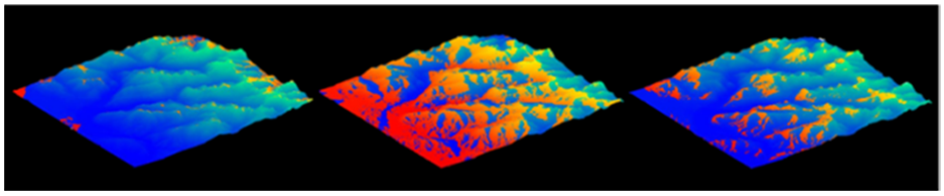
HPC Project: Shade Model Optimization

Troy Axthelm, Jared Baker, Matthew J. Brazell, Nels Frazier

May 12, 2015



1 Introduction

The Shade Model program is designed to model shade given an elevation and location information for a selected land area. With this information a model of watershed can be created for primary use by CI-Water Project for predicting locations for forest fire remediation. The uploaded land area of interest can simply be downloaded off of GIS, the user can select a time interval over the course of a specific day of the year, by default the time increment is 15 minutes. The GPS coordinates of the land area as well as day of the year provide solar angle along with the topography of the land indicates if an area is shaded or not. Currently the program only indicates if an element of the land area is shaded or not shaded, this is represented in the output file as '0' for shaded and '1' for non-shaded. No information of temperature or partial shading is provided and may be a future goal however currently this information alone is sufficient for CI-Water Project's needs.

The original program was created by Troy Axthelm and Jingyu Li funded by CI-Water Project. For more information and location of the open source code see the link below:

https://sites.google.com/site/uwyoshademodel/

As a project for a graduate coarse: Designing and Building Applications for Extreme Scale Systems lead by instructor William Gropp and Professor Craig Douglas, our group intends to make improvements in performance leading to faster simulations using fewer computational resources.

1.1 Initial Benchmarking

1.1.1 Parlib MPI

To test functionality of different Message Passing Interface (MPI) routines and avoid conflicts with Parlib, we decided to remove the Parlib dependency in this particular case. While there are advantages to having an interface library, we are exploring different methods with MPI which are not yet implemented in Parlib, such as MPI derived data types and structures. To effectively pass structures in MPI as an array data type, we needed to utilize a method which was not available in the Parlib implementation. This was a primary driving factor for removing the interface code for this project. There was also a possibility that we may have implemented MPI-3 routines which were not also available in the Parlib implementation.

1.1.2 MPI

An initial profiling of the code in parallel was used to determine trouble areas in the code where performance improvements could be made. The result of using Allinea profiler showed that over 80% of the computing time was spent in a loop in main.c, the specific code for this problem area is shown in Appendix A

1.2 Performance Model

A performance model was created to give us a rough idea on the expected timing in the area described above where over 80% of the computation time is spent. To verify that this is an area that can be improved a simple performance model is created and compared with timing results of the actual code over this loop. This performance model is only to demonstrate an upper limit on performance. The model is defined as:

total time $= n * m * (8r + 5w + 2c)$

assuming the time for write is equal to the time for read then:

total time $= 8(\frac{bytes}{element}) * n * m * (13 * r * \frac{1Mbytes}{1e6bytes} + 2c)$

with n=1966 and m=2054, using the Stream performance the best Rate shown in table 1 an estimate for the performance of a read is 15564.0 MB/s, time for a read is $\frac{1}{15564.0MB/s}$

The compute nodes on Mt. Moran used for testing use Intel chips with 2.6 GHz which gives an estimate of $c = 2.6e9$ flops. The estimated time based on this performance model gives t=0.09159 seconds if we assume 1-4 bytes moved per 104 bytes for each floating point operation gives 0.96% to 3.8% of peak performance giving an estimated time of t=$8.7926e-4$ to t=0.0035 [Sec]

Table 1: Stream Results on Mt. Moran

Function	Best Rate MB/s	Avg time	Min time	Max time
Copy:	14078.4	0.011393	0.011365	0.011444
Scale	14098.2	0.011366	0.011349	0.011390
Add:	15564.0	0.015445	0.015420	0.015477
Triad:	15379.7	0.015635	0.015605	0.015659

1.3 Coding Improvements

1.3.1 Replace equations and inline math functions with constants

An easy method to make small performance improvements as well as make the code more easily readable is to replace inline math functions with constants. This improvement is especially important if it is in a loop. An example of this is the need for π in some trigonometry calculations. If π is defined upfront then it can be used multiple times in the code instead of relying on a library look up to calculate π. This is seen in the file "landReader.c" where $4 * atan(1) = 3.141592653589793 = \pi$.

1.3.2 Linear Memory

To make improvements to the part of the code where the majority of the compute time is taken as discussed above and shown in Apendix A, by improving the way the array's "outData.elevation" is read and "ourData.shading" is written will make significant improvements to performance. As seen in the performance model, a large amount of time is spent reading and writing data from memory. The original code used a 2D array implemented as a list of pointers to pointers of doubles. When reading and writing to memory, this can lead to many cache misses. So to help improve cache efficiency, this data structure was linearized so that all data access could be accomplished from a single contiguous memory section.

1.3.3 Threading

The use of for loop style applications many times generally grant themselves to utilize a shared memory programming paradigms. With this, a small section of the code was instrumented with OpenMP multi-threading. This occurs in the file reader portion where a the original data structure elements are calculated. Even though this was only a small section of the code in main.c, it did provide an area for experimentation of using threads in the application. With the threading model, performance was increased by approximately 40% using 4 threads. While this doesn't like very much gain, the 40% was a global improvement over the entire file reading and data structure element calculations which the file reader dominated the cycles.

1.3.4 MPI Routines

The original version of the code had each MPI process reading each data. By changing this system to have one process read the data and use MPI_Bcast to "broadcast" the data to all other processes this effectively allowed the code to pass structures in MPI as an array data type.

1.4 Results

Using one compute node on Mt. Moran some timings of various stages of optimization show the improvements made as shown in table 2.

To compare the improvements made with the predicted performance model, the maximum and minimum times in the computationally expensive loop described above are provided in table 3. Since the code can exit out of the loop early the resulting land slope is greater than the solar altitude (indicated a shaded area) it

Table 2: Code Timing Results

-	Time [Sec]	Ratio w.r.t. Base Code
Base Code	582.4	1
Entail Math Ops., Less Parlib	573	0.984
Memory Linearization	532	0.914
Threading	79	0.136

is more informative to compare the maximum observed time in the loop. The base code maximum time is on the same magnitude as the performance model prediction where as the improved code is 93.5% faster.

Table 3: Inner Loop Timing Results

-	Base Code Time [Sec]	Improved Code Time [Sec]	Ratio w.r.t Base Code
min	4e-06	4e-06	1
max	1.08e-04	7e-06	0.0648

1.5 Pit Falls

While group projects are challenging having multiple people working on a coding based project becomes an increasingly difficult to manage. Using code repositories can help to mitigate code rework and provides the most updated version to all group members however this still relies on diligent users.

While this group did achieve the main goal of producing a faster code for the Shade Model however there exists a problem within one of the updates made, making this code unable to reproduce 100% repeatable results with the base case. Figure 1 below shows a region on the East part of the grid of no shade when according to the Base code there should be shade.

This error could have been avoid if more test cases were run while also using more commits to work through the changes slower and more carefully.

A B C

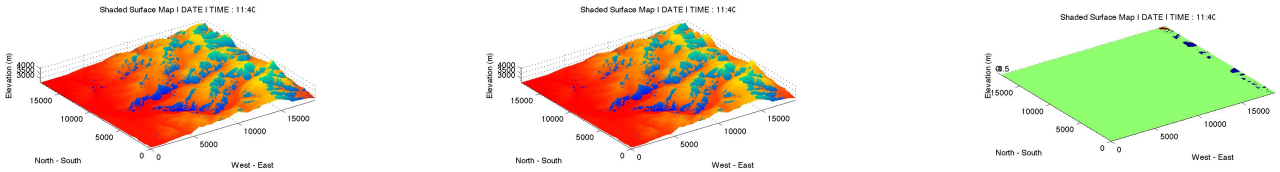


Figure 1: ResultA: Base Test Case (No errors), B: New code missing shaded section, C: New code with elevation removed showing difference between base test case and new code

1.6 Conclusion

The Shade Model is a useful tool for research scientists however improving the performance of the code would allow not only quicker computation in general but also allow larger land areas to be analyzed at once while using fewer computational resources. Using techniques learned from the HPC course this group was able to pin point areas of the code to improve, establish a performance model to give a prediction of possible performance expectation as well as ideas on not only using MPI but also OpenMP to take advantage of multi-core chip capabilities as well as HPC resources. The updated version of the code does not provide an accurate solution however the main goal of our project was achieved by significantly increasing performance (86 % faster than the base case). Not only did our group learn how to implement in an actual application of the HPC course but also how to prevent future downfalls when working in a group setting.

A
Reference Main.c Code

```
//Step along azimuth until off of grid or the observer point is determined to be shaded
while (tempX >= 0 && tempX <= (numCols-1) && tempY >= 0 && tempY <= (numRows - 1))
{

roundTempX = (int)round(tempX);
roundTempY = (int)round(tempY);

if(roundTempX != i ||  roundTempY != j)
{
tempSlope = (ourData[roundTempY][roundTempX].elevation - ourData[i][j].elevation)/sqrt(lenX*lenX + lenY*lenY);

if(solarAlt <= tempSlope)
{
ourData[i][j].shading[k] = 1;
break;
}
}

lenX += stepX;
lenY += stepY;

tempX += stepTempX;
tempY += stepTempY;

}
```