Initial Performance Analysis of mlRho using Vampir

Abhinav Thota, Robert Henschel and Scott Michael
Scientific Applications and Performance Tuning (SciAPT),
University Information Technology Services,
Indiana University, Bloomington, IN, 47408

October 2, 2012

Contents

1 Introduction 2

2 Scalability Tests on Ranger 2
  2.1 SAGA BigJob framework ............................................. 2
  2.2 Benchmarking mlRho on Ranger ................................. 3

3 Trace Results 5
  3.1 mlRho Trace Configuration ................................. 5
    3.1.1 Top Level Function Summary ............................ 6
    3.1.2 Exclusive time spent in each function ................. 6
    3.1.3 Invocations per function .............................. 7
    3.1.4 Memory Usage ........................................ 8
  3.2 mlRho trace results with a different configuration ....... 8
    3.2.1 Invocations per Function ............................ 9
    3.2.2 FLOPS observed over the course of execution ...... 10
  3.3 How the Cache Hits/Misses add up .......................... 10
  3.4 Initial Performance Improvements ......................... 10

4 Conclusions 11
1 Introduction

This document describes the results of a performance study of the mlRho application. The work was done by the Scientific Applications and Performance Tuning group (SciAPT) of Indiana University’s Research Technologies division. Overall, over 120 man hours have been invested into this project so far.

This document outlines two major areas, first a scaling study of mlRho on the Ranger supercomputer and second a detailed performance analysis using the Vampir tool chain. The scaling study was done together with Mike Lynch’s group in preparation for submitting a request for compute time to XSEDE\(^1\). This request was ultimately successful and resulted in an award of 6.2 million core hours on Ranger. The section outlines how one can run many instances of mlRho efficiently on a large cluster. In section 3, we used Vampir to get a better understanding of the runtime characteristics of mlRho. At the end of this section, we make the first step in improving the performance of mlRho, by compiling it with the Intel compiler instead of the default GCC compiler. This yields a speedup of 10%.

This document represents a first steps in the project to improve the performance of mlRho. We are looking forward to discussing the findings with the authors and the users at IU.

2 Scalability Tests on Ranger

In this section we describe some of the results we have seen from scaling studies on an XSEDE resource, Ranger.

2.1 SAGA BigJob framework

BigJob is pilot-job tool available on many XSEDE resources such as Kraken, Ranger and Lonestar. Many researchers have successfully used BigJob [3] to bundle hundreds of smaller jobs into larger, more manageable groups of jobs [1, 4]. The pilot-job provides a container for many sub-jobs, i.e applications run these sub-jobs through the pilot-job and not the resource manager, but the pilot-job itself is submitted to the queue. A major advantage of this approach is that, by bundling jobs, the overall waiting time at the local resource manager is reduced. This reduces the overall time-to-completion and increases throughput [3].

\(^1\)XSEDE (the successor to TeraGrid) supports 16 supercomputers and high-end visualization and data analysis resources across the country (www.xsede.org).
BigJob maintains a list of processors that are allocated after the job request becomes active. It then assigns these processors to appropriate sub-jobs and manages them. For more details on the BigJob implementation see reference [2]. The main benefit of doing this is that instead of submitting thousands of single core job requests to the queue, we can submit hundreds of large job requests ($\approx 500$ to $5000$ cores) to the queue. This reduces the overall job submissions to the queue and thereby, time spent waiting in the queue. This job size is also more appropriate for many of the XSEDE machines, some of which give preference to larger jobs.

We plan to use BigJob to bundle our single-core serial mlRho simulations. We have integrated mlRho with BigJob and conducted several experiments on Ranger which have shown good performance and scalability. The experiments are explained in the next section.

### 2.2 Benchmarking mlRho on Ranger

We used the BigJob installation already in place on Ranger for our benchmarking tests. While each mlRho simulation is a serial job, we bundled many of these simulations using BigJob. The major focus of our tests was to find out how mlRho scales when we increase the number of concurrent mlRho processes.

<table>
<thead>
<tr>
<th>Organism Type</th>
<th>Size of profile in gigabytes</th>
<th>Distance travelled in 24 hrs</th>
<th>SUs/distance</th>
<th>Measured SUs/d/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>F. cylindrus</em></td>
<td>0.72</td>
<td>297</td>
<td>0.08</td>
<td>0.11</td>
</tr>
<tr>
<td><em>P. ornithorhynchus</em></td>
<td>11</td>
<td>20</td>
<td>1.2</td>
<td>0.109</td>
</tr>
<tr>
<td><em>C. familiaris</em></td>
<td>31</td>
<td>7</td>
<td>3.42</td>
<td>0.11</td>
</tr>
</tbody>
</table>

Table 1: The table lists three organisms. The second column shows the size of the genome profile in gigabytes. The third column shows the distance travelled in 24 hours by each mlRho simulation. The fourth column shows the number of SUs (1 SU = 1 hour of computing on 1 CPU) consumed per distance and the fifth column predicts the number of SUs required per distance per gigabyte. We conclude that the number of SUs required per distance is directly proportional to the size of the genome profile.

We took three different organisms based on the size of their genome. We selected three sizes – small (0.72 GB), medium (11 GB) and large (31 GB). We ran the mlRho program with each of these genomes. We started with 16 concurrent mlRho instances which read from the same data file, and let them run for 24 hours. We repeated this for each of these genomes with 16, 32, 64 and 128 concurrent instances. The results are shown in table 1 and figure 1. Figure 1 shows that the distance traveled, irrespective of the genome
Figure 1: After running 16, 32, 64 and 128 instances of mlRho concurrently using BigJob on Ranger, we see that amount of work done increases linearly with the number instances of mlRho. The y-axis shows the total distance travelled by all the sum of all the instances which is then normalized to 16 instances. The simulations were carried out on Ranger.

size and the number of concurrent instances, increases linearly with number of concurrent instances being run. Table 1 shows the distance traveled by each instance in 24 hours for different genomes. This also shows that the distance traveled is a function of the size of the genome data file. As the reader can observe, distance traveled is directly proportional to the size of the genome data file. We were able to estimate the SUs required per distance per GB of genome data file using each of the three genomes – all three estimates are nearly identical. As the distance traveled increases linearly up to 128 processes, we are confident that we can scale up to \( \approx 1000 \) concurrent processes using BigJob without performance deteriorating.
3 Trace Results

Tracing is specialized use of logging to record information about a program’s execution\(^1\). We traced mlRho 1.10 using Vampir\(^2\) on Mason\(^3\).

Our aim when tracing mlRho was to get an understanding of the execution workflow over the entire runtime. For example, we were able to look at the amount of time spent in different function groups, the amount of time spent in each function and also the number of times each function was invoked, etc. This will help us better focus our efforts when we start looking at the code and attempt to make improvements. But it is usually most efficient to have the developers look at all this information, because they are the people who can make the most sense of this information.

3.1 mlRho Trace Configuration

We used the smallest genome that was available to us, Diatom F cylindrus (718 MB). We ran mlRho for only 1 distance and the runtime was \(\approx 420\) seconds, which increased to \(\approx 550\) seconds when we were doing a trace. We filtered the following functions in this trace, i.e. they were not traced:

1. void lik#
2. void likP#
3. double lOne#
4. double logBinProb#
5. double lTwo#
6. void queuePut#
7. QueueItem *queuePeek#
8. QueueItem *queueGet#
9. Node *addTree#

\(^1\)http://en.wikipedia.org/wiki/Tracing_(software)
\(^2\)Vampir provides an easy to use analysis framework which can display program behavior in detail (www.http://vampir.eu/).
\(^3\)Mason at Indiana University is a large memory computer cluster configured to support data-intensive high-performance computing tasks.
10. int coverage#

11. void fillQueueItem#

We chose to filter functions that were invoked millions of times but contributed only a few seconds to the total runtime. By filtering out these functions, we are able to reduce the overhead caused by tracing.

3.1.1 Top Level Function Summary

Below is the top level function summary, which displays the exclusive time spent in each group of functions:

![Figure 2](image)

Figure 2: Note that the time spent in LIBC I/O is very small, only 3.68 seconds out of nearly 550 s runtime. Also, the time spent in LIBC memory is negligible.

3.1.2 Exclusive time spent in each function

Below is the function summary, which displays the exclusive time spent in each function: (the 31 functions with the largest exclusive run times are shown)
3.1.3 Invocations per function

The image below displays the number of times each function is invoked:

Figure 4: Note that the top four functions are GSL functions.
3.1.4 Memory Usage

The image below shows the memory usage over the course of execution:

![Memory Usage Graph]

Figure 5: The maximum amount of memory used at any point is less than 6 Mbytes.

3.2 mlRho trace results with a different configuration

In the previous section, we have shown results from a trace whose runtime was \(\approx 550\) seconds. But in the images below, we did not filter the function ‘void lik#’, which seemed to be calling some GSL functions, which in turn seem to be responsible for most of the FLOPS. We wanted to locate the function which was responsible for all the FLOPS.

Because void lik# is invoked \(\approx 40\) Million times, as many events are written to the trace file. This is a significant overhead and caused the application to run out of memory towards the end of the run. The total runtime for this run was in excess of 2500 seconds, but since that is not an accurate representation, we will only focus on how each function invocation correlates to the peak FLOPS observed. Also, all the GSL functions that were not directly called by mlRho were filtered out this time. The functions that were filtered out are otherwise the same as before.
3.2.1 Invocations per Function

The image below shows the function summary with the number of times each function was invoked:

![Function Summary Image]

Figure 6: Since the run did not complete, the number of times each function was invoked here and in the previous images might not match.
3.2.2  FLOPS observed over the course of execution

The image below shows the FLOPS observed over the course of execution:

![Graph showing FLOPS over time](image)

Figure 7: We are quite sure that the function invoked during the peak FLOPS is void lik#, which in turn might be calling a GSL function. The peak FLOPS observed is \( \approx 700 \) MFLOPS.

3.3  How the Cache Hits/Misses add up

<table>
<thead>
<tr>
<th>Level</th>
<th>Hits</th>
<th>Misses</th>
<th>Accesses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Level 1</td>
<td>N/A</td>
<td>16.35 G</td>
<td>1.336 T</td>
</tr>
<tr>
<td>Level 2</td>
<td>15.96 G</td>
<td>368.5 M</td>
<td>18.15 G</td>
</tr>
<tr>
<td>Level 3</td>
<td>N/A</td>
<td>35.9 M</td>
<td>320.5 M</td>
</tr>
</tbody>
</table>

Table 2: The table shows the total Level 1, 2 and 3 cache accesses, hits and misses. Please note that the numbers may not add up as they come from different runs. The misses are happening only 1% of the time in Level 1 and the vast majority of misses are happening here. Level 2 and Level 3 misses are happening between 10 and 20% of time. Overall, cache misses does not appear to be an issue.

3.4  Initial Performance Improvements

Depending on the machine architecture and operating system, using the right compiler will instantly give us some performance benefits. On each of the machines that we used to test mlRho, we tried different compilers and compiler flags. The results are shown below in table 3.
### Table 3

<table>
<thead>
<tr>
<th>Compiler</th>
<th>Ranger</th>
<th>Mason</th>
</tr>
</thead>
<tbody>
<tr>
<td>GNU</td>
<td>450</td>
<td>414</td>
</tr>
<tr>
<td>Intel</td>
<td>405</td>
<td>364</td>
</tr>
<tr>
<td>PGI</td>
<td>418</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The table shoes mlRho runtimes on different machines with different compilers. The runtimes are in seconds. Compiling with the Intel compiler gave us a 10% speedup compared to GNU/PGI compilers on Mason and Ranger. Future work could include building GSL with Intel compilers.

## 4 Conclusions

In this document, we broadly presented the runtime characteristics of mlRho. We have barely scratched the surface of what is possible, especially taking into account that we did not spend any time deeper understanding the algorithmic idea behind mlRho. We hope that this work is valuable to the authors, and that we can collaborate more in the future. We are currently working in a very similar way with the Trinity developers. SciAPT contributes performance analysis information to the project, which in turn is used by the Trinity developers to improve the runtime performance of Trinity. In parallel we are providing input and writing joint papers. The same approach could be taken with mlRho.

Looking at the various performance counters and numbers, we did not find an obvious place for optimization. After getting a better understanding of the code, it will definitely be possible to come up with ideas how to make the code run faster. One of the things worth trying is compiling GSL using more aggressive optimization levels and a different compiler. Another approach is substituting GSL with the Intel MKL library. While we have not identified I/O as a bottleneck, it is also worthwhile noting that right now, the code uses very little memory. If algorithmic optimizations can possibly be applied at the expense of memory, it is definitely worth looking into. This document represents a first step in a potential project to improve the performance of mlRho. We are looking forward to discussing the findings with the authors and the users at IU.


References


