goterm* database remotely with different block sizes. In summary, the observed behaviour for this category of experiments indicates a considerable improvement in response time with there is an increase in the block size, but only up to the point where the cost of constructing the blocks starts to overweigh the improvement gained by returning larger blocks of data per service invocation. This is due to the reduced number of service calls, which tend to be one of the dominant factors in the cost. Recalling from Figure 2 that the cost of accessing the same data source with single-tuple blocks (i.e. block size = 1) was 218 seconds, reducing this cost to 10.5 seconds with a block size of 120 or 130 is a significant improvement.

Scanning the *protein_intractions* table exhibits similar behaviour (see Figure 6). The only difference is that the optimal performance is achieved when the block size is slightly smaller: 110 tuples per block.

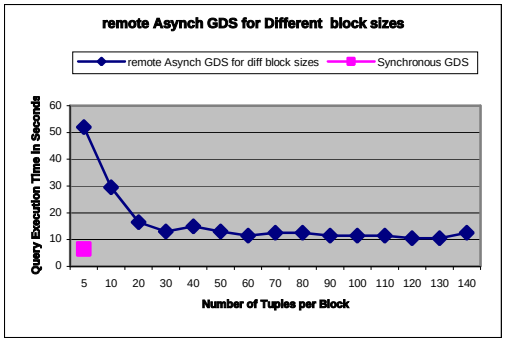

Figure 5. Cost of GDS access with different block sizes.

The reduction in the cost of accessing GDSs propagates improvements to the OGSA-DQP performance too. Motivated by the lessons learnt from these experiments, any access to the underlying databases in OGSA-DQP has been modified to utilise the block aggregation activity. With a block size of 100 the time to get the results of the *protein_goterms* scan via OGSA-DQP reduces from 156.37 seconds to 27.8 seconds. Although this is a significant improvement it is still more than twice the equivalent (i.e. with the same block size) GDS response time. Therefore it is worth analysing the stages internal to OGSA-DQP so that a clearer picture of the whole query processing life cycle can be drawn. The following section aims to do that.

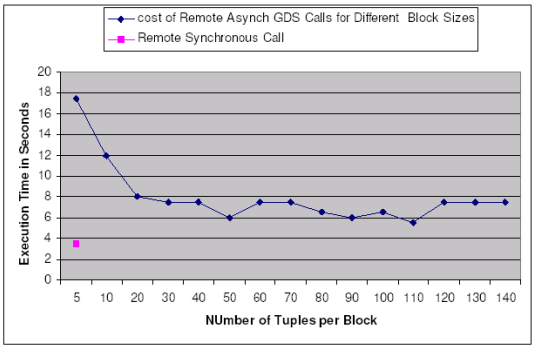

Figure 6. Cost of GDS access with different block sizes.

3. Understanding the behaviour of the internal architecture

OGSA-DQP evaluates a query by instantiating evaluator service instances on available servers, to yield a tree of evaluator services, the leaf nodes of which interact with GDSs to stream data from relational databases. One noteworthy feature of OGSA-DQP is its ability to parallelise the execution of computationally expensive operations across available resources and to perform pipelined execution. This class of experiments, therefore, aims to identify the bottlenecks in the internal architecture by which the query evaluation is achieved, and to assess the impact of parallelism. Two distinct categories of investigation are considered:

1. Investigating the most time consuming operations during the pipeline of processes involved in executing a query by profiling the OGSA-DQP code.

2. Investigating the effect of parallelising the computationally intensive operations across available computational resources as and when they become available.

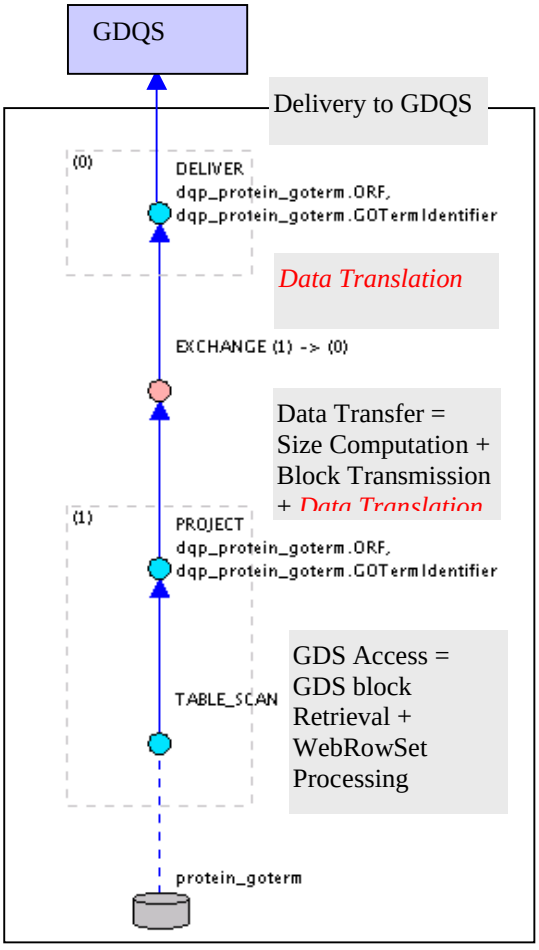

Figure 7. The query plan for the Scan-1 query. The shaded boxes indicate the costs at the corresponding part of the plan.

3.1 An analysis of the OGSA-DQP processing cycle

The analysis is performed for the simple, complete scan of the *protein_goterms database*. The query plan generated for this query is given in Figure 7. Data is retrieved from one machine (with identifier 1 in the figure), and is sent to another one (with identifier 0), which is co-located with the DQP GDQS. The plan is annotated at several places, with a number of costly operations identified as the key costs. Starting from the bottom where the TABLE_SCAN operator accesses the GDS, moving up to the root evaluator where the results are delivered to the GDQS those key costs are:

1. *GDS Access Cost*. This includes the cost of retrieving tuples in block from the GDS via the GDT port type and the processing cost of the returned XML blocks, which are in WebRowSet format. This cost is encapsulated in the cost of the TABLE_SCAN operator.
2. *Data Transfer Cost*. This is the cost of packaging and sending data from an evaluator lower in the query plan to a higher-level evaluator. It includes the block construction cost (because tuples are packaged into blocks before being sent to the receiving evaluator), the cost of calculating the size of a tuple, the cost of sending the block over the network and the cost of translating the blocks into tuples at the receiving evaluator. The last cost is included because the sending evaluator calls the receiving one synchronously, so it has to wait for the data translation to complete before the call returns. The data transfer cost is the cost of the producer thread of the EXCHANGE [5] operator in the query plan.
3. *Data Translation Cost*. This aims to measure the cost of unblocking the data packets at the receiving evaluator. Note that this is also included within the Data Transfer Cost. It related to the cost of the consumer thread of EXCHANGES.
4. *Delivery Cost*. This is the cost of sending data blocks from the root evaluator to the GDQS. Sending data blocks in this case is a service-to-service call over SOAP, from the root GQES to the GDQS.

These costs do not cover all the possible costs that contribute to the query response time, but other remaining costs, such as applying the PROJECT operator, adding and removing the tuples from the queues etc., are thought to be relatively negligible.

OGSA-DQP source code has been instrumented to measure each of the costs above. The results are as follows:

The total response time is 28.6 seconds. The GDS block retrieval takes 9.6 secs and the XML WebRowSet processing (to unpack and map the tuples to the internal format of DQP) costs 5.5 secs. Thus the GDS access cost is 15.1 secs.

Data is transferred from the machine that is local to the database to the machine that receives the results in 26.8 secs, from which 4.8 secs are spent for translation and 1sec for size computation. The latter, is a CPU-intensive activity, and is used to determine the point at which a buffer is adequately filled to be sent across the network. The results are delivered to GDQS in 8.1 secs.

If there were no pipelined parallelism the total response time would be the sum of all these costs, i.e., 50 secs instead of 28.6. However, this is not the case and some of the time slots have overlapping slices. For example, GDS Access is handled by a separate thread and therefore overlaps with the thread that calls the *next()* method on the local root (i.e., EXCHANGE) operator and all the other operators down to the TABLE_SCAN operator. Similarly, delivery to the GDQS happens simultaneously with the retrieval and translation of the incoming data blocks from the lower level evaluators in the consumer EXCHANGE.

The profiling of this query has revealed another aspect of the negative effects of XML. Apart from the increase in the data volume that has to be transmitted over the network, which inevitably leads to a significant increase in the communication-related costs, XML-related processing seems to incur some CPU-related cost as well. This query was expected to have negligible CPU cost. However, the cost to compute the size of buffers and parse and unpack XML blocks is responsible for approx. 20% of the total cost of EXCHANGE, which is the bottleneck for this simple query. This happens despite the fact that, due to an earlier optimisation of the system, XML parsing in OGSA-DQP employs the SAX rather than the DOM model, which usually performs better for these sizes (e.g., [7]).

3.2 The impact of parallelising expensive operations

One of the main claims of Grid DQP has been that it can parallelise expensive operations in a transparent way to the user thereby making it of practical interest for CPU-intensive applications [4]. In this way, multiple instances of the same operator in the query plan can applied to disjoint subsets of a relation or of intermediate results in the plan. This form of parallelism is usually called intra-operator or partitioned. In non-service-based Grid DQP, intra-operator parallelism has been shown to be capable of yielding performance improvements [9]. The last set of experiments deals with intra-operator parallelism in the context of OGSA-DQP.

For this experiment, an additional table from the OGSA-DQP demo databases, *protein_sequences*, has been used. The expensive operation to be parallelised is a call to the `calculateEntropy` method of the `EntropyAnalyser` Web Service, which is part of the publicly available OGSA-DQP demo, too. In OGSA-DQP the OPERATION_CALL generic query operator handles calls to WSs. The test query is:

```
select p.ORF, go.id,
calculateEntropy(p.sequence)
from protein_sequences p,
goterms go, protein_goterms pg
where go.id=pg.GOTermIdentifier and
p.ORF=pg.ORF and
pg.ORF like "YCL0\%" and
go.id like "GO:0\%";
```

To test the effect of parallelising the OPERATION_CALL operator, two different parameters undergo variation. Firstly, the number of available evaluator nodes, which is equivalent to the number of machines available for the deployment of GQES instances that can invoke the web service, is not predefined as before but ranges from two to six machines. Secondly, the number of web service copies is increased by one for each run, from 1 to 6, to ensure that GQES instances increasingly invoke separate web service copies (rather than the same web service).

Figure 8 illustrates the response times for an increasing number of service copies. Each bar group in the figure represents a particular service copy configuration, and indicates the change in response time with respect to the number of available evaluator nodes. For example, the leftmost group shows results for 1 service copy for an increasing number of evaluators (i.e., the first bar indicates the response time when 1 evaluator invokes 1 service, the second bar indicates the response time when 2 evaluators invoke 1 service, and so on). The optimal line links the lowest bar in each bar group, to denote the change in the lowest response time in each configuration. As can be seen, the best response time for each service copy configuration is obtained when the number of evaluator nodes equals the number of available service copies. In this case, each evaluator invokes exactly one service, leading to maximum effective concurrency. Overall, intra-operator parallelism can improve response times by several factors (2.25 in this example).

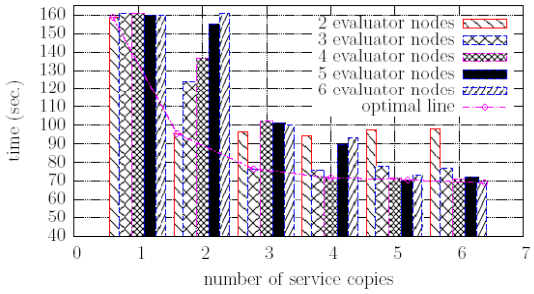

Figure 8. Comparison of operation_call cost for increasing number of available computational resources.

4. Conclusions

Distributed Query Processing (DQP) is, in itself, a complex task, and DQP artefacts comprise many components and require several tens of thousands of lines of code. Especially when applied to new, emerging settings, such as service-oriented Grids, their behaviour may be hard to predict, and new types of bottlenecks may arise. OGSA-DQP is one of the first Grid DQP systems, and as such, it is desirable to investigate its performance and to conduct some profiling. To this end, several types of experiments were performed, which are described in this paper, along with their results and lessons learnt.

The initial group of experiments tried to investigate the impact of the double service layer between the back-end database store and the OGSA-DQP clients. Retrieving data through services may slow down the tuple output rate by an order of magnitude. To mitigate these negative effects, tuples need to be retrieved in blocks so that messaging overheads are amortized across many tuples. However,

beyond a certain point, further increases in the size of the block cause performance degradation. As a result of these experiments, the forthcoming release of OGSA-DQP will retrieve data from GDS-wrapped databases using the block aggregation functionality of OGSA-DAI (the block size will be ~100).

Another group of experiments allowed us to identify several points where considerable resource CPU was being consumed. In particular, parsing and unpacking of the incoming data blocks in XML and constructing the data blocks for retransmission appeared to be taking considerable amounts of time. This is an indirect way in which XML-based communication compromises the performance. In a more direct way, inserting XML tags in the raw data may result in a significant increase of the total data volume. To tackle this, OGSA-DQP 2.0 has already moved towards the SAX rather than the DOM XML parsing model. In the future, alternative delivery mechanisms and formats that are provided by more recent OGSA-DAI distributions will be examined.

The experiments in the last group indicated that parallelising computationally expensive operations such as service calls to web services that perform relatively costly analysis on data, may significantly improve the performance. This observation suggests that providing multiple instances of the same service (or web services that transparently manage multiple processors) is an important prerequisite for high-performance Grids, when combined with the ability of higher level middleware services (such as OGSA-DQP) to utilize such services concurrently when they are available. The beneficial impact of this kind of parallelism is complemented by the pipelining execution, which can reduce query response time significantly by performing many operations concurrently.

References

[1] M. Antonioletti, M. Atkinson, R. Baxter, A. Borley, N. P. Chue Hong, B. Collins, N. Hardman., A. Hume, A. Knox, M. Jackson, A. Krause, S. Laws, J. Magowan, N. W. Paton, D. Pearson, T. Sugden, P. Watson and M. Westhead. The design and implementation of Grid database services in OGSA-DAI, in Concurrency and Computation: Practice and Experience, Volume 17, Issue 2-4, pages 357 – 376, Feb, 2005.

[2] Mario Antonioletti, Malcolm Atkinson, Rob Baxter, Andrew Borley, Neil P. Chue Hong, Patrick Dantressangle, Alastair C. Hume, Mike Jackson, Amy Krause, Simon Laws, Mark Parsons, Norman W. Paton, Jennifer Schopf, Tom Sugden, Paul Watson and David Vyvyan. OGSA-DAI Status and Benchmarks, 4th UK e-Science Programme All Hands Meeting (AHM 2005), 2005.

[3] M.Nedim Alpdemir, Arijit Mukherjee, Norman W. Paton, Paul Watson, Alvaro A.A. Fernandes, Anastasios Gounaris, and Jim Smith. "Service-Based Distributed Querying on the Grid". In Proc. of First International Conference on Service Oriented Computing - ICSOC 2003, LNCS 2910, pages 467-482.

[4] Anastasios Gounaris, Rizos Sakellariou, Norman W. Paton, Alvaro A. A. Fernandes. Resource Scheduling for Parallel Query Processing on Computational Grids. 5th IEEE/ACM International Workshop on Grid Computing, GRID'04, 2004.

[5] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In Proc. of ACM SIGMOD, pages

[6] M. Jackson, M. Antonioletti, N.P. Chue Hong, A.C. Hume, A. Krause, T. Sugden, and M. Westhead. Performance Analysis of the OGSA-DAI Software. OGSA-DAI mini-workshop, 3rd UK e-Science Programme All Hands Meeting (AHM 2004), 2004.

[7] Matthias Nicola and Jasmi John. XML parsing: a threat to database performance. Proceedings of the 12th International Conference on Information and Knowledge Management, CIKM'03, pages 175-178, 2003

[8] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, P. Vanderpilt, Open Grid Services Infrastructure, Version 1.0, June 27, 2003, GFD.15.

[9] Jim Smith, Anastasios Gounaris, Paul Watson, Norman. W. Paton, Alvaro. A. A. Fernandes, and Rizos Sakellariou. Distributed query processing on the grid. International Journal of High Performance Computing Applications, 17(4):353–367, 2003.

[10] JSR 114: JDBC Rowset Implementations http://www.jcp.org/en/jsr/detail?id=114