A COMPARISON OF PARALLELIZATION AND PERFORMANCE OPTIMIZATIONS FOR TWO RAY-TRACING APPLICATIONS

Chen Yang, Yongjian Chen
Programming Systems Lab/MTL
Intel China Research Center
8F, Raycom Infotech Park A, No. 2
KeXue Yuan South Road, ZhongGuanCun
HaiDian Dist, Beijing, China 100086
E-mail: {chen.yang, yongjian.chen}@intel.com

Xiong Fu
Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui, China 230027
Email: fuxiong@ustc.edu

Chu-Cheow Lim
Developer Products Division, SSG, Intel
SC12-305, 2200 Mission College Blvd
Santa Clara, CA 95052-8119, USA
Email: chu-cheow.lim@intel.com

Roy Ju
Google Inc.
1600 Amphitheatre Parkway
Mountain View, CA 94043, USA
Email: royju@google.com

KEYWORDS
Ray-tracing, POV-Ray, PBRT, Parallelization, Performance analysis.

ABSTRACT

Ray-tracing is a compute-intensive technique to render three-dimensional graphics and is embarrassingly parallel. There are many examples of ray-tracing applications from both academic and public domain projects. We ask if the parallelization and optimization issues are similar. This paper uses two sequential ray-tracing applications (POV-Ray (Persistence of Vision Raytracer) and PBRT (Physically-based Ray Tracer)) to compare the issues. We describe every step of the process, from parallelization, debugging/ verification to optimization of the parallelized codes. We use a performance model of the applications to guide us on the speedups' predicted upper bounds. We also profile the execution characteristics of both applications in their final optimized version. The general experience is that POV-Ray needed more efforts to parallelize and debug than PBRT, but less efforts to optimize.

1. INTRODUCTION

Our work is motivated by the industry trend towards multi-core chip (or chip multiprocessors), e.g. IBM's Power4, dual-core Pentium 4 and dual-core Itanium. While much work has been done for compiler auto-parallelization [1,2,4,5,14] and there are commercial products that includes auto-parallelization features (e.g. PGI [10]), it is also well-acknowledged that effective parallel programming often requires a user to manually break down the computation into independent tasks, possibly re-factoring the source code in the process.

This paper compares the issues encountered when we parallelized, debugged and optimized two applications from the same domain. It is *not* on compiler features and tools.

Ray-tracing is a compute-intensive technique to render three-dimensional graphics, by simulating the way light rays travel. With the popularity of computer-generated animation and games, it also has many practical uses.

The general understanding of ray-tracing is that it is embarrassingly parallel because each ray's computation is independent of the others. We picked an embarrassingly parallel application because the focus is on extracting the parallelism and optimizing the implementation of a parallel algorithm, and not designing a new parallel algorithm. Much work has in fact been done on parallel ray-tracing [3,13].

There are many examples of ray-tracing applications from both academic (e.g. PBRT [9], Rayshade (http://graphics. stanford. edu/ ~cek/ rayshade/ rayshade.html), OpenRT (http:// www. openrt. de)) and public domain projects (e.g. POV-Ray [11], PARRT (http:// www. nongnu. org/ parrt/)).

POV-Ray [11] is public domain, with free source code. It is arguably one of the most popular (if not the most) ray-tracing software, though it is not the fastest. Another publicly available ray-tracing code is PBRT (Physically based ray tracer) [9]. PBRT uses more sophisticated data structures and algorithms, and is from academic research written in a literate programming style. We compare the efforts needed to get optimized parallel codes of a faster, more academic-oriented ray-tracing implementation written in a literate programming style vs. a slower, more popular, public domain one.

Both are reasonably well-written, with source code and documentation available. These two codes are also up-to-date, unlike other distributions which have not been updated for a while.

Section 2 will describe how the applications were parallelized, and compare the issues encountered in verifying the correctness of the two parallelized codes. Section 3 describes the tuning of parallelized POV-Ray and PBRT on Linux. When we ported the parallel

Proceedings 20th European Conference on Modelling and Simulation
Wolfgang Borutzky, Alessandra Orsoni, Richard Zobel © ECMS, 2006
ISBN 0-9553018-0-7 / ISBN 0-9553018-1-5 (CD)

POV-Ray to Windows, more optimization was needed due to the nature of the operating system (section 4). We compare the two parallel programs' distribution of task execution times and instruction distribution (section 5). Section 6 concludes the paper.

2. PARALLELIZATION OF POV-RAY, PBRT

From the literature, ray-tracing can be parallelized in two ways: sub-space (scene space) division and sub-scene division. Sub-space division uses the fact that the pixels are independent, so that each pixel's ray is traced in parallel. Since adjacent rays have similar behavior (ray coherence) and have a high probability of accessing the same addresses, there is an opportunity to improve the parallel code's spatial locality and hence its performance. In order to achieve better ray coherence [6] and cache utilization [13], pixels are organized into tiles, so that each independent task performs ray-tracing for a tile, rather than for a pixel. PARRT, PVM POV-Ray and OpenRT all use screen space subdivision [6, 12] as a proven effective approach to parallel ray-tracing. The tile size and number of threads determine the lock contention and load imbalance.

There needs to be a way to assign the created tasks to available processing units (threads). We use a dynamic task assignment scheme to achieve better load balance. The tasks are inserted into a work queue shared by the threads. Each thread gets its task from the shared queue. (For both POV-Ray and PBRT, the threads are created after the input scene has been parsed.) The only difference is that in POV-Ray, all the tiles are inserted into the work queue before ray-tracing begins while in PBRT, each tile is computed on demand, i.e. whenever a thread attempts to get a tile.

The synchronization requirements are similar in both POV-Ray and PBRT. There is a barrier for all the threads before rendering is performed (and after ray-tracing is completed for all the pixels). A lock is used to protect the work queue.

It is not surprising that POV-Ray and PBRT share much similarity in terms of the implementation work done to decompose the sequential computation, and to assign the tasks to threads.

2.1 Comparison Of Parallelization Efforts

The parallelization efforts taken for both applications are different. It is difficult to quantify the impact of the programming language on the amount of parallelization efforts, because of human factors (e.g. the experience of the original authors). We do however find that the object-oriented nature of PBRT's C++ code, with limited use of class inheritance and polymorphism ease the parallelization efforts.

POV-Ray is implemented mainly in C. The ray-tracing computation code references global variables and function calls. The global variable references are scattered in different files and functions. It was therefore difficult to *manually* analyze the dependences to preserve their correctness after parallelization. A lot of efforts were expended to decide whether the variables should be private or shared due to lack of modular design in the POV-Ray code. To illustrate this point, we privatized about 300 variables to be thread-private. Among these 300 variables, about 70 variables are accessible in multiple source files. Table 1 shows how 4 of these 70 variables are accessed. It was more difficult to understand the dependences in POV-Ray because the global variable references are scattered in many different files and functions.

Reference	**Number of distinct locations**	**Number of distinct functions**	**Number of distinct files**
ADC_Bailout	35	10	5
Frame	455	83	23
Opts	1708	165	38
Stats	198	80	44

Table 1: Examples of Variables in POV-Ray Referenced in Multiple Locations

PBRT is implemented in C++. The code structure abstracts the ray tracer into classes. The class definitions provide a static scoping for us to analyze the use-def relationships of the variables, making it easier to analyze the dependence within the program. The use of class inheritance, on the other hand, means that we needed to analyze the relationship of a class's methods with its parent class's methods, and this demanded more efforts in understanding the existing dependences. The use of polymorphism also poses a similar problem. Since polymorphism allows for dynamic binding, we had to be conservative when manually analyzing the dependences of the candidate methods.

As an example, the *Shape* class has an *IntersectP()* method. Other classes *Cones* and *Cylinder* inherit from *Shape*. We call "shape->IntersectP(r);" to compute the intersections of the rays with objects in the scene. To decide if this call is thread-safe, we need to analyze all the "*IntersectP()*" definitions in parent and child classes.

PBRT limits most classes to at most 3 levels of inheritanc. Also the polymorphic functions which we needed to manually analyze, mostly are intersection tests with memory reads only.

For PBRT, the use of an object-oriented language with class encapsulation, together with the limited usage of class inheritance and polymorphism meant that it took less effort to understand the dependences. Our code browsing was limited to static fields in classes, static methods and invocation of object methods, and the static variables declared in the methods. The PBRT code is also more modular in the sense that there is a limited number of write accesses to shared variables.

2.2 Verification and Debugging of Parallelized Code

The next and more difficult step was actually the verification of the parallelized codes, to be confident that the results are valid. We are not ray-tracing experts, so the obvious way is to check the results of a parallel execution against that of a sequential or single-thread execution.

We found two *similar* categories of bugs.

Data races.

We encountered data race bugs in both applications. We used the Thread Checker tool [14] to try to locate the data races. The Intel Thread Checker collects profile data when an instrumented multi-threaded application executes and analyzes the data for errors e.g. data races, deadlocks.

The tool helped to find several data races in POV-Ray, e.g. the result buffer was not protected when the threads write their results separately into the buffer.

PBRT uses dynamic module loading which was not supported by the version of Thread Checker we used. The version we used could capture the race point but was unable to associate it with the source code. The user therefore could not map the problem back to the source code. We found a few data races in PBRT by eye-balling the source code.

Thread Checker depends on instrumentation to analyze data races observed during program execution. The conventional expectation is that the instrumented code has a slow-down of 10x-100x, but it can be worse. An input which took POV-Ray 78s to run on 2 threads, took 36 hours to complete on Thread Checker[1]. We trimmed the original input set from 384x384 (with 32x32 tiles) to be 8x8 with 4x4 tiles. Using the Thread Checker required more efforts than expected.

Random number generator.

A second issue was the random number generators used in POV-Ray and PBRT. With multiple threads accessing the same random number generator, different results were obtained for different runs even with the same number of threads. In POV-Ray, the problem was fixed by pre-generating the random numbers, so that every tile gets the same random number independent of which thread / processor it is executing on.

PBRT uses a Mersenne Twister random number generator. It uses a lot more random numbers so we could not pre-generate the numbers. We tried two approaches: (i) to duplicate the random number generator for each thread and (ii) was to use a parallel version of the Mersenne Twister generator [8].

Summary

In the verification/debugging process, POV-Ray and PBRT definitely share enough similarity such that understanding the problem (e.g. random number generator) in POV-Ray helped the work on PBRT. On the other hand, the Thread Checker was more effective for POV-Ray than PBRT in isolating the data race bugs, and the random number generator solutions employed were quite different for POV-Ray and PBRT, respectively.

3. TUNING PARALLELIZED POV-RAY, PBRT

The next step is to understand the performance bottlenecks. We used a preliminary performance prediction tool (PerfMiner) to predict the parallel performance as the number of processors change. PerfMiner uses a discrete-event simulation approach to estimate the scheduling and synchronization of the tasks. The predictions have a tighter upper bound than models based on Amdahl's Law because they also take load imbalance and synchronization behavior into consideration while Amdahl's law only takes the sequential bottlenecks into account.

Our tool breaks the computation of the parallel program is broken into discrete "computation chunks", between synchronization points. We ran the parallel program on a single thread. A discrete-event simulator uses the measured timings of the chunks and their dependence / synchronization information to simulate how the chunks may be scheduled and how they synchronize on different number of threads.

If a parallel execution on N threads has no additional overheads (e.g. additional cache misses and bus contention), and if each task in a parallel execution takes the same amount of time as it would in the sequential execution, the parallel execution time measured should match the predicted parallel time.

We followed "prediction-guided optimization" methodology, by comparing the predicted and measured speedups. As the graphs will show, we did uncover significant performance bottlenecks every time a gap exists between measured and predicted numbers. Eventually the measured speedups do indeed match the predicted speedups. This experience is the same in both POV-Ray and PBRT, even though the bottlenecks tend to be implementation-specific.

Initial measurements show that both parallel POV-Ray and PBRT scale poorly when compared to the predicted speedups. For example, POV-Ray's execution time was worse for 4 threads than for 2 threads. The measured speedups were far below the predicted speedups. Because we can do a comparison of what is possible (model's prediction) vs. what is happening (actual execution), and the gap is large, we believe it worthwhile to put in more efforts to discover and fix these performance bottlenecks (e.g. using Vtune).

[1] Thread Checker has since reduced its slowdown.

3.1 Privatization

We first looked for privatization opportunities to reduce true-sharing. We protected shared data structure with locks, or access them with atomic operations. Some data structures however need not be shared and can be replicated to improve memory locality.

We used the Vtune Performance Analyzer to identify the performance bottleneck(s) for both POV-Ray and PBRT. The last level cache miss data showed that they both suffered from the usage of shared statistic counters, but to different degrees. Such counters are used to track information, e.g., the number of intersections and the number of traced rays. While the statistics counters were the most important reason for POV-Ray's performance bottleneck, they were not so for PBRT.

POV-Ray has about 130 statistic counters, and only about 40 are used in output only for user's reference information. The shared accesses of these counters by multiple threads were the main performance bottleneck for POV-Ray. In Figure 2, we measured POV-Ray's speedups on a 4-processor 2 GHz Xeon machine (without hyperthreading [2]) using a benchmark input provided in the POV-Ray package.

The "Baseline version" curve gives the speedups of the initial parallel version of POV-Ray. The "Predicted speedup" curve gives an expected upper bound using a performance model based on Amdahl's Law. Figure 2 shows that the speedup after the statistics counters were privatized ("Optimized version (with privatization)") improved and matched the predicted speedup very well.

PBRT also has about 10 statistic counters accessed by the threads, but they were not the main performance bottleneck. The first three lines in Figure 3 show that PBRT's speedup curve improved after privatization ("Optimized version (with privatization)"), but the measured speedups did not match the predicted speedup.

This optimization differs in its effectiveness due to the differences in specific implementations.

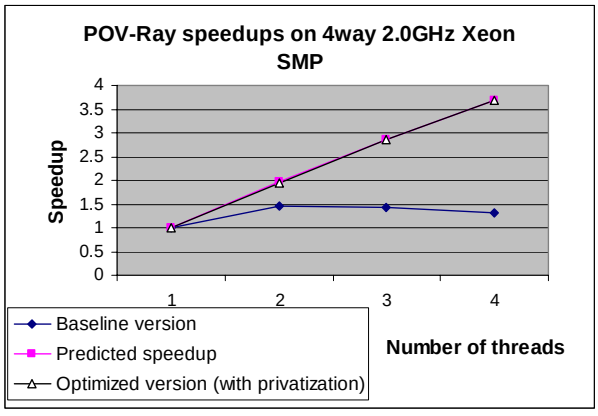

Figure 2: Comparison of POV-Ray Measured Speedups Before and After Privatization

[2] All our measurements do not use hyperthreading.

3.2 Data Structure Replication

A ray may be tested for intersection against the same primitive multiple times. To avoid redundant tests, PBRT uses a technique called mailboxing to quickly determine if a ray has already been tested against a specific primitive. The implementation relies on keeping track of the latest ray that intersects with each primitive within a *MailboxPrim* structure (associated with each primitive), via a *lastMailboxId* field. In the parallel version for PBRT, a straight-forward way to ensure that mailboxing works for multiple threads was to extend the *lastMailboxId* field into an array so that each thread accesses its own element.

PBRT implements acceleration data structures (in the form of a grid or k-d tree) to speed up a ray's test of intersection against all the objects in a scene. As part of the mailboxing implementation, the accelerator data structure keeps track of the current ray with a *curMailboxId* field. In the parallel PBRT, we similarly extend the single integer field into an array of integers.

We also replicated other data structures for each thread before thread creation (e.g. *Sample* object into which the ray-traced results of a sample point are stored).

With the above changes, each thread is able to use the accelerator data structure independently without interfering with other threads.

The fourth line in Figure 3 shows that PBRT's speedup is further improved with data replication. It is however still below the predicted speedup, e.g. for 4 threads, the measured speedup is 3.12 against the predicted speedup of 3.71.

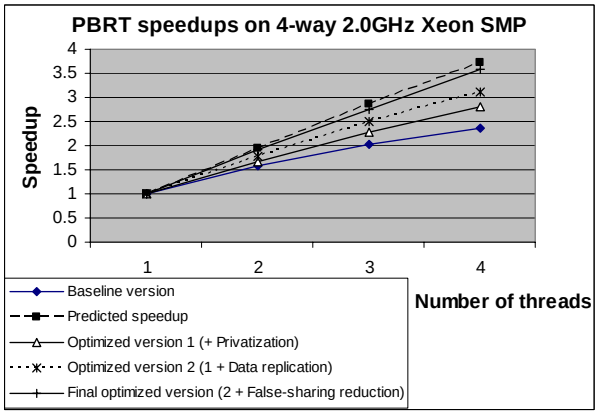

Figure 3: PBRT Speedups with Privatization, False-sharing Reduction and Padding

3.3 False-Sharing Reduction

Section 3.2 describes how we extended the single integer fields to arrays of integers (*lastMailboxId[]* and *curMailboxId[]*) so that multiple threads can do mailboxing independently. But on our evaluation platform, the cache line size at all levels is 64 bytes. Since the intersection tests uses mailboxing and are a significant part of ray-tracing computation, this causes false-sharing for these arrays to become a significant

overhead. To eliminate false-sharing in parallel PBRT, we padded extra bytes in these arrays. The *lastMailboxId[]* array is further modified to *lastMailboxId[][16]*, and similarly for *curMailboxId[]*. The elements accessed by the ith thread (*lastMailboxId[i][0]* and *curMailboxId[i][0]*) are separated from those accessed by the (i+1)th thread by 64 bytes. Each thread should now access a different cache line and false-sharing is eliminated.

In Figure 3, we see that PBRT's speedup, after applying all the three optimizations in sections 3.1 to 3.3, is close to the predicted curve.

In this section, we have seen that the key performance bottleneck for POV-Ray and PBRT was similar, and was due to inter-processor memory interferences. The two applications however needed different solutions to fix the problem. For PBRT, in addition to variable privatization, we had to replicate data structures (including those for mailboxing) and reduce false-sharing by padding.

4. I/O AND HEAP OPTIMIZATIONS

The timings for the optimized parallel POV-Ray (Figure 2) were initially measured on a Linux machine. The Windows version of POV-Ray has a different source base. We also performed the same parallelization and optimization on POV-Ray's Windows code and measured the speedup of the optimized parallel POV-Ray on a 16-processor 3.0GHz Xeon Unisys (Windows) machine. The machine has a Cellular Multi-Processor (CMP) architecture where one Cell consists of 4x Xeon processors with 32MB shared cache.

The measured speedup on the 16-processor Unisys CMP however did not scale well, and diverged from the predicted speedup beyond 10 processors (the first two lines in Figure 5). Further analysis found an abnormal increase in kernel execution time (Figure 4) with increasing number of threads. We suspected the reasons to be either the file operations or blocking of threads when they fail to acquire the locks on the heap.

In order to reduce the file access overheads in parallel POV-Ray, we looked at POV-Ray's scene parsing which uses file input operations that read one byte at a time. We optimized it by loading the file contents into an input buffer, and accessing the bytes via an index pointer to the buffer. This eliminated the calls to fgetc(), ungetc() and fgets().

With this modification, kernel mode time usage was reduced (Figure 4), but the overall speedup (third line in Figure 5) is worse than the baseline version. After the I/O changes, the single-threaded execution time reduced from 1731 to 1718 seconds. With 2 or more threads, the version with I/O changes was about 2 to 13 seconds *slower* than that without the I/O changes.

The Vtune data showed that the memory allocation routines had severe L3 cache misses. There was a contention problem when the threads concurrently call *malloc* and *free* library routines. Every call of *malloc/free* is protected by a heap lock. (From Visual Studio's CRT/source/MALLOC.c file, a *malloc* call will acquire the heap lock before it tries to find a block for allocation, and it will release the lock after the allocated succeeds or fails.) With increasing number of threads, there is a severe contention for the heap lock.

The reason is because each thread now also performs its own initialization and object allocation (after reading the input independently).

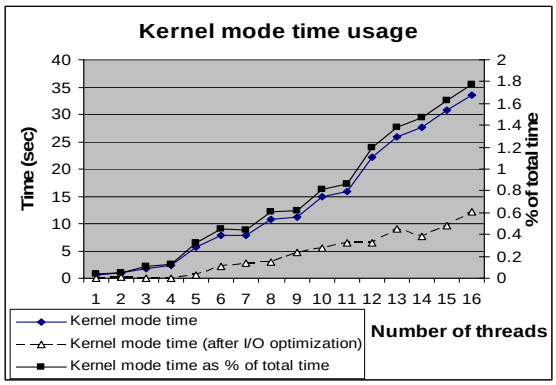

Figure 4: Kernel Mode Usage Time of Parallel POV-Ray (Windows) After I/O Optimization

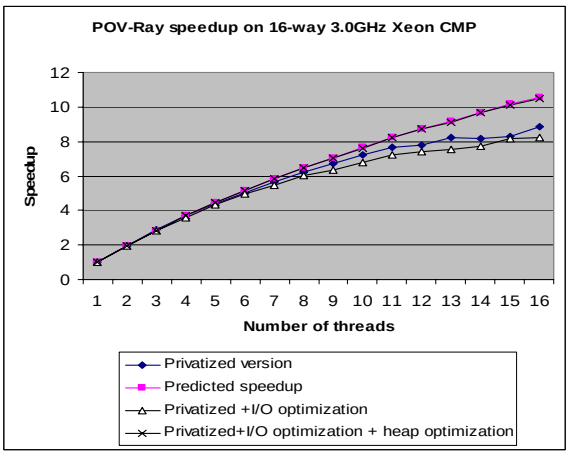

Figure 5: POV-Ray's Speedup Curves After Applying I/O Optimization and Heap Privatization

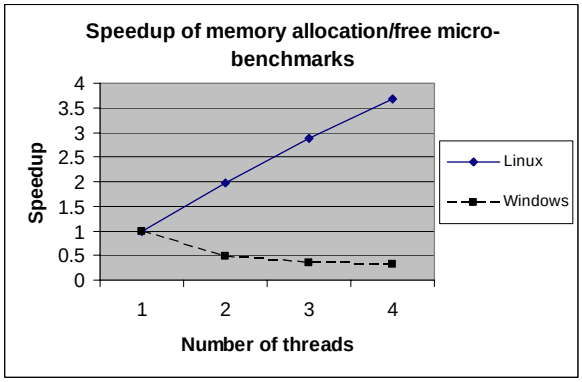

Figure 6: Performance of Micro-benchmark on Memory Management Routines on Linux and Windows Machines

Our solution was to privatize the heap for each thread. Figure 5 shows that POV-Ray's measured speedup was again close to the predicted curve after we applied both I/O optimization and heap privatization. We therefore include both modifications in the final optimized code.

We designed a micro-benchmark to evaluate the extent of the memory routine overhead. We did 1.2 million times of 512-byte allocations, distributed to different threads. We measured the micro-benchmark on a Linux and a Windows machine, up to 4 processors. Figure 6 shows that Linux's memory management library scales better than Windows' library. This experiment ensured that the original optimized parallel version of Linux POV-Ray would not have the same performance problem from the memory library.

The memory management routines show a library problem which we encountered for POV-Ray as a result of porting POV-Ray from Linux to Windows.

5. EXECUTION CHARACTERISTICS

This section compares two execution characteristics of POV-Ray and PBRT: (a) distribution of tile execution times, and (b) distribution of executed instruction.

Distribution of tile execution times

For POV-Ray and PBRT, even though our optimized speedup was close to the predicted upper bound, there appeared to be further room for improvement when we compared against the "ideal speedup". The ideal speedup is based on Amdahl's law and assumes the work is shared in a perfectly balanced way (Figure 7).

In order to explore this possibility, we should look into the load balancing issues.

To compare the distribution of tile execution times for both POV-Ray and PBRT in a more meaningful manner, we used inputs such that both applications generate approximately similar images. The sequential times are 104s and 70s for POV-Ray and PBRT respectively.

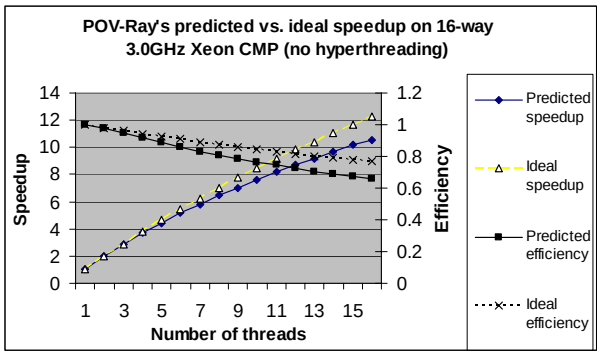

Figure 7: Predicted and Ideal Efficiency of POV-Ray on 16-Way Machine

Figure 8 shows the normalized distribution of the tile execution times for both POV-Ray and PBRT. We collected the execution time of each tile. For each set of timings, we computed the mean and standard deviation and normalized each individual time using the respective mean and standard deviation.

The graph shows that the tile execution times vary widely and load imbalance may be an issue, especially when the number of threads increases. It is well-known that load balancing can be improved if the threads' tasks are finer-grained (e.g. each thread works on a pixel or sample instead of a tile). But if the tasks are too fine-grained, the shared work queue's lock contention increases. One possible future experiment is a thread-private work queue that allows each thread to steal from other work queues when its own becomes empty.

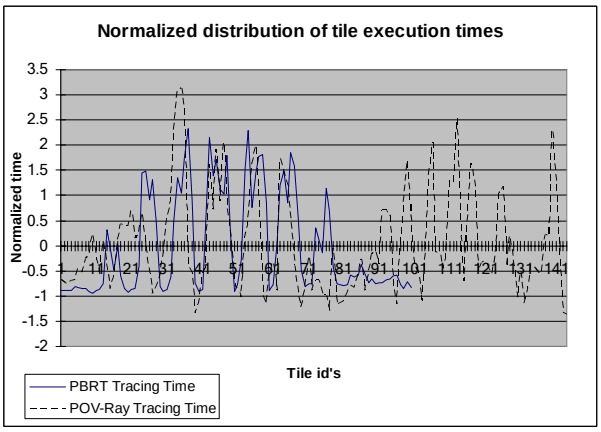

Figure 8: Distribution of Tile Execution Times for POV-Ray and PBRT on Comparable Images

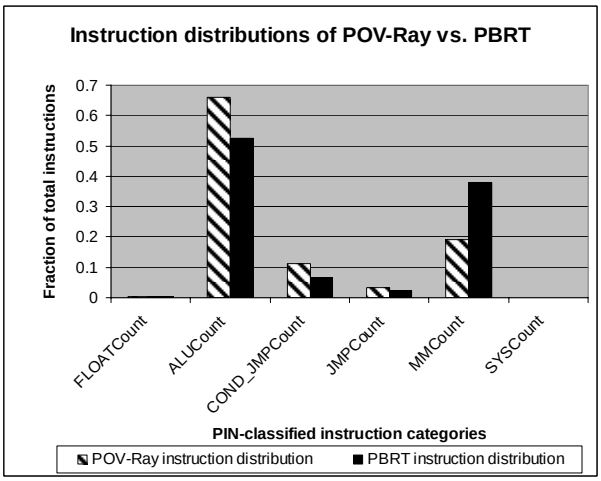

Figure 9: Breakdown of Instruction Categories for POV-Ray and PBRT on Comparable Inputs

Distribution of executed instructions

We also wanted to find out if the two ray-tracing applications differ in the types of instructions they execute. We modified Pin [7] to collect the instruction distribution. The categories provided by Pin for IA32 are floating point, integer, conditional and unconditional jumps, MMX and system calls. We found that PBRT has a higher proportion of MMX/SSE/SSE2 instructions, but we did not glean any

information about the proportion of memory instructions. This data shows that the Intel *icc* compiler is better able to vectorize the sequential PBRT code than POV-Ray.

6. CONCLUSION

This paper compares our experience in parallelizing and optimizing two ray-tracing applications. We classify the similarities / differences between the two applications in the sections 2 to 5: (a) parallelization work, (b) optimization, and (c) platform issues, and (d) general application profile.

Both codes were parallelized using the pthreads library, in a shared memory programming model.

A similarity between the two codes was that they each use a random number generator. The same type of bug occurred when multiple threads use the random number generator in parallel.

Our experience is that the amounts of efforts to modify POV-Ray into a multithreaded code were more than that needed for PBRT, mainly because of the work needed to manually identify the shared and private variables, by eye-balling the codes. The POV-Ray code was less modular with global references scattered among multiple functions and multiple files. This style of programming was probably partly a result of using C and possibly because it is public domain code with different contribution sources.

During optimization, the only similarity was that we applied privatization to both parallel POV-Ray and PBRT. For PBRT, we also replicated the data structure and introduced data padding. This difference in the types of optimizations is expected because optimization issues are usually implementation-dependent.

The parallel POV-Ray needed less optimization effort. The reason is that in the sequential POV-Ray code, there is pre-computation on the scene data before the actual ray-tracing. When we modified the code, the "natural" way of parallelizing was to have each thread perform its own pre-computation and as a result, the threads do not share the scene data. PBRT does not have such pre-computation and the straight-forward way of parallelizing tends to have shared scene data.

The distribution of tile execution times show that load balancing is an equally important issue for both applications with increasing number of threads. We need to understand if the higher proportion of MMX/SSE/SSE2 instructions in PBRT is due to algorithm or implementation differences. It would also be interesting to refine our Pin-modified tool to gather information on proportion of memory instructions.

We have used disparate tools (Vtune, Thread Checker, our performance prediction tool, a Pin-based tool) in this work. Both applications reinforce the need for a comprehensive, integrated tool-chain to make parallel programming easier on multi-core chips.

REFERENCES

1. Allen, Randy and Ken Kennedy. 1987. "Automatic trans Automatic translation of FORTRAN programs to vector form." *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9, No. 4 (Oct), 491 – 542.
2. Allen, Randy, D. Callahan and K. Kennedy. 1987. "Automatic decomposition of scientific programs for parallel execution." *Conference Record of the 14th ACM Symposium on Principles of Programming Languages*, 63 –76, Jan 1987.
3. Chalmers, Alan, Timothy Davis, Erik Reinhard (Editors). 2002. Practical Parallel Rendering. AK Peters, Ltd.
4. Collard, Jean-Francois. 1993. "Code generation in automatic parallelizers." Research Report No. 93-21, Laboratoire de l'Informatique du Parallelisme, Ecole Normal Superieure de Lyon, Instiut IMAG (Jul).
5. Eigenmann, R., J. Hoeflinger, Z. Li and D. Padua. 1991. "Experience in automatic parallelization of 4 Perfect benchmark programs." *Proc. Of the 4th Workshop on Programming Languages and Compilers for Parallel Computer*, Aug 1991.
6. Joy, Kenneth I. and Murthy N. Bhetanabhotla. 1986. "Ray tracing parametric surface patches utilizing numerical techniques and ray coherence." *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*. ACM Press, 279—285.
7. Luk, Chi-Keung, Robert Cohn, Robert Muth, et al. 2005. "Pin: Building customized program analysis tools with dynamic instrumentation." *Programming Language Design and Implementation Conference* (Jun 2005, Chicago, Illinois, USA).
8. Mersenne Twister: A random number generator. 1997. http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/emt.html .
9. Pharr, Matt and Greg Humphreys. 2004. Physically Based Rendering: From Theory to Implementation. Morgan Kaufman (Aug).
10. The Portland Group Fortran. C and C++ Compilers and Tools for Linux and Window. http://www.pgroup.com.
11. POV-Ray – The Persistence of Vision Raytracer. http://www.povray.org.
12. Reinhard, Erik. 1995. *Scheduling and Data Management for Parallel Ray Tracing.* PhD thesis, University of East Anglia, 1995.
13. Wald, Ingo, Carsten Benthin, Andreas Dietrich and Philipp Slusallek, "Interactive Distributed Ray Tracing on Commodity PC Clusters -- State of the Art and Practical Applications". In Lecture Notes on Computer Science 2790, 2003, 99-508.
14. Wolfe, Michael. 1990. "Massive parallelism through program restructuring". *Symposium on Frontiers on Massively Parallel Comuptation* (Oct) 407—415.
15. Intel Thread Checker. http://www.intel.com/software/products/threading/tcwin/
16. Intel Vtune Performance Analyzers. http://www.intel.com/software/products/vtune/