Evaluating Chip Multiprocessor Performance

Project Report – Fall 2015 – 16:332:563 – Computer Architecture I

Manuel O. Maldonado

Email: manuel.maldonado@rutgers.edu

RUID: 165000201

Ming Tai Ha

Email: ming.tai.ha@rutgers.edu

RUID: 133004530

# Abstract

Our team tested three versions of the Intel i7 processor, a Haswell, a Broadwell and an Ivy Bridge, using the NASA Advanced Supercomputing Division (NAS) Parallel Benchmark suite to determine the speedup provided by leveraging the processor’s chip multiprocessor architecture. Our testing showed a linear trend in speedup for all three versions when increasing the number of cores, additionally we found that cache and memory sizes quickly became the bottleneck of parallel programs. We will discuss our hardware configuration, the benchmark suite, assumptions, our approach to testing and results in this report.

# Introduction

The typical CPU performance growth that we have come to know since the 1970’s hit a wall in the early 2000’s. As the number of transistors continued to raise, clock speed and performance, however, was a different story. It became harder to exploit higher clock speeds due to several physical issues, most notably heat, power consumption, and current leakage problems. Since then manufacturers have been able to bridge this performance gap in new chips with three main approaches, only one of which being the same as in the past, these are: hyperthreading, cache, and multicore.

With multicore processors you have two or more actual CPUs on one chip, increasing the overall performance of a processor; for example, doubling the number of cores in a CPU would theoretically double the processing power. Our team set out to test this effect by benchmarking several multicore processors. We ran applications specifically designed to take advantage of the parallel nature of multicore processors on different number of cores, collected run statistics related to execution time and analyzed them to corroborate this theory.

# Hardware

In this section we will describe the two computer systems we used in our benchmarking experiment, each with a different multicore processor.

The first computer system consisted of a laptop running the Ubuntu 14.04 Linux operating system. It had a fourth generation Intel i7-4720HQ processor (based on Intel’s Haswell microarchitecture), with four cores (eight threads total), running at 2.6GHz. The processor also had a shared L3 cache with 6MB, an L2 cache of 256KB per core, and an L1 cache of 64KB per core. The system had a total of 8GB of RAM.

The second computer system was a laptop running the OS X 10.11 (El Capitan) operating system. It had a fifth generation Intel i7-5557U processor (based on Intel’s Broadwell microarchitecture), with two cores (four threads total), running at 3.1GHz. The processor also had a shared L3 cache with 4MB, an L2 cache of 256KB per core, and an L1 cache of 64KB per core. The system had a total of 16GB of RAM.

In both systems, we disabled any power management settings provided by the operating system and both laptops were provided power through the entire run. Finally, we chose to run the benchmark in two different systems to show that even on different systems, the speedup gained from multicore processors is significant, similar and relevant by today’s standards.

The system which we will discuss is the Linux Ubuntu computer system. This is because the processor of this system has 4 cores, whereas the second computer system has only 2 cores. The data for the benchmark tests for the second computer system can be found in the Appendix section.

# Multiprocessing Platform

As our team ran the benchmark in different systems, it became apparent that we needed a common multiprocessing framework as to make sure we were measuring speedup in relation to increasing the number of cores and not because of the implementation of the multiprocessing framework in each system. For this reason, we decided that for a benchmark suite to be eligible it needed to support the Open Multi-Processing (OpenMP) application programming interface (API).

OpenMP is a specification for a set of compiler directives, library routines, and environment variables used to specify high-level parallelism in programs. OpenMP has become a standard platform for parallel programming on shared memory systems and it provides code extensions/directives to make programs run in parallel. It supports most platforms, processor architectures and operating systems, including Linux and OS X, and it is available C, C++, and Fortran.

# Benchmarking Suite

For a benchmarking suite, we decided to go with the NASA Advanced Supercomputing (NAS) Parallel Benchmarks (also known as NPB). These are a small set of programs, five kernels and 3 pseudo applications derived from computational fluid dynamics (CFD) applications, designed to help evaluate the performance of parallel supercomputers. All of these programs leverage the OpenMP platform for multiprocessing needs and were implemented in the Fortran programming language. Complete details on the benchmark suite and each of the programs can be found in the references section of this report, but below is a high-level description of each:

* Five Kernels:
  + Embarrassingly Parallel (EP) benchmark. It generates pairs of Gaussian random deviates according to a specific scheme. The goal is to establish the reference point for peak performance of a given platform.
  + Multigrid (MG) benchmark. It uses a V-cycle MultiGrid method to compute the solution of the 3-D scalar Poisson equation. The algorithm works continuously on a set of grids that are made between coarse and fine. It tests both short and long distance data movement.
  + Conjugate Gradient (CG) benchmark. It uses a Conjugate Gradient method to compute an approximation to the smallest eigenvalue of a large, sparse, unstructured matrix. This kernel tests unstructured grid computations and communications by using a matrix with randomly generated locations of entries.
  + Fast Fourier Transform (FT) benchmark. It contains the computational kernel of a 3-D fast Fourier Transform-based spectral method. FT performs three one-dimensional (1-D) FFT’s, one for each dimension.
  + Integer Sort (IS) benchmark. It performs a sorting operation that is important in “particle method” codes. It tests both integer computation speed and communication performance.
* Three simulated CFD pseudo applications:
  + Block Tri-Diagonal Solver (BT) benchmark. It is a simulated CFD application that uses an implicit algorithm to solve 3-dimensional (3- D) compressible Navier-Stokes equations. The finite differences solution to the problem is based on an Alternating Direction Implicit (ADI) approximate factorization that decouples the x, y and z dimensions. The resulting systems are Block-Tridiagonal of 5×5 blocks and are solved sequentially along each dimension.
  + Scalar Penta-Diagonal Solver (SP) benchmark. It is a simulated CFD application that has a similar structure to BT. The finite differences solution to the problem is based on a Beam-Warming approximate factorization that decouples the x, y and z dimensions. The resulting system has Scalar Pentadiagonal bands of linear equations that are solved sequentially along each dimension.
  + Lower-Upper Gauss-Seidel Solver (LU) benchmark. It is a simulated CFD application that uses symmetric successive over-relaxation (SSOR) method to solve a seven-block-diagonal system resulting from finite-difference discretization of the Navier-Stokes equations in 3-D by splitting it into block Lower and Upper triangular systems.

Next, each benchmark comes in multiple data classes (i.e. data size). The size of the data for each benchmark class used in our testing can be seen in the table below.

Table - Problem sizes and parameters for each of the classes. Empty cells in the table indicate undefined problem sizes.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Benchmark | Parameter | Class S | Class W | Class A | Class B | Class C |
| CG | no. of rows | 1400 | 7000 | 14000 | 75000 | 150000 |
| no. of nonzeros | 7 | 8 | 11 | 13 | 15 |
| no. of iterations | 15 | 15 | 15 | 75 | 75 |
| eigenvalue shift | 10 | 12 | 20 | 60 | 110 |
| EP | no. of random-number pairs | 224 | 225 | 228 | 230 | 232 |
| FT | grid size | 64 x 64 x 64 | 128 x 128 x 32 | 256 x 256 x 128 | 512 x 256 x 256 | 512 x 512 x 512 |
| no. of iterations | 6 | 6 | 6 | 20 | 20 |
| IS | no. of keys | 216 | 220 | 223 | 225 | 227 |
| key max. value | 211 | 216 | 219 | 221 | 223 |
| MG | grid size | 32 x 32 x 32 | 128 x 128 x 128 | 256 x 256 x 256 | 256 x 256 x 256 | 512 x 512 x 512 |
| no. of iterations | 4 | 4 | 4 | 20 | 20 |
| BT | grid size | 12 x 12 x 12 | 24 x 24 x 24 | 64 x 64 x 64 | 102 x 102 x 102 | 162 x 162 x 162 |
| no. of iterations | 60 | 200 | 200 | 200 | 200 |
| time step | 0.01 | 0.0008 | 0.0008 | 0.0003 | 0.0001 |
| (BT-IO) | write interval | 5 | 5 | 5 | 5 | 5 |
| Gbytes written | 0.0008 | 0.022 | 0.42 | 1.7 | 6.8 |
| LU | grid size | 12 x 12 x 12 | 33 x 33 x 33 | 64 x 64 x 64 | 102 x 102 x 102 | 162 x 162 x 162 |
| no. of iterations | 50 | 300 | 250 | 250 | 250 |
| time step | 0.5 | 0.0015 | 2.0 | 2.0 | 2.0 |
| SP | grid size | 12 x 12 x 12 | 36 x 36 x 36 | 64 x 64 x 64 | 102 x 102 x 102 | 162 x 162 x 162 |
| no. of iterations | 100 | 400 | 400 | 400 | 400 |
| time step | 0.015 | 0.0015 | 0.0015 | 0.001 | 0.00067 |

# Methods

We compiled all of the eight benchmark programs on each machine using the GNU *gcc* and *gfortran* compilers and provided the *‘-fopenmp*’ during compilation to make sure we used OpenMP. The only benchmark classes compiled and used were A, B, C, S and W as these were the only ones that our machines’ main memory could handle in terms of data size.

Once compiled, we wrote a script that would automatically run all benchmark programs and classes from one thread/one core to eight threads/four cores. This script also ran five iterations of each benchmark to allow us to calculate the average execution time. We controlled the number of threads by setting the environment variable ‘*OMP\_NUM\_THREADS*’ provided by OpenMP with the assumption that the operating system would handle thread scheduling, and that threads would not share the same core unless there were no more available cores.

For each run, all unnecessary (i.e. third party) programs were manually closed and network connectivity was removed so as to make sure that only the operating system and benchmark were the only things running. We did not, however, run the benchmark on a clean installation of the operating system.

Finally, we collected all raw output for each benchmark run and wrote a python script to export all the execution times into a table format (*.csv file*) for easy data analysis.

# Results

Asd

# Conclusions

Amdhal's law dictates that the speedup between an optimized task and an un-optimized task is affected by the speedup of the we gain from improving the execution time of some part of a task. Nonetheless, we saw speedup from the benchmark tests to be sub-linear to the increase in the number of cores used; at times, we saw a decrease in performance with an increase in the number of cores used. From each benchmark, we found different sources of bottlenecks, which range from data dependencies, to cache misses, to the computational overhead created in timing the benchmarks themselves. For applications which are not data-intensive, a speedup was observed when a benchmark ran with more cores. When, however, there is a large increase in the amount of memory that is required to run a benchmark of a certain class, we either see little increase or even a decrease in performance as the benchmark script is memory-limited.

Unsurprisingly, yet interesting to observe, different applications perform scale differently with a different number of cores. While increasing the number of cores will increase your computational capacity, it may strain the limited memory resources located on the processors. (This an important consideration for shared caches on any level.) Thus, it is also important to increase the size of the cache. This theme, however, is completely in line with a corollary to Amdhal's law, which suggests that there are diminishing returns to optimizing one aspect of the task very well. Since there is an upper bound in the speedup on can attain by optimizing only one part of a task, it is important to optimize all parts of the tasks to see the greatest overall speedup. In this case, we may be able to introduce speedup by having more cores, but we still need a larger cache to see speedup across all benchmarks.

# References

1. http://www.gotw.ca/publications/concurrency-ddj.htm
2. http://ark.intel.com/products/78934/Intel-Core-i7-4720HQ-Processor-6M-Cache-up-to-3\_60-GHz
3. http://ark.intel.com/products/84993/Intel-Core-i7-5557U-Processor-4M-Cache-up-to-3\_40-GHz
4. http://openmp.org/
5. https://www.nas.nasa.gov/publications/npb.html
6. https://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf
7. https://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf
8. https://www.nas.nasa.gov/publications/npb\_problem\_sizes.html
9. http://h21007.www2.hp.com/portal/download/files/unprot/fortran/docs/vfhtml/
10. pg/pguaracc.htm
11. https://github.com/itomaldonado/multiprocessor-benchmarking-exploration

# Appendix

asd