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

Table of Contents

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 1 - 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

In this section we will show the results and discuss each of the benchmarks in the full suite for the first computer system. We will first present the five kernel applications and then show the three pseudo CFD applications. For each, we will show two graphs, the one on the right will present the speedup versus the number of threads/cores per each benchmark class while the one of the right will focus on benchmark class C and show each parallelized sub-section (i.e. subroutine) of the benchmark program. As stated in the Hardware section, we will only focus on the Linux computer system for results discussion but will include the charts of the OS X computer system in the Appendix section.

IS

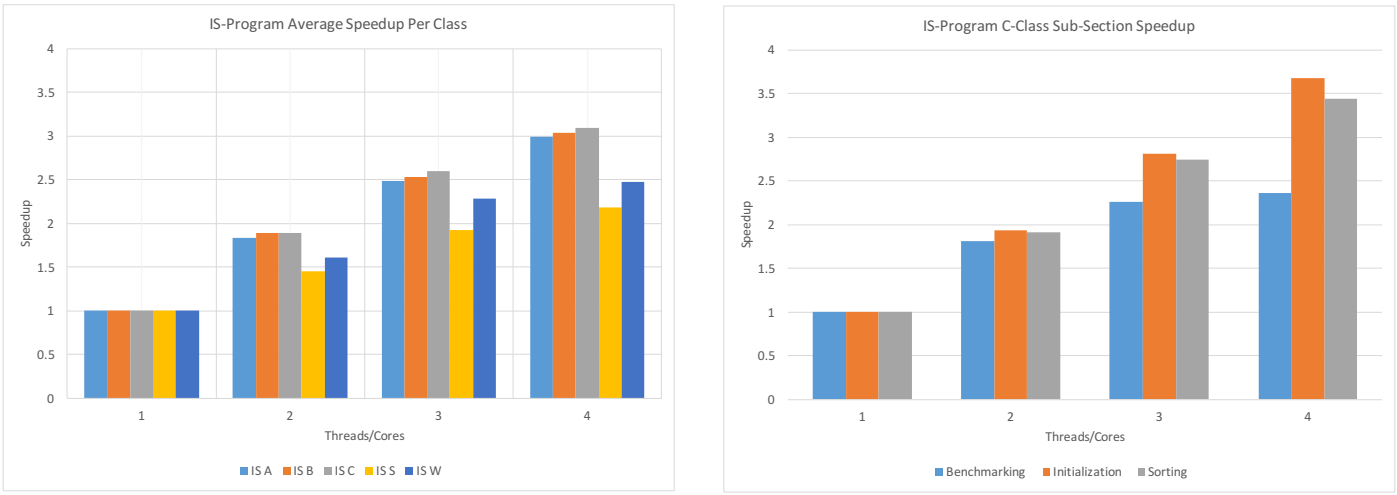


Figure 1 - IS Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

As seen in the left graph of Figure 1 above, class S and W being the smaller classes in terms of data size do benefit from additional cores but the speedup increase is not as much because of the overhead from spawning new threads. On the right graph, we focus on class C were we see that the '*Benchmarking*' sub-section of the kernel (used for calculating benchmarking statistics of the program) does not benefit as much as the other sub-sections from increasing the number of cores. This brings the overall performance benefit down but still quite good with an increase in speedup of **+0.6969**/core.

EP

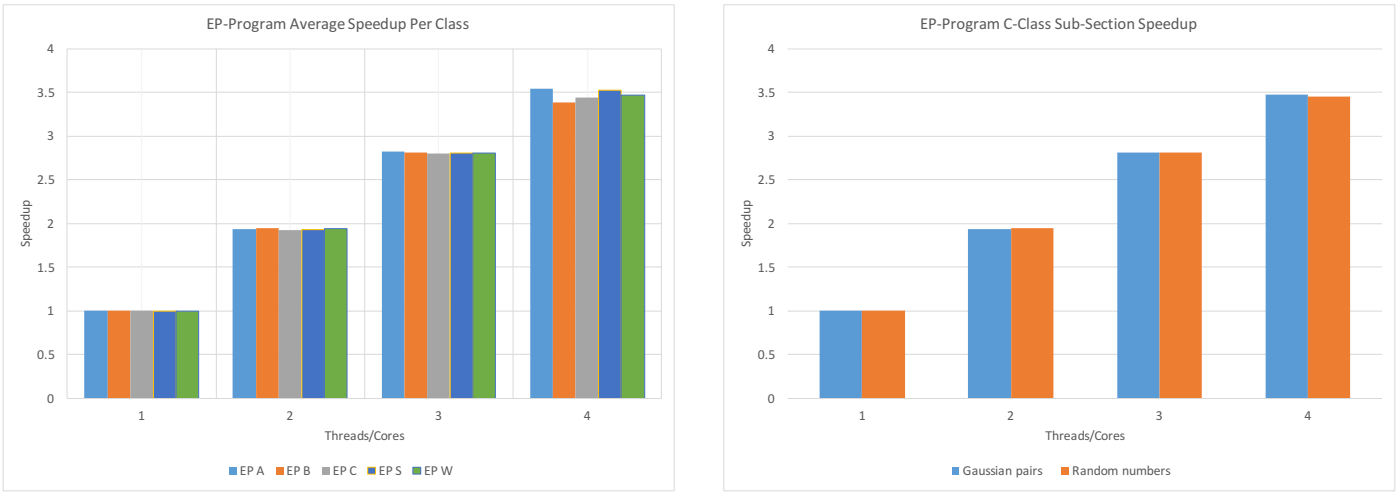


Figure 2 - EP Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

Here as the name suggests, this is a highly (and embarrassingly) parallel job. As seen in Figure 2 above, all classes benefit greatly from increasing the numbers of cores, even the sub-sections within the kernel both benefit from it. The average speedup for this benchmark is **+0.8187**/core. The speedup not being the theoretical/linear can easily be explain by the fact that the computer system is sharing resources with the OS (which we cannot control) as well as overhead added by spawning the extra threads/processes and results being aggregated. This result nonetheless shows the practical speedup ceiling of our computer system.

CG

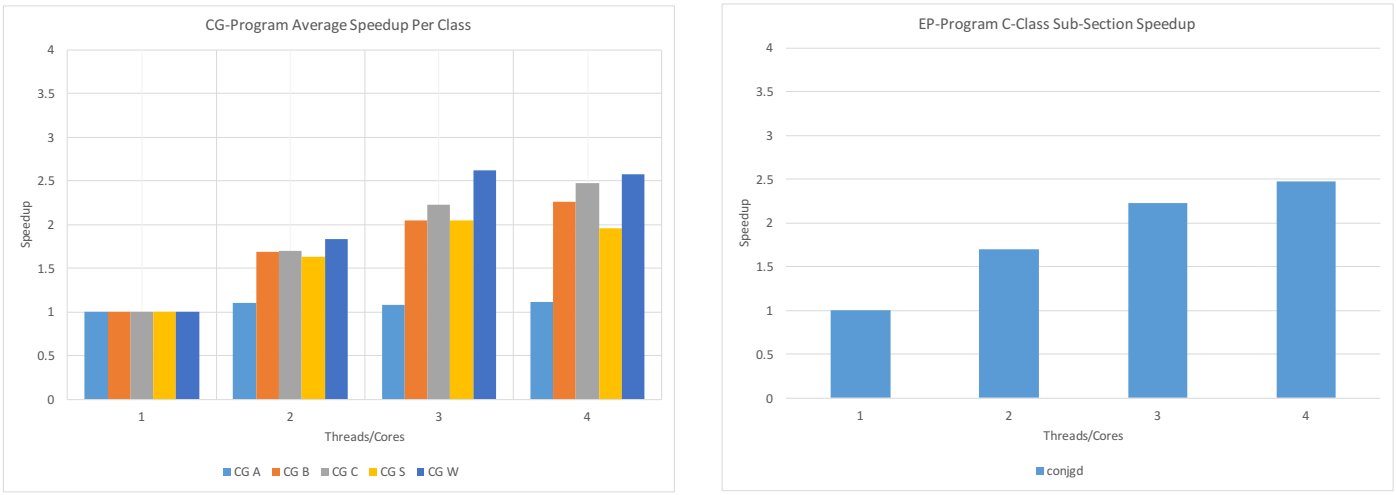


Figure 3 - CG Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

As seen in the left graph of Figure 3 above, CG also benefits greatly from increasing the number of cores, although smaller classes (S and A) do not benefit as much with class A having almost no benefit. Focusing on class C we see that the '*conjgd*' sub-section the the one mainly responsible for the increase in performance of the overall kernel. As explained by the the kernel description, this benchmark does a lot of random memory accesses which can bring the performance of our benchmark down as our computer system has a relatively small cache (L1-64KB) per core and the data size is quite large. We now start to see the bottleneck in performance because of cache sizes. The speedup for CG is of about **+0.4938**/core.

MG

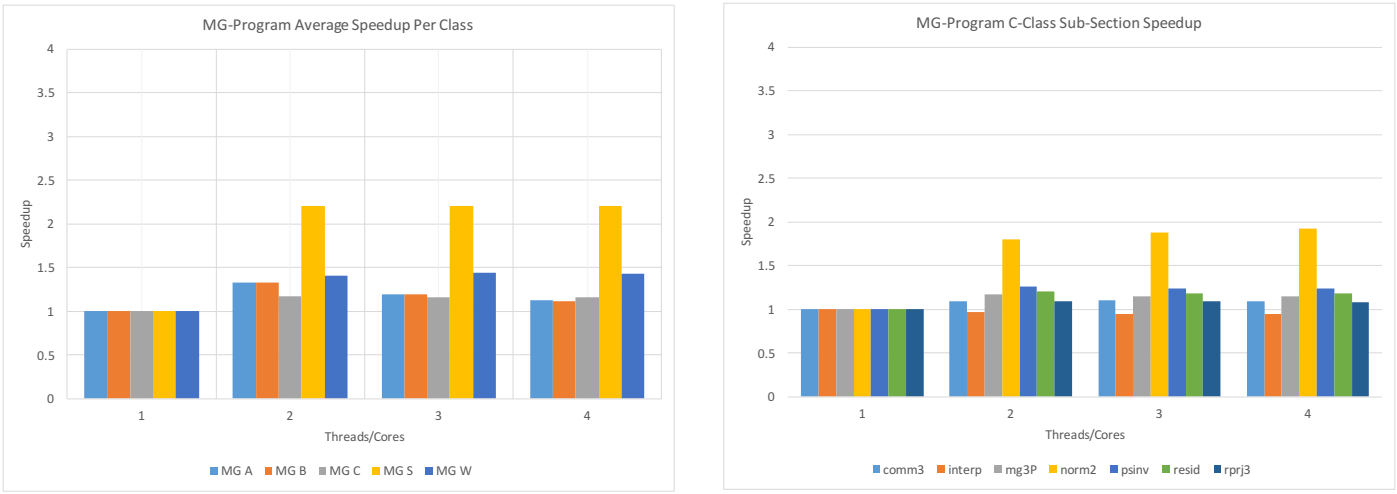


Figure 4 - MG Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

In the left graph of Figure 4 above, we can see that MG does not benefit as much as the benchmarks we have discussed until this point. Only the smallest class (S) benefits but it reaches a maximum of around 2.0 speedup, with two cores, which means that it won't benefit anymore, this is because of the amount of data being worked on. This benchmark traverses large distances when reading data, where we can see the cache bottleneck making an impact here again. Class S doesn't get impacted by the cache bottleneck because the amount of data being worked on is so small that it fits in the cache quite easily leading to fewer cache misses. The average speedup for this benchmark is **+0.0472**/core.

FT

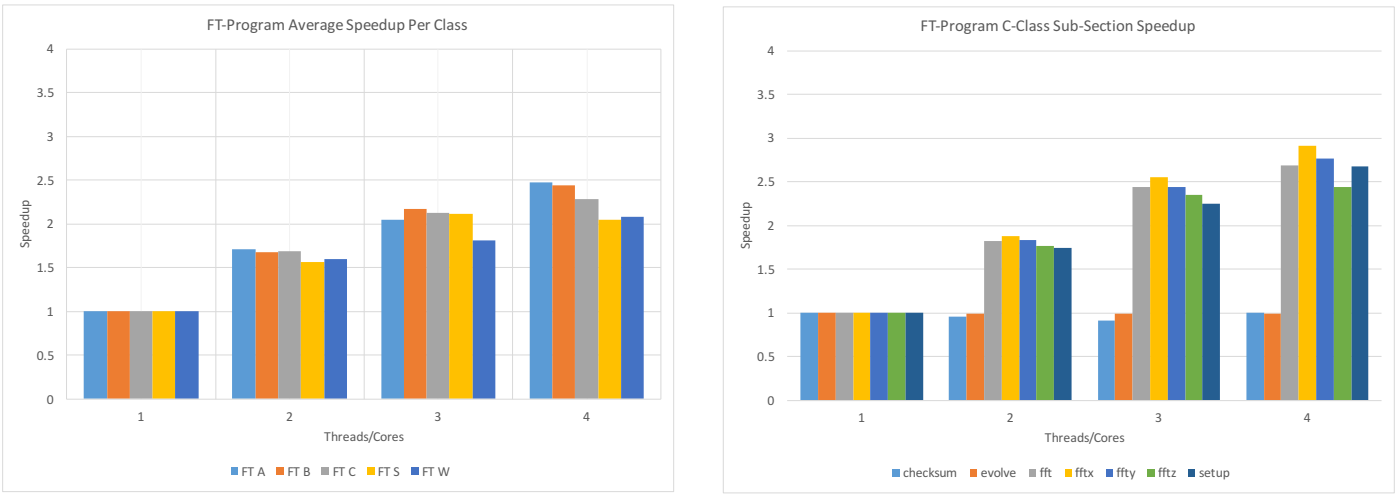


Figure 5 - FT Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

For the FT benchmark, similar to the IS benchmark, it takes advantage of multiple cores quite will with a nice linear growth. For FT specifically we can see in the right graph of Figure 5 above that the sub-sections '*checksum*' and '*evolve*' do not increase speedup as we increase the number of cores as these tasks are strictly sequential in nature; these two bring the overall benefit down. The average speedup for FT was about **+0.4304**/core.

BT, SP & LU

We decided to merge all three CFD pseudo applications into one section as they all deal with the same type of problem. As before, we will show the result graphs first and the the discussion of them.

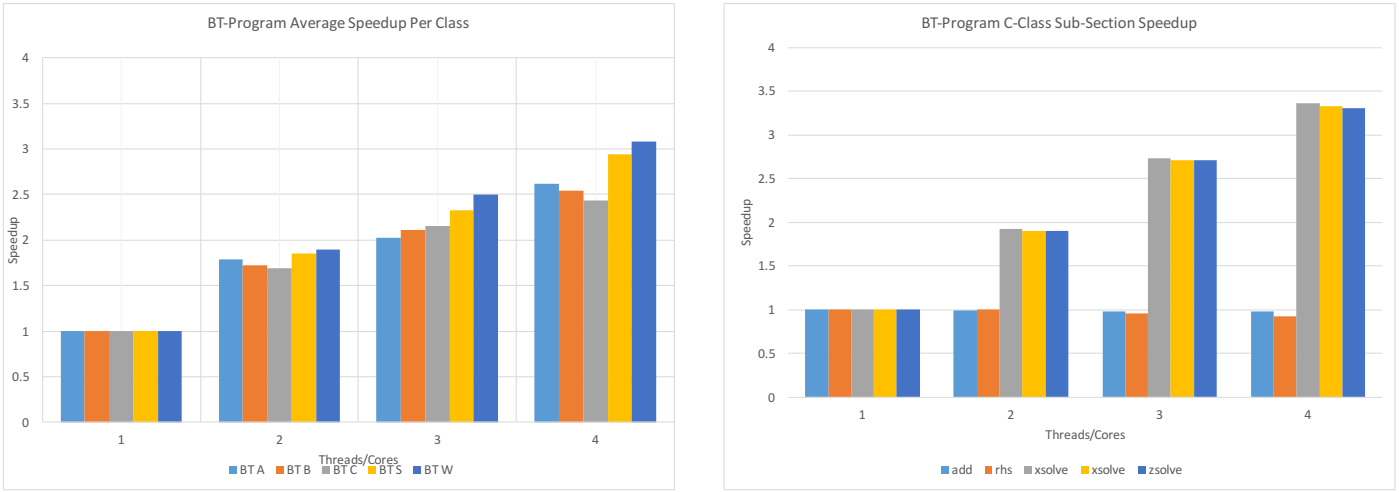
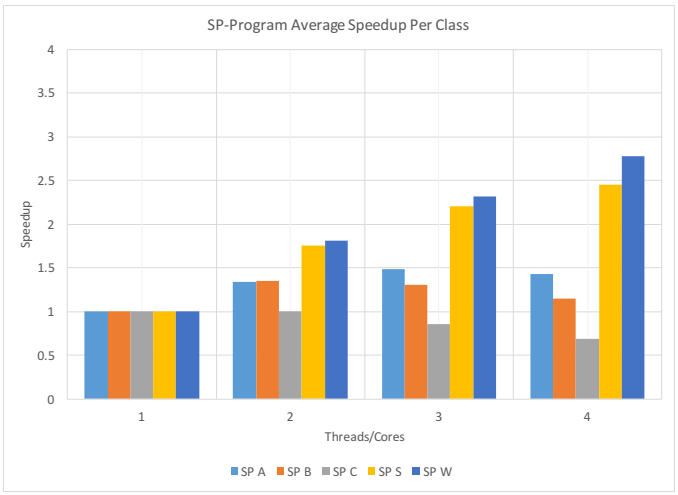


Figure 6 - BT Speedup versus threads/cores of all classes (left) and sub-section for class C (right).



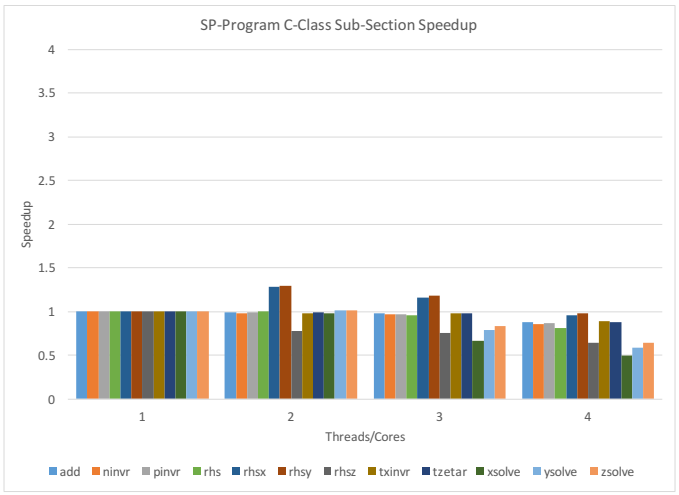
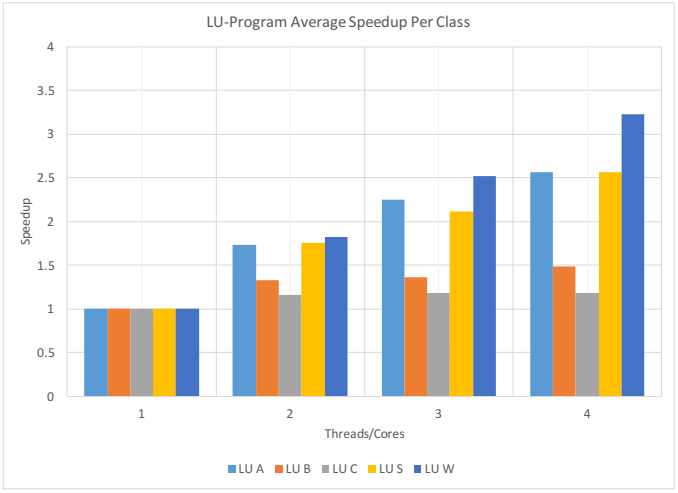


Figure 7 - SP Speedup versus threads/cores of all classes (left) and sub-section for class C (right).



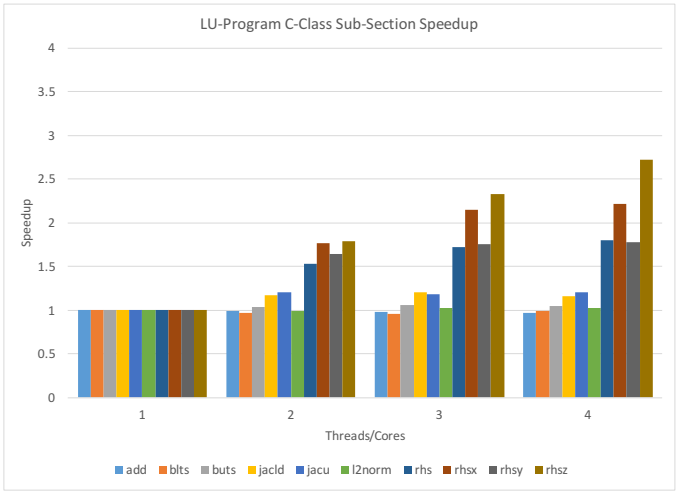


Figure 8 - LU Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

In Figure 6, Figure 7 and Figure 8 above we see that there are several limiting factors to the speedup, mainly due to the '*rhs**' and '**solve*' sub-sections (where * = x,y,z). It should be noted that the nomenclature of the subroutines are derived from the NAS Parallel Benchmark Technical Report and thus are named in uniform manner:

1. There are high requirements for solving the Navier-Stokes equations. Since we are trying to solve the Navier-Stokes equations in 3D, a matrix cube with 100 entries on each side equates to having a matrix cube of 10^6 elements. Since double precision floating point numbers are 8B each, a matrix cube of length 100 occupies 8MB of space. Given that the shared L3 cache is only of 6MB, it is impossible to store the entire matrix. As a the number of cores increase, the shared L3 cache will be our bottleneck; inevitably, the performance speedup will suffer. Note that the above explanation does not take into account the synchronization instructions that occur when two

processors are reading/writing to the same memory location, nor does the dataset size take into account the auxillary matrices used in the intermediate calculations

2. The equations which defined the individual terms in a matrix were large, often with more than 10 operands. While Fortran allows a variable to be defined as a long sequence of algebraic operations, the assembler will see only a sequence of instructions with at most two operands. This decomposition of a long Fortran variable assignment to a sequence of instructions will inevitably lead to a true data dependence. Given that the operands can be any algebraic operation, they will inevitably lead to unpredictable data dependencies. Moreover, since there are many algebraic operations, there may also be structural hazards as there are only 4 ALU's per core.
3. The code for the benchmarks took advantage of the column-major order array storage scheme native to Fortran, but did not always take advantage of loop-blocking. Moreover, there were instances when the memory accesss to the array elements were of non-unit stride. This increased the number of conflict misses that occurred in the cache.

Finally, to keep the same format as earlier benchmark results, the average speedup for BT is **+0.4750**/core, for SP is **-0.1071**/core (which is our worst performing benchmark) and for LU is **+0.0584**/core.

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/vf-html/pg/pguaracc.htm
10. https://github.com/itomaldonado/multiprocessor-benchmarking-exploration

Appendix

In this section we include extra graphs and charts for other benchmark runs.

Computer System #2: Speedup Results

Bellow are the speedup results for the OS X computer system. We decided to include the speedup graphs in the same format as our results section to keep consistency. Moreover, the main reason we decided to include them in this section is to show the impact of increasing the number of threads well beyond the amount of threads/cores supported by the system. As seen in the following graphs, after two threads/cores the speedup reaches a maximum and plateaus, after four threads (shred among two cores) speedup may begin to decrease because of conflicts trying to schedule more than two threads per core, resource sharing, etc.

IS

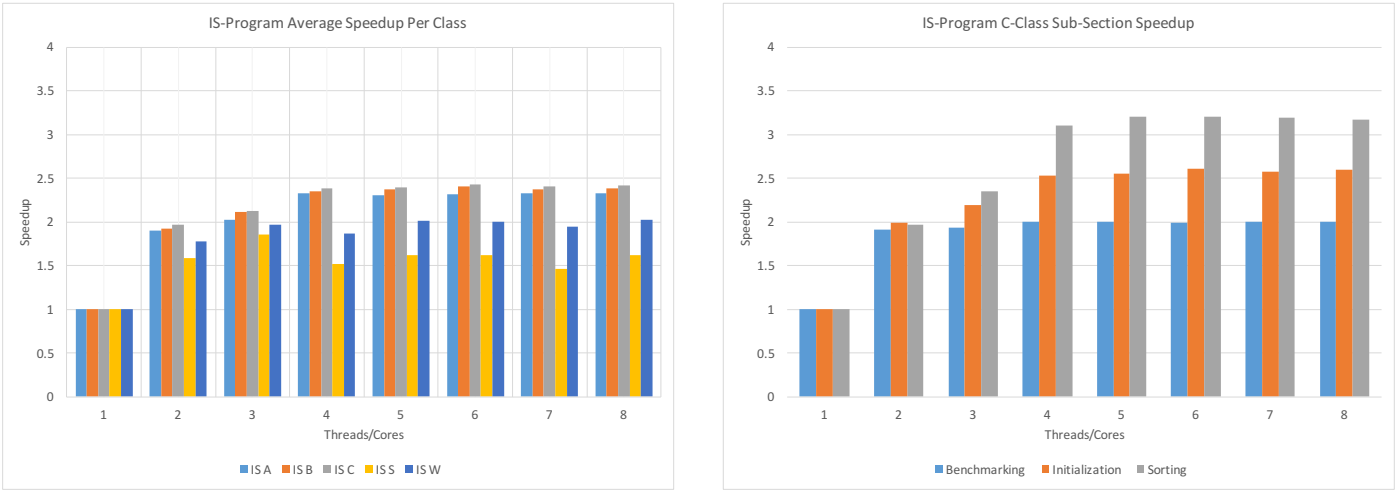


Figure 9 - OSX - IS Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

EP

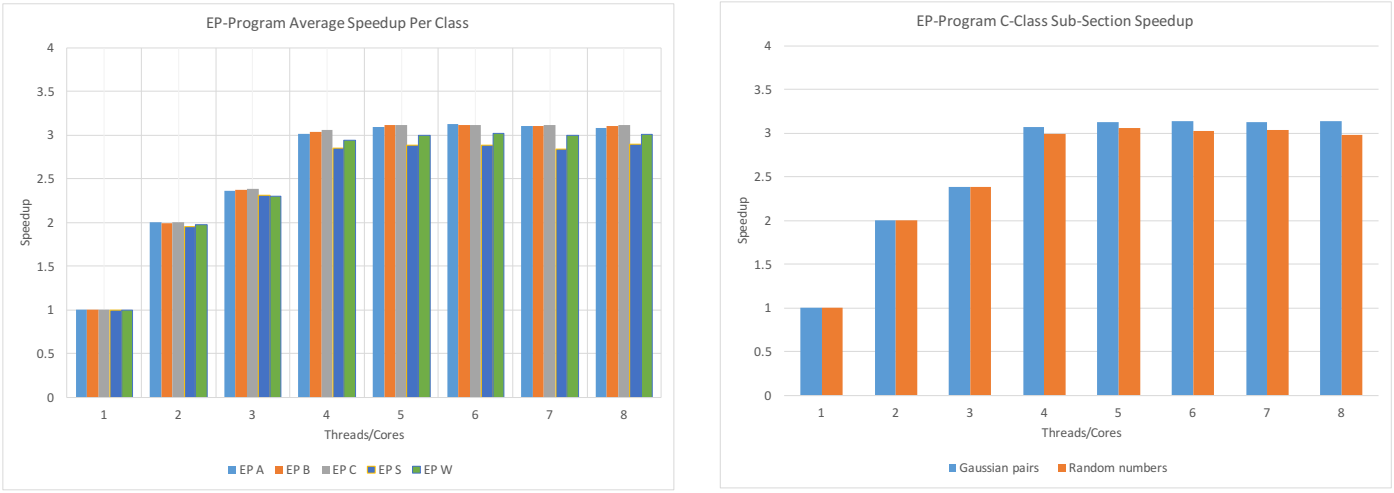


Figure 10 - OSX - EP Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

CG

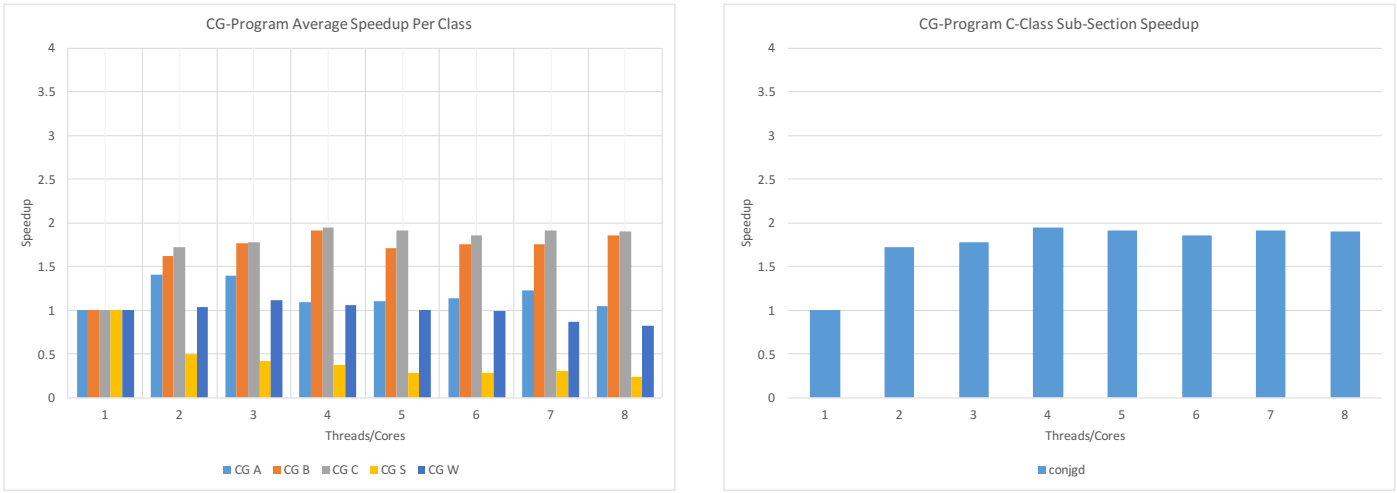


Figure 11 - OSX - CG Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

MG

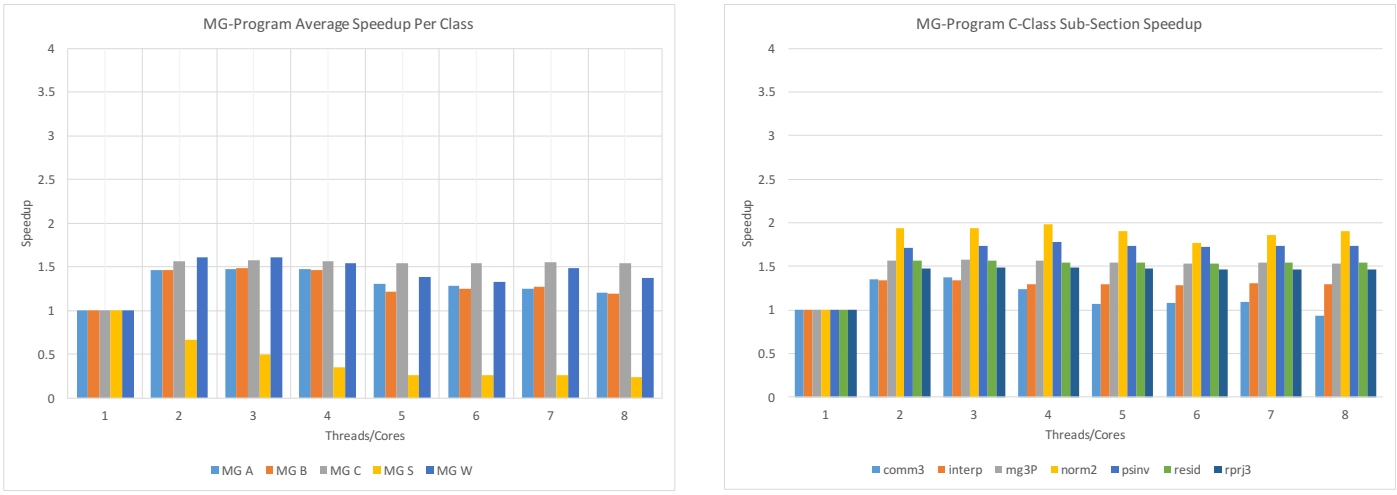


Figure 12 - OSX - MG Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

FT

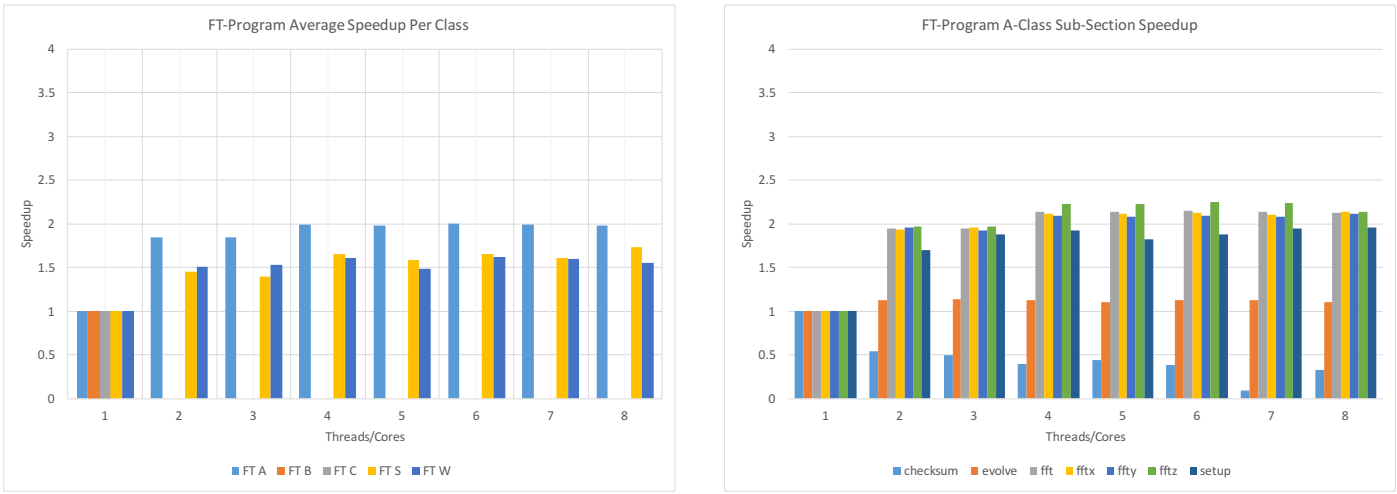
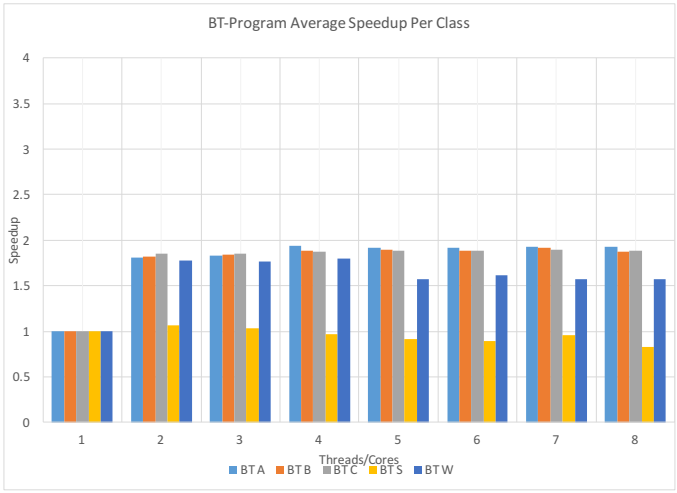


Figure 13 - OSX - FT Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

BT



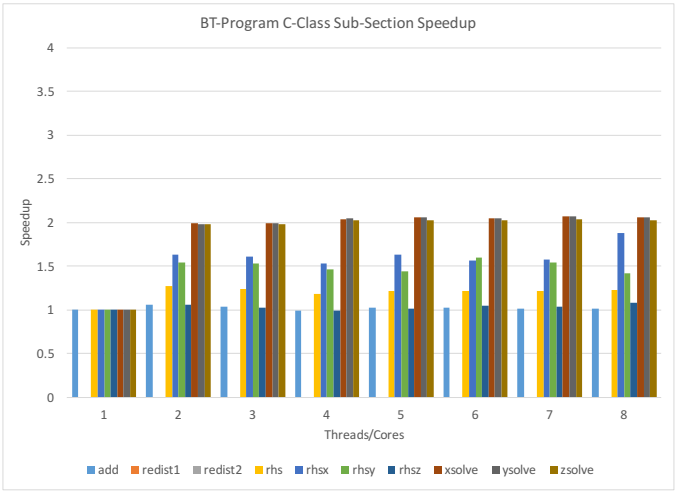
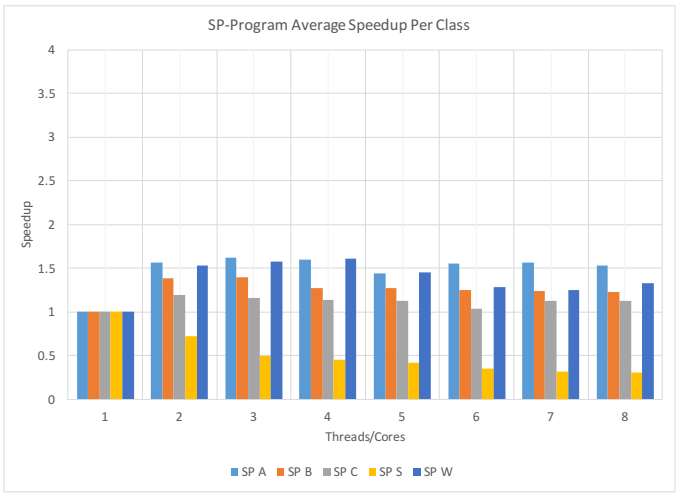


Figure 14 - OSX - BT Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

SP



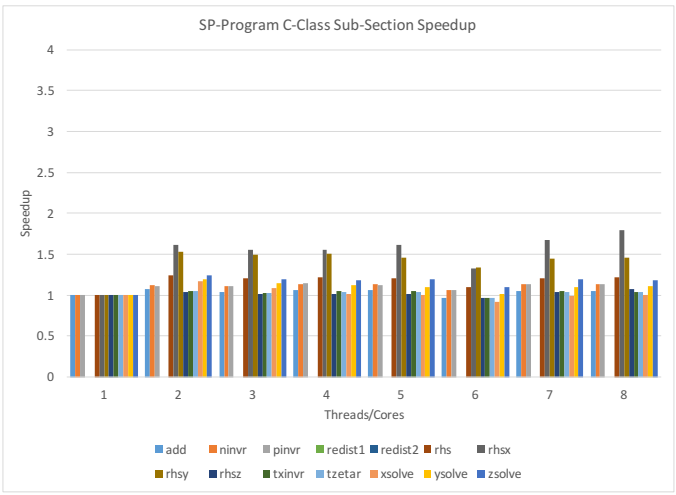
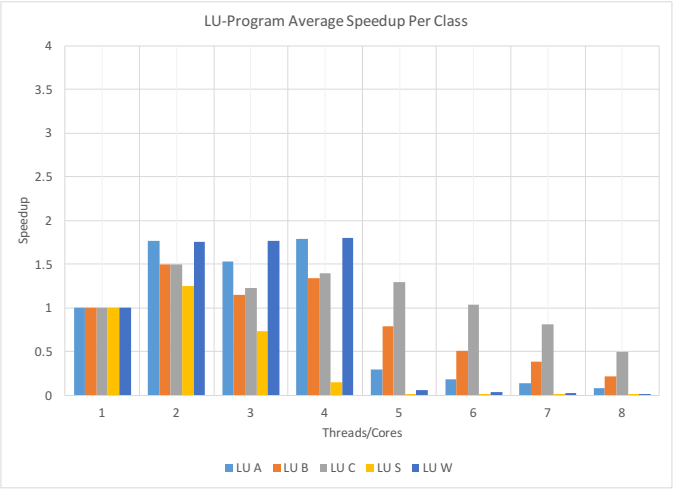


Figure 15 - OSX - SP Speedup versus threads/cores of all classes (left) and sub-section for class C (right).

LU



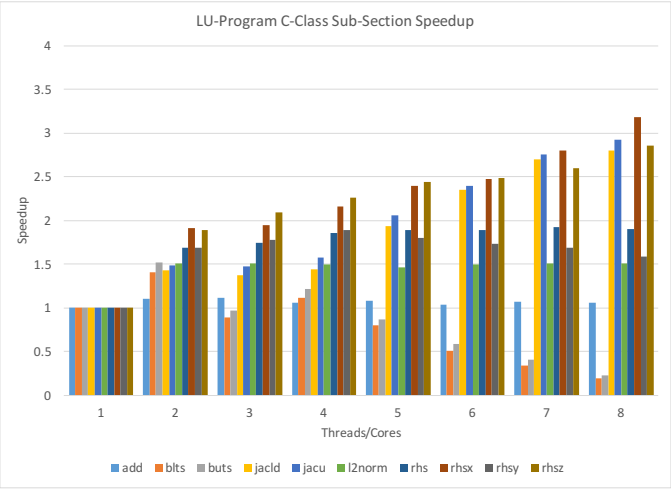


Figure 16 - OSX - LU Speedup versus threads/cores of all classes (left) and sub-section for class C (right).