# Evaluating Chip Multiprocessor Performance

Group Members:

*Manuel Maldonado*

*Ming Tai Ha*

# Introduction

- Evaluate the performance of Chip Multi-Processor (CMP) architecture
- Done by running benchmarks and studying their performance
- Provide explanation for any bottleneck

# Hardware

- Computer #1:
  - OS: Linux Ubuntu 14.04
  - 4th Gen i7 4720HQ processor @2.6GHz, Haswell[1] microarchitecture
  - L1 cache 64KB per core, L2 cache 256 KB per core, L3 cache 6MB shared
  - 4 cores, 8 threads

- Computer #2:
  - OS: OS X 10.11 El Capitan
  - 5th Gen i7-5557U processor @3.1GHz, Broadwell[2] microarchitecture
  - L1 cache 64KB per core, L2 cache 256KB per core, L3 cache 4 MB shared
  - 2 Cores, 4 thread

# Multi-Processing Platform

- OpenMP:
  - Specification for a set of compiler directives, library routines, and environment variables used to specify high-level parallelism in programs[3]
  - A standard platform for parallel programming on shared memory systems
  - Provides code extensions/directives to make programs run in parallel
  - Available for Fortran and C/C++ programs

# Benchmark Suite – NPB 3.3.1

- NASA Advanced Supercomputing (NAS) Parallel Benchmarks:
  - Small set of programs designed to help evaluate the performance of parallel supercomputers[4]
  - Derived from computational fluid dynamics (CFD) applications
  - Five Kernels and three pseudo-applications[5][6]
  - All benchmarks are Fortran programs and use OpenMP
    - *Note:* Fortran uses '*Column-Major*' storage order[7]

# Five Kernels

- *Embarrassingly Parallel (EP):*
  - Used to provide an estimate the upper achievable limits for FP performance.
- *Multigrid (MG):*
  - Highly-structured long/short distance data communication
- *Conjugate Gradient (CG):*
  - Typical unstructured gird computations (e.g. matrix vector multiplication)
- *3-D Fast Furrier Transforms (FT):*
  - Rigorous test of long-distance communication performance
- *Integer Sort (IS):*
  - Integer computation speed and communication performance

# Three Programs – Simulated CFD

- *Block Tri-Diagonal Solver (BT):*
  - Solves Block-Tridiagonal systems of 5×5 blocks in sequential manner along each dimension
- *Scalar Penta-Diagonal Solver (SP):*
  - Similar structure to BT but solving Scalar Penta-Diagonal bands of linear equations sequentially along each dimension.
- *Lower-Upper Gauss-Seidel Solver (LU):*
  - 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.

# Benchmark Classes

- The NPB benchmark comes in multiple classes (sizes):
  - Class S:
    - small for quick test purposes
  - Class W:
    - workstation size (a 90's workstation; now likely too small)
  - Classes A, B, **C**: (our focus)
    - standard test problems; roughly 4X size increase going from one class to the next
  - Classes D, E, F:
    - large test problems; roughly 16X size increase from each of the previous classes

# Methods[8]

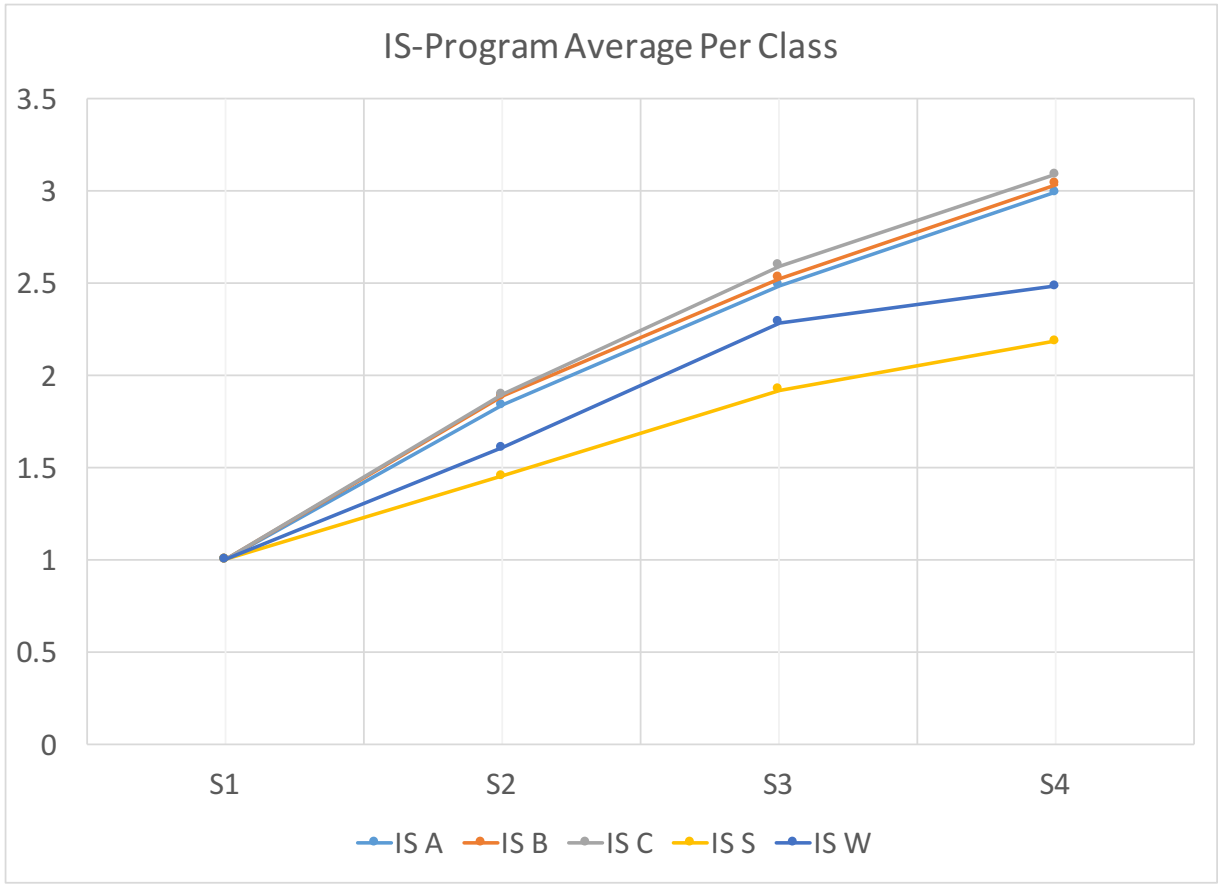
- Compilers used: GNU *gcc* and *gfortran*
- Compilation done using the '*-fopenmp*' flag on both
- Classes A, B, C, S and W were compiled and run
- Ran all classes with a defined number of threads and iterations
  - Sets '*OMP_NUM_THREADS*' to control the number of threads used by OpenMP
  - OS took care of thread/core scheduling.
  - Explored using '*OMP_PROC_BIND*' and 'OMP_PLACES' for CPU affinity
- All benchmark (raw) output was capture and kept
- Times compiled and averaged, used for speedup calculations

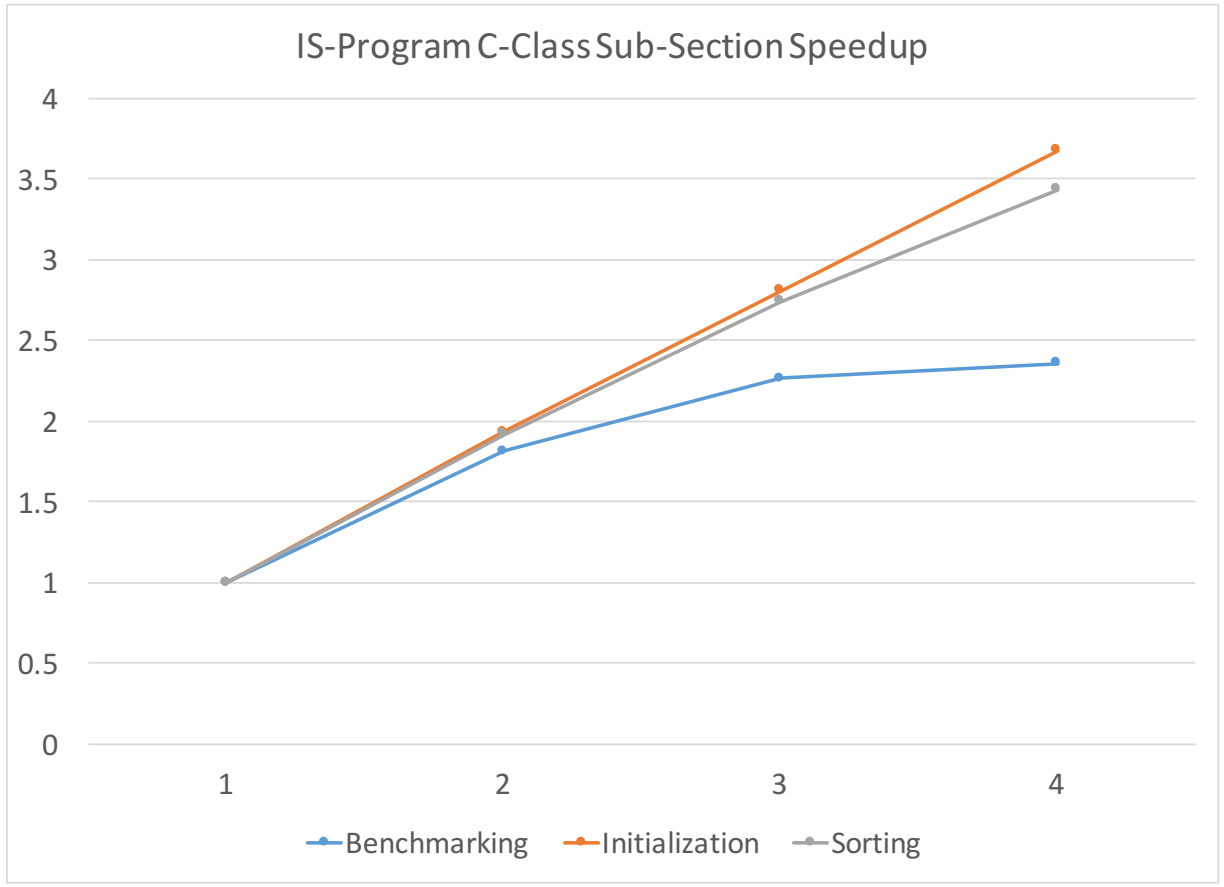
# Computer #1: Speedup Results

Speedup per program class and speedups for all sub-sections for class C programs.

4 cores, 1-4 threads.

# IS

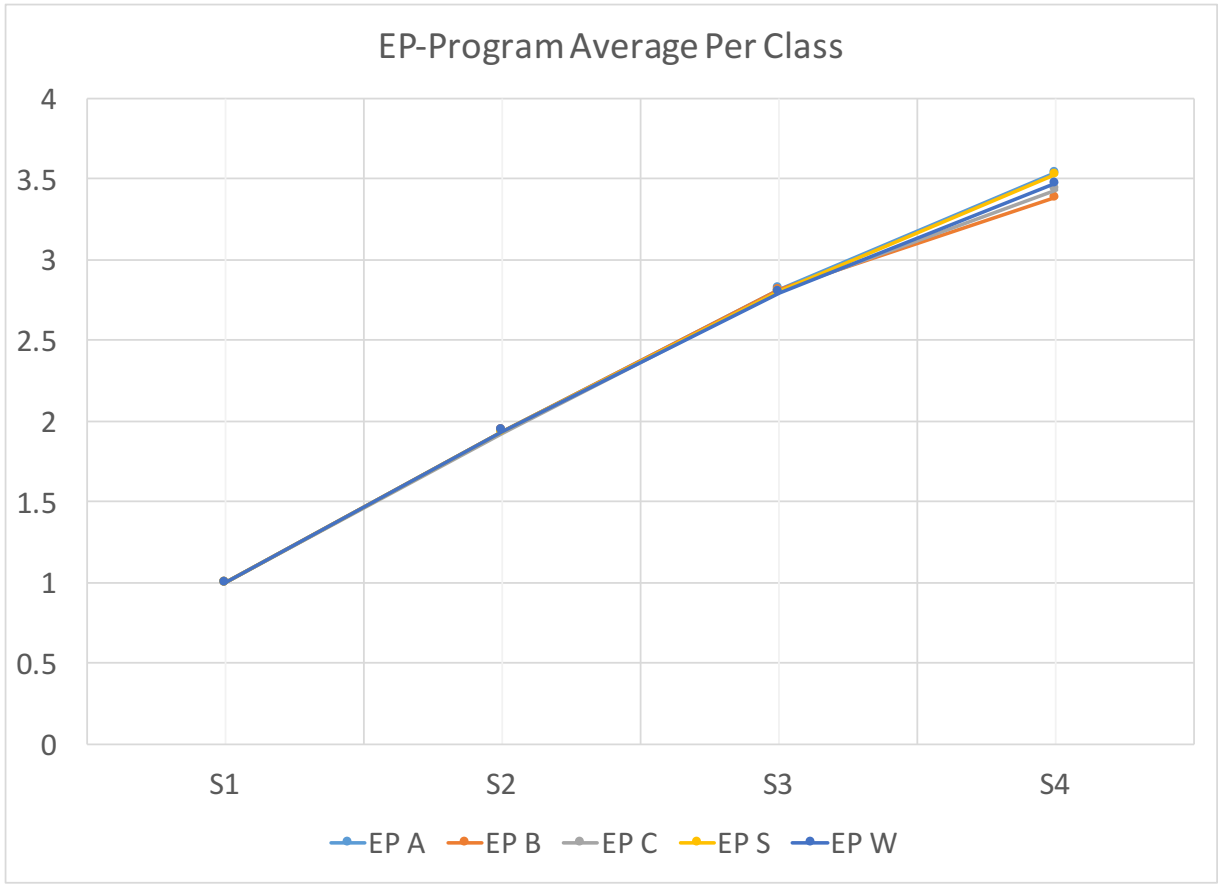

IS-Program Average Per Class
3.5
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
IS A
IS B
IS C
IS S
IS W

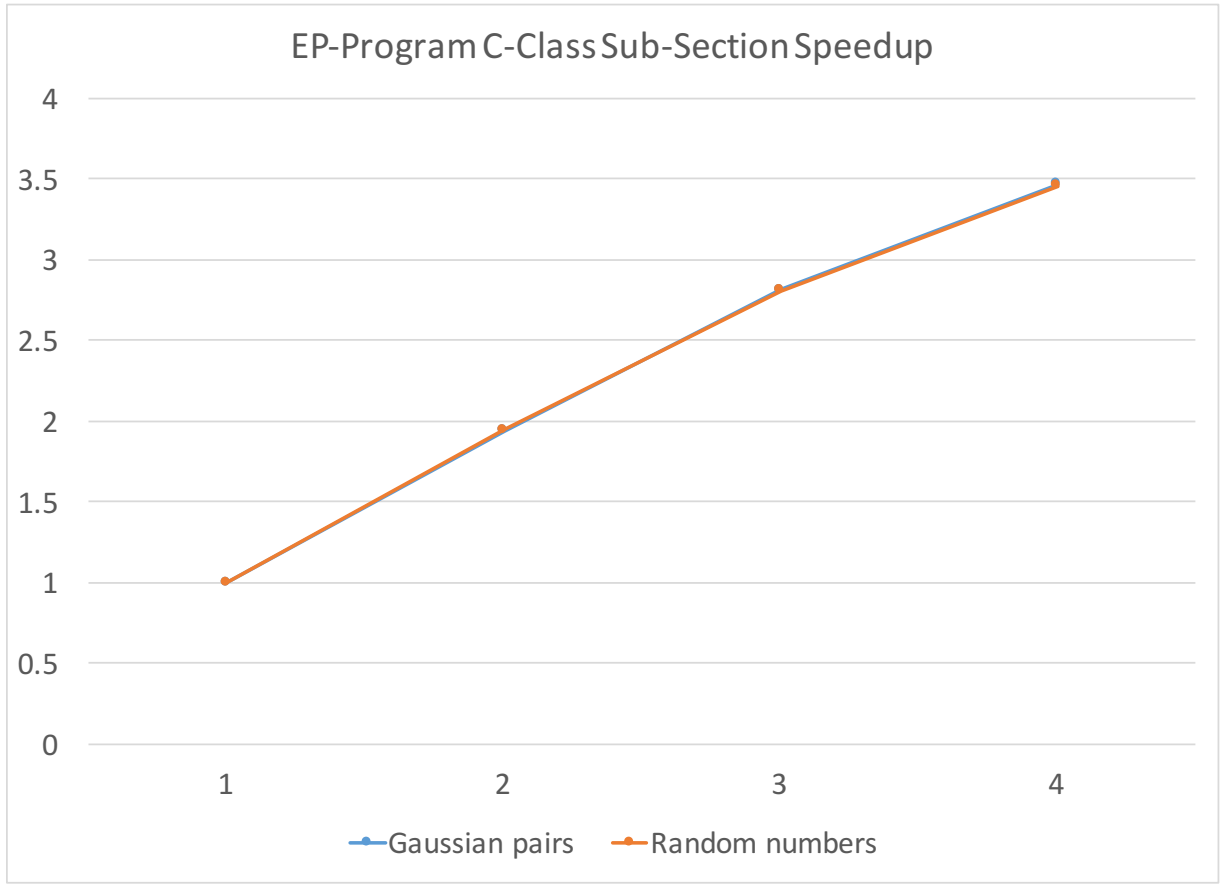

IS-Program C-Class Sub-Section Speedup
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
Benchmarking
Initialization
Sorting

# IS – Discussion

- Benefits greatly of parallelization and multi-processes, speedup of about +0.6969/core
- Suffers from data dependencies on smaller classes as the same number of threads need to share a smaller amount of work
- The ‘benchmarking’ sub-process is a for-loop going through all the data first, suffers from some data dependencies which bring the overall speedup down slightly

# EP


EP-Program Average Per Class
4
3.5
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
EP A
EP B
EP C
EP S
EP W

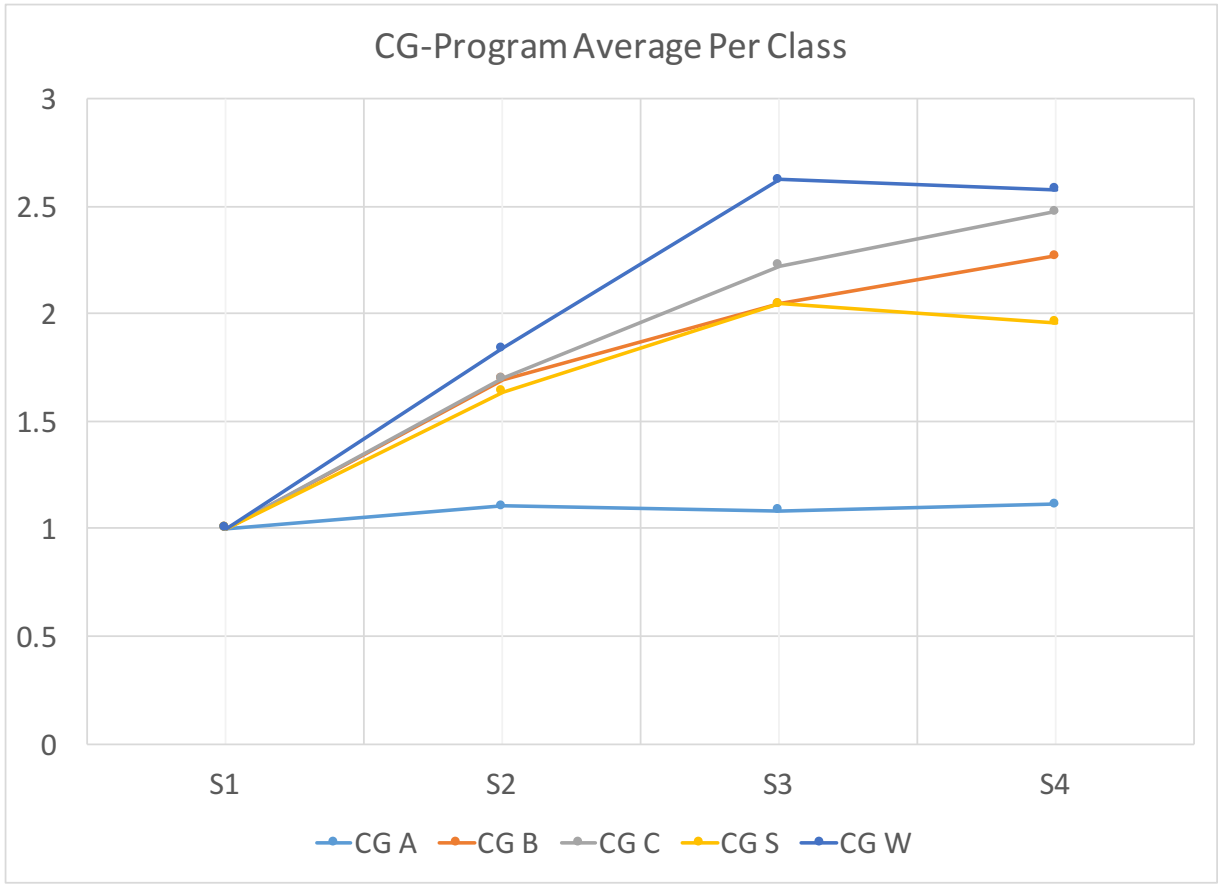

EP-Program C-Class Sub-Section Speedup
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
Gaussian pairs
Random numbers

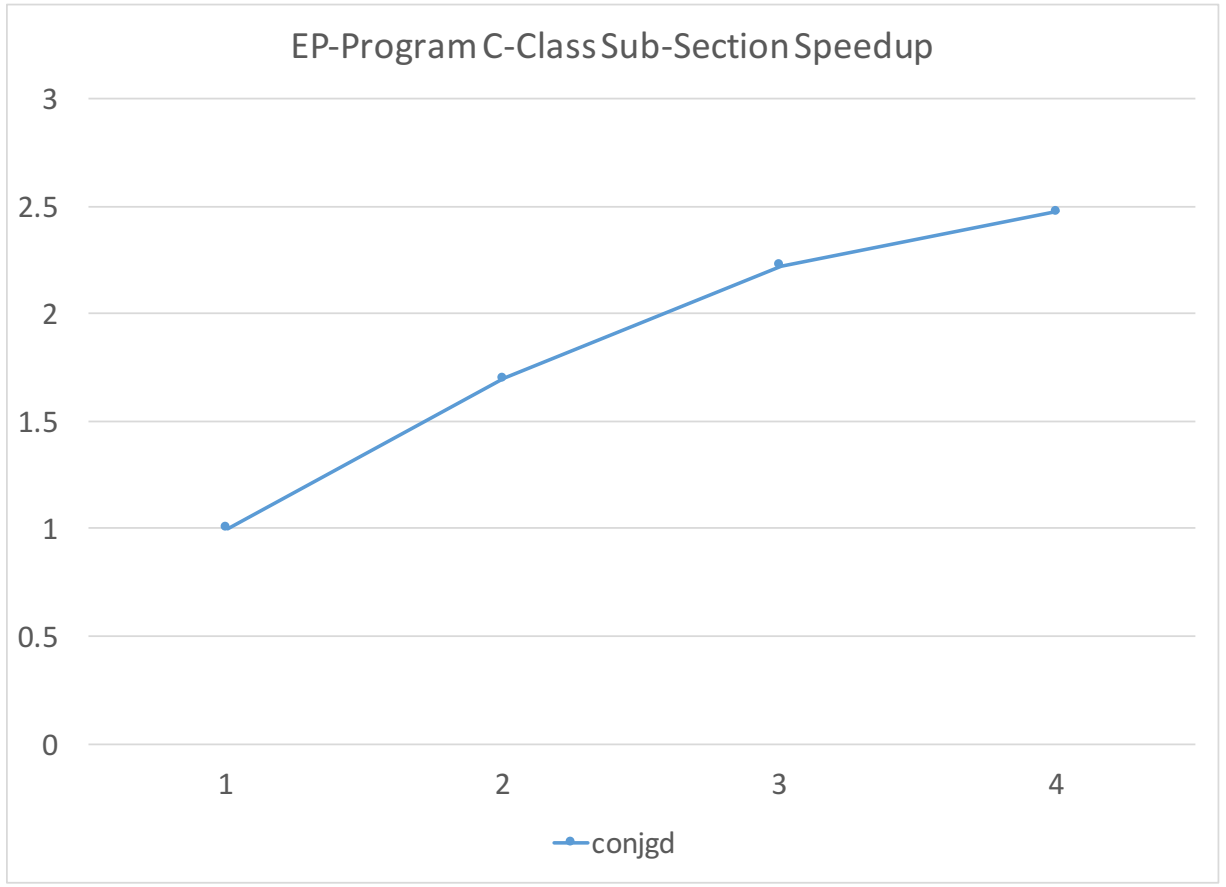
# EP – Discussion

- As the name suggests, this is a highly parallelizable job. Speedup of about +0.8187/core

- Speedup not being 4.0x can be explained by OS sharing resources (which we don’t control), thread creation/destruction overhead results being aggregated.

# CG

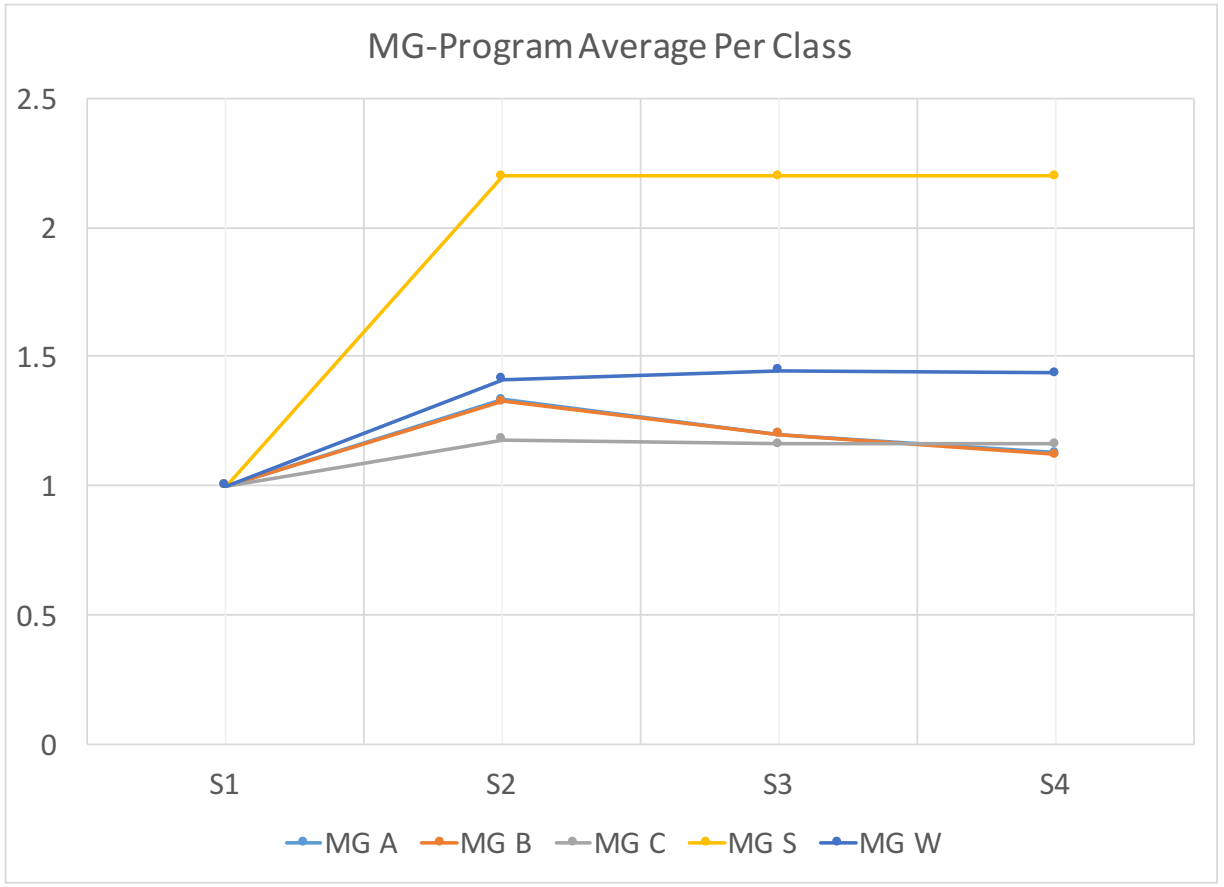

CG-Program Average Per Class
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
CG A
CG B
CG C
CG S
CG W

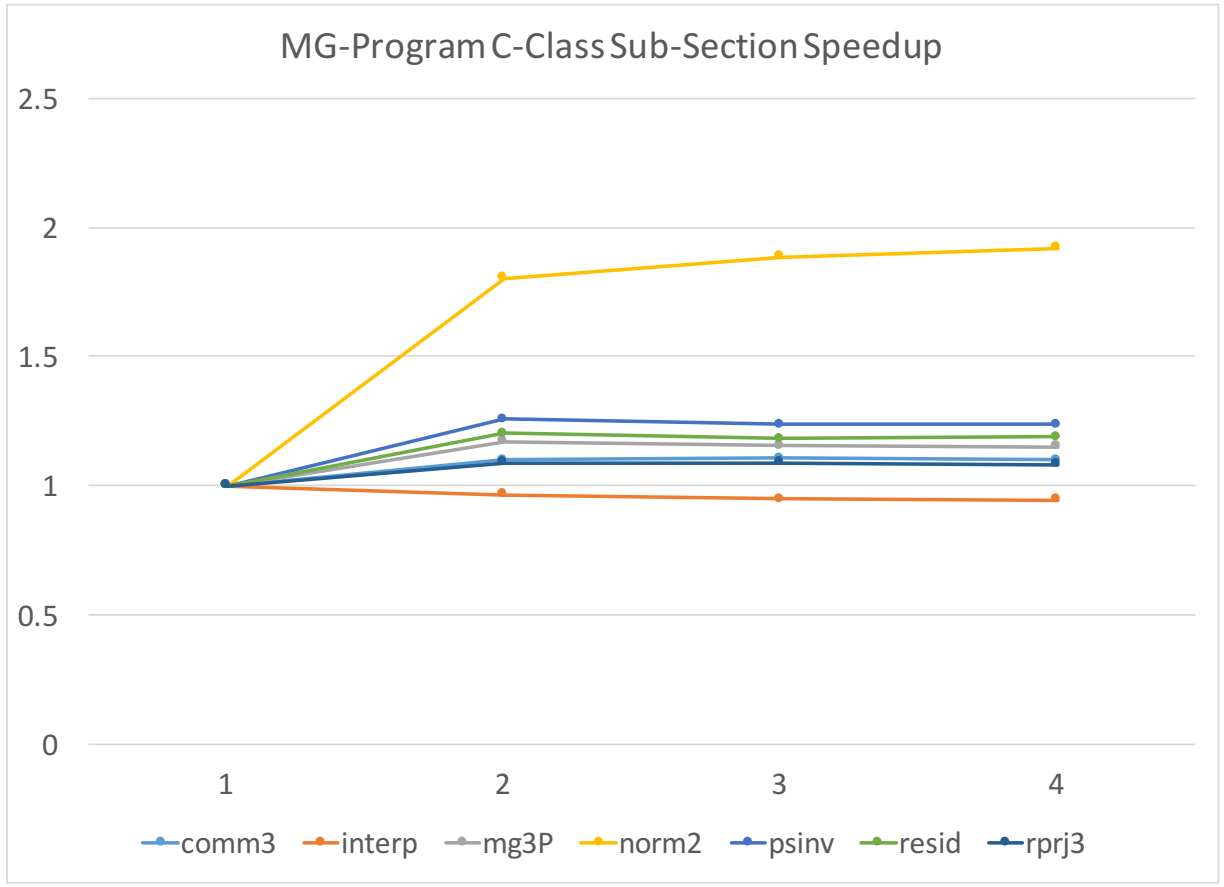

EP-Program C-Class Sub-Section Speedup
3
2.5
2
1.5
1
0.5
0
1
2
3
4
conjgd

# CG – Discussion

- Also benefits rather well from parallelization and multi-processes, speedup of about +0.4938/core.
- Smaller classes don’t benefit as well from parallelization.
- Since memory access is done randomly cache performance will impact it more.

# MG


MG-Program Average Per Class
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
MG A
MG B
MG C
MG S
MG W


MG-Program C-Class Sub-Section Speedup
2.5
2
1.5
1
0.5
0
1
2
3
4
comm3
interp
mg3P
norm2
psinv
resid
rprj3

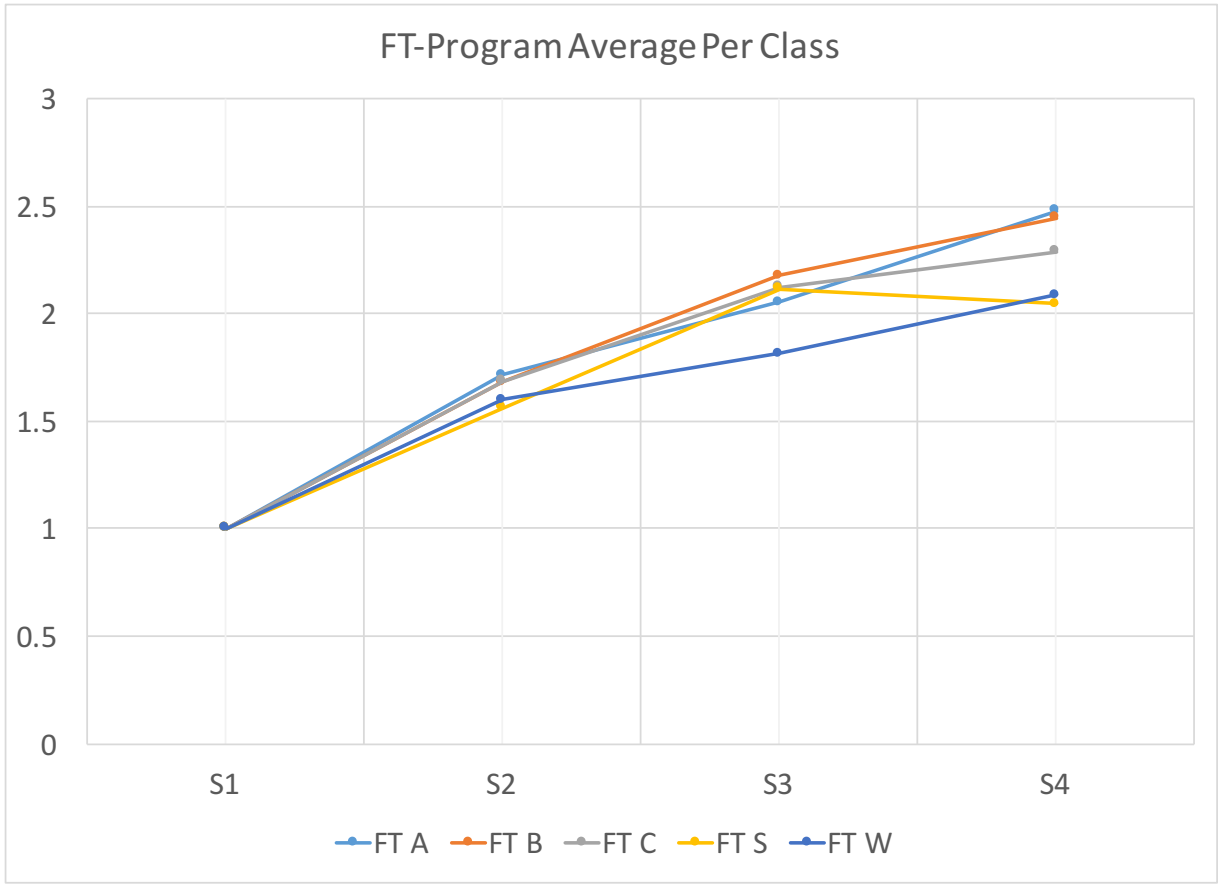
# MG – Discussion

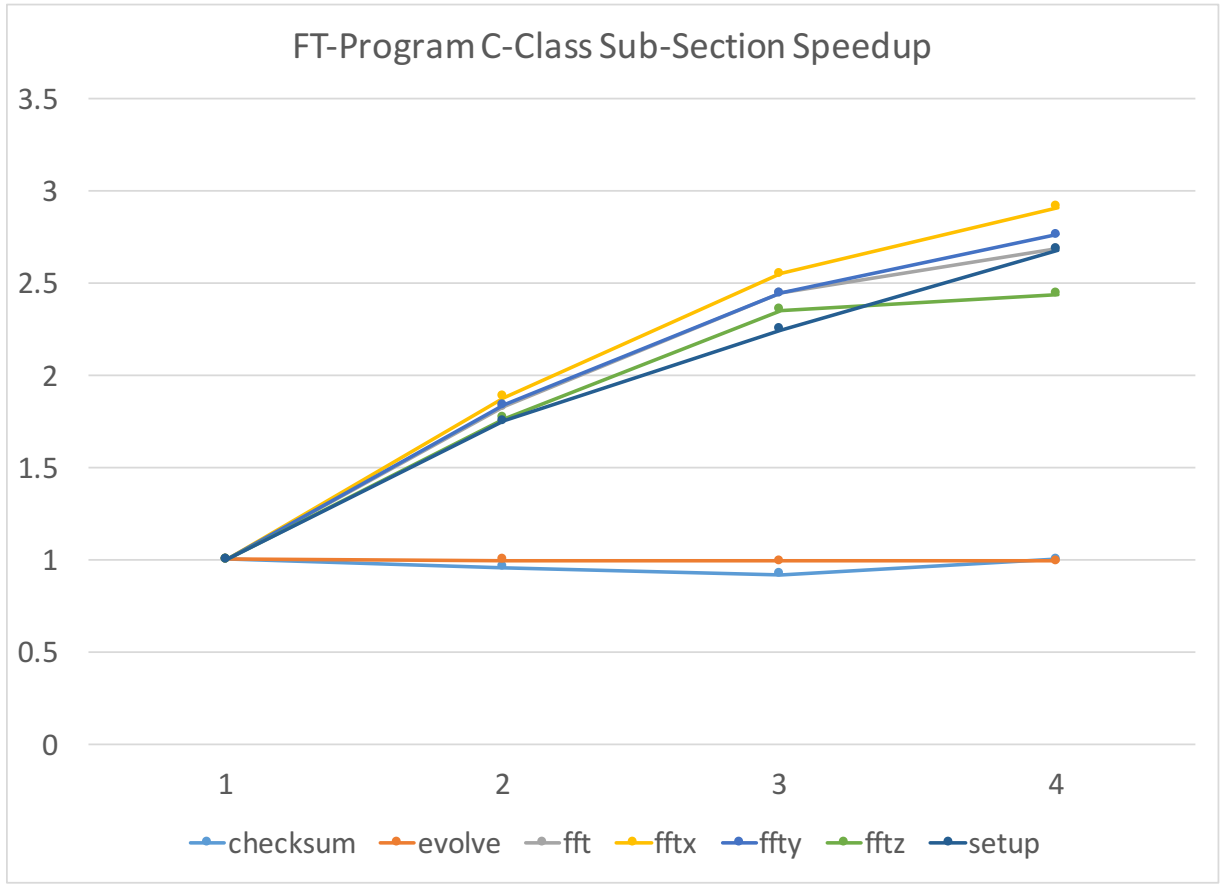
- This program traverses large distances when reading data, impact of the cache can bee seen here. Speedup of about +0.0472/core.

- The smallest class benefitted the most since it is possible to fit most of the data in caches.

- Although we get a speedup, the bottleneck of this program is memory architecture performance.

# FT


FT-Program Average Per Class
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
FT A
FT B
FT C
FT S
FT W


FT-Program C-Class Sub-Section Speedup
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
checksum
evolve
fft
fftx
ffty
fftz
setup

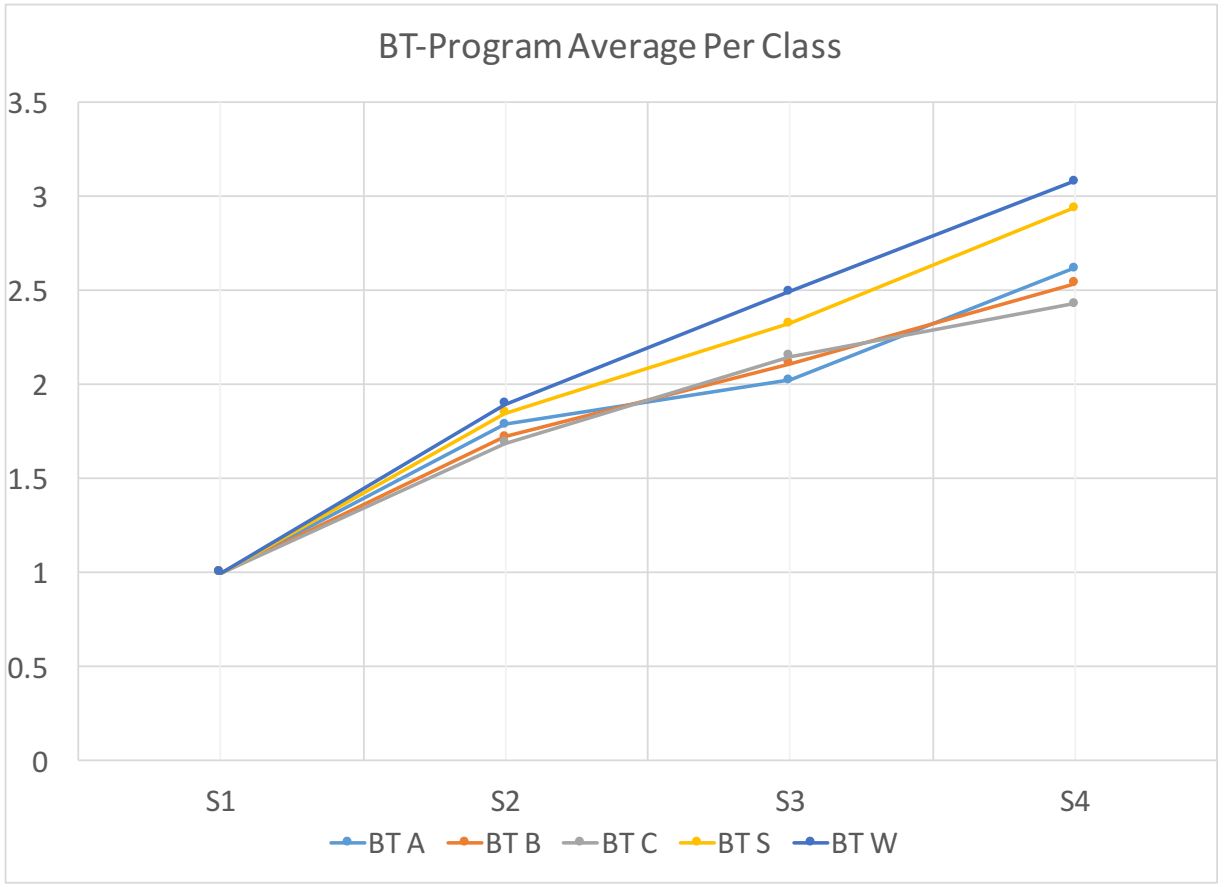
# FT – Discussion

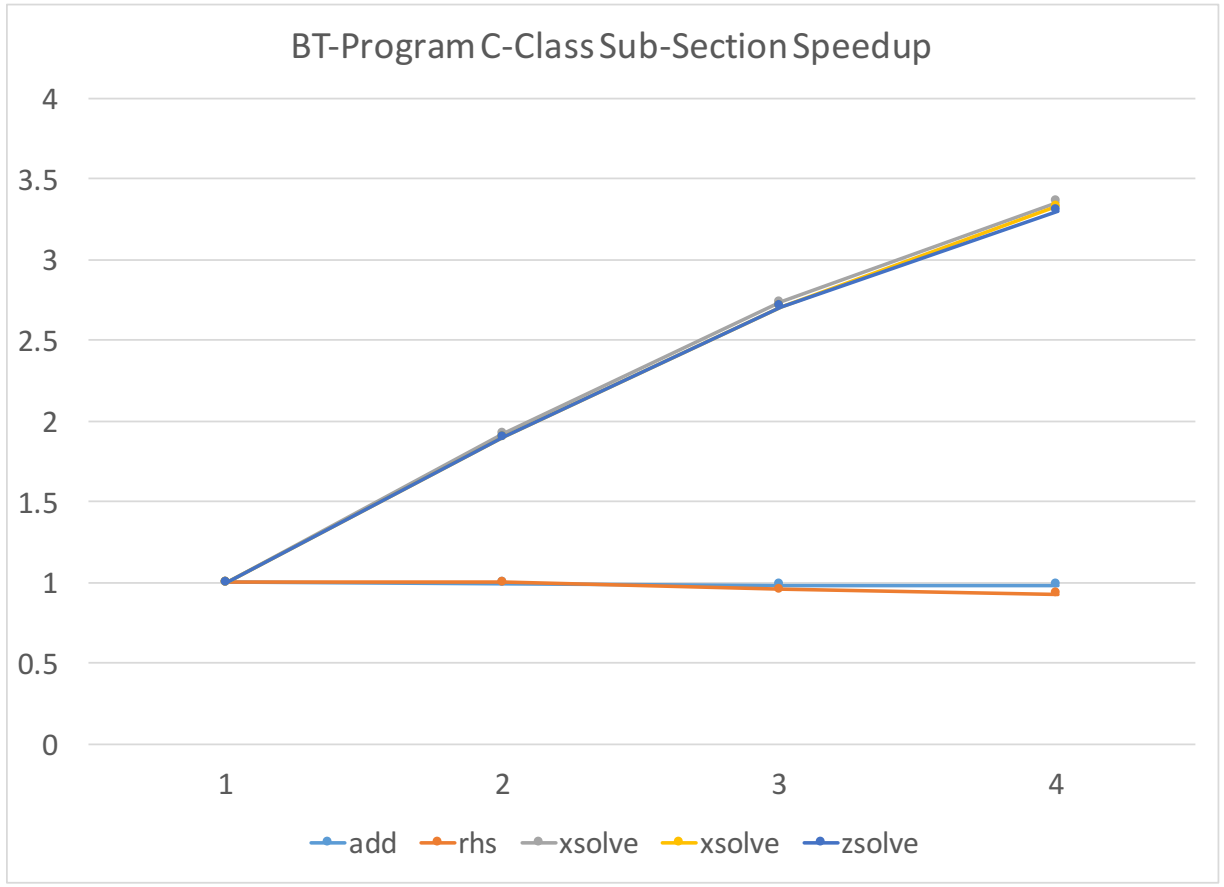
- Similar to the IS program, this also benefits greatly from parallelization and multi-processes. Even the smaller classes. Speedup of about +0.4304/core.

- Suffers from data as threads do share some of the calculation.

- Bottleneck of this program is the verification sub-process done via `*checksum*` and the `*evolve*` sub-process. Both are linear/sequential tasks.

# BT


BT-Program Average Per Class
3.5
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
BT A
BT B
BT C
BT S
BT W


BT-Program C-Class Sub-Section Speedup
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
add
rhs
xsolve
xsolve
zsolve

# BT – Discussion

- Speedup of about +0.4750/core.

- Loop was nested so that the fastest changing variable is the leftmost to ensure unit stride. This was not often enforced and so were more cache misses. This can be found in most components of the BT benchmark.

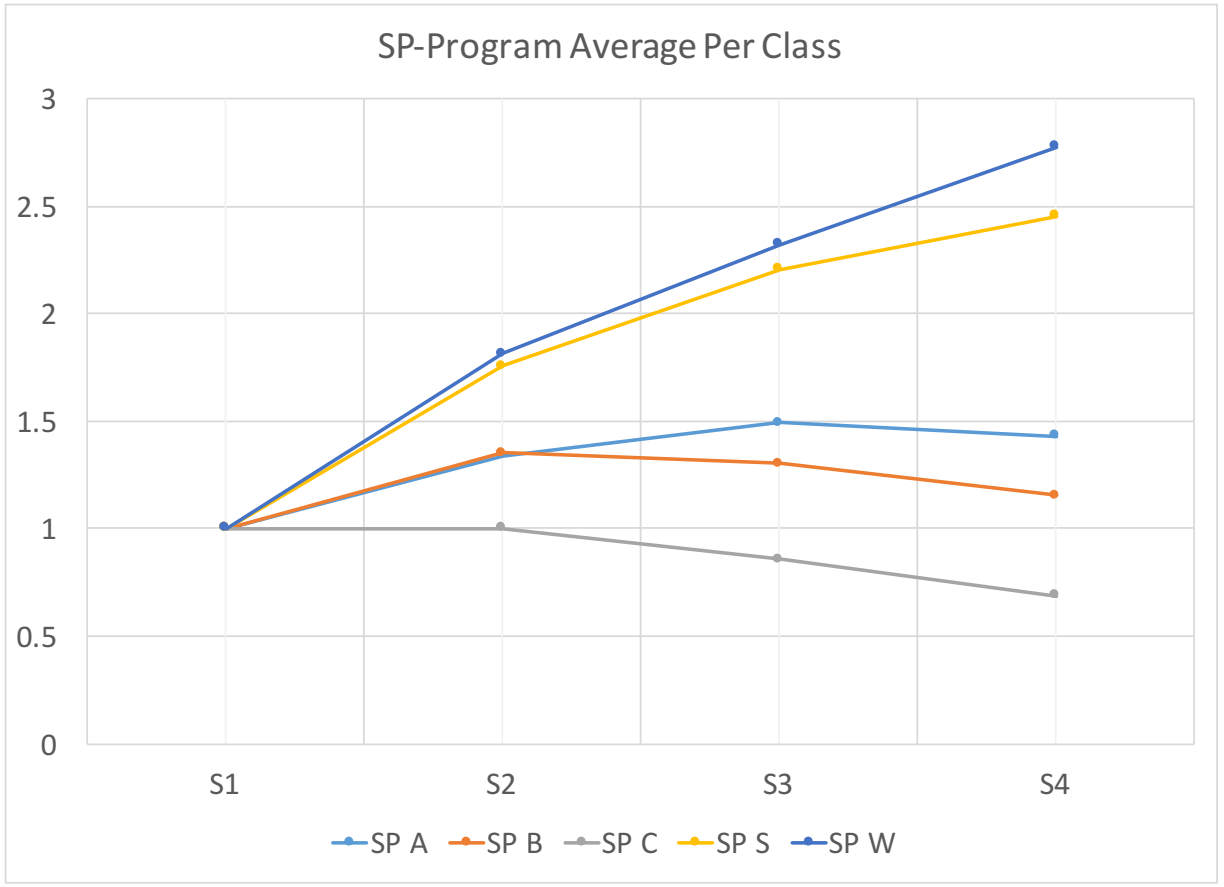
- The equations are very long and thus create data dependence. There is also a low number of arithmetic units. Lots of array elements are used and thus there are many data memory accesses.

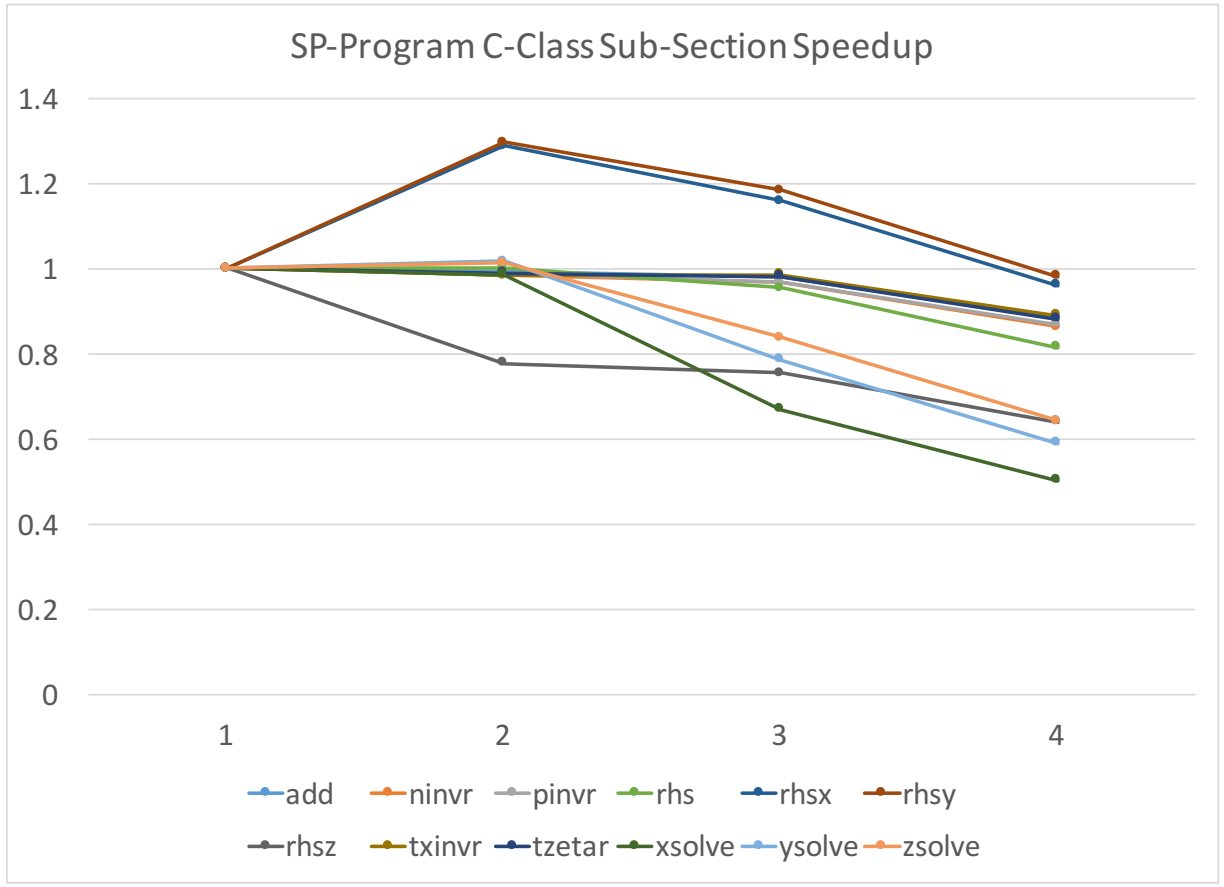
- High memory requirements. Problem size scaled cubicly, given the stencil was performed on 3D matrices. There were also many auxilliary matricies used to calculated other terms used in the CFD simulations. Given the number of floating point operations and terms required, it used up cache memory. We see drop in performance as number of cores increased because of restricted cache size.

**The above points are prominent in xsolve, ysolve, and zsolve. Appears also in rhs*

- True data dependencies in rhs& subroutine since many terms depend on terms calculated in previous iterations of the loop.

# SP


SP-Program Average Per Class
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
SP A
SP B
SP C
SP S
SP W


SP-Program C-Class Sub-Section Speedup
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
add
ninvr
pinvr
rhs
rhsx
rhsy
rhsz
txinvr
tzetar
xsolve
ysolve
zsolve

# SP – Discussion

- Speedup of about -0.1071/core.

- Loop was nested so that the fastest changing variable is the leftmost to ensure unit stride. This was not often enforced and so were more cache misses. This can be found in most components of the SP benchmark.

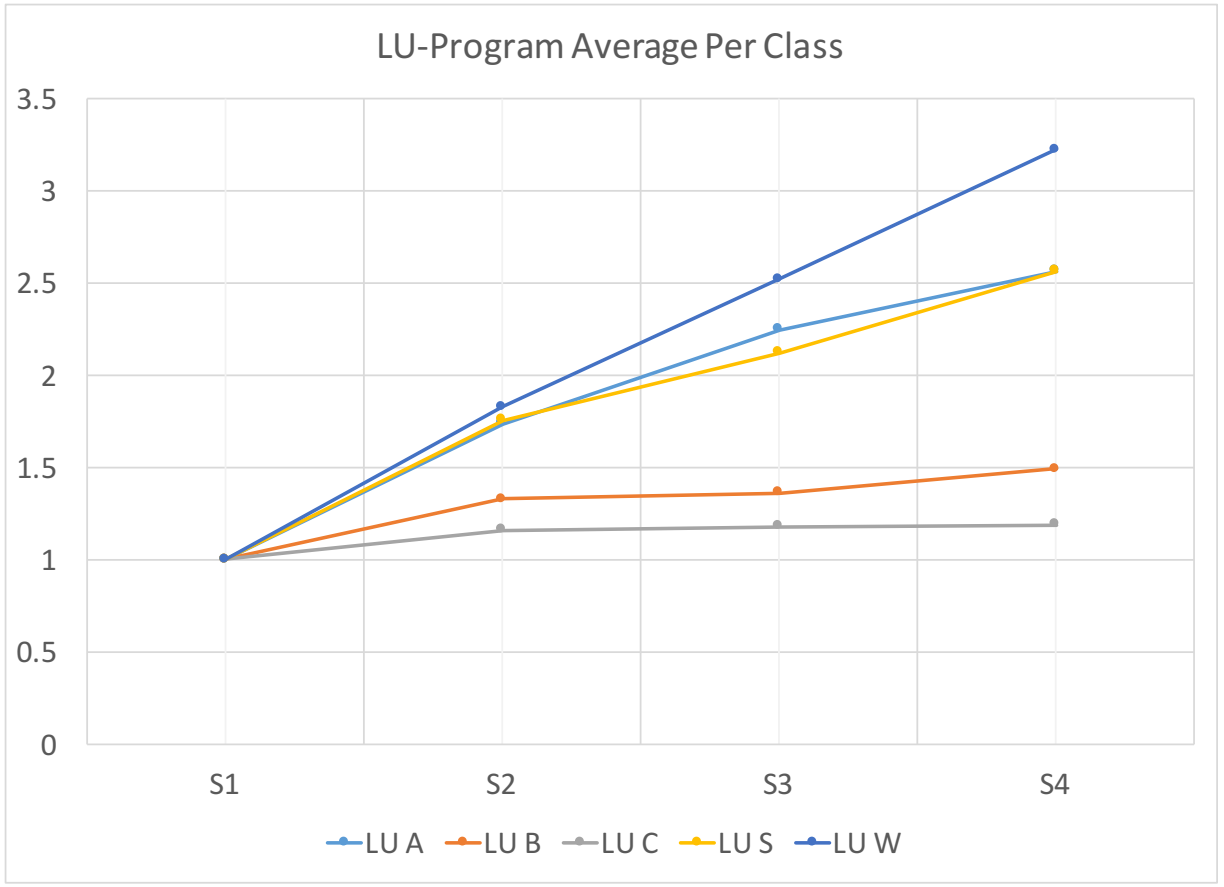
- The equations are very long and thus create data dependence. There is also a low number of arithmetic units. Lots of array elements are used and thus there are many data memory accesses.

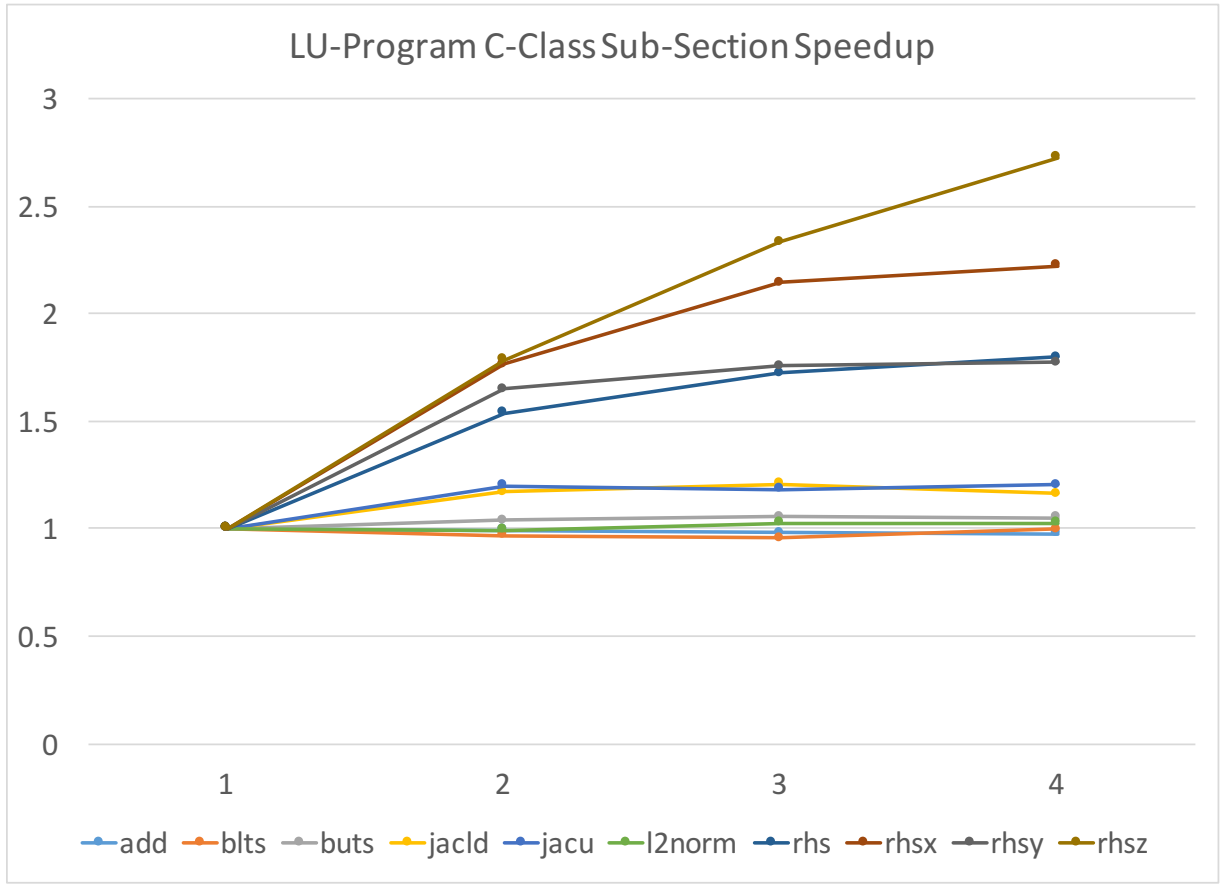
- High memory requirements. Problem size scaled cubically, given the stencil was performed on 3D matrices. There were also many auxiliary matrices used to calculated other terms used in the CFD simulations. Given the number of floating point operations and terms required, it used up cache memory. We see drop in performance as number of cores increased because of restricted cache size.

**The above two points are prominent in rhs*, xsolve, ysolve, zsolve, . Appears also in rhs**

- True data dependencies in rhs& subroutine since many terms depend on terms calculated in previous iterations of the loop.

# LU


LU-Program Average Per Class
3.5
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
LU A
LU B
LU C
LU S
LU W


LU-Program C-Class Sub-Section Speedup
3
2.5
2
1.5
1
0.5
0
1
2
3
4
add
blts
buts
jacld
jacu
l2norm
rhs
rhsx
rhsy
rhsz

# LU - Discussion

- Speedup of about +0.0584/core.

- Loop was nested so that the fastest changing variable is the leftmost to ensure unit stride. This was not often enforced and so were more cache misses. This can be found in most components of the LU benchmark.

- The equations are very long and thus create data dependence. There is also a low number of arithmetic units. Lots of array elements are used and thus there are many data memory accesses.

- High memory requirements. Problem size scaled cubicly, given the stencil was performed on 3D matrices. There were also many auxilliary matricies used to calculated other terms used in the CFD simulations. Given the number of floating point operations and terms required, it used up cache memory. We see drop in performance as number of cores increased because of restricted cache size.

**The above two points are prominent in jacld, blts, jacu, buts. Appears also in rhs**

- True data dependencies in rhs subroutine since many terms depend on terms calculated in previous iterations of the loop.

# Conclusions

- Speedup heavily influenced by data dependencies and architectural hazards
- Theoretical speedup should increase linearly with number of processors, in reality, other processes (e.g. OS, etc.) interfere as well as overhead when creating threads and aggregating results
- Large array sizes can quickly consume cache space, increasing likelihood of cache misses and page faults

# References


1. https://en.wikipedia.org/wiki/Haswell_(microarchitecture)
2. https://en.wikipedia.org/wiki/Broadwell_(microarchitecture)
3. http://openmp.org/
4. https://www.nas.nasa.gov/publications/npb.html
5. https://www.nas.nasa.gov/assets/pdf/techreports/1994/rnr-94-007.pdf
6. https://www.nas.nasa.gov/assets/pdf/techreports/1999/nas-99-011.pdf
7. http://h21007.www2.hp.com/portal/download/files/unprot/fortran/docs/vf-html/pg/pguaracc.htm
8. https://github.com/itomaldonado/multiprocessor-benchmarking-exploration

# Appendix

Extra charts and results

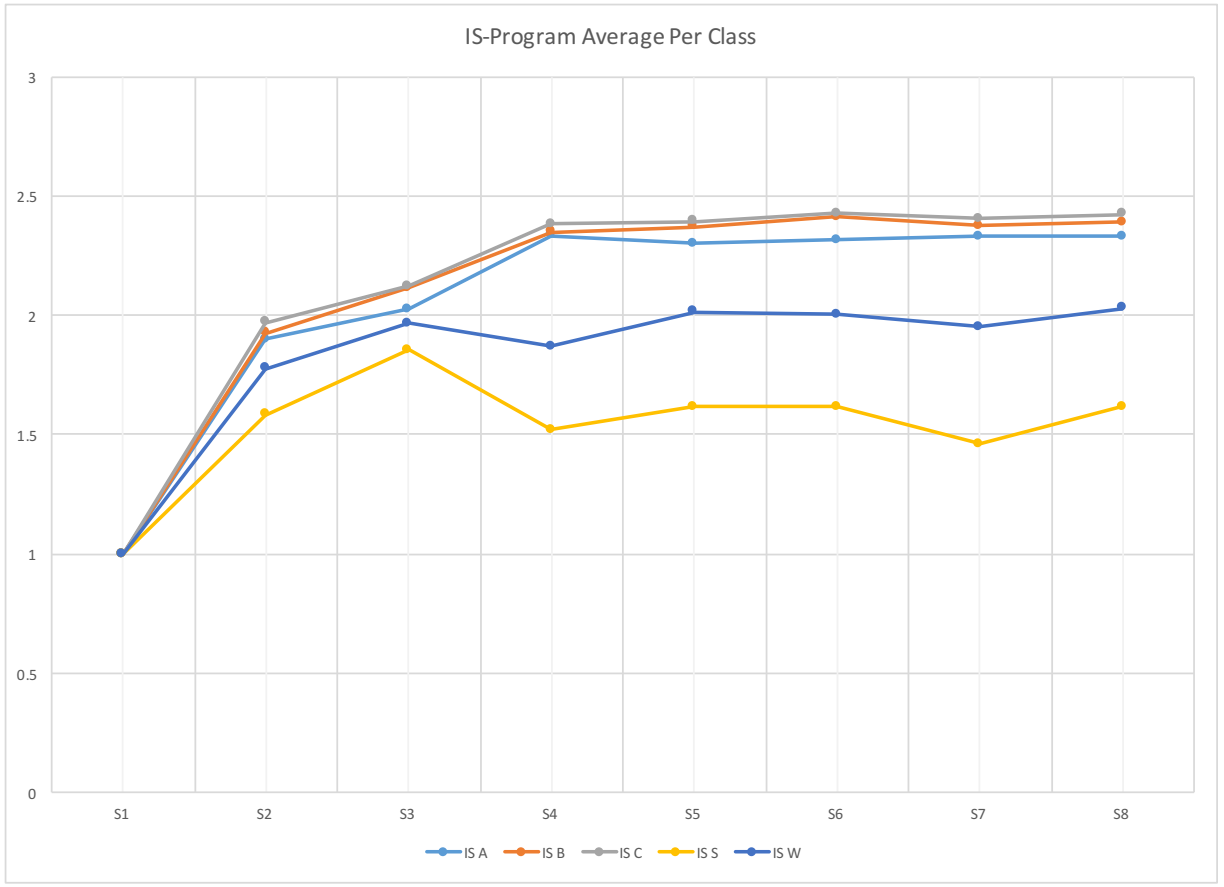
# Computer #2: Speedup Results

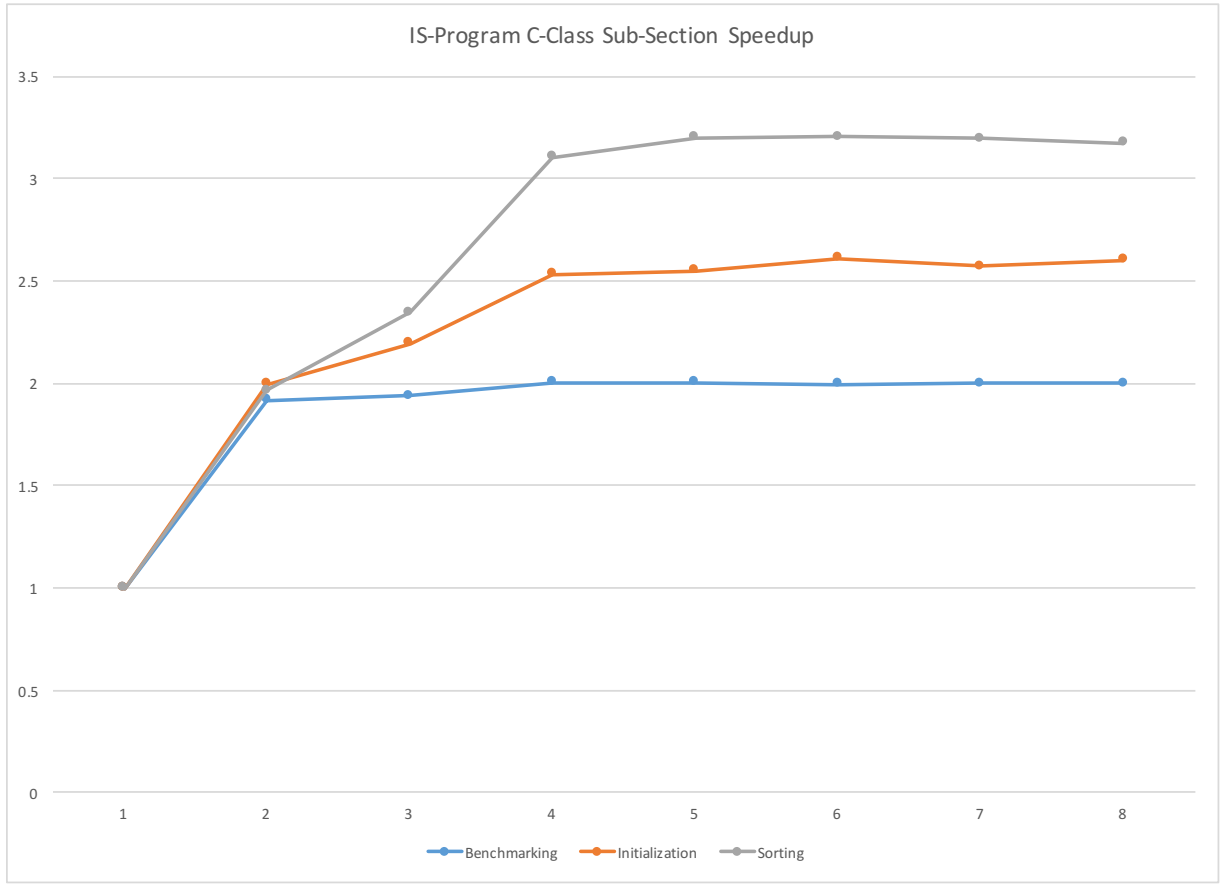
Speedup per program class and speedups for all sub-sections for class C programs.

2 Cores, 1-8 threads.

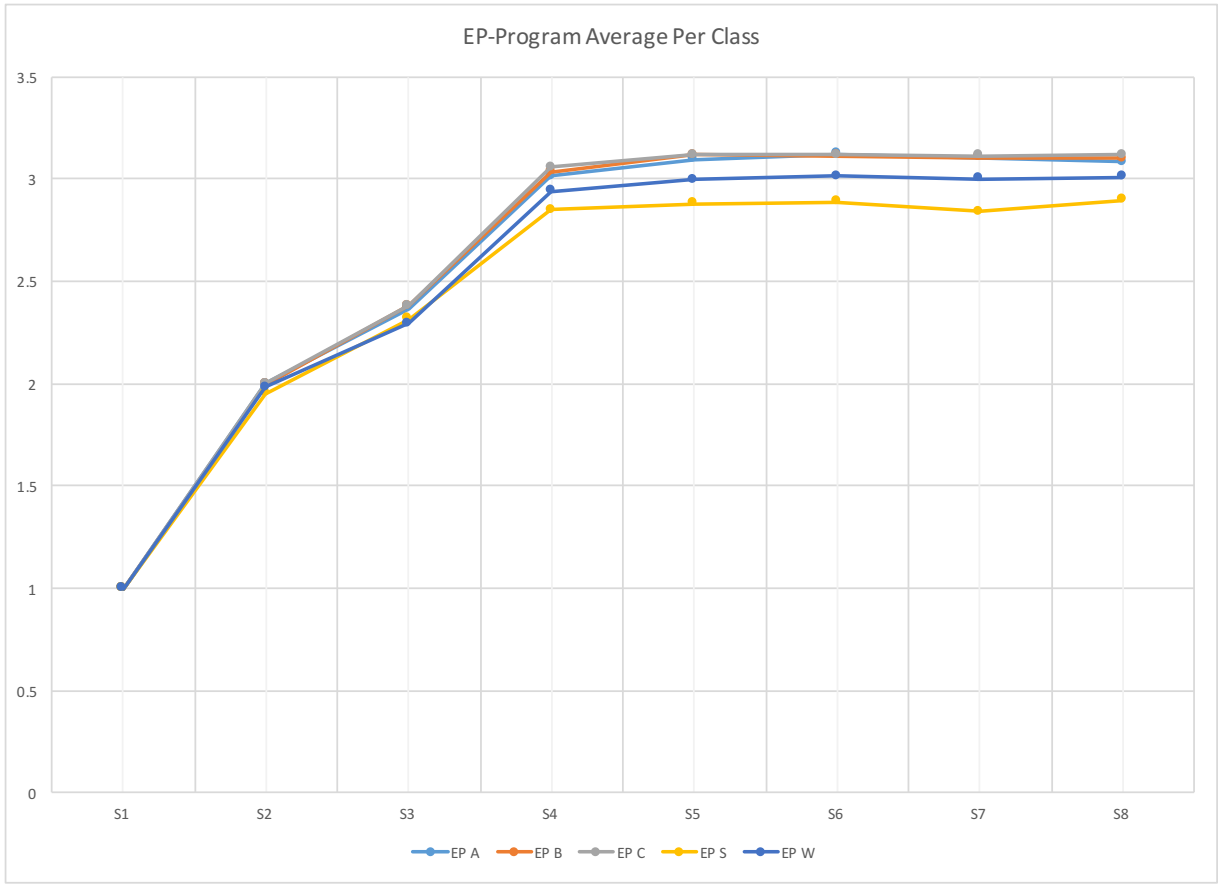
Demonstrates negative performance when overscheduling threads.

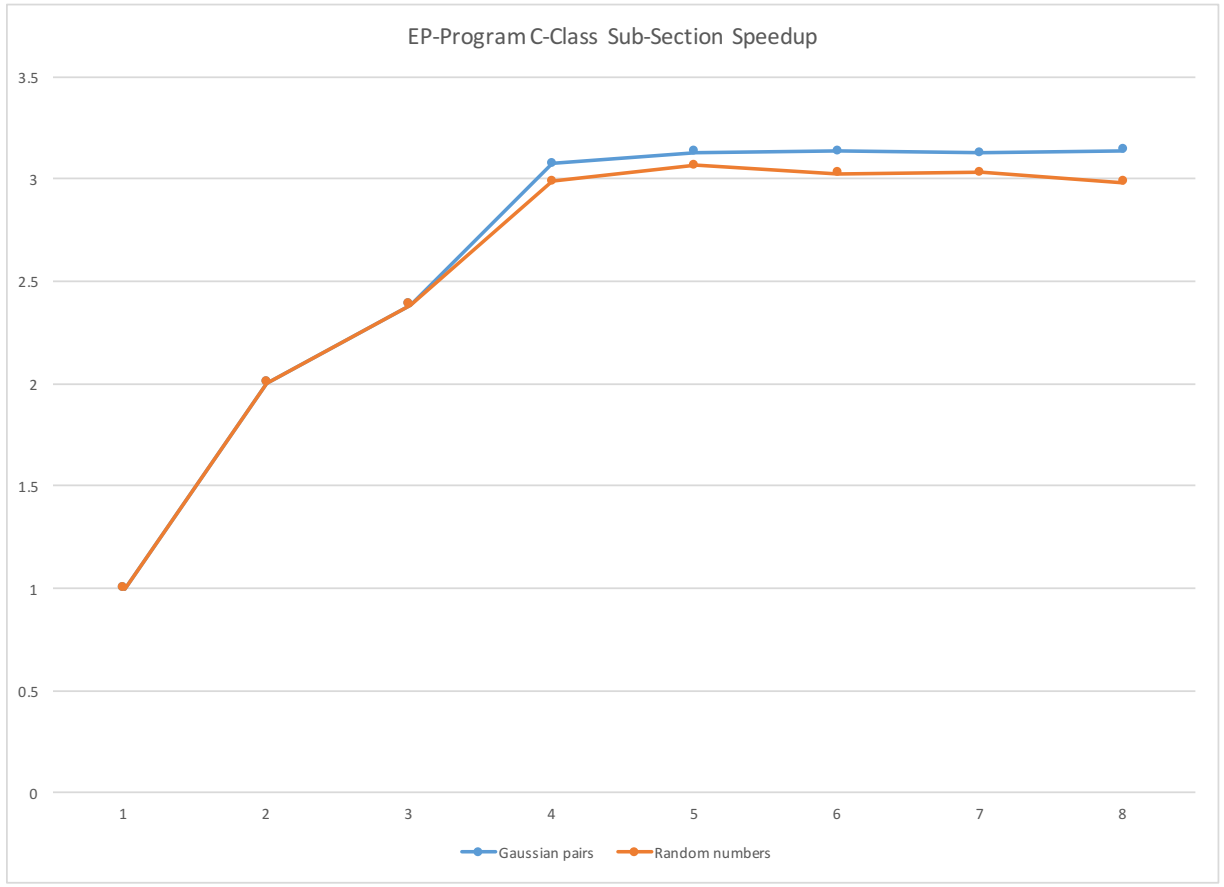
# IS


IS-Program Average Per Class
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
S5
S6
S7
S8
IS A
IS B
IS C
IS S
IS W

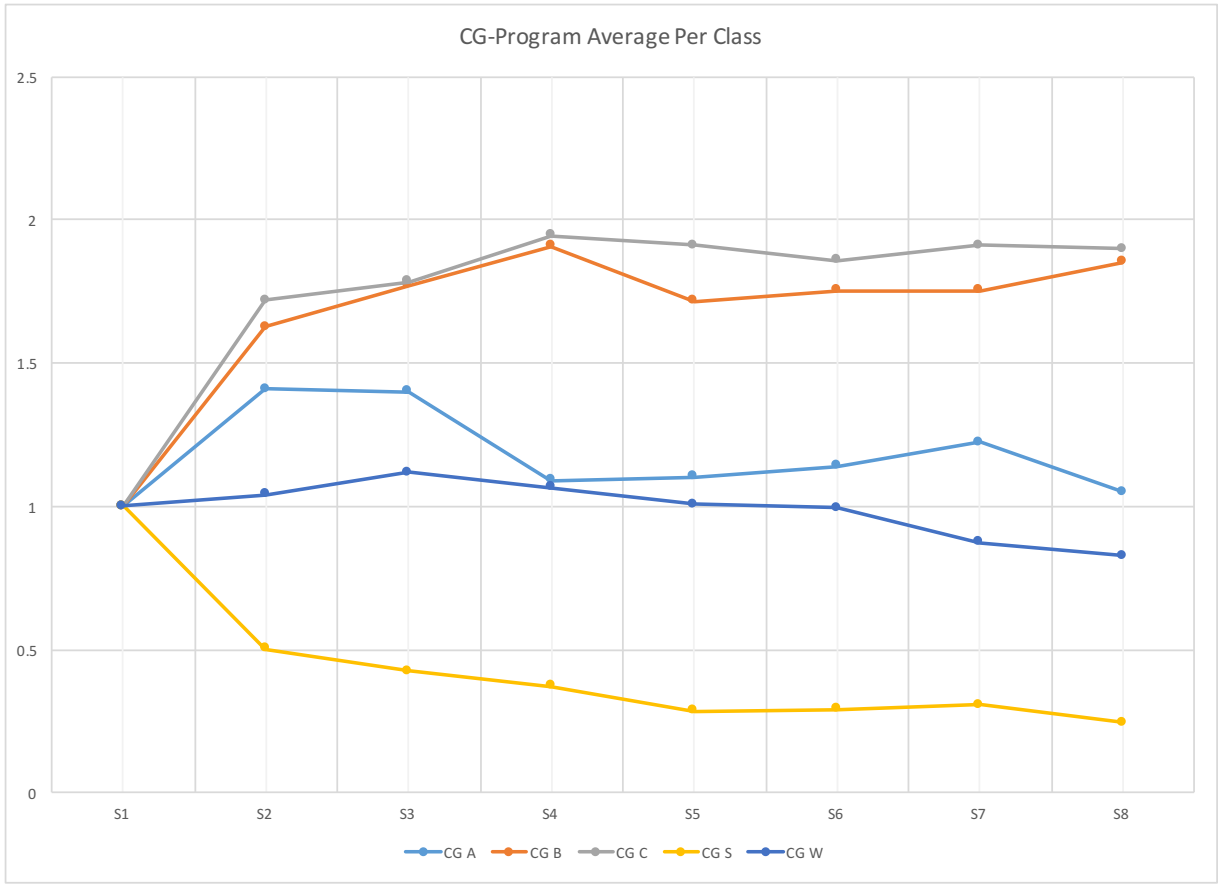

IS-Program C-Class Sub-Section Speedup
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
Benchmarking
Initialization
Sorting

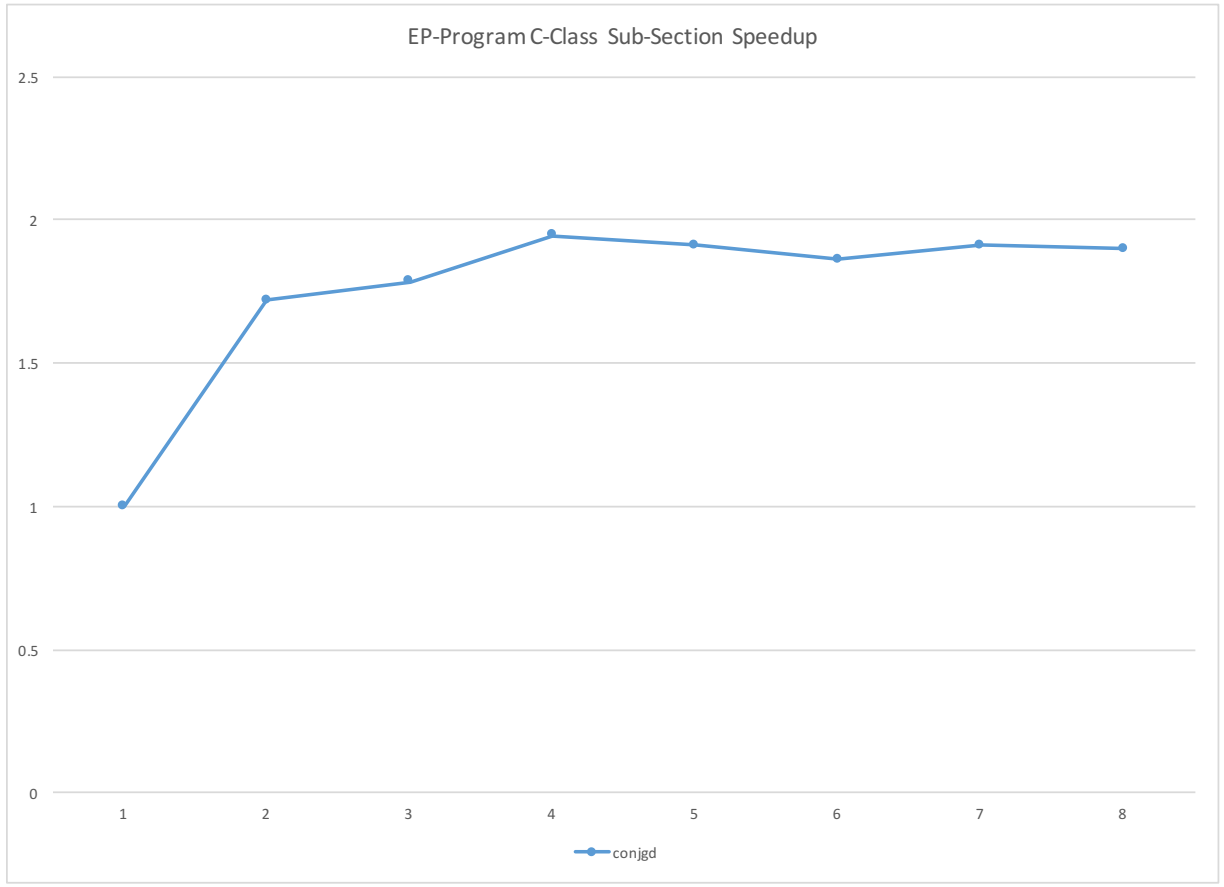
# EP


EP-Program Average Per Class
3.5
3
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
S5
S6
S7
S8
EP A
EP B
EP C
EP S
EP W

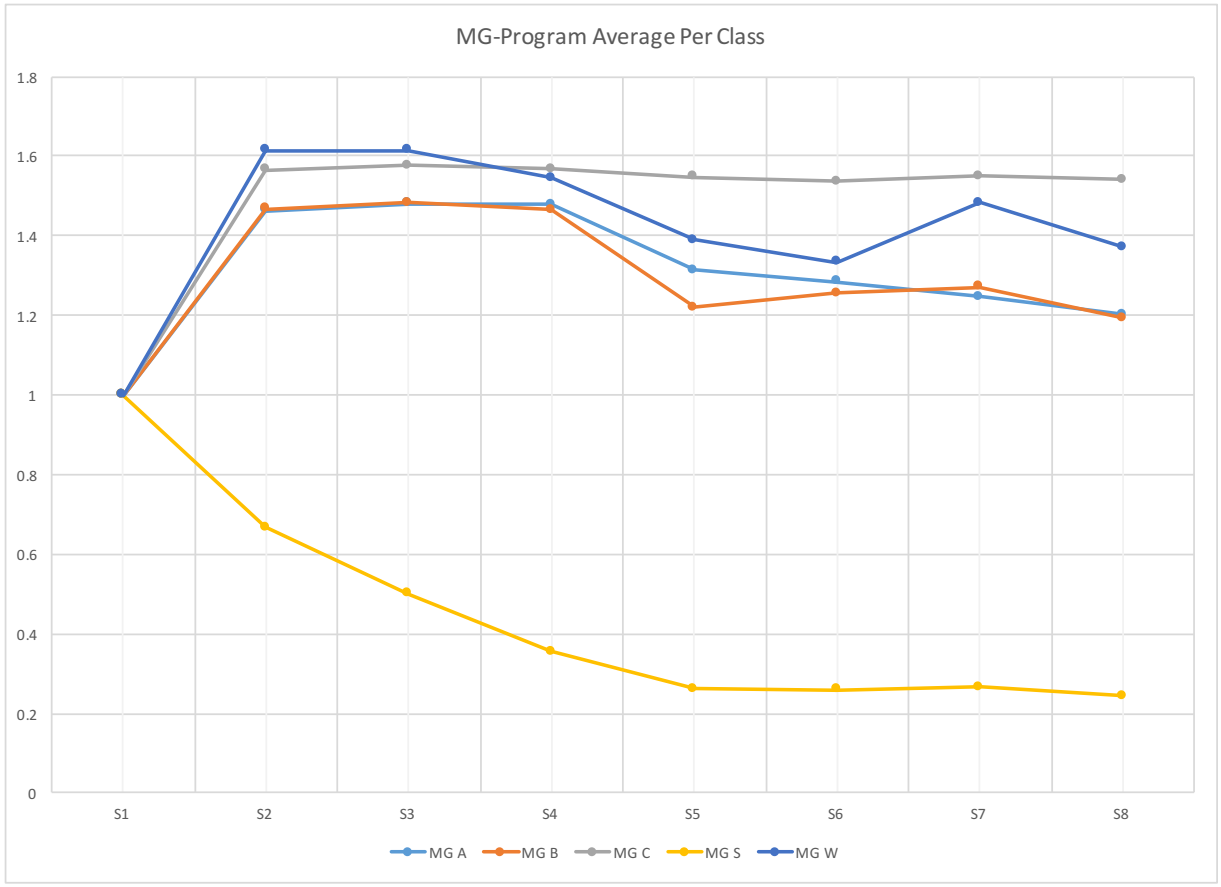

EP-Program C-Class Sub-Section Speedup
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
Gaussian pairs
Random numbers

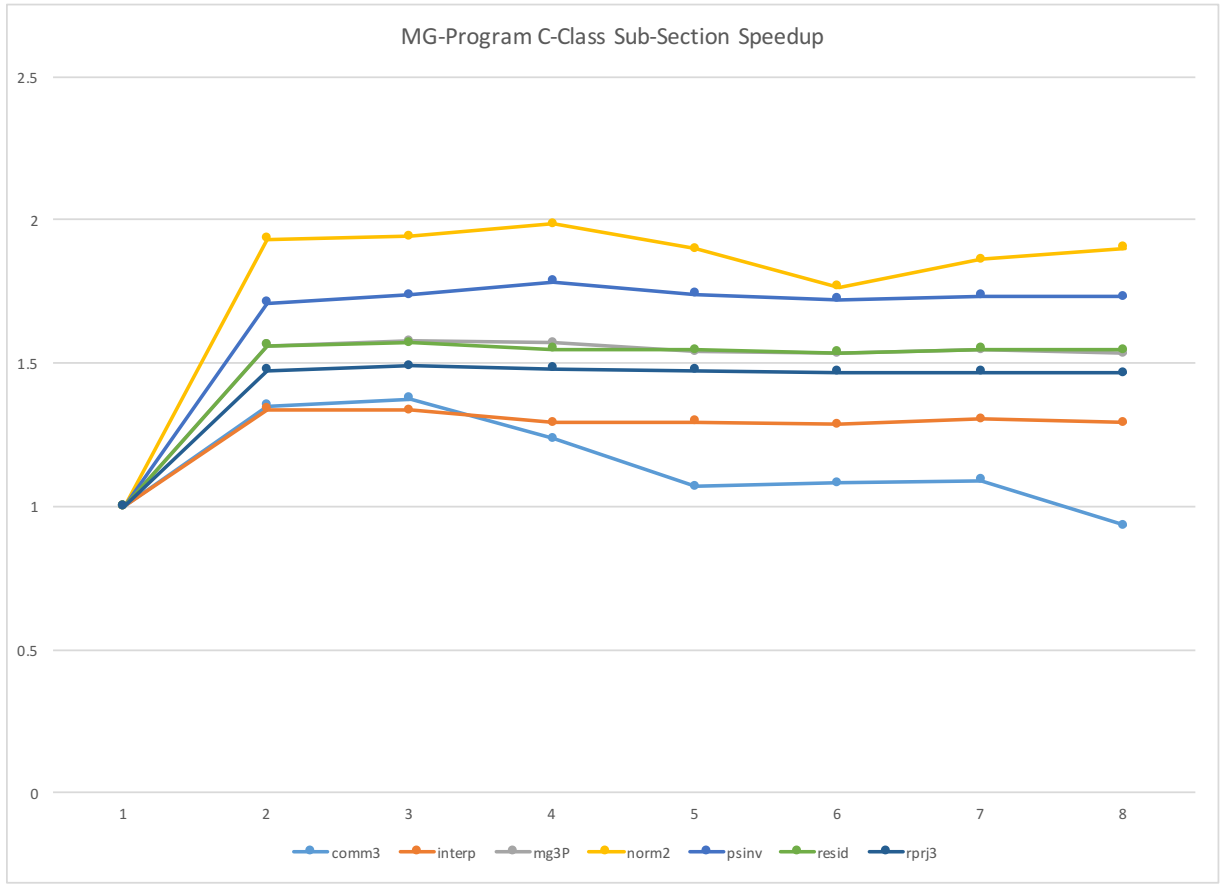
# CG


CG-Program Average Per Class
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
S5
S6
S7
S8
CG A
CG B
CG C
CG S
CG W

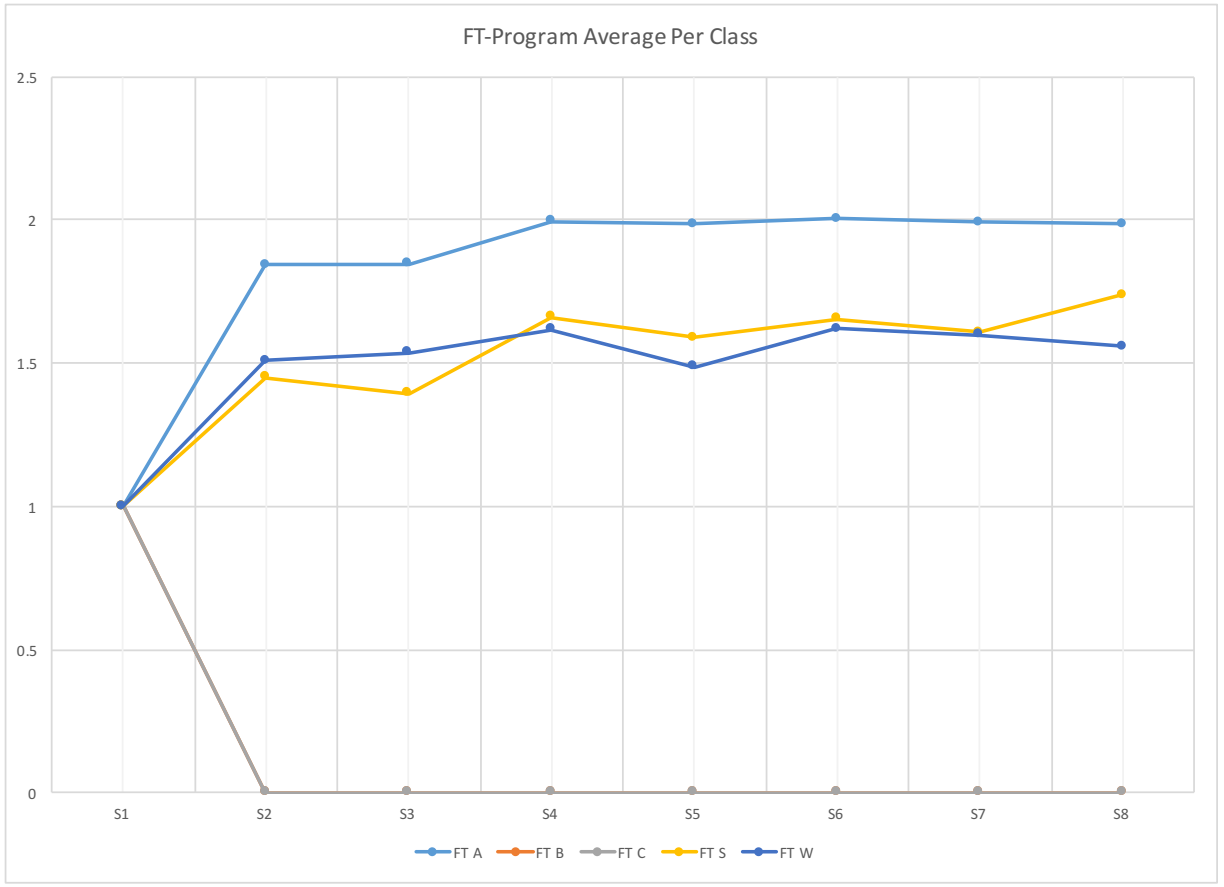

EP-Program C-Class Sub-Section Speedup
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
conjgd

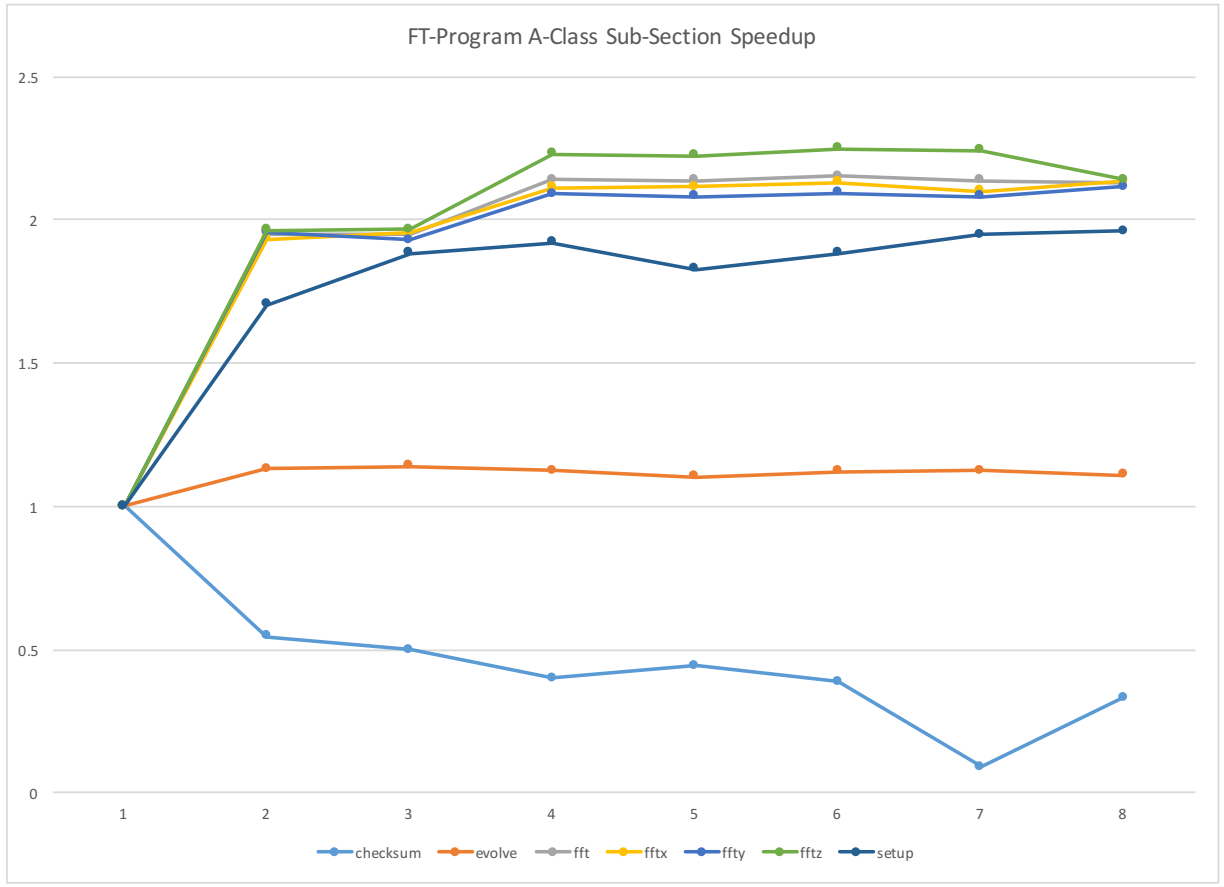
# MG


MG-Program Average Per Class
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
S1
S2
S3
S4
S5
S6
S7
S8
MG A
MG B
MG C
MG S
MG W

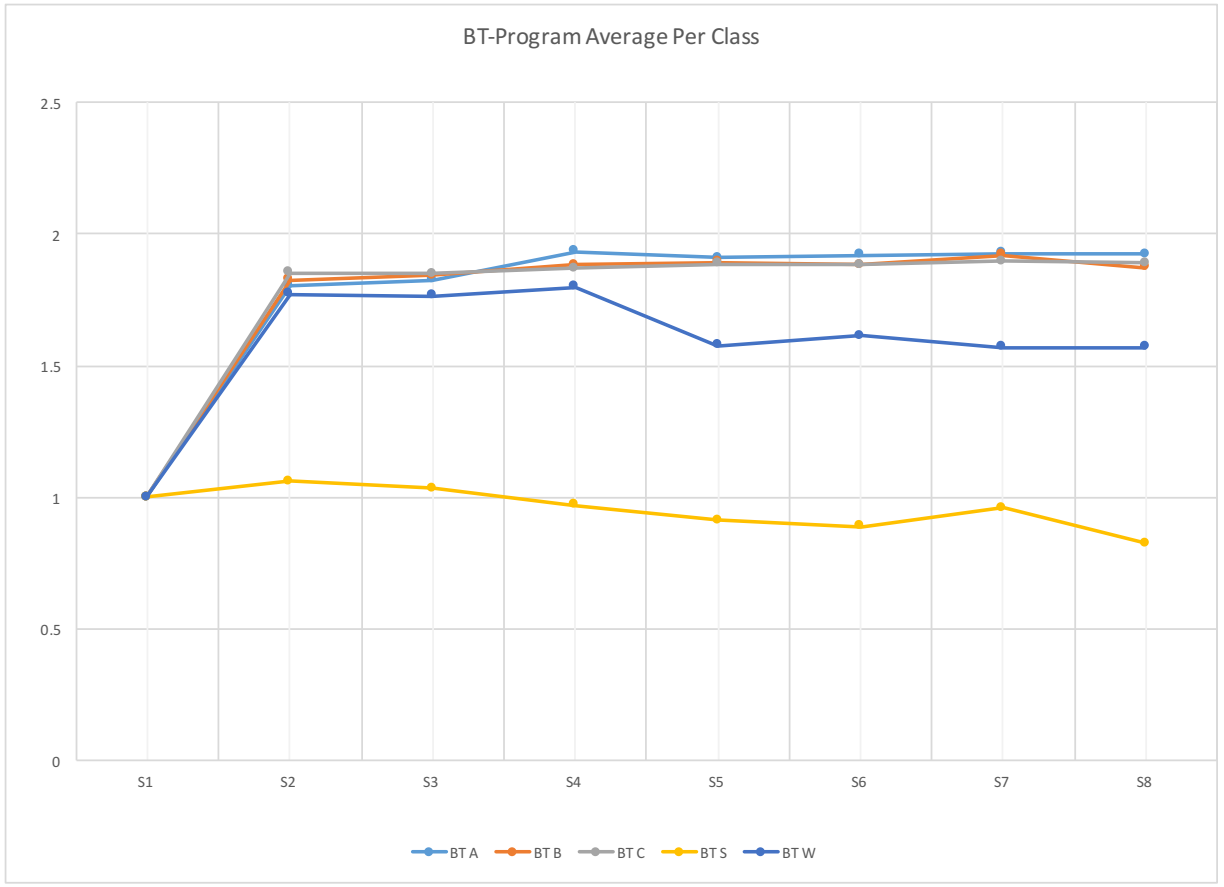

MG-Program C-Class Sub-Section Speedup
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
comm3
interp
mg3P
norm2
psinv
resid
rprj3

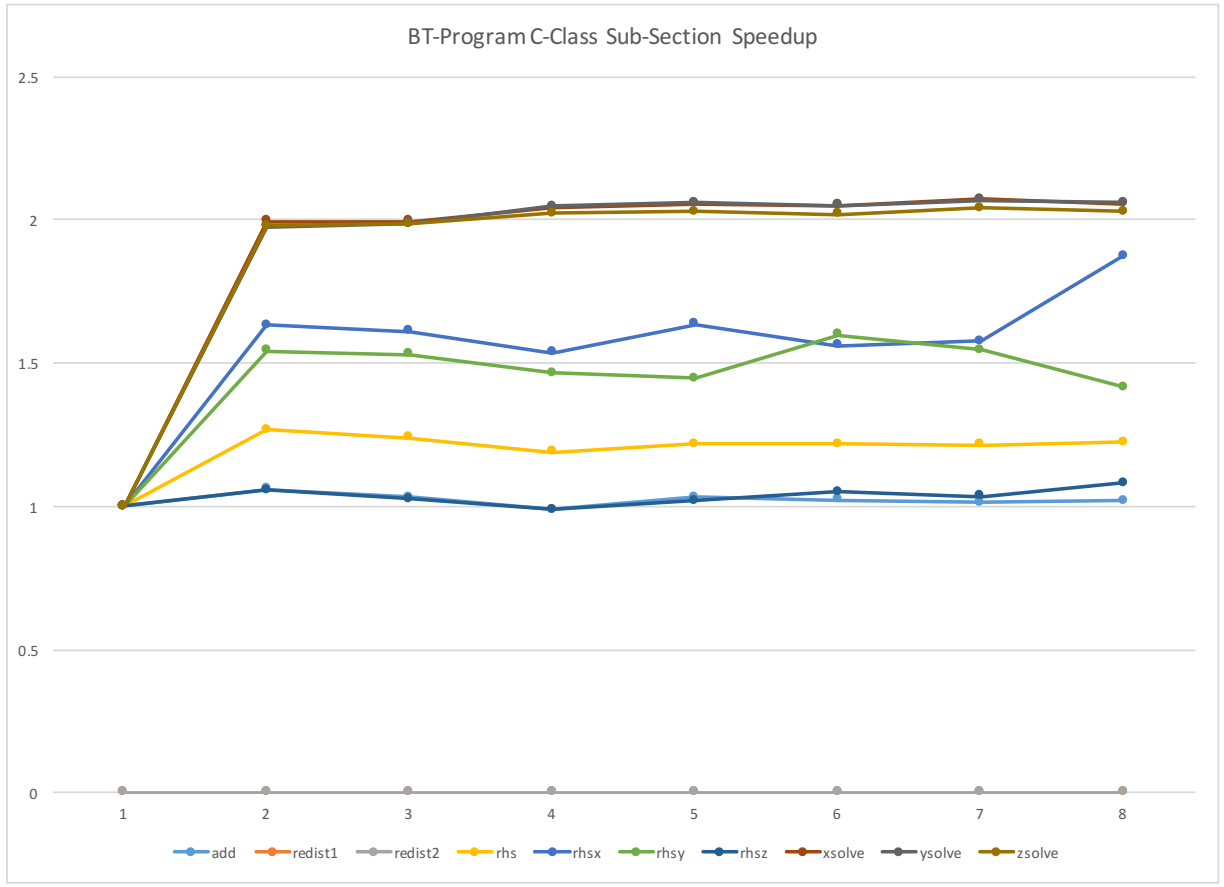
# FT *A Class


FT-Program Average Per Class
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
S5
S6
S7
S8
FT A
FT B
FT C
FT S
FT W

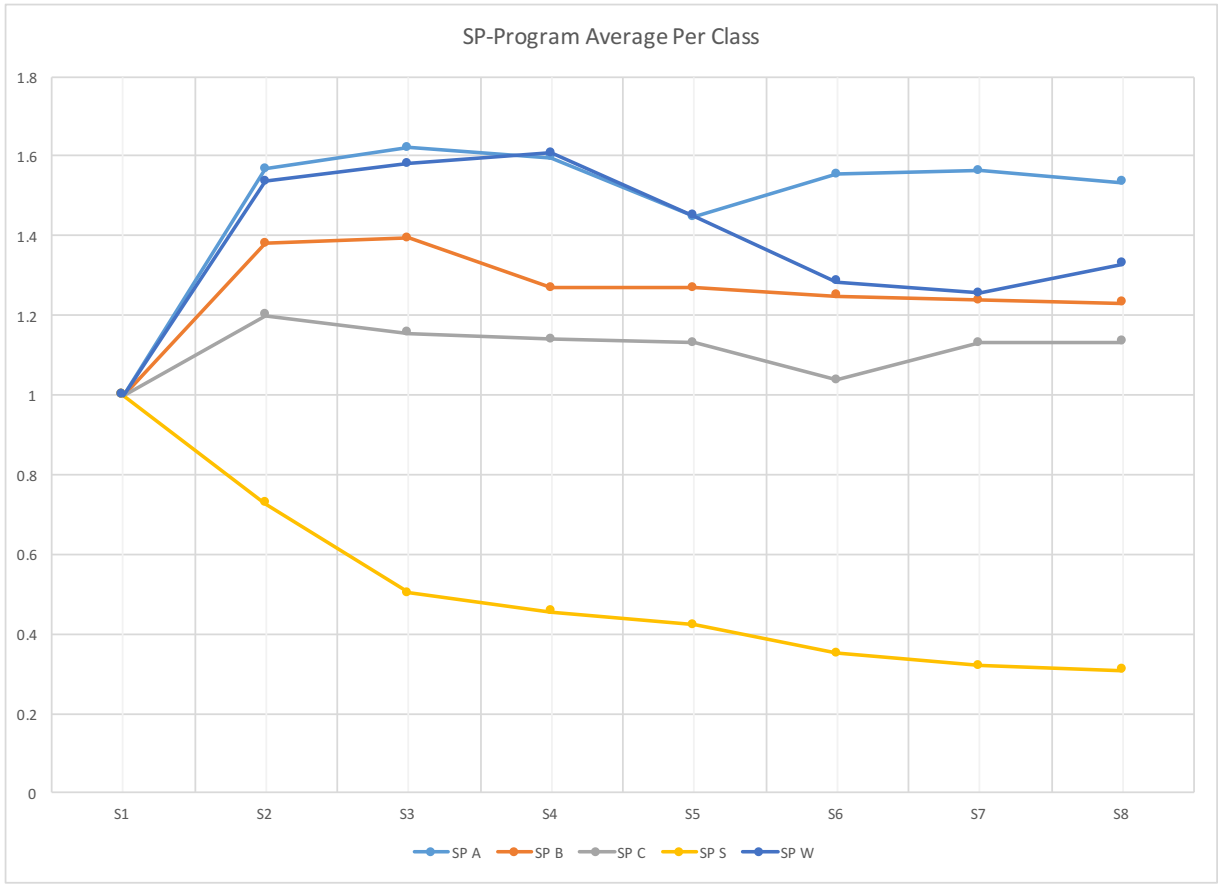

FT-Program A-Class Sub-Section Speedup
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
checksum
evolve
fft
fftx
ffty
fftz
setup

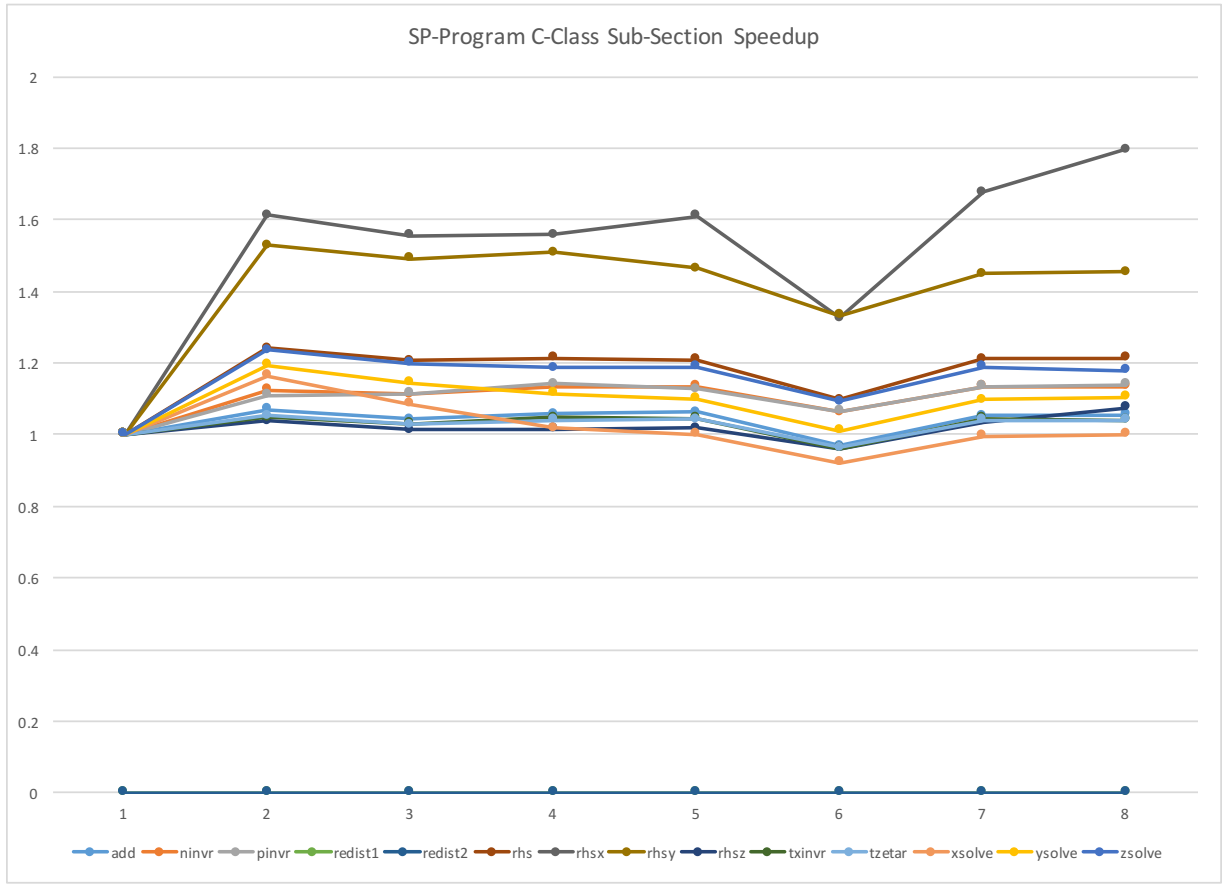
# BT


BT-Program Average Per Class
2.5
2
1.5
1
0.5
0
S1
S2
S3
S4
S5
S6
S7
S8
BT A
BT B
BT C
BT S
BT W

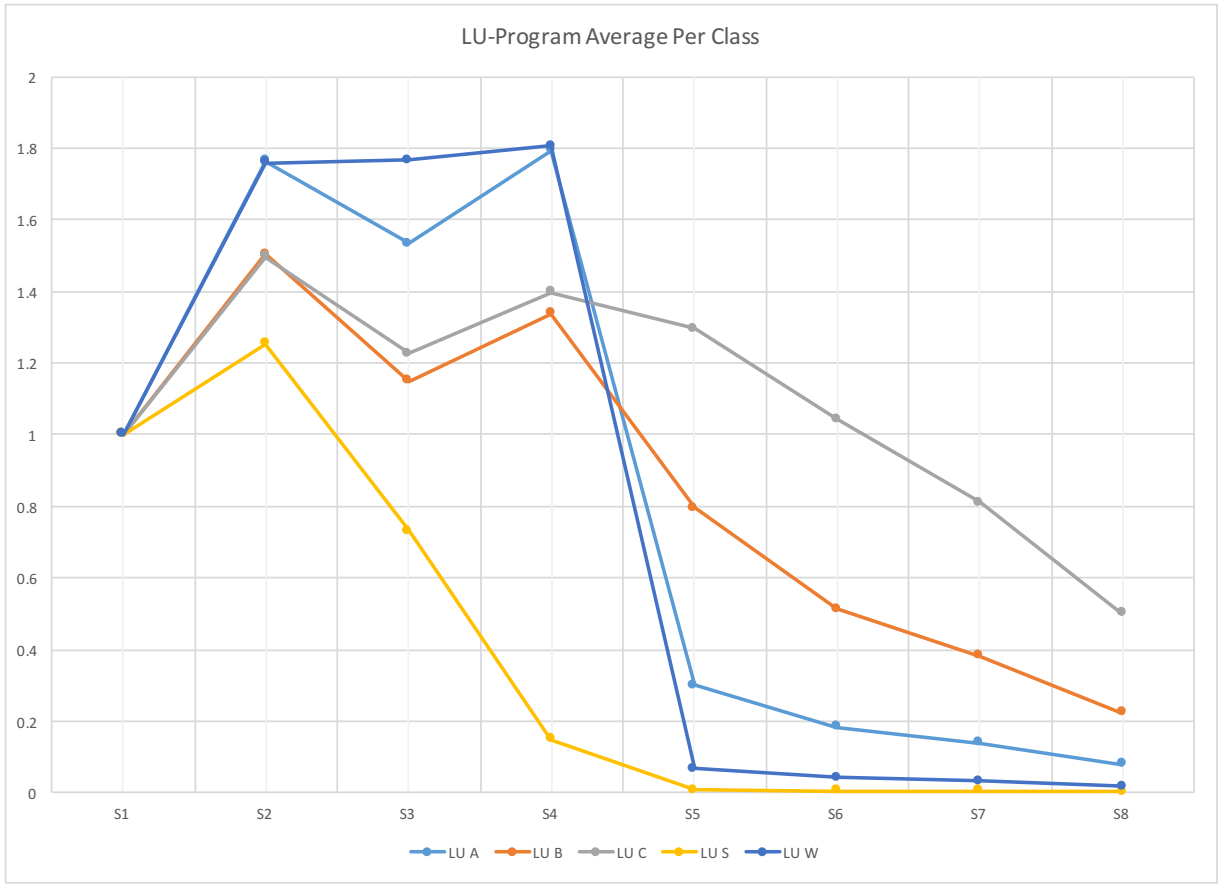

BT-Program C-Class Sub-Section Speedup
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
add
redist1
redist2
rhs
rhsx
rhsy
rhsz
xsolve
ysolve
zsolve

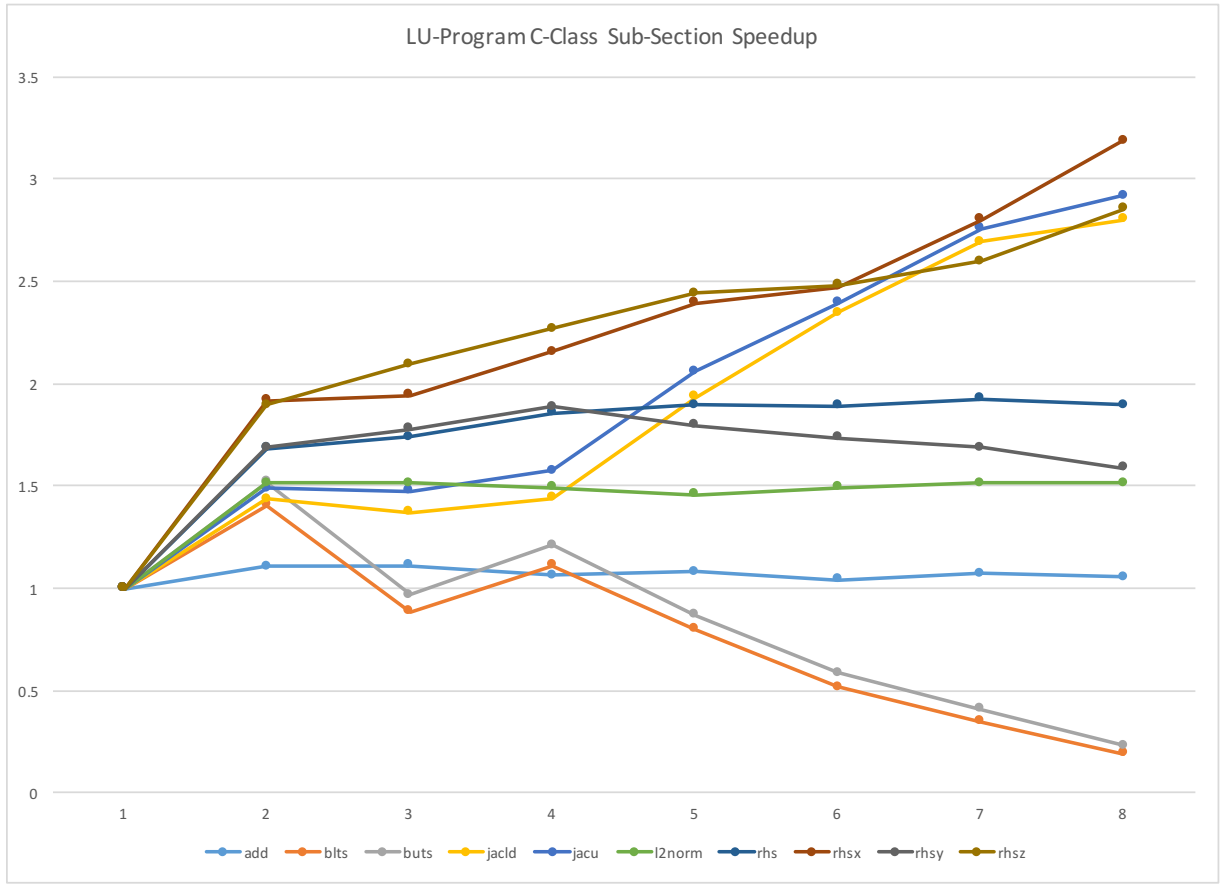
# SP


SP-Program Average Per Class
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
S1
S2
S3
S4
S5
S6
S7
S8
SP A
SP B
SP C
SP S
SP W


SP-Program C-Class Sub-Section Speedup
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
1
2
3
4
5
6
7
8
add
ninvr
pinvr
redist1
redist2
rhs
rhsx
rhsy
rhsz
txinvr
tzetar
xsolve
ysolve
zsolve

# LU


LU-Program Average Per Class
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
S1
S2
S3
S4
S5
S6
S7
S8
LU A
LU B
LU C
LU S
LU W


LU-Program C-Class Sub-Section Speedup
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
add
blts
buts
jacld
jacu
l2norm
rhs
rhsx
rhsy
rhsz