TSQR on EC2 Using the Nexus Substrate

Benjamin Dagnon

TSQR is an algorithm that trades flops for reduced number of messages, and has already been proven efficient in grid computing[1]. The goal of this project was to implement TSQR on Amazon’s EC2, while exploring the Nexus substrate.

Nexus

The Nexus substrate attempts to address composability in cluster computing through resource allocation and job scheduling. While Nexus allows for MPI and MapReduce to run on top of it, this project makes use of its native APIs. We – myself and Ben Hindman – decided against MPI in large part due to others having already implemented TSQR using MPI, though I was unable to obtain a fully functioning implementation for comparison. A MapReduce implementation may be novel and worth exploring, but it has workflow and communication constraints that are not in place when implementing directly on top of Nexus.

The native Nexus API, from a user-developer’s perspective, is simple. There is a master scheduler (master) in control of some number of slave daemons (one each corresponding to a single node), and this master attempts to efficiently and fairly offer these resources to client frameworks. These frameworks can accept any number of given offers in part or in whole, specifying CPU count and memory for each offer. Tasks are then run on slave nodes using an executor specified by the framework, and this executor runs and manages any tasks for a given node.[2] Along with resource offers, frameworks also receive status updates for nodes and both frameworks and executor can use framework messages as a form of non-bulk communication.

Design Choices

Given the large design space allowed on Nexus, my approach was to implement a solid basic design which could later be modified and refined to enable more performance. After the choice of platform, the most important initial decision was to make the implementation file-based. The very real risk of file I/O becoming the limiting factor was overshadowed by the ease with which a file-based implementation allowed other features, such as shorter task lifespans and fault tolerance.

The framework’s implementation uses a scheduler which wraps around other phase schedulers that are operated sequentially. The active phase scheduler receives a subset of the Nexus API calls, and is responsible for accepting offers. There are two existing phase schedulers: one for handling factorization, and another that enables the application of the \( QR \) formed by the first type of scheduler to be applied to an \( nxn \) input matrix (such as the \( R \) value output by the first scheduler). The phase scheduler models the overall \( QR \) factorization as a tree, and determines the assignment of slave nodes to tree nodes on-the-fly.

\( QR \) tasks are modeled simply: they take inputs, operate on them and generate outputs. This allows them to be naive operators with no knowledge of the scheduler or overall architecture, outside of issues such as the shape and format of their inputs and outputs. I/O is modeled as data streams, and the desired sections of each input stream is accumulated in a buffer before being operated on. The stream abstraction allows switching between HDFS and local file systems (notably, on systems without HDFS installed), and may make an adoption of socket-based communication somewhat simpler. Eventually I wrapped this operator object in another layer to handle parts of the accumulation phase; it currently is used to cache \( R \) values being written to disk, in order to eliminate a file access if the same node is used to operate on the parent in the reduction.

Implementation Details

This implementation uses Java, and C via JNI. Testing a JNI wrapper versus straight C to execute LAPACK’s dgeqrf algorithm on an i5 2.53 GHz Intel MacBook Pro, the increase for matrices of size was less than 6%. While performance is acceptable, there is currently a resulting increase in the application’s memory footprint. Java’s NIO package allows direct allocation of buffers, but this is an optional part of the API which didn’t function on OS X and therefore was made a lower priority for testing on EC2. There are other unexplored avenues for minimizing RAM, such as allocating the buffer in C and passing it back to Java. Otherwise, one could move the data input/output for the operator into C.

There are several known flaws with the current implementation. Most serious is that results have only been verified with \( ||A - Q \ast R|| / ||A|| \) using the Frobenius norm, while the orthogonality of \( Q \) hasn’t yet been verified. However all core calculations are done using standard LAPACK functions, such as DGEQRF, and
casual inspection of the results of applying the $Q$ factors to an identity matrix with the results of using DORGQR on the same elementary reflectors and tau values are promising.

Other flaws affect results in a way that most likely hurt scaling. $QR$ calculations and disk storage do not take advantage of the structure when $R$ values are input. (This is insignificant in the given benchmarks, however there is an obvious trend in performance that needs addressing for significant scaling to be possible.) Benchmarks were done with a significant amount of logging, and assertions were enabled. There is a legacy use of sockets for some communication, and redundant communications of the hostnames to executors.

**Testing**

Testing was done on a cluster of four EC2 nodes of size m1.large. To compare run times across different numbers of nodes, the scheduler was modified to limit the total number of unique slaves it assigned to tasks. For simplicity, and to attempt to compensate for memory issues, only one CPU was used per node. As suggested elsewhere[1], calculating $Q$ should double the overall time; therefore, benchmarking was simplified to focus on the $R$ calculations.

**Results**

This implementation was benchmarked using two numbers for the running time: the overall running time, and the sum of times for all accepted offers from just before the offer was accepted until just after it returned as finished. Further, nodes logged times into six sections:

- Read time for leafs.
- Read time for inner nodes.
- Write time for leafs.
- Write time for inner nodes.
- Calculation time for leafs.
- Calculation time for inner nodes.

Figure 1 shows a breakdown of costs for four nodes as the width of a matrix varies, keeping the overall size constant. The communication time held relatively the same regardless of width, while the thinnest matrices ($n = 16$) had significantly lower calculation times than the widest matrices ($n = 512$). Thus the computation time and the ratio of I/O to computation time mirror each other. Inner node compute times weren’t drawn, as their small values heavily distorted the graph, but were accounted for in the ratios.

Figures 2 and 3 show strong scaling for various widths of matrices. Thinner matrices have noticeably worse scaling, and scaling is noticeably better when considering node-times than overall times.

As benchmarking was done, it became apparent that there was a significant difference between the overall runtime and the time it took to execute on the nodes (whether tracked on the nodes themselves or by the framework). Figure 4 reflects this disparity, showing that the node times relative to the overall reported times had a very similar relationship among different numbers of domains for a different number of nodes. More research is needed, but this is most likely a sign of overhead and may be fallout from having a framework or scheduler with inefficient code. If this can be eliminated, scaling would be improved significantly.

Although it is highly unlikely to be related to the disparity in times mentioned above, there is the issue of how to best evaluate performance of very composable algorithms where overall execution time wouldn’t accurately reflect the resources used. For instance, an MPI reduction using Nexus requires all involved nodes for the duration of the reduction, whereas this framework only uses nodes at distinct times during the reduction.

Finally, Figures 5 and 6 show changes in execution time as the number of domains is increased one a given problem size. The decrease in time for computation of leaf nodes is notable, while one can see the upward trend of the times for inner nodes.

This project is not proof of the viability of TSQR on EC2, however I believe it shows that at least for larger $n$ there is most likely a solution possible even using a file-based approach.
Figure 1: Performance for four nodes as $n$ varies, while overall size is static.

References


Figure 2: Strong scaling for $n = 512$
Figure 3: Strong scaling for $n = 16, 128, 512$

Figure 4: Disparity between overall run time and the sum of all node times as # domains varies.
Figure 5: Performance scaling with change in # domains for one node.

Figure 6: Performance scaling with change in # domains for four nodes.