Benchmarking and Optimising Software for Improved Multiprocessor Performance

Waide Tristram[1] and Karen Bradshaw[2]
Department of Computer Science, Rhodes University
E-Mail: [1]g05t1067@campus.ru.ac.za [2]k.bradshaw@ru.ac.za

***Abstract*—Parallel computers have become widely available with the introduction of multicore processors, however software will need to be optimised to make use of the additional processors. A number of tools can aid the process of optimising code for parallel execution, such as parallel debuggers, profilers and parallel libraries. This paper describes an approach to evaluating the effectiveness of a selection of tools and optimisations. Additionally, an automated benchmark tool is described along with several modifications that assist in the benchmarking of optimisations. The benchmark tool has been tested and the preliminary results show an impressive increase in performance when using OpenMP.**

***Index Terms*—Benchmarking, Multicore Processors, Software Optimisation, Parallel Tools**

I. Introduction

MULTIPROCESSOR computer systems are currently on the rise with the majority of processors shipped by manufacturers such as Intel being multicore based designs [1]. The current trend towards more processor cores instead of higher clockspeeds can be attributed to the physical limitations of modern processor designs [2], [3]. This has serious implications for the design and development of applications as writing and debugging parallel software is difficult and requires more expertise than sequential programming [3]. Tools and frameworks to help programmers make the transition from sequential to parallel programming are becoming available, such as OpenMP and Thread Building Blocks, and many others are in development [1], [2]. This research aims to identify and evaluate the most effective tools and optimisations for harnessing the processing power of multiprocessor computers. It is hoped that these techniques can be used to improve the performance of transaction based services, thus alleviating the need to distribute the workload over multiple servers.

This paper provides a brief overview of related work. It then goes on to describe the approach chosen for investigating the various available tools and optimisations. The focus of the paper is the work carried out on the automated benchmarking of target software, which is detailed in Section III. The paper then presents preliminary results captured using the automated benchmarking tool and finally, our conclusions.

II. Related Work

Utilising all the available CPU cores of a computer system requires that software be able to spread its workload across multiple processors [3]. This is typically achieved using multi-threading, although other techniques such as message passing can be employed [3]. There are limits to how far one can parallelise a program, and these affect the performance and scaling of software on parallel computers [2], [4].

The problem lies with Amdahl's law, defined in (1).

$$S = \frac{1}{(1-P) + \frac{P}{N}}, \tag{1}$$

where S represents the speedup, P the proportion of the program that executes in parallel and N the number of processors [4]. This law states that both speedup and scaling are dictated by the sequential portion of the program. This means that given a P of 0.9 and N of 10, the speedup is only ~5.26x, which is not ideal.

It is therefore crucial to make every effort to reduce the sequential portion to increase speedup [4]. However, code that works well with four processors may not scale with eight or even sixteen processor cores [2].

Even with concurrent code, there can be bottlenecks. Inefficient locking mechanisms can cause a drastic decrease in performance under load [4]. Lock granularity also needs to be considered as coarse granularity can result in high contention if a method holds the lock without actually needing it [5].

Identifying the appropriate sections of code to optimise requires profiling the target program with a parallel aware profiler such as TAU [6]. Although optimising hotspots can be achieved by manually implementing multi-threading, the use of parallel abstractions such as OpenMP, Cilk and Thread Building Blocks is recommended [2], [7], [8].

III. Research Design

The aims of this research require that various tools and optimisations be evaluated to gauge their effectiveness in improving software performance on multiprocessor systems. Accordingly, the approach taken focuses on experimentation and benchmarking of optimisations and tools.

The first step is to evaluate the available profilers, debuggers, libraries, and compilers that are targeted at parallel programming. Concurrent to investigating tools, research will be carried out to identify suitable optimisation techniques for improving multi-threading performance. These optimisations usually require code changes, such as modifying sequential algorithms, instead of using a tool. Once a list of suitable tools and optimisations has been compiled, some basic experimentation will be carried out in order to gain familiarity before commencing with the proper investigation.

Selected software, such as the SIP Express Router (SER) and other transaction based applications, will be benchmarked and profiled to record their baseline performance

The authors would like to acknowledge the financial support of Telkom, Comverse, Stortech, Tellabs, Amatole Telecom Services, Mars Technologies, Bright Ideas 39, and THRIP through the Telkom Centre of Excellence in the Department of Computer Science at Rhodes University.

before optimisation. Thereafter, iterative optimisation and testing of each tool and technique will be performed. This involves using the automated benchmarking tool, which is described in Section III-A. Each optimisation is applied to a fresh copy of the source, which is then compiled and benchmarked. Combinations of tools and techniques will be tested where possible. Once satisfactory results have been gathered, they will be analysed to draw conclusions and recommendations about each of the tested tools and optimisations.

A. Automated Benchmarking Tool

This project requires repetitive testing and benchmarking of changes, which, if done manually would be both tedious and prone to error and inconsistency. As a result, an automated benchmarking tool has been developed. Work thus far has focused on implementing and refining such a tool.

Instead of building the automated benchmarking tool from scratch, a similar tool was found and subsequently modified. The Computer Language Benchmarks Game [9] is a website that compares language implementations using a variety of benchmarks. To automate the process of benchmarking dozens of language implementations for each test program, the author of the site, Isaac Gouy, developed a benchmark tool written in Python [9]. The sourcecode for the benchmark tool is available under the GPL, thus allowing it to be downloaded and freely modified.

B. Modifications to the Automated Benchmarking Tool

Better result and run information logging has been implemented, such that instead of overwriting previous results and run logs, the measurements and logs for each run are stored according to the date and time of the run. Additional information is also recorded for each run, such as the environment and build options used for the run. All of the files for the benchmark tool and the test programs are added to a Subversion (SVN) repository to allow for revision control of the benchmark suite. The SVN revision information is then recorded along with the logs for each run. This provides a number of useful features, such as the ability to revert to a specific set of changes to replicate a test run and to generate a *diff* file showing the code changes between runs.

The tool has also been modified to allow for the testing of different compilation flags using the same compiler. The ability to build projects using tools such as *make* has been added, instead of only allowing single source code files. Work is currently underway to allow for measuring programs with a specific benchmark tool (such as *SIPp* for SIP implementations like SER) and further work includes building a web interface for viewing results.

IV. Preliminary Results

To demonstrate the automated benchmarking tool, a number of simple benchmark tests were performed. A C++ implementation of the Mandelbrot fractal algorithm was benchmarked using the GNU GCC compiler. The hardware and software used for the test are listed in Table I, while the results can be viewed in Table II.

Regarding the actual benchmarking process, the automated benchmark tool made setting up the test and gathering results very easy. Looking at the results, it can be seen that there is no significant difference between the -Os, -O2, and -O3 optimisation flags, however -O1 is noticeably slower. More interestingly, the use of OpenMP allows for all the CPU cores to be utilised, providing a significant increase in performance, whilst only requiring the addition of two pragma lines and a compilation flag.

Table I
Hardware and Software Configuration

Component	Specification
CPU	Intel Q9400 Quadcore 2.66GHz
RAM	4GB DDR2 800MHz
OS	Gentoo Linux AMD64
GCC	Version 4.3.3

Table II
Benchmark Results

Flags	CPU(s)	Mem(KB)	CPU Load(%)				Time(s)
-Os	22.459	31 144	29	2	75	1	22.499
-O1	24.037	30 480	2	1	1	100	24.053
-O2	22.453	31 136	100	0	2	3	22.483
-O3	21.856	31 168	4	1	99	0	21.867
-Os -fopenmp	22.34	28 476	96	98	97	98	5.806
-O1 -fopenmp	26.401	31 288	95	97	100	97	6.863
-O2 -fopenmp	22.034	29 432	97	99	99	96	5.708
-O3 -fopenmp	22.589	30 512	99	98	96	98	5.835
Common compiler flags: -mfpmath=sse -msse3							

V. Conclusions

As the number of CPU cores on modern processors is increasing steadily, software needs to be designed with the future in mind. Various tools exist to aid multiprocessor development, but it is necessary to identify the most effective tools. This requires extensive testing of the various tools and optimisations, and therefore a benchmark tool has been developed to make testing easier. Preliminary tests show that tools such as OpenMP can have a significant positive effect on multiprocessor performance.

References

[1] R. Merritt. "Chip industry confronts 'software gap' between multicore, programming." EETimes. Apr. 3, 2008. [Online]. Available: http://www.eetimes.com/news/semi/showArticle.jhtml?articleID=207001633 [Accessed: Jun. 3, 2009].

[2] K. Carlson. "Sd west: Parallel or bust." Dr. Dobb's Journal. Mar. 7, 2008. [Online]. Available: http://www.ddj.com/hpc-high-performance-computing/206902441 [Accessed: Jun. 2, 2009].

[3] B. Hayes, "Computing in a parallel universe," *American Scientist*, vol. 95, pp. 476–480, 2007.

[4] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, March 2008.

[5] B. Goetz. "Threading lightly, part 2: Reducing contention." Sept., 5 2001. [Online]. Available: http://www.ibm.com/developerworks/java/library/j-threads2.html?ca=drs- [Accessed: Apr. 3, 2009].

[6] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[7] M. Frigo, "Multithreaded programming in cilk," in *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*. New York, NY, USA: ACM, 2007, pp. 13–14.

[8] T. Mattson and M. Wrinn, "Parallel programming: can we please get it right this time?" in *DAC '08: Proceedings of the 45th annual conference on Design automation*. New York, NY, USA: ACM, 2008, pp. 7–11.

[9] I. Gouy. "Computer language benchmarks game." May 2009. [Online]. Available: http://shootout.alioth.debian.org/u32q/faq.php [Accessed: May 28, 2009].

Mr Waide Tristram completed his Honours in Computer Science in 2008 under the guidance of Dr Karen Bradshaw in the Department of Computer Science at Rhodes University. Waide is now in his first year of MSc in Computer Science and working for the Centre of Excellence (CoE) at Rhodes University.