Towards Modeling a Complex Geological Simulation

David Apostal
University of North Dakota
Grand Forks, ND 58202 USA
david.apostal@und.edu

Sara Faraji Jalal Apostal
University of North Dakota
Grand Forks, ND 58202 USA
sara.farajijalal@und.edu

Ron Marsh
University of North Dakota
Grand Forks, ND 58202 USA
rmarsh@cs.und.edu

Travis Desell
University of North Dakota
Grand Forks, ND 58202 USA
tdesell@cs.und.edu

ABSTRACT

Data motion is a significant factor affecting runtime performance. Data-intensive applications are subject to the effects of data motion more so than other applications. This research uses abstract machine models to calculate runtime performance expectations for a geological simulation program. The models are based on the time to execute double-precision floating-point instructions and the time to load operands and store results for those operations. Two extremes of cache memory are considered and provide expected bounds for the program's runtime performance. In one case, cache memory is unlimited; once a datum is read into cache, it is always available. In the other case, the cache is extremely limited and each datum is removed from cache memory after it is used once. A model for inter-process communication is also incorporated.

This research shows that simple models can provide accurate expectations for runtime performance for some operations. For one operation studied herein the observed runtime performance only exceeded the expected upper bound in one case by 2%. The models can become less accurate if memory utilization is high. This may occur if only one core is used or if a change to an algorithm results in larger data structures.

Keywords: abstract machine, performance analysis, conduction, heat flow

1. INTRODUCTION

One use of geothermal energy is in the mining of hydrocarbon deposits. The geothermal energy can replace natural gas for heating water used during the extraction process.

Heat flow data can be used to estimate the subsurface temperatures of layers of rock with known thermal conductivity.

SpringSim-HPC, 2016 April 3-6, Pasadena, CA, USA
©2016 Society for Modeling & Simulation International (SCS)

Subsurface temperature can be used to create maps that describe the extent, depth, and temperature of geothermal resources. The maps help to identify regions with potential for geothermal exploitation.

Simulations associated with the study of geothermal resources can be data intensive applications. A well-established geological model has been used to measure heat flow beneath the Earth's surface [5][6]. Arc, a software implementation of the geological model, provides the basis for the abstract machine models in this study.

Arc was originally written in Fortran. The Fortran code was ported to C++ and additional features were added as part of a student project. The Arc software can measure heat flow across a two- or three-dimensional rectilinear grid of temperatures. A variation of the C++ version of the program was created to run on distributed memory parallel computers.

Both the desktop and parallel versions of Arc use a finite differencing approach to calculate heat flow during a period of simulation. Given the current temperature at a point, the program calculates what the temperature will be after a time step. A point's next temperature is stored in a separate grid data structure so that the point's current temperature can be used as input when calculating the next temperatures for other near-by points. This process is performed on all points in the model repeatedly for the duration of the simulation.

Recently it was observed that some nodes in the parallel version of Arc were idle while other nodes completed computations. The code was measured and changes were made until there was an improvement in the runtime performance. As a result the Arc program was modified to use non-blocking MPI point-to-point communication instead of blocking MPI function calls. This resulted in a performance increase of 28%, depending on environment [2]. However, there was no analysis to determine how fast the program or parts of it might be able to run. Simple abstract machine models can provide performance expectations for key operations, and help developers make effective use of their time when tasked with finding performance gains.

2. RELATED WORK

Effective use of powerful high performance parallel computer systems requires paying attention to many parts of the application. This can be done with fairly simple model-based analysis that provides a performance expectation. Models can be developed to provide expectations for how much time an operation might take. When compared against observed runtimes, models can help to identify performance bottlenecks. Models can also be developed based on proposed algorithms or code changes to help predict the performance impact of changes.

Hager *et al.* used simple models to predict performance and power [7]. They constructed a model to predict the energy required to solve a problem and then developed guidelines for energy-efficient execution of parallel program.

Zhong *et al.* modeled the performance of scientific programs in order to execute them on hybrid (multicore and GPU) platforms [12]. Algorithms based on the models were shown to support data partitioning in heterogeneous distributed-memory systems.

Simple models for sparse matrix-matrix multiplication were used by Scharpff *et al.* in [9].

Ang *et al.* used abstract machine models to explore architectures that may lead to exascale computing levels [1]. The models help hardware architects and software developers communicate and study new approaches that minimize data movement.

3. APPROACH

In order to understand the performance of the parallel Arc geological program, two simple models based on the existing code were created. An initial model was created to give an expectation of the time required to calculate the temperature differences between a grid point and its neighbors. This provided some guidance when developing the model of conductive heat flow. The conductive model was built upon the temperature difference model. Both models were based on the time cost of floating-point operations, the time to load operands to those operations, and the time to write results to memory.

The models do not include several items that are present in the existing code. Overhead associated with loops, calls to subroutines, and conditional statements have been omitted from the models. Also, the cost of measuring times of certain operations was removed from results.

Two variations of the models were created. A limited cache memory model provided an expected upper bound on the times for calculating temperature difference and conductive heat flow. With limited cache data would be flushed from cache in order to make room for the data required by the next calculation. Also, an unlimited cache memory model provided a lower bound on the expected performance of each operation.

3.1. Modeling Temperature Difference

Heat flows from warm areas to cooler areas. In Arc the temperatures above, below, and to each side in a 2-D grid affect the conduction at a given point. A stencil for temperature difference at one point is shown in Figure 1. The center point is dependent on its neighboring points. The difference in temperature between two points is adjusted by the conductivity and heat production at each point and the distance between the points. In all there are 10 floating-point operations in the current code to calculate temperature difference at one grid point. In a 3-D grid, a stencil for temperature difference includes points from adjacent slices.

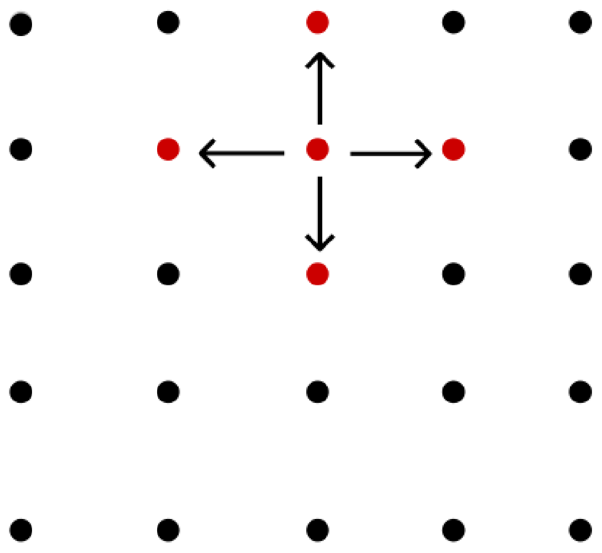

Figure 1. Heat flow at one grid point in a 2-D grid is based on its temperature difference from four neighboring points.

Reading the operands for those 10 floating-point operations and writing the results are also parts of the abstract model for temperature difference. These are examples of data movement, an important aspect to modeling performance. Two abstract machine models were considered. One model has unlimited cache memory; if a grid point is loaded into cache as part of a temperature difference calculation, that point's temperature will remain in cache memory until the program ends. The other model has extremely limited cache memory where no data remains in cache after it is used to calculate a temperature difference.

Assuming unlimited cache memory, the geological simulation code requires on average one double-precision (8-bytes) read operation for each point in the grid. Also, the next temperature of a point is written two times in the Arc conduction code. In total there are three double-precision read or write operations.

A worse-case scenario would be if the data for calculating temperature differences were flushed from cache immediately after the temperature difference at a point had been calculated and written to the next temperature grid. In this situation there would be five double-precision reads for the point in question plus two stores to the next temperature grid. This scenario requires 7 double-precision read or write operations.

3.2. Modeling Conduction

The conduction model builds on the model for temperature difference. When calculating conduction, the difference in temperature between a point and its neighbors is adjusted by conductive and radioactive heat production properties and the distances between points. In all there are 37 floating-point operations in the current code to calculate conduction at one grid point. However, the memory access pattern for conduction is more complicated than the stencil shown previously. With unlimited cache memory there are seven read operations and two write operations. With limited cache there are 23 reads and two write operations.

In fact, the 37 floating-point operations and nine or 25 I/O operations only applies to interior points of the grid. Points at the four corners are dependent on only two neighbor points, and require only 22 floating-point operations. The other non-corner edge points are dependent on three neighbors, and require 30 floating-point opertions. There are also reductions in the number of I/O operations for these two sets of points. However, in this study the modeling is intended to give a general expectation of performance and not a precise estimate. Therefore, the 37 floating-point operations with nine or 25 I/O operations is acceptable.

3.3. Modeling Inter-process Communication

When Arc is run with using parallel processes, the program divides the grid as evenly as possible among the processes. Each process calculates conductivity for a subset of the grid points. Consider a grid with 1000 rows. If there are two processes working together, each process will calculate conduction for 500 rows. If there are four processes, each will work on 250 rows. However, if there are 16 processes, the first 15 processes will work on 63 rows and the last process will work on the remaining 55 rows. Arc uses asynchronous MPI messaging to exchange data among pairs of processes.

With parallel processes conduction for some grid points depends on temperature data managed by another process. In other words the stencil crosses the process boundary. Processes must exchange rows of data with their neighboring process or processes. With just two proesses and a 1000 row grid, the first process will send its last row (row 500) to the second process; the second process will send its first row (row 501) to the first process. This is called a halo exchange. With four or more processes the first and last process exchange halo rows with their one neighbor process. The middle processes must exchnge data with two neighbor processes.

4. EXPERIMENTAL ENVIRONMENT

The programs used in this work were run on the Stampede supercomputer. Each Stampede compute node includes two Intel Xeon E5-2680 2.7 GHz processors. The nodes have 32 GB of main memory and 20240 MB of cache memory according to the /proc/cpuinfo file on a compute node. The Stampede nodes run CentOS 6.3 and are connected by an FDR Infiniband network. The Extreme Science and Engineering Discovery Environment [10] project provided access to Stampede.

Programs were compiled with Intel compiler version 15.0.2. MPI programs used Intel MPI (impi) version 5.0.2. Unless otherwise noted, the compiler options used were -O3 and -xHost. The O3 option is recommended for applications that have loops that heavily use floating-point calculations and process large data sets. The xhost option to the Intel compiler enbles the highest level of vectorization available on the processor. In the case of the Intel Xeon (Sandy Bridge) processors in Stampede, Streaming SIMD Extensions (SSE), SSE2, SSE3, SSE4.1, SSE4.2, and Advanced Vector eXtensions (AVX) are all available [3].

The better-case and worse-case performance models are functions of the time for a floating-point operation (c) and the time for read (r) or write (w) operations. The number of floating-point, read, and write operations determines the performance expectation for each model.

The values for c, r, and w were determined from the Stampede supercomputer. The inverse of the processor frequency was used for the value of c. The Xeon E5 processors on Stapmede run at 2.7 GHz. Therefore, c = 3.7e-10 seconds. A turbo mode for the E5 processors may be available depending on factors including workload, the number of running cores, the estimated power consumption, and the processor temperature. If the Stampede processors ran at full turbo mode (3.5 GHz), then c would equal 2.86e-10 seconds.

A well-known benchmark program was used to test if the inverse of processor frequency was reasonable for estimating the time for floating-point operations. The LINPACK benchmark program solves a system of linear equations. Over 97% of the instructions in the two main subroutines are floating-point add or multiply operations [4]. The source code [11] was compiled with -O3 and -xHost options and was run on a single core of a Stampede compute node using a 200x200 matrix. The average of the KFLOPS reported by the benchmark was 2,916,673.486 KFLOPS. This is roughly 3.43e-10 seconds per floating-point operation. The inverse of processor frequency estimate is about 1.08x slower than the LINPACK benchmark. A comparison of LINPACK performance and the ”inverse of processor frequency” models for the Xeon E5 and E5 Turbo Mode is shown in figure 4.

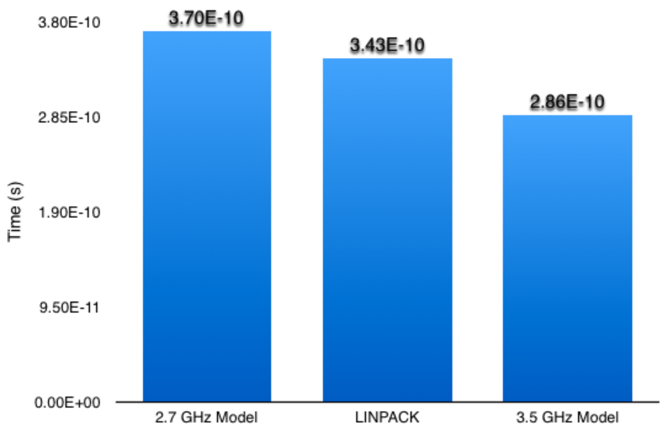

Figure 2. The inverse of CPU frequency is a close approximation of the LINPACK results.

The values for r and w can be derived from a memory bandwidth benchmark like STREAM. The STREAM benchmark provides a measure of the sustainable memory bandwidth over large arrays of double-precision data [8].

The arrays are set to a length that is at least four times the size of the largest cache. This ensures that the results are reflective of sustained memory access, including cache misses, and not sustained cache access. The STREAM Copy kernel reads a data element value from one array and writes the value to another array at the same index as shown in Algorithm 1.

```
for $j \leftarrow 0$ to $N$ do
    $c[j] \leftarrow$a[j]
end
```

Algorithm 1: The STREAM Copy kernel algorithm

The STREAM benchmark was compiled arrays with 20 million double-precision elements. This is more than large enough for a Stampede compute node’s largest cache size of 20480KB. The observed bandwidth on Stampede was 7665.4 MB/s on a single core. That is the data transfer rate. A single r or w is an operation on a double-precision (8-bytes) datum. Therefore, $r = w = 2.09e\text{-}9$ seconds. This may be found by:

1. Convert to bytes per second,
2. Convert to doubles per second,
3. Divide by two,
4. Invert the result.

5. RESULTS

The Arc simulation was run with an input file containing 1000 x 60 data points. The runtime measurements started after the data file was read. The program simulated 20 million years with each iteration taking 10,000 years. Runtime performance data was gathered using from one to 16 MPI tasks on the cores of a single Xeon E5 node. With one core the number of temperature difference or conduction operations is 120,000,000. The number of operations decreases by half as the number of cores doubles. All observed performance runtime results are the averages of five runs.

5.1. Temperature Difference Models

The abstract machine model for temperature difference with unlimited cache memory is in Equation 1. Using (1) a temperature difference operation for a single grid point might take 9.97E-09 seconds. The total time for conduction using a single core is expected to be 1.196 seconds.

$$Time = 10c + r + 2w \tag{1}$$

The abstract machine model for temperature difference using limited cache memory is in Equation 2. Using (2) a temperature difference operation for a single grid point might take 1.83E-08 seconds. The total time for conduction is expected to be 2.196 seconds when using one Stampede core.

$$Time = 10c + 5r + 2w \tag{2}$$

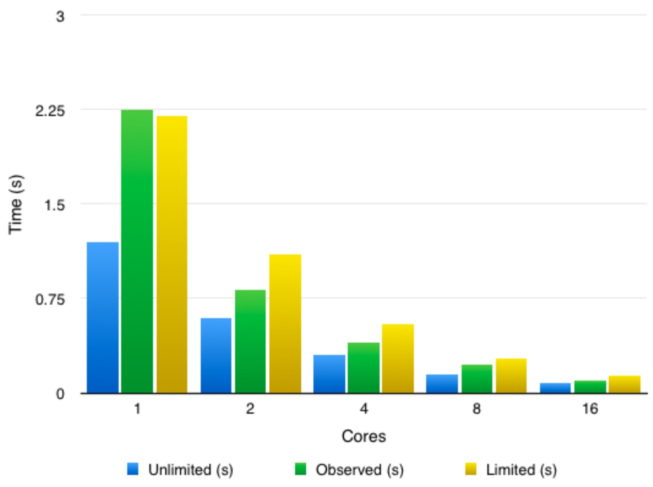

Figure 3. The two models almost always provide upper and lower performance bounds for the Arc temperature difference operations.

Figure 3 shows the relative differences between the observed temperature difference results and the models with unlimited and restricted cache memory. Table 1 shows the observed runtimes for temperature difference operations exceed

Table 1. The details of the observed runtimes and expected performances for the temperature difference operation. All times are in seconds.

Cores	Unlimited Model	Observed	Limited Model
1	1.196	2.250	2.196
2	0.598	0.820	1.098
4	0.299	0.397	0.549
8	0.150	0.222	0.275
16	0.075	0.094	0.137

the limited cache model by 2% when using a single core. All other observed runtimes are between the expected bounds of the unlimited and limited cache memory models.

5.2. Conduction Models

The abstract machine model for conduction with unlimited cache memory is in Equation 3. Using (3) a conduction calculation for a single grid point can be expected to take 3.29E-8 seconds. Using a single core to perform Arc conductions for 2,000 iterations across 60,000 grid points might take 3.948 seconds.

$$Time = 37c + 7r + 2w \tag{3}$$

The abstract conduction model with limited cache memory is in Equation 4. Using (4), a single conduction operation might take 6.59E-8 seconds. Using the limited model for conduction 120,000,000 times on Stampede can be expedted to take 7.908 seconds.

$$Time = 37c + 23r + 2w \tag{4}$$

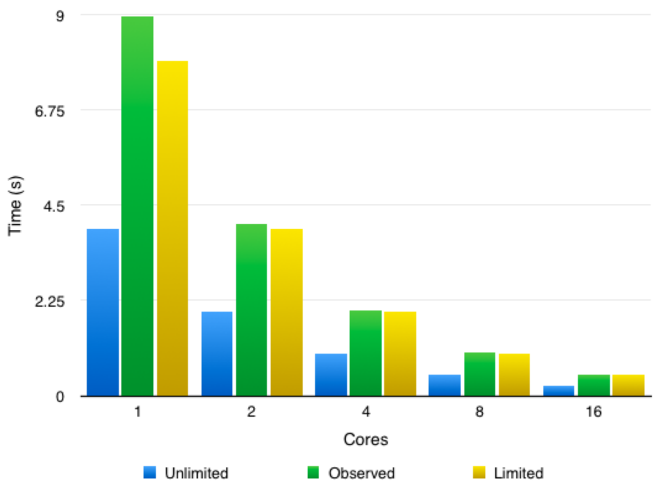

Figure 4. The two models almost always provide upper and lower performance bounds for the Arc temperature difference operations.

Table 2. The details of the conduction runtime and expected performances before any changes to the code. All times are in seconds.

Cores	Unlimited Model	Observed	Limited Model
1	3.948	8.96	7.908
2	1.974	4.06	3.954
4	0.987	2.02	1.977
8	0.494	1.01	0.989
16	0.247	0.50	0.494

Figure 4 shows the relative performance of the conduction runtimes and the two variations of the conduction model. Table 2 shows that the observed runtimes for the conduction operations are all outside of the expected bounds of the unlimited and limited cache memory models. The observed runtime exceeded the limited cache model by 13% when using a single core. For all other observations the expectation of the limited cache model is exceeded by 3% or less.

5.3. Inter-process Communications

A simple model using latency, bandwidth, and message size can provide an expectation for inter-process communication. Equations 5 shows this model.

$$TransmissionTime = latency + size \div bandwidth \tag{5}$$

Although information on latency and bandwidth for Stampede is not readily available, it is possible to calculate these values. All that is needed is a simple ping-pong program that measures the time required to send and receive a message between two processes. The program is rerun using increasingly larger messages. As with the geology simulation program, the ping-pong program uses non-blocking MPI communication functions.

With the times that were measured from the ping-pong test using different sized messages, one can calculate a best-fit regression line. The inverse of the slope of the line is the bandwidth. The value for latency comes from the time to send and receive a message of size 0 bytes.

A bandwidth of 3.6 GB/s for Stampede was calculated using the ping-pong data in Table 3 and Figure 5. The latency for Stampede was found to be 1.41e-6 seconds.

Using (5) one can calculate the expected time to send a single message. However, processes running Arc exchange halo data on every iteration of the program. In fact, Arc sends an 8-byte message 60 times per iteration (one message per column value). This costs 1.393344e-5 seconds. For 2000 iterations, the total cost of a single halo exchange is 0.0279 seconds.

5.4. Changes to the Code

There were two characteristics of the code that stood out during the development of the conduction performance

Table 3. The ping-pong results with different message sizes.

Message Size	Time (s)
0	1.41E-06
4	2.13E-06
16	2.15E-06
64	2.74E-06
256	2.66E-06
1024	3.85E-06
4096	7.48E-06
16384	1.05E-05

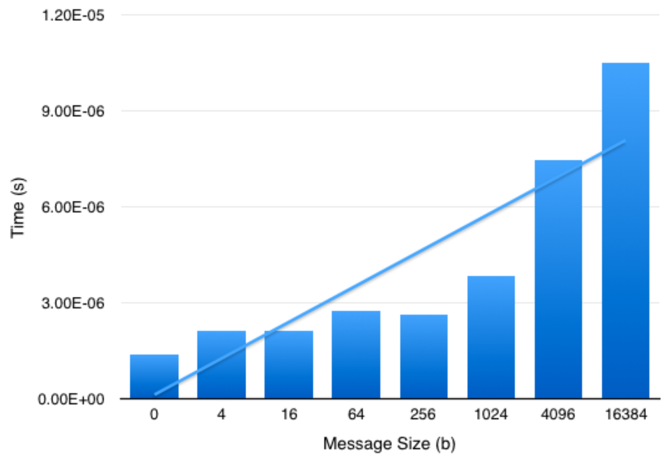

Figure 5. Results of ping-pong test on Stampede for different message sizes.

model. Each may be contributing to the runtime performance observed.

First, the next temperature array for each grid point was written to memory three times per conduction operation in different subroutines. If this write is going through to main memory rather than just updating the cache memory, then that may account for some of the runtime performance. A temporary next temperature scalar variable was created and used in all but the final update to the next temperature array.

It was also observed that values associated with conductivity and heat production were accessed indirectly through lookup tables. It was thought that these were artifacts of attempts to minimize the size of the input file. The file input process was modified so that these values would be accessible without using lookup tables. This increased the total memory required by the program. Equations 6 and 7 show the updated best-case and worst-case conduction models.

$$Time = 37c + 20r + w \tag{6}$$

$$Time = 37c + 5r + w \tag{7}$$

The conduction runtime results after making changes to the code are shown in Figure 6 and Table 4. It appears that the code changes improved performance in all cases except when a single process was used.

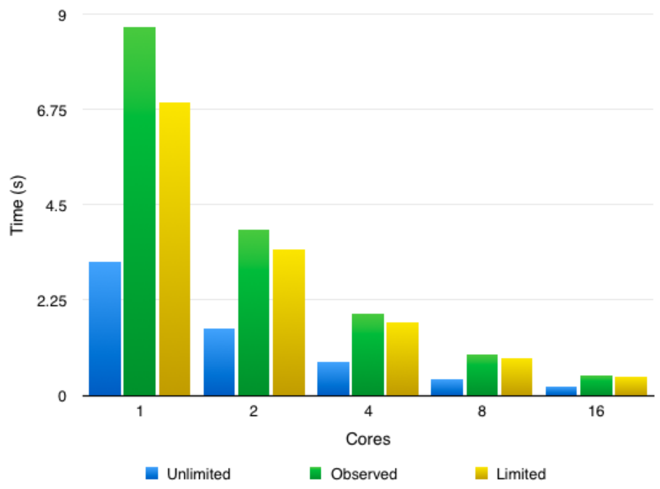

Figure 6. Observed and modeled performance of conduction after code changes.

Table 4. The details of the observed runtime and modeled performances after changes to the code.

Cores	Unlimited Model	Observed	Limited Model
1	3.144	8.69	6.912
2	1.572	3.92	3.456
4	0.786	1.94	1.728
8	0.393	0.97	0.864
16	0.197	0.47	0.432

The performance of the Arc conduction operation improved between 3% to 6% after the code was changed. However, Table 4 shows that the observed runtimes for conduction operations are still all outside of the expected bounds provided by the two abstract machine models. The observed performance is between 26% (one core) and 9% (16 cores) slower than the limited memory abstract machine.

A possible performance improvement was also identified in the MPI halo exchange code. As mentioned previously, the code sends 60 eight-byte messages for each iteration. Considering again Equation (5), latency is significantly larger than the bandwidth factor. So, the cost of latency is included in each PMI message. If a single message of 60 eight-byte values is sent, the cost of latency is reduced significantly. With this improvement the cost of sending a single message with 60 eight-byte values should be 3.6344e-7. For 2000 iterations, the total cost of a halo exchange with one other process is 0.000727 seconds.

6. DISCUSSION/CONCLUSION

In this work we considered two major operations within the Arc geological simulation program. Abstract machine models based on temperature difference and conduction operations were created and evaluated for accuracy. The runtime performance of the temperature difference and conduction operations were very close to the expectations of the limited memory models. The runtime performances exceeded the limited memory expectation by 2% and 13% respectively, for the temperature difference and conduction operations when using one core.

After modifying the conduction operation and removing memory accesses, the simulation performance improved slightly (from 3% to 6%). A side effect of these changes was that more memory was required for the conduction data structures. The observed times were approximately 1.26 to 1.09 times slower than the expected times of the limited memory model. For such simple models, these differences were considered acceptable.

These were combined with a simple model for inter-process communication. The models combined to give an expectation of how long the geological simulation might run. In fact, the simulation ran much slower than the combined models. More analysis of the inter-process communication code is needed to understand how to improve the networking model.

Data motion tends to dominate the performance of the conduction operation and the entire Arc simulation. However, it may be possible to improve the accuracy of the models. The simple models in this study only considered the number of loads and stores in an operation. The models were less accurate when using just one core or when more memory was required by Arc. Future work will attempt to understand the extent, if any, that memory utilization affect runtime performance. It may be possible to make the models more accurate when running on systems with different amounts of memory or when the size of the data grid changes.

REFERENCES

1. J. A. Ang, R. F. Barrett, R. E. Benner, D. Burke, C. Chan, J. Cook, D. Donofrio, S. D. Hammond, K. S. Hemmert, S. M. Kelly, H. Le, V. J. Leung, D. R. Resnick, A. F. Rodrigues, J. Shalf, D. Stark, D. Unat, and N. J. Wright. Abstract machine models and proxy architectures for exascale computing. In *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*, pages 25–32, Nov 2014.

2. David Apostal, Kyle Foerster, Travis Desell, and Will Gosnold. Performance improvements for a large-scale geological simulation. In *Procedia Computer Science*, volume 29, pages 256–269, June 2014. 14th International Conference on Computational Science (ICCS 2014).

3. Intel Corp. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, September 2015.

4. Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15(9):803–820, 2003.

5. W.D. Gosnold. Basin-scale groundwater flow and advective heat flow: An example from the northern great plains. *Geothermics in Basin Analysis*, pages 99–116, 1999.

6. Will Gosnold, Jacek Majorowicz, Rob Klenner, and Steve Hauck. Implications of post-glacial warming for northern hemisphere heat flow. *Transactions of the Geothermal Resources Council*, page 12, 2011.

7. Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, 28(2):189–210, 2016.

8. John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.

9. T. Scharpff, K. Iglberger, G. Hager, and U. RÃ$\frac{1}{4}$de. Model-guided performance analysis of the sparse matrix-matrix multiplication. In *High Performance Computing and Simulation (HPCS), 2013 International Conference on*, pages 445–452, July 2013.

10. John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkins-Diehr. Xsede: Accelerating scientific discovery, Sept.-Oct. 2014.

11. Bonnie Toy, Will Menninger, and Jack Dongarra. Linpack benchmark (c source code), 1994. available at: http://www.netlib.org/benchmark/linpackc.new.

12. Ziming Zhong, V. Rychkov, and A. Lastovetsky. Data partitioning on heterogeneous multicore and multi-gpu systems using functional performance models of data-parallel applications. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 191–199, Sept 2012.