Parallel Ray Tracing in pH: An Exploration of Functional Programming

Wes Beebee, Karin Cheung, and Jeffrey Sheldon
December 3, 1999

6.837 Final Project

Abstract

Traditionally, computer graphics development has relied on the use of imperative, sequential programming languages, such as C or C++, to achieve the necessary computational efficiency. This paper explores the introduction of functional programming as a higher-level substitute for imperative languages, through the development of a ray tracer in pH. This implementation choice arises from pH's purely functional infrustructure as well as its ability to execute in parallel. Furthermore, our ray tracing experiment incorporates the challenges of performance optimization, as well as lessons learned in data flow analysis and producer/comsumer relationships. Finally, detailed performance results were recorded in execution comparisons between two different compilers, GHC and pHC, as well as various multi-processor arrangements.

Introduction

In computer graphics research, much development of realistic graphics displays concentrates its efforts on creating efficient surface-rendering techniques, specifically methods of visible-surface detection and illumination modeling. While numerous algorithms have been devised to approach this problem, developers commonly use ray casting and ray tracing procedures, where a ray is sent out from each pixel position to locate surface intersections. This simple idea proves extremely effective in determining visible surfaces in a scene, obtaining global reflection and transmission effects, producing shadow effects, incorporating transparency features, and considering multiple light-source illumination.

Generally, computer graphics software, such as that of ray casting and ray tracing, focuses on computational efficiency. Therefore, implementations of these methods typically employ languages such as C and C++, which function through state mutation, using a specified order of execution. This traditional method of programming, however, has several weaknesses. First, while the ability to mutate state has its advantages, such intrusiveness allows possibilities of incorrectly modifying memory locations, one of the most frequent problems in computer programming. Furthermore, the sequential order of execution restrains the speed of the software, by forcing the system to evaluate each instruction line by line. This sequential characteristic of languages like C, including Java, Pascal, and so on, is referred to as being imperative.

One can easily imagine a more abstract programming language that no longer permits modification of the machine state, such that the order in which instructions are executed no longer matters, except in certain data dependency situations. In fact, many research circles have developed an interest in creating such a system, namely a functional language. Functional languages emphasize a clean design with minimal conceptual overhead, in the belief that issues of implementations and speed are best left to the compiler. One such functional language, developed at the MIT Laboratory for Computer Science, is pH,

a parallel dialect of the functional language Haskell.

What exactly is a functional language? A functional language like pH is inherrently difference from an imperative language such as C. A functional language is a single expression which is executed by evaluating that expression. The idea of functional programming is very similar to that of a spreadsheet program. In a spreadsheet, one specifies the value of each cell in terms of the value of the other cells. *The focus is on what is to be computed, not how it is computed.*

- We do not specify the order in which expressions are evaluated, but instead take for granted that they will be computed with respect to their dependencies
- We do not tell the program how to allocate memory, rather we expect the program will allocate memory as it is needed
- We specify a value by its expression, rather than a sequence of commands which calculates its value

For example, SQL, the standard database query language, is inherently functional. A query within SQL consists of projections, joins, and selections, tools which allow the programmer to say what relation should be computed without saying how it should be computed. This focus on a high level ‘what’ without the low level ‘how’ is the crucial difference between a functional language and an imperative one.

In regard to rendering techniques, an experimentation of ray casting and ray tracing in pH allows a high level abstraction of the core rendering system that is purely functional. A lighting model, for instance, could be abstracted out of the care of the rendering engine, making it very simple to swap one lighting model for another. Furthermore, pH is a fully type-safe language, unlike C++, which holds promise in reducing debugging time and allowing static analysis to assist in program verification. Possible rendering-efficiency benefits of pH will also be analyzed in this exploration. Since pH relies on the functional nature of the language and the lack of a sequential order of events, the compiler is free to analyze data dependencies to determine opportunities for parallelism.

Therefore, the remainder of the paper analyzes our experimentation in creating a ray tracer in pH. After an initial discussion of design motivations, this study will describe the design of the ray tracer, including a description of the syntax definitions, the world representation, the ray tracer itself, and the actual painting of pixels to the screen. Next, this exploration will review what aspects of using a functional language made the system easier or more difficult to implement, as well as a proposal of why these advantages or limitations existed. Lastly, the paper will discuss the results of the final ray tracer, using benchmarking tests to evaluate the value of implementing a ray tracer in a parallel, functional language such as pH.

Goals

This project seeks to introduce a new way of thinking about computer graphics, a method of application-building that has the potential to revolutionize the way graphics software is developed today. In analyzing the current standards of application development, we have noted several shortcomings of graphics programming languages which are commonly used today in building applications like those we have implemented in class, ray casting and ray tracing. The necessity of creating very fast, very efficient programs in graphics has driven the field further into the realms of low level programming, leading to tools created mostly in C. While such tools must be commended for their efficiency capabilities, design

and implementation with such tools burdens the software developer with the need to translate naturally abstract ideas into low-level code.

Thus, this project explores the possibility of relinquishing the need for low-level programming while retaining efficiency characteristics. Our group hoped to accomplish this through the replacement of imperative languages with the use of functional programming techniques. While functional programming naturally makes graphical software development much easier in terms of abstraction, the challenge of optimization remains. Therefore, in our development of a ray tracer in a functional language, we aim to successfully create a ray tracer in a purely functional manner, using optimization techniques such that it potentially may equal the efficiencies found in C programming. At this point, we should also note that pH is still under development within LCS, so many of the compiler issues are not optimized to their full potential. Since pH is a parallel dialect of Haskell, we can also use better developed Haskell compilers use as a control case to factor out the problems caused by pH's underdevelped codegen engine. However, this project should provide a fair evaluation of how well functional languages work in graphics applications, namely a ray tracer, with promise of showing even better results, especially for parallelism, when pH development is completed.

Anaylzed in more detail, most computer graphics applications tend to be very CPU intensive tasks in which computation times is often the limiting factor. This has naturally led the majority of graphics work being done in low level language, most commonly C, which allows the implementation to occur very near the hardware. Historically this decision has been clearly justified as the need for efficiency demanded careful hand design and tuning of the project. Working at such a close machine level, however, leads to many drawbacks.

From a strictly software engineering perspective one lacks many of the higher level abstractions which are typically used to accelerate development and guard against errors. C is kind enough to provide a type system of sorts, but only a truly weak one. The language itself is not type safe and through some of its more creative pointer manipulations, allows for essentially arbitrary behavior. This has the natural consequences of allowing a great many more programming and design errors to slip through the development cycle until a much later stage. Beyond the issue of type safety, more modern languages also admit the use of such constructs as higher order procedures and true abstract data types.

Modern system architectures have also changed substantively in recent years. In the past improving system performance involved looking at ways to reduce redundant computation, accelerate loops, speed array access and the like. These changes could be expressed in low level languages, but are no longer relevant due to compiler advances. Modern code acceleration deals with far more subtle issues that arise in rearranging code and attempt to allow the instruction stream to make use of the multiple execution units on a single chip. These sorts of optimizations are nearly impossible for a programmer to carry out forcing the job to the compiler (and in many cases the actual hardware). To perform this sort of analysis on machine code requires inferring much about the higher level goal of the code thereby constraining the analysis to simple changes in small code fragments. Analysis of this sort can be performed far more effectively at a higher level. A strictly imperative language like C allows for more structure and more powerful analysis, but still remains little more than a readable, machine-portable assembly language. By using more modern, functional, representations of a program, the programmer more directly represents the work which needs to be done, rather than strictly defining how the processor should carry out that work.

One particularly interesting example of this kind of analysis, especially in light of the growing numbers

of SMP's, is parallel computation. While traditional techniques for parallelism have involved either explicit management by the programmer, or truly complex (and sometimes unreliable) techniques for attempting to extract parallelism from a sequential program, this parallelism can be more directly expressed by not allowing the programmer to express sequentiality in the first place.

Clearly working in a higher level functional language can offer many benefits over traditional imperative implementations, but graphics work, which tends to be CPU bound, unlike many other applications, may be faced with substantial performance hits when working with such languages. As such we wished to explore the feasibility of implementing a highly CPU bound graphics algorithm, ray tracing, in such a language. We choose to work with pH as it provides a particular interesting example due to its ability to execute in parallel. Clearly such an implementation is possible, but with the overhead typically associated with the runtime systems of such functional languages, optimizing the program becomes an interesting challenge. While compiler technologies have far surpassed human ability at managing registers and reodering instructions to prevent pipeline stalls, inefficiencies due to poorly designed dataflow structures cause substantial delays in many programs. It would, therefore, be interesting to look at ray tracing, and see if it will be possible to create a reasonably efficient functional implementation, which can also execute in parallel.

Achievements

We were able to complete a functioning raytracer in pH/Haskell and were able to do so using completely functional constructs. (The pH language has certain non-functional extensions designed to be judiciously used.) In terms of performance, we had somewhat mixed results. As one might have expected neither running the code under Haskell nor pH was able to produce a ray tracer which would perform as well as a well-implemented C version. The Haskell implementation, however, did perform reasonably well when compared to the C version. The performance of the pH version was quite disappointing as it often ran well over an order of magnitude slower than the Haskell runtime. Futhermore, while speed clearly remains an issue, many of the benefits of programming in a higher level language were apparent. The pH was indeed able to spawn across multiple processors without difficultly, although performance did not maintain a linear increase in the number of processors.

Oddly enough one of our greatest concerns when entering into the project was the task of actually performing the graphics output. While foreign function interface support existed in Haskell, no such support existed in pH. Therefore, we circumvented this lack of support by simply writing the graphics data to standard out and then using an external viewer to read the pixel colors and draw them to the screen. In the end the resulting performance loss was not very significant since the raytracing calculations more than swamped any required I/O time.

In the process of initially implementing the ray tracer, we admittedly ran into a number of implementations problems which speaks poorly for supporting our claim that working with additional, and higher level structure can accelerate and ease program development. Nearly every one of these problems, however, could be traced back to issues arising from the fact that many of the tools we are using are still under development in the Laboratory for Computer Science at MIT. On the whole, however, the resulting system seemed to represent the ray tracer in a clearer, more concise, manner than the original C implementation. Of course in our initial implementation of the system we also showed the danger of writing in a high level language by creating a ray trace which through the course of rendering even a small thumbnail would, ignoring garbage collection, attempt to allocate many gigabytes of memory. Some thought about what we had actually written, however, was able to produce a version of

the code which took nearly three orders of magnitude less memory and retained the clean structure.

By carefully studying the flow of data through the ray tracing algorithm (which will be described later) we were able create what we feel is a reasonably efficient implementation when running with Haskell. Unfortunately the same cannot be said about the same code running under pH. This was partially expected as pH's code generation phase remains somewhat primitive and, among other things, does not yet perform register allocation. Nonetheless the dramatic difference between the two was surprising. On the other hand this does suggest that once a reasonable implementation of codegen is developed for pH, reasonable execution times can be achieved.

The use of the pH code on multiple processors performed extremely well when switching across a low number of processors. The synchronization overhead seemed to be negligible when using two processors, and remained small using up to four processors. Unfortunately using a greater number of processors did not substantially improve performance. This is likely being caused by two factors. The current garbage collector is not parallel, and thus require all but one of the processors to be idle when garbage collecting. There are also some synchronization costs involved in handling the locks used by the scheduler to maintain a work queue. A detailed performance analysis chart can be found in the appendix section of the paper.

Individual Contributions

While work on the ray tracing project was not cleanly divided among our group members, each member contributed to a significant portion of the project development.

To a rough approximation, work on the project required the completion of four major sections:

- Designing a workable rendering world
- Implementing ray tracing in a purely functional manner (requiring work on ray casting, shadowing, reflection, transmission, transparency, matrix handling and other graphics tools)
- Developing a means of extracting rendered pixel data to be painted to the screen from within the pH runtime
- Benchmarking
- Optimization
- Writing this paper

Karin Cheung: Participated in developing a rendering world in which to store graphical objects. Some issues involved in this design included user friendliness, usability, and efficiency concerns. Also involved in the implementation of the ray tracer, contributing to the development of each of the tools stated above. Lastly, helped write the final project paper.

Jeffrey Sheldon: Also participated in developing the rendering world and implementing ray tracing procedures. Furthermore, ran many benchmarking tests on the resulting program, including tests on both the pH compiler and GHC (Glasgow Haskell compiler), as well as work on multiple processor machines. Also, helped with paper writing.

Wes Beebee: Participated in developing a method of extracing rendered pixel data to be painted on the screen, and implemented ray tracing procedures. Also, contributed to both benchmarking and optimization procedures, including a setup of the Glasgow compiler on Athena. Optimization involved

analyzing the flow of information through the program and finding and attempting to remove inefficiencies. Also, helped with paper writing.

Lessons Learned

A raytracer relies on recursion to cast new rays every time an object is hit, and casting and shading rays relies on large procedures to determine which object is hit, what the normal is, and how the properties of the object affect which colors the ray will accumulate. Each of these large procedures has many variables, and requires a large frame to hold the values of those variables. Since the memory for the frame cannot be deallocated until the final recursion for the ray, many large stack frames must be present at the same time. Since memory allocation and deallocation is one of the slowest operations a computer can perform, if many huge stack frames accumulate, the raytracer will slow down.

Since many different procedures could potentially have access to the same variables at the same time in a parallel raytracer, the property sets of a given object, for instance, and an execution of a parallel raytracer is a tree, the frames containing variables must be heap allocated to keep them alive and accessable to multiple parts of the raytracer at a time (multiple children of a given thread).

Furthermore, since heap allocated stack frames must be allocated and garbage collected (a slow process), it is critical to the speed of the raytracer that calculated properties of the objects: intersection, color, and normal data, that depend on the path of a given ray, be short-lived.

In practice, dataflow analysis designed to reduce memory usage makes 2-3 orders of magnitude in speed improvements. For instance, since many rays are processed in a raytracer at the same time, at the top level, the raytracer function is mapped over the set of rays to be cast. When creating the rays to be cast, we initially provided one function to create the list of rays, and another to map the raytracer over them. This program took over 2 Gigs of memory to execute! So, we put the list generator code in the list argument for the map function that maps the raytracer over the rays to determine their colors. This reduced memory usage to 200 Mb, a savings of 2 orders of magnitude.

We unknowingly discovered a producer/consumer relationship problem in the raytracer. Since the ray generator code is pretty fast compared to the ray tracer code, the list of rays were generated faster than they could be consumed, and different copies of the raytracer started consuming the list asynchronously, generating numerous stack frames that completely swamped the machine. Since the list of rays must remain in memory for the entire time the raytracer is running, that memory could not be reclaimed, and added additional memory overhead. When we placed the producer next to the consumer, the compiler optimized away the intermediary list, and produced the rays in lock step with the raytracer, so rays could be generated and immediately cast. This eliminates the list overhead, and allowed the scheduler to run on only as many rays as the processors and memory could handle efficiently at a time.

After fixing this producer/consumer relationship problem, we found other producer/consumer relationship problems in the code. When new rays are cast, the previous computations on the old ray could be happening asynchronously, while the new, reflected ray, is blocking on parts of the old computations. Thus, stack frames for both are allocated on the heap at the same time, reducing the available heap memory, and slowing down the computation. This problem required strictness analysis to determine which parts of the ray tracer needed to know what information about the ray and objects need to be available, and at what times.

The more information that you need to know earlier, the faster the computation becomes, since you do not need to block later on that information. However, if you choose to be completely strict about everything, you lose all of the parallelism implicit in the code. Therefore, we used pattern matching in the method signature to ensure that the data necessary to execute a given method (that can be pattern matched) is available at the start of execution of the method. This approach helps prevent extra, partially empty, stack frames from being allocated on the heap, and preserves the inherent fine-grain parallelism of the raytracer.

The inlining of small functions, such as adding two vectors or taking their dot product, also becomes a tricky matter when dealing with heap-allocated frames. We agressively profiled and inlined all the functions that could be inlined without reducing performance. However, this traditional optimization did not yield a very large performance increase (<10%), compared to the large performance increase of looking at producer/consumer relationships. The reason for that is simple. Inlining functions reduces the number of frames allocated, but may increase the size of those frames that are allocated. This tradeoff made inlining a minor optimization compared to inlining in C.

We also relied on the non-strictness of pH to collapse let blocks, and allocated less frames while increasing the potential number of computations performed when the computations were relatively minor (such as multiplies and adds of vectors). Since the critical path is the same in either case, the raytracer performed faster when it had less memory to allocate, since it only had to block on the computations if the results were actually used. This style improved the speed and readability of the code, and contrasts sharply with the short-circuiting style used in the C raytracer. Once again, this was a comparatively minor optimization compared to the strictness analysis and optimization of consumer/producer relationships.

In short, the optimizations performed on the pH version of the ray tracer were markedly different to the C version, and emphasized the flow of data throughout the raytracer algorithm rather than short-circuiting of control flow as in C. Control flow optimizations are not as important in an implicitly parallel, functional ray tracer, since data flow determines control flow.

Interestingly enough we did actually have to learn about properly handling some of the geometric transforms required to correctly map between different spatial representations. Naturally this sort of logic was required for the ray tracer used in the problem set, but in this case the actual work was simply performed by a callout into an Open Inventor library.

As mentioned already, we were quite suprised with the importantance of the codegen portion of the compiler which accounted for the bulk of the speed difference between the Haskell runtime and the pH runtime. Previous experience did not seem to indicate as profound of a difference between the two systems. This suggests that the highly numerical computations involved in a graphics algorithm such as ray tracing are highly suceptible to performance problems when faced with a system lacking strong code generation. In retrospect this is perhaps not terribly suprising as many computations tend to be bound by memory access time, while a raytracer apparently is limited by actual computation. For instance, under the Haskell compiler, the checker example running at a resolution of 200x200, took 7 min., 51 sec. to run on a Sparc Ultra 5. Under the pH compiler, however, it took 8 hours, 18 min., 29 sec. Apparently, suboptimal pH code generation made a factor of 64 difference in performance.

When implementing the system we did encounter one particular problem which we were suprised to discover was actually far more difficult to formulate in a functional style than in a traditional style. Even

more oddly it was not really a problem specific to ray tracing, but the far more general problem of matrix inversion. The resulting implementation is moderately insightful from a mathematical perspecitive, is quite difficult to follow. This was moderately disappointing and stood in sharp contrast to the rest of the implementation. For further information, the code for matrix inversion in pH can be found in the appendix section of this paper.

Acknowledgements

Jan-Willem Maessen implemented a large portion of the pH compiler, and has helped us debug our ray tracer with it. He gave us hints as to optimizations to perform, and a mini-tutorial of the pH language. He worked hard to fix compiler bugs that our raytracer revealed, as well as bugs in the standard libraries. He gave us advice as to how to set up the pixel pusher, and helped us debug and design the I/O routines. He helped obtain a parallel machine to run on (Xolas), and set up and debugged the pH compiler on it. He fixed linking errors on Xolas, and dealt with numerous operating system/compiler incompatibilities that occurred on the parallel machine due to an operating system upgrade by the system administrators. He has given immense support to our project over its entire duration, and, without a doubt, we would not have been able to do this project without his help.

In return for Jan's help, we have agreed to give him a copy of our raytracer as it is currently his only recent real test case and application for the pH compiler. This raytracer shall serve as a target for future compiler optimizations as well as help gain recognition for the pH language.

Professor Arvind, head of the Functional Language and Architecture Group at LCS, and lecturer of 6.827 (Multithreaded Parallelism: Languages and Compilers), taught us the pH language and the theoretical foundations for understanding implicit parallelism and compilers of implicitly parallel functional languages. Without this specific knowledge of the pH compiler, we probably would not have been able to work around bugs in the pH compiler or optimize our program. He also gave us specific help in determining which optimizations to do, and helped us obtain access to Xolas. He has been very helpful and encouraging. We hope that our project will further his research goals, and forge a mutually beneficial link between graphics research and functional language research.

Professor Seth Teller, head of the Computer Graphics Group at LCS, and lecturer of 6.837 (Computer Graphics), taught us how to build a ray caster and a ray tracer, as well as provided the opportunity to build most of one in C (Assignment #6), a valuable prerequisite to this project. He has provided us with the opportunity and time to do this project as the final assignment in 6.837. He has provided necessary feedback and guidance to help direct the goals and implementation of our project.

Scott Blomquist referred to us by Mitch Mitchell, and Christopher Hill, was the system administrator who made it possible for us to gain access to Xolas for benchmarking. Xolas was a 72-processor machine which has since been broken up into 9 8-processor Sparcs. This break up left all the machines in a non-functional state. The unfortunate timing of this break up delayed benchmarking, but Scott allocated and set up one of the machines in time for us to run our tests. As a tribute to the goals of implicit parallelism, we did not have to debug our program on the SMP. It worked on the SMP from the first time we ran it, with none of the data races or synchronization problems that usually plague multithreaded programs. In a sense, our insensitivity to the delay in access to the SMP was a proof of concept for our raytracer.

The Glasgow Haskell Compiler provided a useful benchmark to demonstrate the difference between the

highly optimized code generation of the Haskell compiler versus the less optimized code generation of the pH compiler. Future developments to the code generation of the pH compiler should enable pH single processor performance to be equal to Haskell compiler performance. As more time and money is spent on developing pH, these performance payoffs shall be realized.

We apologize in advance to those who we may have forgotten to acknowledge here. This raytracer was the product of the collaborative effort of many people and research groups at the MIT Laboratory of Computer Science and elsewhere.

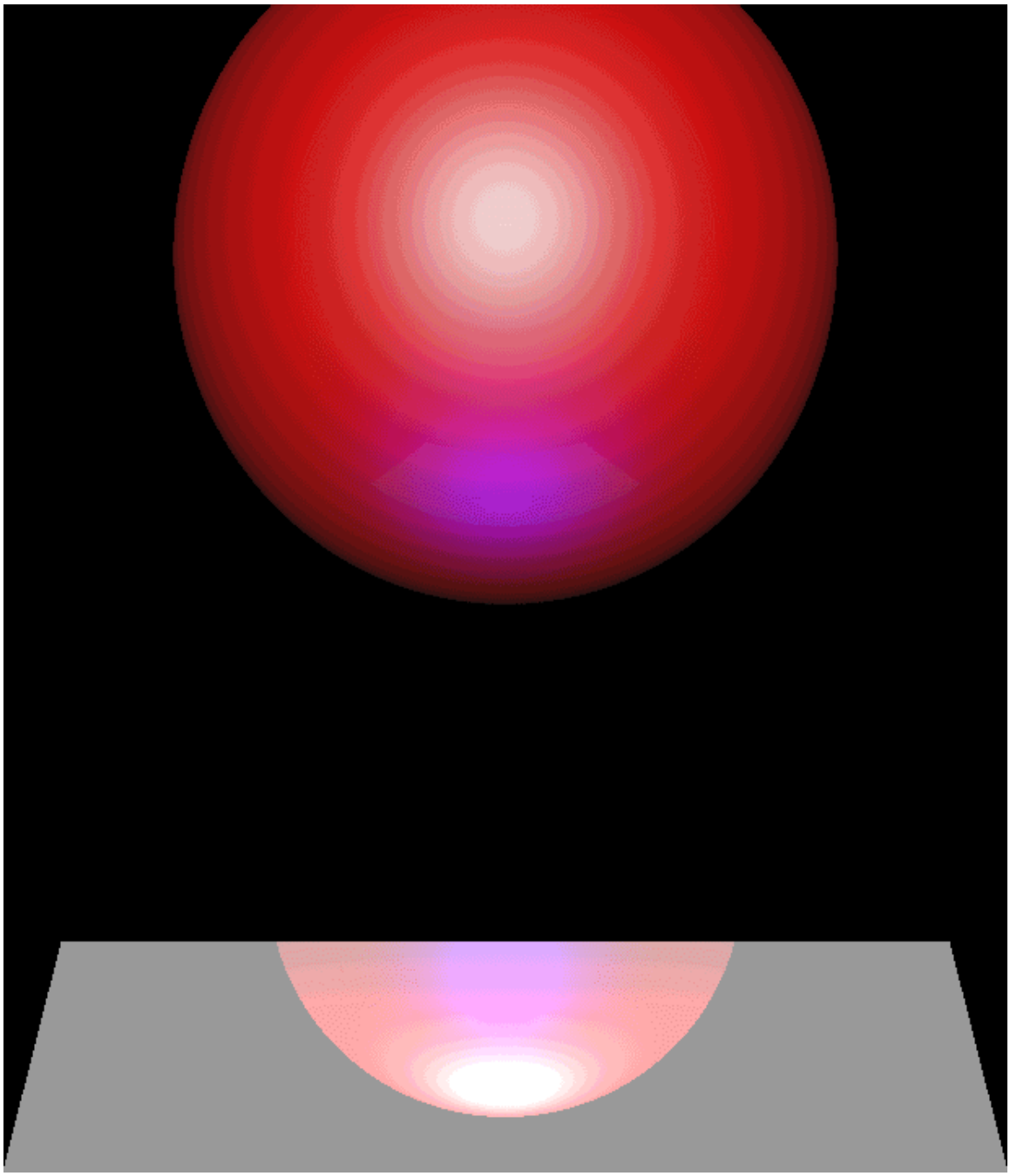
Bibliography

Haskell 98 Report available at `http://haskell.org`.
Nikhil, Rishiyur S., and Arvind, *Implicit Parallel Programming in* pH, 1999.

Appendix

Sample Output

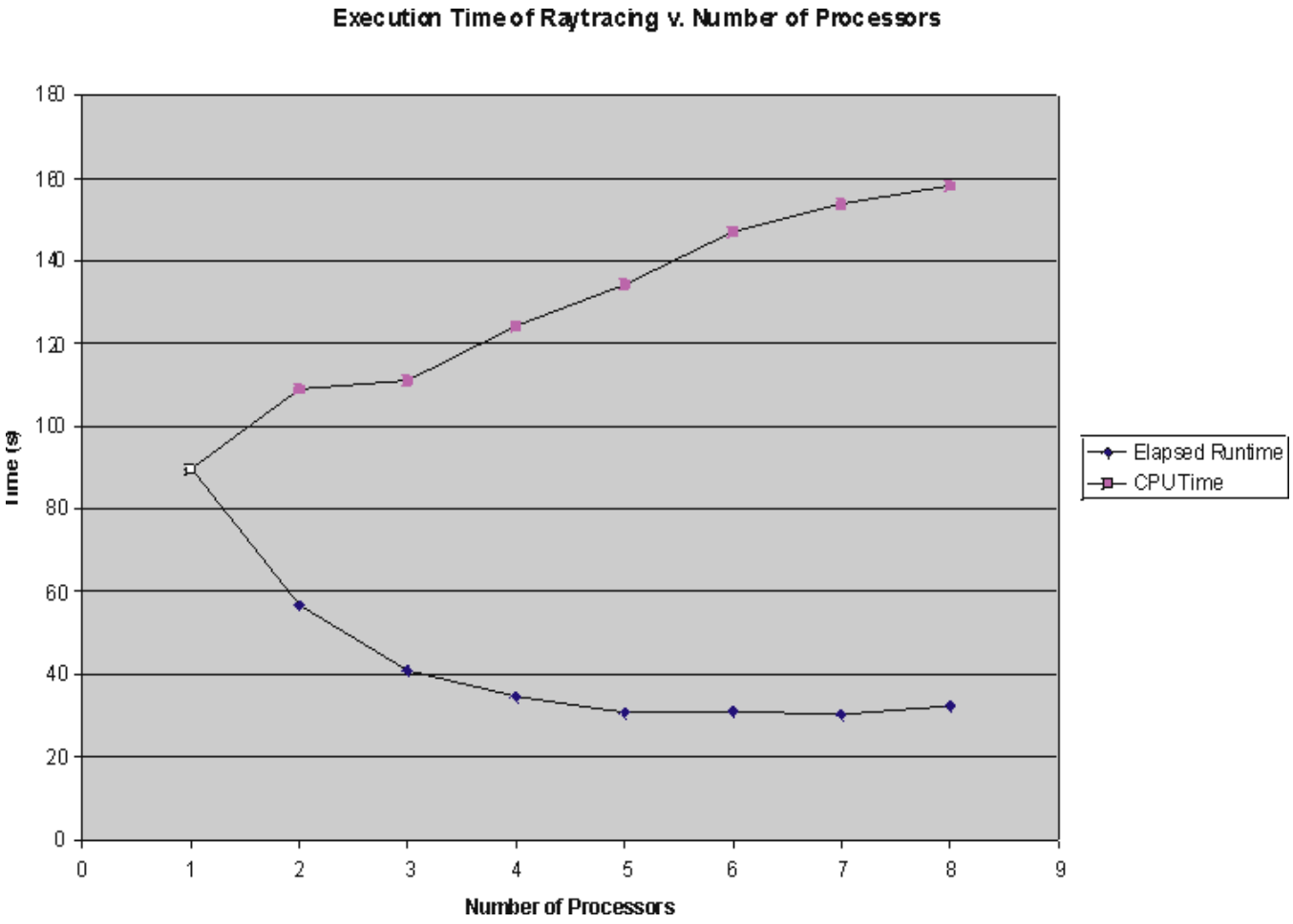
coolreflect



Performance Data

Test	Machine	Compiler	Size	Elapsed	User	System	CPU Usage
coolreflect	ultra 5	ghc	50	0:03	3.0	0.0	
			100	0:13	13.0	0.0	
			150	0:29	29.0	0.0	
			200	0:51	51.0	0.0	

		phc	50	1:03	61.0	0.0	
			100	3:49	228.0	0.0	
			150	9:12	551.0	0.0	
			200	18:16	1093	0.0	
	xolas0 (1)	phc	50	1:29.94	89.51	0.25	99.7
	xolas0 (2)	phc	50	0:56.75	108.94	4.00	199.0
	xolas0 (3)	phc	50	0:40.85	111.01	8.99	294.7
	xolas0 (4)	phc	50	0:34.71	124.19	10.45	387.8
	xolas0 (5)	phc	50	0:30.80	134.25	13.16	478.6
	xolas0 (6)	phc	50	0:30.95	147.06	21.18	543.5
	xolas0 (7)	phc	50	0:30.31	153.74	18.16	567.1
	xolas0 (8)	phc	50	0:32.36	158.17	23.00	559.8
coolreflect.p	ultra5	ghc	50	0:02	2.0	0.0	
			100	0:09	9.0	0.0	
			150	0:20	20.0	0.0	
			200	0:37	36.0	0.0	
		phc	50	2:44.48	163.52	0.21	
			100	11:01.44	659.77	0.23	
			150	26:54.38	1610.03	0.31	
			200	53:40.62	3212.70	0.46	
	O2	cc	200	0:08			
			400	0:30			
			600	1:09			
			800	2:08			
checker	ultra 5	ghc	50	0:31.21	30.4	0.04	
			100	1:58.91	117.58	0.04	
			150	4:27.29	264.64	0.07	
			200	7:51.65	465.76	0.12	
		phc	50	8:44.59	517.52	0.34	
			100	33:17.98	1993.79	0.14	
			150	1:19:27.05	4754.09	0.18	
			200	8:18:29.62	29862.62	0.18	
	O2	cc	200	0:16			
			400	0:57			
			600	2:09			
			800	3:54			



Matrix Inversion in pH/Haskell

```
invert :: Matrix -> Array (Int, Int) Double
invert a =
    let
    ((1,1),(n,_)) = bounds a
    new = array((1,1), (n, 2*n))
           [ (j, (\ (x, y) ->
                 if (y <= n)
                 then a!(x,y)
                 else if (x == y-n)
                 then 1
                 else 0) j) | j <- range ((1,1), (n, 2*n))]
    b = gauss_jordan new
    in
    array((1,1), (n, n)) [(j, (\ (x, y) -> b!(x, y+n)) j) |
                          j <- range ((1,1), (n,n))]

gauss_jordan :: Matrix -> Array (Int,Int) Double
gauss_jordan a =
    let
    bn@((1,1),(n,m)) = bounds a
    aa = array (1,n) ( [(1,a)] ++ [(k+1,eliminate (aa!k) (l!k) k)|k<-[1..n-1]])
    l = array (1,n-1) [ (k, multipliers (aa!k) k) | k<- [1..n-1]]
    multipliers ak k = array (k+1,n) [ (i, ak!(i,k)/ak!(k,k)) | i <- [k+1..n]]
```

```
eliminate ak lk k = array ((k+1,k+1), (n,m))
          [ ((i,j), ak!(i,j) - ak!(k,j)*lk!(i)) | i<-[k+1..n],j<-[k+1..m]]
u = array bn [((i,j), if (i <= j)
                      then aa!(i)!(i,j)
                      else 0.0) | i <-[1..n], j<-[1..m]]
backmultipliers bk k = array (1, k-1)
                         [(i, bk!(i,k)/bk!(k,k)) | i <- [1..k-1]]
b = array(1,n) [ (k, backmultipliers (bb!k) k) | k <- [1..n]]
backeliminate bk lk k = array ((1,1), (k-1, m))
         [((i,j), bk!(i,j) - bk!(k,j)*lk!(i)) | i<-[1..k-1],j<-[1..m]]
bb = array(1,n) ([(n,u)]++[(k-1,backeliminate (bb!k) (b!k) k)|k<-[2..n]])
bu = array bn [((i,j), if (i <= j)
                       then bb!(i)!(i,j)
                       else 0.0) | i<-[1..n], j<-[1..m]]
cmult = array (1,n) [ (k, 1/(bu!(k,k))) | k <- [1..n]]
in
array((1,1),(n,m)) [((i,j), (cmult!i) * (bu!(i,j))) |
                    i <- [1..n], j <- [1..m]]
```

Note the mutually recursive array comprehensions and cyclic dependencies used to express Gauss-Jordan elimination and back substitution without using mutation. This is the one case in our code where the pH/Haskell version is more complicated due to the functional nature of the code. However, an equivalently parallel version using multiple threads in C would have been far more complicated, due to the necessity of multiple synchronization objects to prevent data races caused by mutation.

Code Execution Models

The pH execution model differs substantially from a traditional C environment. The pH runtime includes an activation frame tree, blocking read semantics, and employees its own scheduler to deal with extremely small threads.

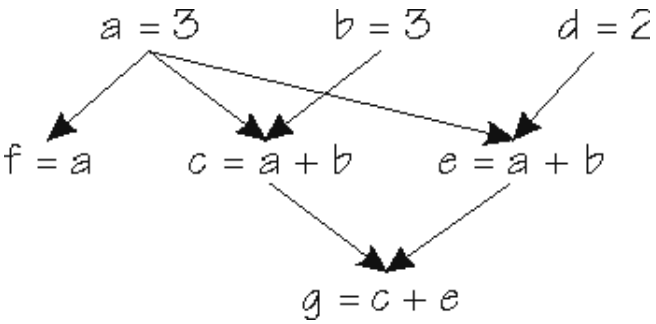
Due to the parallel nature of the computation, as well as the non-strict semantics of the language, a normal C-style call stack cannot be used and instead a call tree must be used instead. Since any procedure may actually call many other procedures concurrently, and since these may return in an arbitrary order, a tree of active activation frames must be maintained.

The semantics of the pH language call for essentially all potentially parallel computations, such as the bindings in a let block, to be spawned in parallel. This, however, leaves open the question of synchronization to ensure that a value is computed before it is used. The issue of a variable holding the wrong value does not arise as long as one uses strictly functional constructs. To solve the concurrency issues, pH uses a notion of blocking reads. When a location in memory for a variable is allocated, it is set to be empty. Should any pH code attempt to read the value for that variable while it is empty, the read will block and the code will be suspended. Once blocked, control will pass to another active thread from the active thread queue. When the variable's value is filled, all threads which were blocked on it are then reactivated. The pH schedule is implemented using a global work queue, and by having the threads themselves, upon blocking, to pass control to the next item on the work queue. This form of scheduling is required as most pH threads are on the order of dozens of machine instructions. This allows for a large number of threads to be spawned at relatively low cost, and prevents synchronization problems.

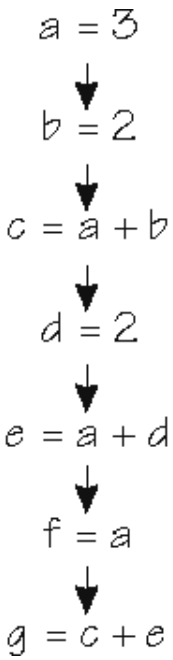
Take the following very simple pH/Haskell code sample:

```
let
  a = 3
  b = 2
  c = a + b
  d = 2
  e = a + d
  f = a
  g = c + e
in
g
```

The data dependencies in the code can be diagrammed as follows.



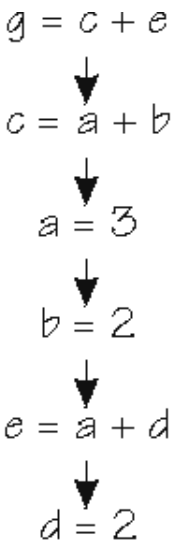
If this were implemented using an imperative structure such as C the flow of execution would perfectly mirror the ordering of the statements as shown below:



Of course this assumes the compiler is not optimizing, since an example as simple as this can be static reduced to a single constant assignment.

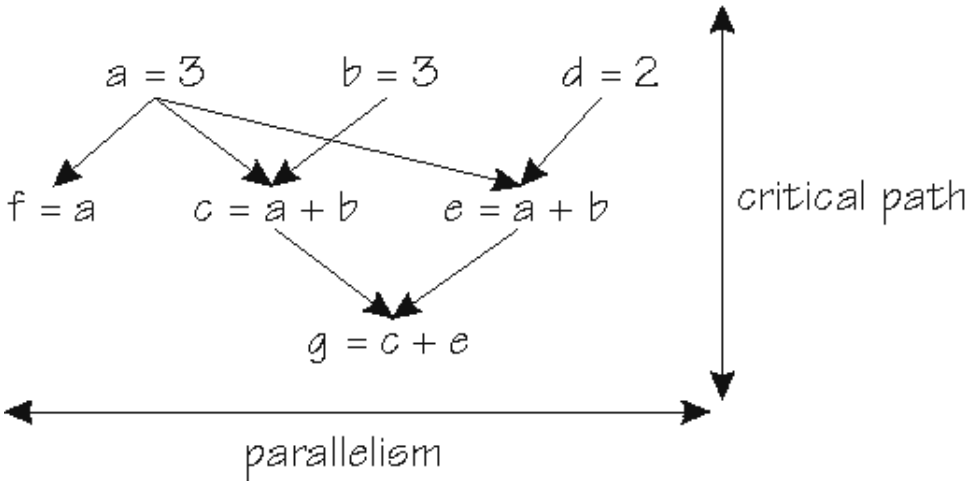
When run in haskell, the evaluation of the final expression takes a lazy evaluation approach. If necessary, subexpressions are then evaluated causing the flow of evaluation to proceed in a reverse

order.



f = a is not executed

pH, however, attempts to execute all the statements. If those which depend on previous values are evaluated, they will block until such values become available. This leads to the following execution tree:



Ray Tracer File Format

```
ModelTree = Tree [ModelTree] |
            ShapeNode Shape |
            PropertyNode Property
Color  = (Double, Double, Double)
Point = (Double, Double, Double)
Point2D = (Double, Double)
Vec = (Double, Double, Double)
Ray = Ray Point Vec

ConeParts = ConeSides | ConeAll
```

```
CylinderParts = CylinderSide |  CylinderAll

Shape = Cube (Double, Double, Double) -- width height depth
      | Cone (Double, Double) ConeParts -- bottomRadius height parts
      | Cylinder (Double, Double) CylinderParts -- radius height parts
      | Sphere (Double, Double) -- radius
      | Polygon [Point2D]

Property = DiffuseColor Color |
           AmbientColor Color |
           SpecularColor Color |
           EmissiveColor Color |
           Shininess Double |
           Transparency Double |
           MatrixTransform Matrix |
           Translate Vec |
           Scale (Double, Double, Double) |
           XRotate Double |
           YRotate Double |
           ZRotate Double |
           Eta Double

Projection = Orthographic | Nonorthographic

Light = DirectionalLight Double Vec Color
      | PointLight Double Point Color
      | SpotLight Double Point Vec Double Color Double

FileInput = Data ModelTree [Light] (Int, Int) Point Projection
```

The file format and internal data structure for the ray trace uses a nested tree structure similar to that which is used by OpenInventor. This allows for simple composition of subscenes. This representation is not necessarily the easiest to manipulate when rendering a scene as the properties associated with a given instance of a shape primitive are not immediately available and would require searching upwards through the tree to find the reaching definition. To actually describe the world in a manner that all relevant properties of an object are defined in the along with the definition of the object would not only require a far more verbose and awkward world description, but would also require the compostition of all transformation between the defined object’s space and world space to be precomputed. This would make editing portions of the scene graph generally unmanageable and produce a world representation which would be barely human readable at best. Using a tree representation allows the user to specify the world in a more natural hierarchal manner.

The file format mirrors the data structure almost exactly since the parser was automatically generated from the description of the data structure. The above BNF is essentially identical to the code which was used to generate it.