PARALLELIZATION FOR GEOPHYSICAL WAVEFORM ANALYSIS

A Senior Honors Thesis

by

DEREK EDWARD KURTH

Submitted to the Office of Honors Programs
& Academic Scholarships
Texas A&M University
in partial fulfillment of the requirements of the

UNIVERSITY UNDERGRADUATE
RESEARCH FELLOWS

April 2002

Group: Computer Science

PARALLELIZATION FOR GEOPHYSICAL WAVEFORM ANALYSIS

A Senior Honors Thesis

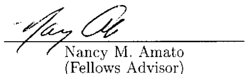
by

DEREK EDWARD KURTH

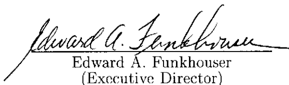
Submitted to the Office of Honors Programs
& Academic Scholarships
Texas A&M University
in partial fulfillment for the designation of

UNIVERSITY UNDERGRADUATE
RESEARCH FELLOW

Approved as to style and content by:



Nancy M. Amato
(Fellows Advisor)



Edward A. Funkhouser
(Executive Director)

April 2002

Group: Computer Science

526-C
18

ABSTRACT

Parallelization for Geophysical

Waveform Analysis. (April 2002)

Derek Edward Kurth
Department of Computer Science
Texas A&M University

Fellows Advisor: Dr. Nancy M. Amato
Department of Computer Science

The use of parallel processors can be a very effective method for improving the running time of large or complex calculations. The Standard Template Adaptive Parallel Library (STAPL) is being developed by Dr. Lawrence Rauchwerger at Texas A&M University to aid the parallel programmer by providing standard implementations of common parallel programming tasks.

Our research involves using STAPL to apply parallel methods to a problem that has already been solved sequentially: Seismic ray tracing. In short, we are modelling the paths of seismic waves as they travel through a known earth model (i.e., an earth region whose properties we know how to model mathematically). By studying the solution to this problem, it is hoped that a more difficult problem may one day be solved: Given the source and end locations of seismic waves, their travel times from source to end location, and their initial and final amplitudes, determine the properties of the earth region through which they traveled.

Parallel methods apply well to this problem and are important because, for complex earth models, the computation costs can grow very large. Our early results have shown that a parallel version of the ray tracing code running on 8 processors ran 5 times as fast as the original sequential version. We hope to optimize the program for multiple platforms and hardware configurations. To assist in the design and

understanding of the algorithms, we are also developing a visualization tool.

ACKNOWLEDGMENTS

Dr. Nancy Amato has given innumerable bits of wisdom, insight, advice, and encouragement that have all helped to make me a better researcher, and possibly a better person. Thanks also go to Paul Thomas, who worked extensively on this project and was always willing to explain things to me until I understood them, and to Dr. Rick Gibson and his students for all of their help.

And thanks to my Mom and Dad, who are the most wonderful and supportive parents I could ever hope for.

TABLE OF CONTENTS

Page

ABSTRACT iii

ACKNOWLEDGEMENTS v

TABLE OF CONTENTS vi

LIST OF TABLES viii

LIST OF FIGURES ix

CHAPTER

I INTRODUCTION 1

II SEISMIC RAY TRACING 3

A. The Wavefront and Interpollation 4
B. Reviewing the Sequential Code 7

III PARALLELIZATION 10

A. Automatic Parallelization 11

IV RESULTS AND DISCUSSION 12

V VISUALIZATION 16

VI CONCLUSION 18

REFERENCES 19

VITA 21

LIST OF TABLES

TABLE — Page

I — Automatic parallelization of the code: Before and after STAPL translation 11

II — The time (in seconds) taken by the program for different number of processors used (rows), for various input sizes (columns) 12

III — Speedup obtained by STAPL for different number of processors used (rows), for various input sizes (columns) 13

LIST OF FIGURES

FIGURE Page

1 Ray Tracing Overview 4

2 Interpollating the Wavefront 5

3 Chart: Varying Input Parameter Values 6

4 Chart: CPU Intensive Functions 9

5 Speedup of RayField code with STAPL for an input size of 1600 Rays . . 13

6 Speedup of RayField code with STAPL for an input size of 4900 Rays . . 14

7 Speedup of RayField code with STAPL for an input size of 7225 Rays . . 15

8 Comparison of speedup obtained for various input sizes using 8 CPU threads 15

9 Visualization of the wavefront at a particular timestep. 17

CHAPTER I

INTRODUCTION

"All I wanted to say," bellowed the computer, "is that my circuits are now irrevocably committed to calculating the answer to the Ultimate Question of Life, the Universe, and Everything." He paused and satisfied himself that he now had everyone's attention, before continuing more quietly. "But the program will take me a little while to run."

Fook glanced impatiently at his watch.

"How long?" he said.

"Seven and a half million years," said Deep Thought.

– The Hitchhiker's Guide to the Galaxy, [1]

Even hundreds or thousands of years probably seems like a long time to wait for the solution to a single problem, let alone seven and a half *million* years. But with technological growth on a steady incline, mankind is becoming capable of studying more and more complex problems at a growing rate. One of our greatest weapons against computationally complex problems is the ability to divide a computation and have multiple computers work together to determine the solution. This approach is called parallel computing, and it will certainly show itself to be an important paradigm to embrace as we attempt to solve more complex problems in the future.

Another sort of obstacle that comes up frequently in scientific computing is what is often referred to as "reinventing the wheel." In developing computer programs to perform a desired computation, a researcher may spend too much time writing code

---

The journal model is *IEEE Transactions on Automatic Control.*

that has already been written – code to search a list of numbers, or to store data in a certain way, for example. These kinds of routine processes can be time consuming to code, and rewriting such ubiquitous functions robs researchers of time that could be spent attacking the real problem.

The solution to the problem of reinventing the wheel is what is known as a code library. A code library is a collection of functions and data structures that are already written and thoroughly tested. Such code is written in a generic way, allowing application to virtually any computational task.

In the C++ programming language, the most thoroughly tested and standardized library is called STL, the Standard Template Library. [9] And, as a result of research conducted under Dr. Lawrence Rauchwerger at Texas A&M University, a new C++ library is being developed that includes parallel versions of STL functions and constructs and some additional parallel constructs. This new library is called STAPL, the Standard Template Adaptive Parallel Library. [2]

The goal of my research as an Undergraduate Research Fellow has been to use STAPL to solve a large and computationally complex problem from the field of Geophysics. This problem is known as Seismic Ray Tracing.

CHAPTER II

SEISMIC RAY TRACING

A Texas oilman goes to see his dentist for a routine checkup. "Everything looks fine," the dentist tells him.

"Go ahead and drill anyway," says the oilman. "I feel lucky today." [4]

While this kind of dental practice is probably a bad idea, you can imagine that this oil investor would like to have reliable, scientifically obtained information about the location of oil in the Earth, just as the dentist can determine the presence of cavities in his mouth before drilling. Seismic ray tracing may one day be able to provide that kind of information about the contents of the Earth.

Seismic ray tracing is a key method for earth modeling and data analysis in Geoscientific fields [3]. To understand the concept of seismic ray tracing, imagine some source of seismic waves sitting on the surface of the Earth. Once waves leave the source, they may be bent by the different material layers of the Earth as they travel, until some of them are directed back towards receivers on the Earth's surface. This situation is pictured in Fig. 1. In our code, we are interested in utilizing a new class of ray tracing solutions known as the wavefront construction approach, which will make the final product a unique and useful tool for geoscientists. [7, 8, 11]

The problem we are currently solving is this: given a known earth model (i.e., a section of earth that we know how to model mathematically), determine how waves propagate through the model. We are able to do this by solving a system of differential equations ("ray tracing equations") which lead to knowledge of the ray paths through the medium, wave amplitudes along the path, and the travel time of a particular ray to a particular point. It is hoped that through study of the way the Earth can be modelled in this manner, eventually the problem of modelling an *unknown* earth

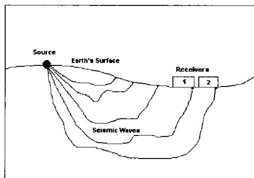


Fig. 1. Ray Tracing: A seismic wave source is located on the surface of earth. Seismic rays travel through the interior of earth, bending as they travel due to the various media and eventually bounce back to the surface of earth where they are detected by receivers (called geophones).

region can be solved by ray tracing methods. This method is useful for both isotropic (uniform in all directions) and anisotropic models, since it has been shown [10] that ray tracing methods are useful in estimating anisotropic properties.

A. The Wavefront and Interpollation

At any point in time, the location of all of the rays propogating through the earth model constitutes the **wavefront**. This can be thought of as a mesh of points in three dimensions identifying all of the ray locations at that time. A very important thing to consider is whether the number of rays present in the system is sufficient to accurately describe the earth model. We may begin our ray calculations with fewer than 10 rays, and as they travel through the earth their paths may spread so far apart that it is difficult to say with accuracy how rays between them would behave. If the situation reaches this point, we **interpolate**, adding rays to the system to describe it more accurately. To test whether interpolation is necessary, we compute a value called the paraxial correction to travel time [6] for a point midway between

two adjacent rays in the wavefront. If this value is too great, we will add a ray at that midpoint, and we perform this test for all sets of adjacent rays (it is actually performed on "patches" of four rays at a time, as shown in Fig. 2) in the wavefront before proceeding to the next timestep to calculate the next wavefront. This ensures that at any point in time, the model is accurately described by the rays before the next wavefront is calculated.

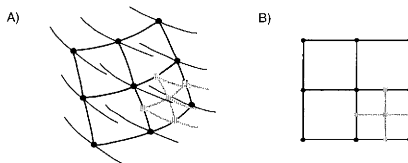


Fig. 2. (A) Schematic illustration of a portion of a wavefront mesh, with one interpolated cell (gray lines). (B) Logical topology of the mesh in the algorithm, where each ray and wavefront surface element is uniquely related to a pair of take-off angles. The grey lines indicate the new boundaries of the interpolated patch on the wavefront surface. [5]

The main action the code takes is to compute where rays will be at the next time step and to interpolate when necessary. There are two important parameters involved in this process: frontDTau and dTestTau. The value of frontDTau defines the timestep between two wavefronts, so it essentially defines how often a wavefront should be calculated. Calculating the positions of all the rays in a wavefront for the next timestep can be a computationally intensive process, so it is important to choose a value for frontDTau that will be small enough that the wavefronts describe the model well, but large enough that the program is not performing this calculation excessively. Note also that the Runge Kutta method used to calculate the positions

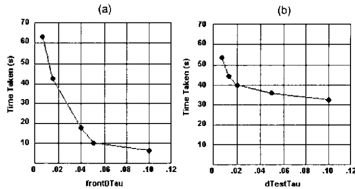


Fig. 3. (a) Increasing the value of frontDTau causes the wavefront to be computed *less* frequently, so the time taken by the CPU decreases. Increasing this value too much, however, will lead to an inaccurate model of the earth structure due to lack of information about the wavefront. (b) Increasing the value of dTestTau increases the amount of error allowed before interpolation will occur, so this also decreases CPU time. Clearly, increasing this value too much will introduce undesireable error.

of rays in the next wavefront will be more accurate the smaller frontDTau is. The graph in Fig. 3 illustrates the increase in computation time as frontDTau is made smaller (that is, the computation occurs more frequently).

The value of dTestTau defines the maximum error in travel time prediction before interpolation should occur. What this means is, when the program is testing whether to interpolate, it will look at two points in the wavefront and compute the paraxial correction to travel time of a point between those two (this is related to the travel time to the midpoint). If the error in this computation is greater than dTestTau, interpolation occurs as shown in Fig. 3(b). Basically, in a quadrilateral formed by 4 rays in the wavefront, rays for the midpoints of the edges of the interpolated quadrilateral are added, as is a ray in the center of the quadrilateral. "Because the interpolation is based on the travel time error, which in turn is directly related to the curvature of the wavefront between rays, the mesh will automatically be refined in

areas where the wavefront changes most rapidly." [5]

B. Reviewing the Sequential Code

In the earliest stage of this research we received code to perform the seismic ray tracing algorithm sequentially (that is, on a single processor). This code was developed under Dr. Rick Gibson of the Texas A&M Geosciences department. Our first task was to compile and run this code on several hardware platforms (HP, SGI, etc). Then, we began studyign how best to parallelize it.

As I mentioned earlier, the STAPL library is a parallel version of the sequential library called STL. Sequential code written using STL can be automatically transformed into parallel code by replacing STL components with STAPL components. However, this incredible usefulness requires that you write your original code using STL. The code we received did not make full use of STL constructs and functions, so we were first required to rewrite some portions of the code to use STL so that STAPL would produce the best results.

Once the code was reengineered in this way, we began looking at which areas of the code would benefit the most from parallelization. To do this, we ran a software profiling program called CXPerf, provided by HP with the HP-UX platform, on the code. This type of profiling software analyzes code as it runs to determine how often each function is called, how much time each function takes, and other statistical data that help identify the time sinks (the most time-consuming portions) of the code.

As may be clear from the previous section, the most important and CPU-intensive calculations performed by this code involve deciding when to interpolate and computing the attributes of the interpolated rays. In the initial steps, there may be only a few rays (less than ten for many test cases). However, as these rays disperse through

the media, interpolation will need to occur to maintain an accurate model. When the program completes, there may be hundreds or thousands of rays, so it is clear that the computational complexity can grow rather dramatically as interpolation is necessary.

With that in mind, we look at the results of our profile of the code. We discovered that there are two main time sinks in the computation, with the first step taking approximately 40% of the time and the second step taking approximately 55% of the time. These two steps in the calculations are implemented in two functions in the RayField class, the class which contains the wavefront information for each time step of the simulation. Setting up the new rays is done in the RayField constructor. Testing the wavefronts and interpolating when necessary is done in the InterpolateFront() function in the RayField class. We quickly noticed that 95.58% of the processing time came from the InterpolateFront() function. But, if we neglect the time taken for the functions that InterpolateFront() calls, it uses only 0.6% of the CPU time. So, it is obvious that InterpolateFront() calls many functions which are CPU intensive. In particular, a function called PatchTest() uses 81.81% of the CPU time. The five most CPU-intensive functions from the code are shown in the bar graph in Fig. 4.

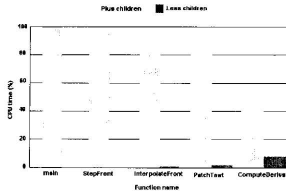


Fig. 4. This graph represents the five most CPU intensive functions from the ray tracing code. The lighter bars represent the percent of the total running time that each function takes including child functions that are called. The darker bars exclude these child functions.

CHAPTER III

PARALLELIZATION

"Modern 3-D seismic surveys produce large volumes of... data that require very rapid processing methods" [5], so we turn to parallelization. The idea behind parallelizing this code is that since the interpolation calculations are so complex, and the number of rays to calculate can increase exponentially as interpolation occurs, we will try to find an efficient way to divide these calculations among multiple processors running simultaneously.

As it turns out, this is theoretically rather easy. The calculations for each ray can be performed completely independently from calculations for the other rays, so they can easily be divided among the available processors. With the code now running using STL constructs and functions, we were able to parallelize it using STAPL.

Code written using STL can very easily be translated to its parallel analogue. STL is composed of three basic units: containers, algorithms and iterators. STL containers are converted to pContainers in STAPL, algorithms are converted to pAlgorithms, and iterators are converted to a new STAPL construct called pRanges. This automatic conversion saves the programmer from much of the complexity of parallel programming, since parallel code written without STAPL would require the programmer to develop parallel containers, algorithms, and iterators – a hefty task. STAPL provides the option for either automatic or manual parallelization. Automatic parallelization is performed during a pre-processing phase in which the STL constructs are directly replaced by their parallel equivalents. Although manual parallelization is faster in some cases, when we tried parallelizing the ray tracing code manually we obtained running time results that were virtually identical to those obtained via automatic parallelization.

Before Translation (STL)		After Translation (STAPL)
`#include<start_translation>`		
`accumulate(x.begin(),x.end(),0);`	-->	`pi_accumulate(x.begin(),x.end(),0);`
`for_each(x.begin(),x.end(),foo);`	-->	`pi_for_each(x.begin(),x.end(),foo);`
`#include<stop_translation>`		

Table I. Automatic parallelization of the code: Before and after STAPL translation

A. Automatic Parallelization

Automatic parallelization is performed during a preprocessing phase which converts the STL constructs to their parallel equivalents. To parallelize a single function or a portion of a function, two pre-processing tags are added to the program: `'#include<start_translation>'` is inserted immediately before the section of code that should be parallelized, and `'#include<stop_translation>'` is inserted at the end of this code section. These `include` statements cause any STL function between them to be replaced by the equivalent STAPL function. An illustration of STL to STAPL conversion is shown in Table I.

CHAPTER IV

RESULTS AND DISCUSSION

To study the speedup obtained by STAPL we varied the number of processors and recorded the resulting running times of the program. The program was run for a homogeneous, anisotropic earth model with a range of values for the input parameters. The experiments were done in dedicated mode on an HP V2200 machine with 16 PA-8200 CPU's running in 4GB of physical memory in the PARASOL Laboratory in the Department of Computer Science at Texas A&M University. The results of the experiments are summarized in Table II. The speedup obtained for some input sizes is shown in Tables II and III and in Figures 5–8. All times reported are the minimum over five executions of the experiment.

CPU Threads	1600 Rays	2500 Rays	3600 Rays	4900 Rays	6400 Rays	7225 Rays
1	282.3	497.8	555.3	780.4	1089.5	1289.5
2	165.4	287.8	315.4	440.7	365.6	663.7
3	145.6	265.3	279.5	390.2	512.1	604.7
4	102.3	190.7	196.7	277.2	384.6	421.2
6	67.8	114.3	127.6	171.1	230.1	265.8
8	56.0	98.2	106.5	145.4	194.6	227.6

Table II. The time (in seconds) taken by the program for different number of processors used (rows), for various input sizes (columns)

CPU Threads	1600 Rays	2500 Rays	3600 Rays	4900 Rays	6400 Rays	7225 Rays
1	1	1	1	1	1	1
2	1.71	1.73	1.76	1.77	1.71	1.94
3	1.94	1.87	1.98	2.00	2.13	2.13
4	2.76	2.61	2.82	2.81	2.83	3.06
6	4.16	4.35	4.35	4.56	4.73	4.85
8	5.04	5.07	5.21	5.37	5.60	5.66

Table III. Speedup obtained by STAPL for different number of processors used (rows), for various input sizes (columns)

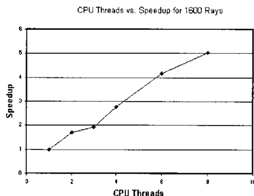


Fig. 5. Speedup of RayField code with STAPL for an input size of 1600 Rays

We note that STAPL was able to attain scalable speedup. As seen in Table III and Figures 5–6, when 8 processors were used we were able to attain a speedup of approximately 5.5. Performance gains remained consistent for various input parameters.

STAPL was able to get better performance improvements for larger input sizes. The performance improvement with eight processors for various input sizes (number of rays) is given in Figure 8. The speedup was highest when 7225 rays were used. As the

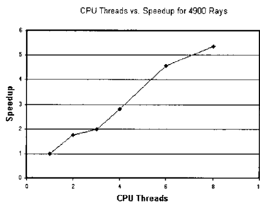


Fig. 6. Speedup of RayField code with STAPL for an input size of 4900 Rays

number of rays increases, the computation required at each time step increases. The greater the computation done in each time step is, the lower the fractional overhead involved in parallelization will be.

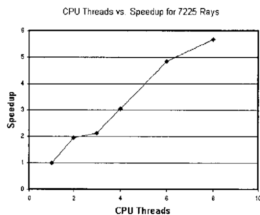


Fig. 7. Speedup of RayField code with STAPL for an input size of 7225 Rays

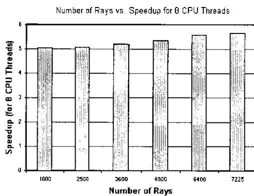


Fig. 8. Comparison of speedup obtained for various input sizes using 8 CPU threads

CHAPTER V

VISUALIZATION

With the the ray tracing code successfully parallelized, we look to visualization as a means to improve user-friendliness and to identify areas for improvement of the algorithm. Before the development of the visualization tool, the code would output a cryptic textual representation of the rays, which left much to be desired when it came to interpreting the data. Now that a 3-dimensional view of the wavefront is available, it is easy to view how the wavefront changes over time, and particularly we can see when and where interpollation is occuring.

Whenever interpollation occurs, we are improving the model at the cost of increased running time. Therefore, if we can identify areas where interpollation is occuring needlessly, we will be able to fine-tune our algorithm (or perhaps just choose better test parameters) to reduce uneccessary interpollations. With the current visualization tool, we are able to see the wavefront for each time step, so we can see when rays are added to the system. Then we can consider the model analytically to determine whether more rays are really justified.

An example image of the wavefront as shown in our visualization package is pictured in Fig. 9. The code for this package was written by Dr. Gibson and his students; my work with it has been in improving certain parts of the design, such as allowing the ability to load wavefront information from a file rather than rerunning the program (which can take a while for large models) every time you wish to view the wavefronts. It is hoped that these types of improvements will lead to more rapid analysis of the algorithm's behavior so that improvements can be made.

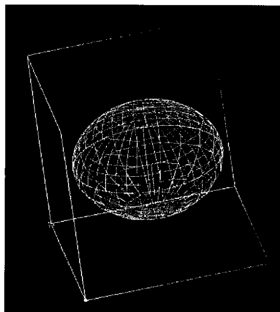


Fig. 9. Visualization of the wavefront at a particular timestep.

CHAPTER VI

CONCLUSION

Parallelization of the ray tracing code has led to significant speedup, and for relatively little effort. That effort could have been minimized further if the sequential code had originally made full use of STL. Likewise, further improvements to the ray tracing algorithm will hopefully be made as a result of a fast and versatile visualization package. Ultimately, we hope that the study of how the Earth can be modelled using ray tracing will provide insight into how wave data can be best used to predict the internal structure of unknown sections of the Earth.

REFERENCES

[1] Adams, Douglas. The Hitchhiker's Guide to the Galaxy. New York: Random House, 1979.

[2] P. An, A. Jula, S. Rus, S. Saunders, T. Smith, G. Tanase, N. Amato, and L. Rauchwerger. STAPL: An adaptive, generic parallel programming library for C++. In *Proc. of the 14th International Workshop on Languages and Compilers for Parallel Computing (LCPC)*, Cumberland Falls, Kentucky, Aug 2001.

[3] Aki, K., and Richards, P., 1980b, Quantitative seismology, vol. 2: W.H. Freeman, San Francisco.

[4] Dingus, Anne. "Yee-ha!" *Texas Monthly* January 2002: 93.

[5] Gibson, R. and Amato, N. M., An Adaptive Wavefront Construction Algorithm for Optimal

[6] R. L. Gibson, Jr., A. G. Sena, and M. N. Toksöz. Paraxial ray tracing in 3-D inhomogeneous, anisotropic media. *Geophys. Prosp.*, 39:473–504, 1991.

Seismic Ray Tracing, NSF proposal, ITR Program, 9/1/00-8/31/03.

[7] Lambaré, G., Lucio, P., and Hanyga, A., 1996, Two-dimensional multivalued traveltime and amplitude maps by uniform sampling of a ray field: Geophys. J. Int., **125**, 584–598.

[8] Leidenfrost, A., Ettrich, N., Gajewski, D., and Kosloff, D., 1999, Comparison of six different methods for calculating traveltimes: Geophys. Prosp., **47**, 269–298.

[9] Musser, David and Derge, Gillmer and Saini, Atul. STL Tutorial and Reference Guide, Second Edition. Addison-Wesley, 2001.

[10] Pérez, M., Gibson, Jr., R. L., and Toksöz, M. N., 1999, Detection of fracture orientation using azimuthal variation of P-wave AVO reponses: Geophysics, **64**, 1253–1265.

[11] Vinje, V., Iversen, E., and Gjøystdal, H., 1993, Traveltime and amplitude estimation using wavefront construction: Geophysics, **58**, 1157–1166.

VITA

Derek Edward Kurth will graduate from Texas A&M University in the Spring of 2002 with a degree in Computer Engineering (Computer Science Track). He will then go on to Carnegie Mellon University to pursue a Master's Degree in Robotics.

Permanent address:

1108 E. Miller
Angleton, TX 77515

The typist for this thesis was Derek Edward Kurth.