Using stream queries to measure communication performance of a parallel computing environment

Erik Zeitler and Tore Risch
Department of Information Technology, Uppsala University, Sweden
{erik.zeitler, tore.risch}@it.uu.se

Abstract

We have developed a data stream management system that supports declarative stream queries running over high data volumes in a supercomputing environment. To enable specification of massively parallel computations our query language provides processes as query language objects. The queries call process construction functions that execute stream sub-queries assigned to a CPU. Such queries can be used to define query functions that parallelize computations. The CPU assignment is normally automatic, but can also be influenced by the user. We show how this enables performance measurements of different communication topologies in a heterogeneous hardware environment containing a Linux cluster and a BlueGene.

1. Introduction

LOFAR [13] is currently building a radio telescope using an array of 25,000 omni-directional antenna receivers whose signals are digitized into data streams of very high rate. Scientists perform computations on these data streams to gain scientific insight. The LOFAR antenna array will be the largest sensor network in the world. The receivers produce raw data streams that arrive at the central processing facilities at a rate, which is too high for the data to be saved on disk. Furthermore, advanced numerical computations are performed on the streams in real time to detect astronomical events as they occur. For these data-intensive computations, LOFAR utilizes an IBM BlueGene supercomputer combined with conventional Linux clusters.

To enable stream processing in heterogeneous and massively parallel environments of LOFAR's kind we have developed a data stream management system called SCSQ (Super Computer Stream Query processor, pronounced *sis-queue*) [22]. SCSQ transparently executes on a variety of hardware platforms and operating systems, including MS Windows, Linux, and BlueGene. To support transparent streaming in a heterogeneous environment consisting of clusters with different communication subsystems, SCSQ features internal drivers that currently support MPI and TCP for carrying streams.

Continuous queries (CQs) are declaratively specified in a query language, SCSQL (pronounced *sis-kel*). To maximize throughput of streams and computations it is important to parallelize CQs into continuous subqueries, each executing as a separate process on a CPU. To enable a customized parallelization, SCSQL provides *stream processes* (SPs) as first-class objects in queries. The user associates subqueries with SPs. Massively parallel computations are defined in terms of sets of subqueries, executing on sets of stream processes.

Properties of the different CPUs, communication mechanisms, and operating systems substantially influence query execution performance. These properties are stored in a database, which is used by the query optimizer when assigning an SP to a CPU.

In implementing the query optimizer, it is crucial to understand how different strategies to distribute computation and communication influence the execution performance. It is particularly important for our application to maximize the bandwidth of the data streams from the receivers into the compute nodes of the BlueGene. The incoming streams are critical paths of the application since a sub-optimal input data rate will slow down the entire stream processing chain. In this paper, we use SCSQL queries in order to measure the streaming bandwidth of different communication topologies between a back-end Linux cluster and the BlueGene, as well as between compute nodes inside the BlueGene. SCSQ optionally allows the user to influence the choice of CPU to which an SP is assigned. We use this facility to specify different communication topologies in SCSQL.

In summary, we present the following contributions:

- The introduction of stream processes enables specification of massively parallel computations in the query language SCSQL.
- We show how SCSQL can be used to measure streaming bandwidth inside a BlueGene using different communication topologies. The results from

these measurements provide a basis for automatic CPU allocation strategies inside BlueGene.
- Analogously, inbound streaming bandwidth from a Linux cluster to BlueGene is measured using different communication topologies specified in SCSQL to provide a basis for automatic set-up of inbound streaming communication.

Before presenting the results, we give an overview of the SCSQ system and the heterogeneous hardware environment in which the experiments were performed. An introduction to the query language SCSQL is also given, and we show how to formulate mapreduce [8] and radix fft [12] queries using SCSQL.

2. The SCSQ system

In this section, we first describe the LOFAR hardware environment that is used for our experiments. Then, we present the overall SCSQ architecture and finally we describe the features of SCSQL that are used in the experiments.

2.1. Hardware environment

Figure 1 illustrates the stream dataflow in the LOFAR hardware environment. Users interact with SCSQ on a Linux front-end cluster. Another Linux back-end cluster first receives the streams from the sensors where they are pre-processed. Next, the BlueGene processes these streams. The output streams from the BlueGene are then post-processed in the front-end cluster and the result stream is finally delivered to the user. Thus, three computer clusters are involved.

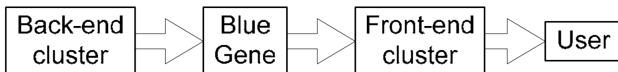

Figure 1. Stream data flow in the LOFAR environment.

The hardware components are characterized by different architectures. The BlueGene features dual PowerPC 440d 700MHz (5.6 Gflops max) compute nodes connected by a 1.4 Gbps 3D torus network, and a 2.8 Gbps tree network. The time it takes for a compute node to send data to another one depends on the relative locations of these nodes in the torus, and how loaded the nodes between them are. Each compute node has a local 512 MB memory. The compute nodes run the compute node kernel (CNK) OS [15], a simple single-threaded operating system that provides a subset of UNIX functionality. One important limitation of CNK is the lack of support for server capabilities (no *listen(), accept()* or *select()*). Each compute node has two CPUs, of which normally one is used for computation and the other one for communication with other compute nodes. A native MPI implementation is used for communication between BlueGene compute nodes, whereas communication with the Linux clusters utilizes I/O nodes that provide TCP or UDP. Each I/O-node is equipped with a 1 Gbit/s network interface. I/O nodes are only used for communication, and cannot be used for computations. In LOFAR's BlueGene, there are 6144 dual processor compute nodes, grouped in processing sets of 8 compute nodes and one I/O node.

The Linux front and back-end clusters are IBM JS20 computers with dual PowerPC 970 2.2GHz processors. Each computer in the back-end cluster has a 1 Gigabit Ethernet interface connected via a switch to the BlueGene.

2.2. SCSQ architecture

Figure 2 illustrates a query that is set up for execution in the hardware environment. SCSQ users interact with the client manager, in which they specify CQs using SCSQL. The execution of a CQ forms a directed acyclic graph of running processes (RPs), each executing the subquery specified in one SP.

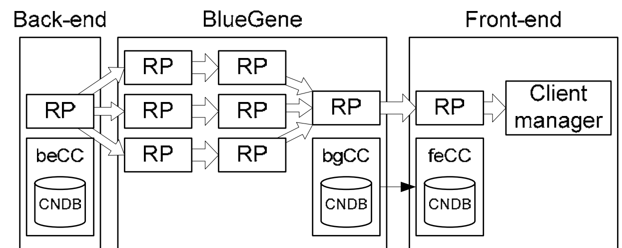

Figure 2. Set-up of a CQ for execution in SCSQ. Wide arrows indicate data streams.

The execution of CQs may be stopped either by explicit user intervention or by a stop condition in the query that makes the stream finite. When a CQ is stopped, its RPs are terminated. RPs regularly exchange control messages, which are used to regulate the stream flow between them and to terminate execution upon a stop condition.

When a user submits a CQ, it is optimized and started in the client manager. When the client manager identifies an SP, the sub-query of that SP is registered with the coordinator of the cluster where the sub-query is to be executed (*feCC*, *bgCC,* or *beCC* in Figure 2). Then, the coordinator starts an RP to execute the sub-query. In addition, an RP can dynamically start new RPs by requesting them from the cluster coordinator of the cluster where the new RP is started.

Since the BlueGene lacks server functionality, subqueries from the client manager to be executed on the BlueGene are registered with the *feCC*. The *bgCC* retrieves new sub-queries from the *feCC* by polling. As BlueGene compute nodes can execute only one process,

each new RP in BlueGene is assigned to a new compute node.

Each cluster coordinator maintains an internal *compute node database* (CNDB) containing the properties and status of the possibly thousands of compute nodes in its cluster. A *node selection algorithm* in the cluster coordinator starts the new RP on a suitable compute node by querying its CNDB. Currently, a naïve node selection algorithm is used, returning the next available node.

2.3. Running Processes

An RP has the components shown in Figure 3. It is responsible for i) compiling its subquery into a local *Stream Query Execution Plan, SQEP* and interpreting it, ii) delivering the result to other RPs, its *subscribers*, iii) dynamically requesting new RPs from a coordinator if needed, iv) retrieving its input data from other RPs, its *producers*, and v) monitoring the execution of its SQEP.

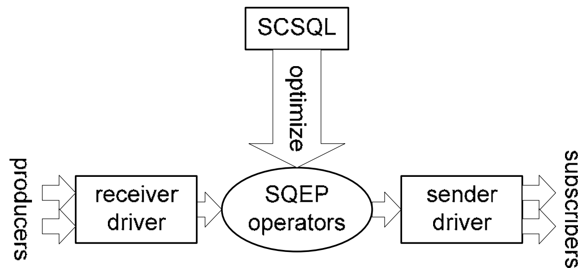

Figure 3. A SCSQ running process.

The operators in the SQEP are executed when data arrives. Incoming data is buffered in a *receiver driver* and de-marshaled (materialized) into objects. Streams of materialized data objects are delivered to the operators of the SQEP. The objects resulting from the operators are passed on to the *sender driver*, which marshals them and sends the buffer contents to subscribers. Objects are dynamically de-allocated when no longer needed by any operator. The sender and receiver drivers can use various network protocols for carrying the streams. We have implemented stream carrier protocols based on MPI and TCP. SCSQ supports the use of MPI on any MPI enabled cluster. MPI is always used inside the BlueGene as that is the only allowed protocol, while TCP is always used when communicating between clusters. The MPI sender and receiver drivers contain double buffers so that one buffer can be processed while the other one is read or written.

2.4. SCSQL

SCSQL is a query language similar to SQL, but extended with streams and *stream processes* as first-class objects. Stream processes allows dynamic parallelization of continuous queries, which is used in this paper to measure the performance of a massively parallel and heterogeneous computing environment. This section introduces SCSQL.

All data in SCSQ is represented by *objects* in SCSQL. The relation between first-class objects in SCSQL is illustrated in Figure 4. A *stream* is an object that represents (possibly unbounded) sequences of any kind of objects. The result of a continuous subquery is a stream. Continuous subqueries are assigned to *stream processes*. Users of SCSQL define parallel and distributed stream computations by assigning continuous subqueries to stream processes.

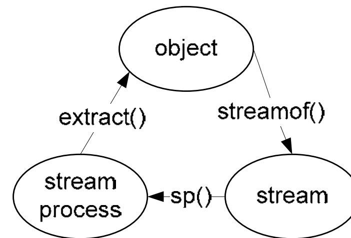

Figure 4. The relation between streams, stream processes and objects in SCSQL.

The function *sp(s, c)* assigns the subquery *s* to a new stream process to be run in cluster *c*. The function *extract(p)* requests elements (objects) from the subquery assigned to stream process *p*. If *p* ever terminates, *extract(p)* also terminates. The function *streamof(e)* transforms the output of any expression *e* to a stream. This is useful when a stream output is desired from functions that do not naturally return streams, e.g. *count()*, which returns a single integer.

To enable easy handling of sets of parallel stream processes, the function *spv(**s**, c)* assigns each subquery in the set ***s*** to a new stream process on some compute node in the cluster *c*, and returns a set (bag) of handles to the assigned stream processes.

The function *merge(**p**)* generalizes *extract()* by requesting elements from each stream process in ***p***. *merge()* terminates when (if ever) the last stream process in ***p*** terminates.

The use of the data types representing streams and stream processes allows specification of parallel and distributed CQs with different topologies. *merge()* provides stream combinations, while variables bound to sets of stream processes provide parallel execution.

For example, the distributed grep mapreduce [8] query using 1000 parallel *grep* calls is specified in SCSQL as follows:

```
1 merge(spv(
2      select grep("pattern", filename(i))
3      from integer i
4      where i in iota(1,1000)));
```

Line 1 contains the reduce predicate. In this case there is no reduction, so there is no function outside the merge.

On line 2, the subquery performs a *grep* for a pattern on the ith filename in a table. Each subquery executes in a separate process. Line 4 specifies the degree of parallelism, in this case 1000 processes. *iota(n,m)* generates all integers from *n* to *m*. In this example, *iota()* is used to generate 1000 duplicates of the select stream, and to provide a key to the *filename()* table.

Splitting of streams is specified by referencing common variables bound to stream processes, as illustrated by the following query function, which implements the radix2 parallelization of FFT [12] for a stream source named *s*.

```
1  create function radix2(string s)
2                               ->stream
3  as select radixcombine(merge({a,b}))
4  from sp a, sp b, sp c
5  where a=sp(fft(odd (extract(c))))
6    and b=sp(fft(even(extract(c))))
7    and c=sp(receiver(s));
```

The *receiver()* function returns a stream of 1D arrays of signal data. *odd(x)* and *even(x)* obtain odd and even elements from array *x*, respectively. *radixcombine()* combines the results from the partial FFT algorithms working in parallel.

Optionally, the SCSQL user can constrain the allowed compute nodes for the node selection algorithm by specifying a *node allocation query* as an extra argument to *sp()* and *spv()*. This query returns a stream of allowable compute nodes in preferred allocation order, called the *allocation sequence*. The allocation sequence is passed to the node allocation algorithm of the cluster coordinator when it allocates the RP for an SP. The node selection algorithm will choose the first available node in the allocation sequence. (In case the stream contains no available node, the query will fail.) Thus, allocation sequences allow the user to restrict and prioritize the node selection order.

In the next section we show how we utilize allocation sequences to enforce different communication topologies. This helps us determine how to achieve maximum streaming bandwidth. The gained knowledge will be used to improve the node selection algorithm.

3. Streaming performance

Using SCSQL and its allocation sequence option, we set up different communication topologies and measure how they influence the streaming bandwidth. The following experiments are performed:

1. The streaming bandwidth between RPs executing on compute nodes inside the BlueGene is measured. For good performance of *extract()* and *merge()*, SCSQ buffers incoming elements in the receiver driver. Different buffer settings for MPI streams inside the BlueGene are evaluated. Furthermore, explicit node selections are used to measure different communication topologies inside the BlueGene.
2. The bandwidth is measured for communicating streams from RPs in the back-end cluster to RPs inside the BlueGene. The impact on the bandwidth of different node selections in the back-end cluster and in the BlueGene is measured. As the system uses TCP for communication between the back-end cluster and BlueGene, we rely on the buffering of the TCP stack in this case.

In all experiments, the streams contain arrays of numerical data, as required by the application. The bandwidth is computed by measuring the total time to communicate a finite stream of 3MB arrays between stream processes. Small array sizes increase processing overhead, and we are primarily interested in communication performance, hence the large array size. Each experiment was performed five times in order to achieve low variance in the measurements.

3.1. Intra-BG streaming

The following experiments are performed:

1. We measure the bandwidth of point-to-point communication between two RPs, which execute on different BlueGene compute nodes.
2. We measure the bandwidth of stream merging from two RPs to a third one.

In the experiments, we vary the buffer sizes of the communication subsystem. We also compare the usage of double and single buffering.

Figure 5 illustrates the set-up of the point-to-point measurement. *a* generates a stream of large arrays and *b* counts the total number of arrays in the finite stream extracted from *a*. The result of the count is sent to the front-end. Since only one number is transmitted from *b* to the client manager, the total time measured is dominated by the time for streaming the data from *a* to *b*.

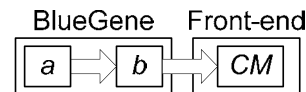

Figure 5. Intra-BG point to point streaming.

The sending RP *a* (i) generates the arrays, (ii) marshals them into a send buffer and (iii) transmits the send buffers when they are full. The receiving RP *b* (iv) receives buffers, (v) de-marshals the buffer contents, (vi) allocates new arrays, (vii) counts them, and (viii) de-allocates them. Only the result of the count is streamed to the front-end. The query is expressed in SCSQL as follows:

```
1  select extract(b)
2  from sp a, sp b
3  where b=sp(streamof(count(extract(a)))
4              'bg',0) and
```

```
5      a=sp(gen_array(3000000,100),'bg',1);
```

gen_array() generates the finite stream of 100 arrays of size 3MB each. The calls to *sp()* assign the streams to new stream processes on a compute nodes in the 'bg' (BlueGene) cluster. The function *count()* counts the number of elements in a bag. The function *streamof()* makes a stream of the output of *count()*. Allocation sequences are specified in the third arguments of the *sp()* calls as single node identifier values (0 and 1), since we want to exactly specify the selected node here. The selected node cannot be busy in this query since we know what nodes are allocated and where they are located in the BlueGene.

Figure 6 shows the bandwidth of intra-BG point-to-point streaming. As can be seen, the optimal buffer size is 1000 bytes for both single and double buffering. The drop-off above the 1000-byte buffer size is probably due to cache misses. The performance degradation for buffers smaller than 1000 bytes buffer size is because 1K is the smallest message size that can be exchanged in the BlueGene 3D torus. Furthermore, we observe that double buffering pays off for large buffers. A number of bumps are clearly seen in the double-buffer curve. No explanation for this phenomenon can be found, but it is nevertheless statistically significant.

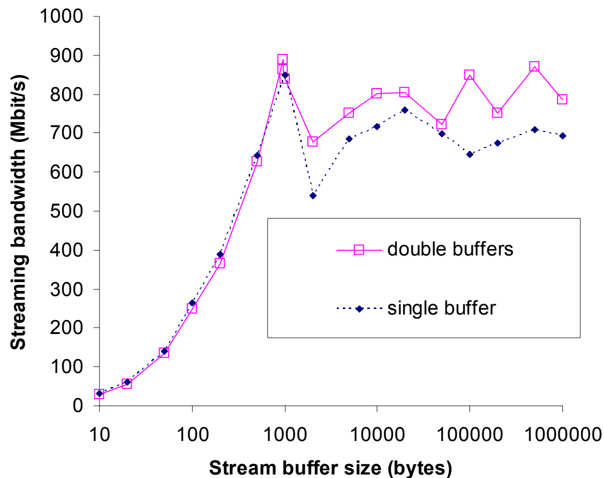

Figure 6. Point to point streaming performance.

For stream merging, we measure the throughput when two RPs send data to a third one. When messages are sent between non-adjacent nodes in BlueGene, they must be routed through the communication co-processors of the nodes in between. Communication will be slower if these co-processors are busy.

Since the enumeration of compute nodes in the BlueGene 3D torus is known, it is easy to specify the two communication topologies in Figure 7 using the allocation sequence feature of SCSQL. In both cases *c* merges data from the streams of *a* and *b*. The two experiments are defined in SCSQL by varying *x* and *y* in the following query:

```
1  Select extract(c)
2  from sp a, sp b, sp c
3  where c=sp(count(merge({a,b})), 'bg',0)
4  and a=sp(gen_array(3000000,100),'bg',x)
5  and b=sp(gen_array(3000000,100),'bg',y);
```

count() counts the total number of arrays in the merged streams *a* and *b*. The explicit node selections *0, x*, and *y* on lines 3–5 specify the exact BlueGene node numbers where the RPs execute.

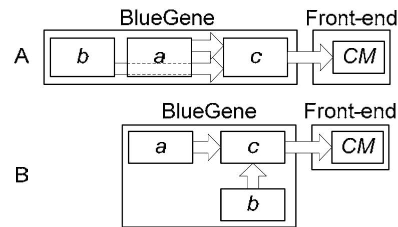

Figure 7. Alternative BlueGene node selections for stream merging.

Figure 7A shows a *sequential node selection*, where MPI messages from *b* to *c* are routed through the communication co-processor of *a*. Here, *x=1* and *y=2* to select compute nodes arranged as in figure 7A. Figure 7B shows a *balanced node selection*, where messages from *a* and *b* are sent directly to *c* over individual communication channels. Here, *x=1* and *y=4* to select compute nodes arranged as in figure 7B.

Figure 8 shows the total streaming input bandwidth at node *c* for stream merging using the two node selection strategies. Both single and double buffering is evaluated. Analogously to the results of the point-to-point streaming experiment shown in Figure 6, we expected double buffering to pay off for large buffers.

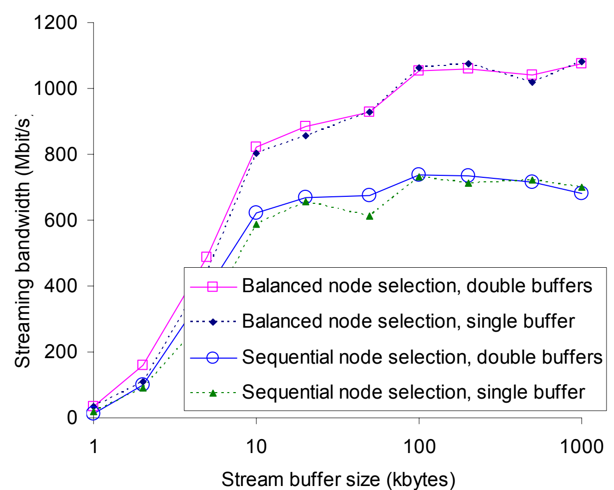

Figure 8. Alternative BlueGene node selections for stream merging.

We observe the following:

1. The streaming bandwidth depends highly on the compute nodes to which the RPs are allocated. This is because of the topology of the BlueGene 3D torus interconnection network.
2. The benefit of double buffering is less significant than that of point-to-point communication.
3. Finally, an interesting observation is that buffers smaller than 10K are much slower for stream merging than for point-to-point communication.

The reason for better performance for large buffers when merging streams is that the single-threaded communication co-processor of *c* must handle data streams from both *a* and *b*. In *c*, it switches between receiving messages from *a* and *b*. Less frequent switching improves communication. By contrast, for point-to-point communication, all messages come from the same source, so the co-processor does not pay any switching penalty. Thus, sending larger but fewer messages is beneficial for stream merging while the opposite holds for point-to-point communication.

3.2. BG inbound streaming

We conducted experiments for six different ways to inject data streams into the BlueGene, named Query 1 through Query 6. The inbound streaming bandwidth of each query is measured for different numbers of parallel input streams by altering a query variable *n*. In all experiments, the total number of arrays in all the finite streams produced in the back-end cluster is counted. The output data from the query is a single integer. Thus, the time to execute the query is dominated by time for streaming the data from the back-end cluster into the BlueGene.

Query 1 investigates the streaming bandwidth when all streams 1 through *n* are sent from a single node in the back-end cluster through a single I/O node into a single compute node in the BlueGene. Query 2 differs from Query 1 in that several compute nodes in the back-end cluster are injecting data into BlueGene. This is to investigate whether parallelization over several compute nodes in the back-end cluster will improve the streaming bandwidth compared to that of Query 1. Queries 3 and 4 transfer all data over one single I/O node but parallelize the receiving over several compute nodes in BlueGene. This is to see whether parallelizing the receiving compute nodes will improve the streaming bandwidth in comparison to all streams being received on a single compute node. Queries 5 and 6 are analogous to Queries 3 and 4 but parallelize the data injection into BlueGene over several I/O nodes. By intuition, query 6 can be expected to achieve the highest streaming bandwidth of all queries, since parallel back-end compute nodes inject data through parallel I/O channels.

The distribution pattern of Query 1 is shown in Figure 9. All streams are produced on a_1 through a_n, executing on the same compute node in the back-end cluster. All streams are sent to *b* inside BlueGene, which merges and counts them. *c* extracts the count from *b* unchanged and sends it to the client manager in the front cluster. The reason to include *c* in this query is to make all experiments comparable.

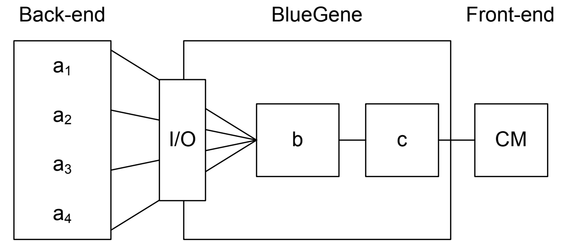

Figure 9. Execution distribution of Query 1.

Query 1 is formulated in SCSQL as follows:

```
 1  select extract(c) from
 2  bag of sp a, sp b, sp c,
 3  integer n
 4  where c=sp(extract(b),'bg')
 5  and   b=sp(count(merge(a)), 'bg')
 6  and   a=spv(
 7    (select gen_array(3000000,100)
 8    from integer i where i in iota(1,n)),
 9              'be', 1)
10  and n=4;
```

The explicit node selection on line 9 assigns all back-end SPs to compute node 1 in the back-end cluster.

The execution distribution of Query 2 is shown in Figure 10. In this query, a_1 through a_n execute on different compute nodes in the back-end cluster. All streams are sent to *b* inside the BlueGene, which merges them and counts the total number of arrays. *c* passes on the count unchanged as before.

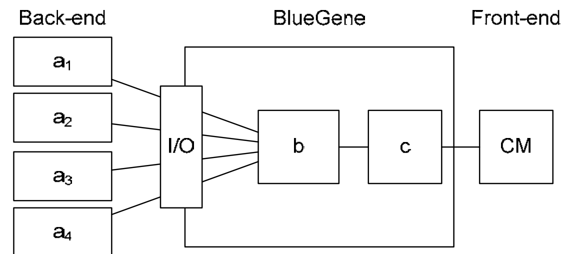

Figure 10. Execution distribution of Query 2.

Query 2 is formulated in SCSQL as follows:

```
 1  select extract(c) from
 2  bag of sp a, sp b, sp c,
 3  integer n
 4  where c=sp(extract(b), 'bg')
 5  and   b=sp(count(merge(a)), 'bg')
 6  and   a=spv(
 7    (select gen_array(3000000,100)
 8    from integer i where i in iota(1,n)),
```

```
 9               'be', urr('be'))
10  and n=4;
```

Only the last argument to *spv()* in line 9 differs from Query 1. Here we want to assign each SP in *a* to different compute nodes. The node allocation function *urr(cl)* retrieves a stream from the CNDB of cluster *cl* of compute node identifiers where each identifier represents a new available node in the cluster in a round-robin fashion. This allocation sequence stream is later shipped back to the cluster coordinator by the *spv()* call to be used by the node selection algorithm. By shipping stream handles we avoid unnecessary data shipping.

The execution distribution of Query 3 in Figure 11 parallelizes the aggregation over several RPs, each one running on a separate receiving BlueGene compute node. Compute nodes belonging to the same *pset* use the same I/O node for inbound communication. All streams from the back-end cluster are sent to the BlueGene compute nodes through a single I/O node by specifying b_1 through b_n to belong to the same *pset*.

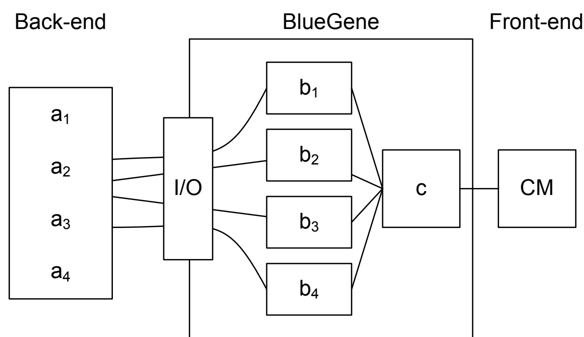

Figure 11. Execution distribution of Query 3.

Query 3 is defined by the following SCSQL query.

```
 1  select extract(c) from
 2  bag of sp a, bag of sp b, sp c,
 3  integer n
 4  where c=sp(streamof(sum(merge(b)),
 5             'bg'))
 6  and   b=spv(
 7    (select streamof(count(extract(p)))
 8     from sp p
 9     where p in a),
10               'bg', inPset(1))
11  and a=spv(
12   (select gen_array(3000000,100)
13    from integer i where i in iota(1,n)),
14               'be', 1)
15  and n=4;
```

On line 10, the processor selection function *inPset(k)*, which returns a stream of compute node identifiers in *pset* number *k*, forces all SPs to belong to the same *pset*.

The execution distribution of Query 4, shown in Figure 12, differs from Query 3 in that the back-end RPs run on different compute nodes.

Query 4 is defined by the following SCSQL query.

```
 1  select extract(c) from
 2  bag of sp a, bag of sp b, sp c,
 3  integer n
 4  where c=sp(streamof(sum(merge(b))),
 5  'bg')
 6  and   b=spv(
 7    (select streamof(count(extract(p)))
 8    from sp p
 9    where p in a),
10               'bg', inPset(1))
11  and a=spv(
12   (select gen_array(3000000,100)
13    from integer i where i in iota(1,n)),
14               'be', urr('be'))
15  and n=4;
```

The only difference from Query 3 is the call to *urr()* on line 14, enforcing all RPs to execute on different compute nodes in the back-end cluster.

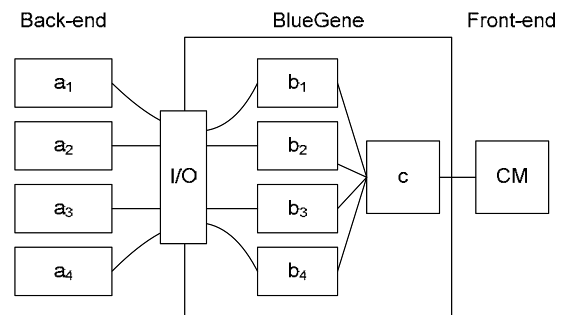

Figure 12. Execution distribution of Query 4.

The execution distribution of Query 5, shown in Figure 13, utilizes different I/O nodes for the communication of streams from the back-end cluster.

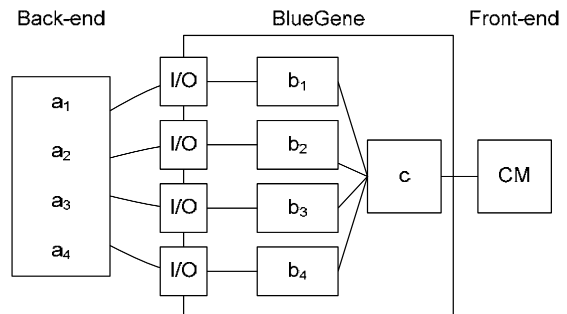

Figure 13. Execution distribution of Query 5.

The following SCSQL query defines Query 5.

```
 1  select extract(c) from
 2  bag of sp a, bag of sp b, sp c,
 3  integer n
 4  where c=sp(streamof(sum(merge(b))),
 5             'bg')
 6  and   b=spv(
 7   (select streamof(count(extract(p)))
 8    from sp p
 9    where p in a),
10               'bg', psetrr())
11  and a=spv(
```

```
12    (select gen_array(3000000,100)
13     from integer i where i in iota(1,n)),
14              'be', 1) and n=4;
```

This query differs from Query 3 in the processor selection on line 10. The function *psetrr()* returns a stream of BlueGene compute node numbers, where each succeeding node number belongs to a new *pset* in a round-robin fashion. This will parallelize the inbound communication over different I/O nodes, since compute nodes belonging to different *pset*s will use different I/O nodes.

Finally, the execution distribution of Query 6 is shown in Figure 14. This query differs from Query 5 in that back-end stream processes run on different nodes in the back-end cluster.

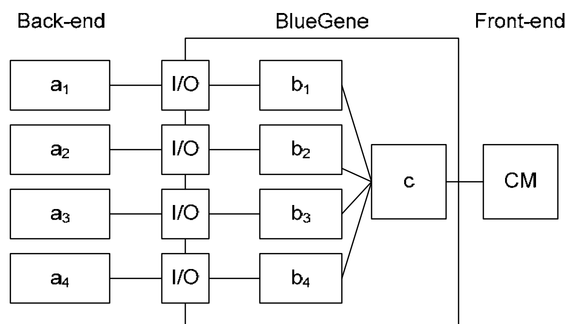

Figure 14. Execution distribution of Query 6.

The following SCSQL query defines Query 6.

```
1   select extract(c) from
2   bag of sp a, bag of sp b, sp c,
3   integer n
4   where c=sp(streamof(sum(merge(b))),
5                'bg')
6   and   b=spv(
7     (select streamof(count(extract(p)))
8      from sp p
9      where p in a),
10              'bg', psetrr())
11  and a=spv(
12    (select gen_array(3000000,100)
13     from integer i where i in iota(1,n)),
14              'be', urr('be'))
15  and n=4;
```

The difference from Query 5 is the call to *urr()* on line 14, assigning all SPs to different compute nodes in the back-end cluster.

Figure 15 compares the BG inbound streaming bandwidth for Queries 1 through 6. *n* is the number of RPs in the back-end cluster that inject streams into the BlueGene. The y-axis measures the total inbound streaming bandwidth from the back-end cluster into the BlueGene compute nodes.

We observe the following:

(1) Queries 1 through 4 all communicate using a single I/O node on the BlueGene. They all have significantly lower bandwidth than that of Queries 5 and 6. Thus, as expected, it is favorable to use many I/O nodes.

(2) The streaming bandwidth of Queries 3 and 4 are slightly better than that of Queries 1 and 2. Changing from one to two receiving BlueGene compute nodes off-loads the communication burden, while further increasing the number of receiving compute nodes is not worthwhile since the total streaming bandwidth does not increase. Hence, it pays off to increase the number of receiving compute nodes from one to two even if there is only one I/O node available. This communication topology should be used in the node selection algorithm when compute nodes are available but the number of I/O nodes is limited.

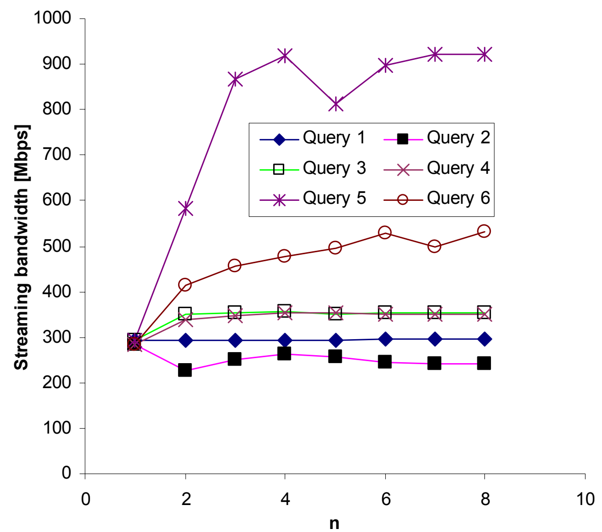

Figure 15. Results for queries 1 through 6.

(3) As can be seen, the best streaming bandwidth is achieved for Query 5, which peaks at ~920 Mbps. It is surprising that a single 1 Gbps connection from the back-end cluster is faster than four separate 1 Gbps connections as in Query 6. It is thus faster to inject streams over different I/O nodes from the same back-end cluster compute node than from different back-end compute nodes. This indicates coordination problems in the I/O node when communicating with many outside nodes. The conclusion is that the node selection algorithm should attempt to co-locate back-end RPs to the same compute node until saturation.

(4) Similarly, the streaming bandwidth of Query 1 is better than that of Query 2, indicating that it is better to run as many RPs on the same back-end node as possible rather than running them on different back-end nodes.

(5) In Query 5, there is a significant performance dip for *n*=5. This is probably because there were only

four I/O nodes available on the BlueGene partition where the experiments were performed. For n>4, compute nodes have to share I/O nodes and therefore the bandwidth decreases. In this case, the node selection algorithm could resort to increase the number of receiving compute nodes as in observation (2) above.

We are currently investigating how to extend the node selection algorithm with the above knowledge.

4. Related work

There are many data stream management system (DSMS) implementations, some of which execute on a single node [3] [5] [7] [9] [14] [17] [19], and some are distributed [1] [4] [10] [11] [16] [20] [21]. Some of these implementations provide high-level SQL-like query languages such as STREAM [3] and TelegraphCQ [5]. In [3], streams are treated as continuously updated relations, while in [5] they are implemented as external functions emitting tuples as an unbounded bag. Unlike all other DSMS projects, the SCSQ data model treats both streams and processes as first class objects. Stream processes allow users to specify massively parallel and distributed computations in CQs by dynamically starting stream processes at run time. Furthermore, the SCSQ user can optionally even influence the location for the node assignments, which has been used in this paper to measure communication performance.

Tribeca [19] provides *pipes* as first-class objects in its query language. These pipes are similar to our stream data type but Tribeca provides no parallelization of their execution, and no dynamic process creation. Similarly, WaveScope [9] provides a stream processing language where arbitrary computations can be specified as functions over streams in a non-distributed stream processing environment.

SPC [11] was evaluated using the Linear Road Benchmark [2] on a highly parallel PC cluster. However, the distribution is manual and SPC has no query language as SCSQL.

Dynamic load balancing for distributed DSMSs has been studied in [4] [20] [21]. In Borealis [1], a central coordinator migrated stream processing operators between nodes using load statistics [21]. Medusa nodes migrate operators between each other using computational economy methods [4]. In D-CAPE [20], different initial distribution and redistribution strategies were experimentally evaluated. The explicit optional node placement primitives of SCSQ can be used for static load balancing, as was done in this paper to measure performance. In addition, SQCQL provides user primitives to specify how to logically parallelize algorithms dynamically through stream processes. This is orthogonal to load balancing.

Distributed execution of expensive user-defined stream query functions has been studied in GSDM [10]. GSDM distributes its stream computations by selecting and composing distribution templates from a library. By contrast, all distribution topologies are expressed as SCSQL queries. In [10], only one parallelization topology (partition, compute, and combine) for user-defined functions is provided. Mapreduce [8] also provides another special distribution topology, namely map and reduce. SCSQL allows the specification of any communication topology. Sawzall [17] features a high-level language that enables compact specifications of massively parallel mapreduce tasks. However, Sawzall is restricted to the mapreduce distribution topology. Furthermore, Sawzall lacks many advanced operators for aggregation and computation, whereas SCSQ features all common stream operators including window aggregation.

5. Conclusions and future work

We presented the SCSQ system, which is a DSMS that runs in a massively parallel hardware environment featuring a BlueGene. Several different kinds of clusters are included in the execution of a continuous query.

The query language SCSQL provides both *streams* and *stream processes* as first class objects. The users of SCSQ are thereby given control over the parallelization of stream queries and functions. Users specify parallel computations by assigning sub-queries to stream processes executed in parallel. We have shown how to parallelize mapreduce and radix FFT using SCSQL.

Furthermore, SCSQL also allows the user to specify allocation sequences that restrict and prioritize which compute nodes to be chosen for execution. Using such allocation sequences, we specified different physical communication topologies for a mapreduce-like query. These experiments measured different topologies for inbound streaming into BlueGene. The measurement showed that in order to achieve reasonable performance, a considerable amount of I/O nodes must be designated to handle input streams. In our experiments, we also discovered that it is favorable to use as few as possible input compute nodes in the back-end cluster. This indicates that the BlueGene I/O is a bottleneck. These experiments provide basis for extending the node selection algorithm.

Moreover, our experiments show that the flexibility of the query language provides a powerful tool for investigating the streaming performance of any computer environment. These experiments can easily be repeated on other kinds of clusters to understand their streaming performance and to provide basis for specific node selection algorithms.

SCSQL allocation sequences were also used to measure the bandwidth of communication between compute nodes inside BlueGene using native MPI. The impact of

buffer sizes and double buffering used in the MPI communication was measured for different topologies. The optimal stream buffer size for MPI communication inside BlueGene was highly dependent on whether point-to-point or merging stream communication was performed. In general, the buffer should be much larger in the case of stream merging. Double buffering proved to be less important in our experiments.

Furthermore, the location of the BlueGene compute nodes highly affects the inter-node communication since data may be routed through intermediate nodes in the 3D-torus of BlueGene. We showed that stream merging performs up to 60% better if no busy intermediate nodes are involved in the communication.

We are currently experimenting with refinements of the node selection algorithm for the BlueGene based on the results of this paper. It should be investigated whether it is possible to parameterize the node selection algorithm so that it can be used in any parallel hardware environment. In the current hardware configuration, we have only four I/O nodes and four nodes in the back-end cluster. It remains to be investigated what happens for large amounts of back-end and I/O nodes. It is also important to analyze the performance of continuous queries involving expensive functions. Further measurements could be made using benchmarks such as The Linear Road Benchmark [2]. The goal is to understand how to distribute streams and computations optimally in a heterogeneous hardware environment.

Acknowledgements

This work is supported by ASTRON.

References

[1] D. J. Abadi et al, "The Design of the Borealis Stream Processing Engine", Proc. CIDR 2005 Conf., Asilomar, CA.

[2] A. Arasu et al, "Linear Road: A Stream Data Management Benchmark", Proc. VLDB 2004 Conf., Toronto, Canada, pp 480–491.

[3] A. Arasu et al, "STREAM: The Stanford Data Stream Management System", http://infolab.stanford.edu/stream/.

[4] M. Balazinska, H. Balakrishnan, and M. Stonebraker, "Contract-Based Load Management in Federated Distributed Systems", Proc. 1st Symp. on Networked Systems Design and Implementation, USENIX Association 2004, pp 197 – 210.

[5] S. Chandrasekaran et al, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World", Proc. CIDR 2003, Asilomar, CA.

[6] M. Cherniack et al, "Scalable distributed stream processing", Proc. CIDR 2003, Asilomar, CA.

[7] C. Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk, "Gigascope: A Stream Database for Network Applications", Proc. SIGMOD 2003 Conf., San Diego, CA, pp 647–651.

[8] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters", Proc. 6th Symp. on OS Design and Implementation, USENIX Association 2004, pp 137 – 150.

[9] L. Girod, Y. Mei, R. Newton, S. Rost, A. Thiagarajan, H. Balakrishnan, S. Madden, "The Case for a Signal-Oriented Data Stream Management System", Proc. CIDR 2007, Asilomar, CA.

[10] M. Ivanova and T. Risch, "Customizable Parallel Execution of Scientific Stream Queries", Proc. VLDB 2005, Trondheim, Norway, pp 157–168.

[11] N. Jain et al, "Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core", Proc. SIGMOD 2006, Chicago, IL, USA.

[12] V. Kumar, A. Grama, A. Gupta, and G. Karypis, "Introduction to Parallel Computing", The Benjamin Cummings Publishing Company, Inc., 1994.

[13] LOFAR, http://www.lofar.nl/.

[14] S. Madden, M. A. Shah, J. M. Hellerstein, and Vijayshankar Raman, "Continuously adaptive continuous queries over streams", Proc. SIGMOD 2002 Conf., Madison, WI, USA, pp 49–60.

[15] J. E. Moreira et al, "Blue Gene/L programming and operating environment", IBM J. of Research and Development, 49(2/3), 2005, pp 367–376.

[16] K. W. Ng and R. R. Muntz, "Parallelizing user-defined functions in distributed object-relational DBMS", Proc. IDEAS Symp. 1999, Montreal, Canada, pp 442–450.

[17] R. Pike, S. Dorward, R. Griesemer, S. Quinlan, "Interpreting the Data: Parallel Analysis with Sawzall", Scientific Programming Journal 13:4, pp. 227-298.

[18] E. A. Rundensteiner et al, "CAPE: A Constraint-Aware Adaptive Stream Processing Engine", in N.Chaudhry, K. Shaw, and M. Abdelguerfi (eds.): "Stream Data Management", Advances in Database Systems Series, Springer 2005, pp 83–111.

[19] M. Sullivan, A. Heybey, "Tribeca: A System for Managing Large Databases of Network Traffic", Proc. USENIX Conf., New Orleans, 1998.

[20] T. Sutherland, B. Liu, M. Jbantova, E. A. Rundensteiner, "D-CAPE: Distributed and SelfTuned Continuous Query Processing", Proc. CIKM 2005 Conf., Bremen, Germany.

[21] Y. Xing, S. Zdonik, J.-H. Hwang, "Dynamic Load Distribution in the Borealis Stream Processor", Proc. ICDE 2005 Conf., Tokyo, Japan.

[22] E. Zeitler, T. Risch, "Processing high-volume stream queries on a supercomputer", Proc. ICDE 2006 Workshops, Atlanta, GA, USA.