COMP 605:Introduction to Parallel Computing
Topic: Performance - Serial

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 01/30/17
Last Update: 01/30/17

Table of Contents

Timing serial or Parallel Code

Performance measurements help determine computer efficiency.

What/how to measure?

- CPU_time? System? Hardware? I/O? Human?
- What is start/stop time, how to compute?
- Where to time? Critical blocks?
- Subprograms? Overhead?
- Difference between T_{wall}, T_{cpu}, T_{user}
- Data type: integer, char, float, double, ...

Units/Metrics?

- Time: seconds, milliseconds, micro, nano, ...
- Frequency: Hz ($1/sec$)
- Scale: Kilo, Mega, Giga, Tera, Peta, ...
- Operation counts:
  - FLOPS: floating point operations per second
  - instruction level?
  - atomic level?

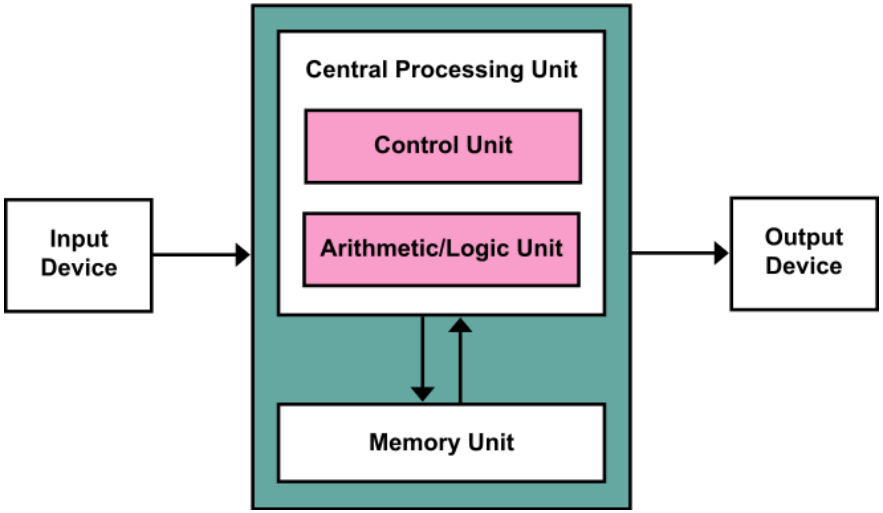
In general, performance is measured not calculated

Large Variety of Performance Tools & Metrics

- "Simple" profiling tools
  - Unix *top* command
  - PBS *qstat* command
  - gprof (GNU Profile)
  - in-code timers *(C CPU_TIME()*
- Advanced profiling tools
  - Allinea DDT
  - Tuning and Analysis Utilities (TAU)
  - integrated performance monitor (IPM)
  - others

Von Neumann electronic digital computer

- Central processing unit:
  - arithmetic logic unit (ALU)
  - processor registers
- Control unit:
  - instruction register
  - program counter
- Memory unit:
  - data
  - instructions
- External mass storage
- Input and output mechanisms



Source: http://en.wikipedia.org/wiki/Von_Neumann_architecture

Total Program Time

Total computer program time is a function of a large number of variables: computer hardware (cpu, memory, software, network), and the program being run (algorithm, problem size, # Tasks, complexity)

$$T = \mathcal{F}\left(ProbSize,\ Tasks,\ I/O,\ \ldots\right)$$

Source: `http://en.wikipedia.org/wiki/Wall-clock_time`

Instructions vs Operations: Floating Point ADD

- Machine instructions
  - specify the name of an operation
  - locations of the operands and the result
- higher level languages: use an operation

  *void store(double * a, double * b, double * c) {*
  **c = *a + *b; }*
- assembler code (X86) for the ADD operation:

```
        .text
        .p2align 4,,15
.globl store
        .type store, @function
store :
        movsd  (%rdi), %xmm0 # Load *a to %xmm0
        addsd  (%rsi), %xmm0 # Load *b to %xmm0
        movsd  %xmm0, (%rdx), # Store to *.c
        ret
```

Source: See Victor Eijkout book

Where to time the code?

- We will focus on timing high level language operations and functions
- Look for where the most work is being done.
- You don't need to time all of the program
- Critical Blocks:
  - Points in the code where you expect to do a large amount of work
  - Problem size dependencies
  - 2D matrix: $\vartheta(n * m)$, Binary Search Tree: $\vartheta(\log n)$
- Input and Output statements:
  - STDIO/STDIN
  - File I/O

Wallclock Time: T_{wall}

Computer Program Run Times

- T_{wall}: A measure of the real time that elapses from the start to the end of a computer program.
- It is the difference between the time at which the program finishes and the time at which the program started.

$$T_{wall} = T_{CPU} + T_{I/O} + T_{Idle} + T_{other}$$

Source: `http://en.wikipedia.org/wiki/Wall-clock_time`

Wallclock Time: T_{wall}

$$T_{wall} = T_{CPU} + T_{I/O} + T_{Idle} + T_{other}$$

- T_{Wall}: The total (or real) time that has elapsed from the start to the completion of a computer program or task.
- T_{CPU}: The amount of time for which a central processing unit (CPU) is used for processing instructions of a computer program or operating system.
- $T_{I/O}$: The time spent by a computer program reading/writing data to/from files such as /STDIN/STDERR, local data files, remote data services or databases.
- T_{Idle}: The time spent by a computer program waiting for execution instructions.
- $T_{overhead}$: The amount of time required to set up a computer program including setting up hardware, local and remote data and resources, network connections, messages.

Serial Histogram (Pacheco IPP)

ser_hist.c

```
/* File:      histogram.c
 * Purpose:   Build a histogram from some random data
 *
 * Compile:   gcc -g -Wall -o histogram histogram.c
 * Run:       ./histogram <bin_count> <min_meas> <max_meas> <data_count>
 *
 * Input:     None
 * Output:    A histogram with X's showing the number of measurements
 *            in each bin
 *
 * Notes:
 * 1.  Actual measurements y are in the range min_meas <= y < max_meas
 * 2.  bin_counts[i] stores the number of measurements x in the range
 * 3.  bin_maxes[i-1] <= x < bin_maxes[i] (bin_maxes[-1] = min_meas)
 * 4.  DEBUG compile flag gives verbose output
 * 5.  The program will terminate if either the number of command line
 *     arguments is incorrect or if the search for a bin for a
 *     measurement fails.
 *
 * IPP:  Section 2.7.1 (pp. 66 and ff.)
 */
 #include <stdio.h>
#include <stdlib.h>
void Usage(char prog_name[]);
void Get_args( char* argv[],  int* bin_count_p, float*   min_meas_p ,float* max_meas_p, int* data_count_p);
void Gen_data(float min_meas, float max_meas, float   data[],  int data_count);
void Gen_bins(float min_meas , float max_meas/, float bin_maxes[], int   bin_counts[], int   bin_count);
int Which_bin(float data, float bin_maxes[], int bin_count, float min_meas);
void Print_histo(float bin_maxes[], int bin_counts[],  int bin_count, float min_meas);
```

ser_hist.c (cont)

```
int main(int argc, char* argv[]) {
   int bin_count, i, bin;
   float min_meas, max_meas;
   float* bin_maxes;
   int* bin_counts;
   int data_count;
   float* data;

   /* Check and get command line args */
   if (argc != 5) Usage(argv[0]);
   Get_args(argv, &bin_count, &min_meas, &max_meas, &data_count);

   /* Allocate arrays needed */
   bin_maxes = malloc(bin_count*sizeof(float));
   bin_counts = malloc(bin_count*sizeof(int));
   data = malloc(data_count*sizeof(float));

   /* Generate the data */
   Gen_data(min_meas, max_meas, data, data_count);

   /* Create bins for storing counts */
   Gen_bins(min_meas, max_meas, bin_maxes, bin_counts, bin_count);

   /* Count number of values in each bin */
   for (i = 0; i < data_count; i++) {
      bin = Which_bin(data[i], bin_maxes, bin_count, min_meas)
      bin_counts[bin]++;      }
#  ifdef DEBUG
   printf("bin_counts = ");
   for (i = 0; i < bin_count; i++)
      printf("%d ", bin_counts[i]);
   printf("\n");
#  endif

   /* Print the histogram */
   Print_histo(bin_maxes, bin_counts, bin_count, min_meas);

   free(data);
   free(bin_maxes);
   free(bin_counts);
   return 0;
}  /* main */
```

ser_hist.c (cont)

```
/*--------------------------------------------------------------------
* Function:  Usage
 * Purpose:   Print a message showing how to run program and quit
 * In arg:    prog_name:  the name of the program from the command line
 */
 void Usage(char prog_name[]);  {
   fprintf(stderr, "usage: %s ", prog_name);
   fprintf(stderr, "<bin_count> <min_meas> <max_meas> <data_count>\n");
   exit(0);
}  /* Usage */

/*--------------------------------------------------------------------
 * Function:  Get_args
 * Purpose:   Get the command line arguments
 * In arg:    argv:  strings from command line
 * Out args:  bin_count_p:   number of bins
 *            min_meas_p:    minimum measurement
 *            max_meas_p:    maximum measurement
 *            data_count_p:  number of measurements
 */
 void Get_args( char* argv[],  int* bin_count_p, float*   min_meas_p ,float* max_meas_p, int* data_count_p)  {
   *bin_count_p = strtol(argv[1], NULL, 10);
   *min_meas_p = strtof(argv[2], NULL);
   *max_meas_p = strtof(argv[3], NULL);
   *data_count_p = strtol(argv[4], NULL, 10);

#  ifdef DEBUG
   printf("bin_count = %d\n", *bin_count_p);
   printf("min_meas = %f, max_meas = %f\n", *min_meas_p, *max_meas_p);
   printf("data_count = %d\n", *data_count_p);
#  endif
}  /* Get_args */
```

ser_hist.c (cont)

```
/*---------------------------------------------------------------------
* Function:  Gen_data
 * Purpose:   Generate random floats in the range min_meas <= x < max_meas
 * In args:   min_meas:    the minimum possible value for the data
 *            max_meas:    the maximum possible value for the data
 *            data_count:  the number of measurements
 * Out arg:   data:        the actual measurements
 */
void Gen_data(float min_meas, float max_meas, float   data[],  int data_count)   {
   srandom(0);
   for (i = 0; i < data_count; i++)
      data[i] = min_meas + (max_meas - min_meas)*random()/((double) RAND_MAX);
#  ifdef DEBUG
   printf("data = ");
   for (i = 0; i < data_count; i++)
      printf("%4.3f ", data[i]);
   printf("\n");
#  endif
}  /* Gen_data */
```

ser_hist.c (cont)

```
/*-------------------------------------------------------------------
 * Function:  Gen_bins
 * Purpose:   Compute max value for each bin, and store 0 as the
 *            number of values in each bin
 * In args:   min_meas:   the minimum possible measurement
 *            max_meas:   the maximum possible measurement
 *            bin_count:  the number of bins
 * Out args:  bin_maxes:  the maximum possible value for each bin
 *            bin_counts: the number of data values in each bin
 */
void Gen_bins(float min_meas , float max_meas/, float bin_maxes[], int   bin_counts[], int   bin_count);
 {
   float bin_width;
   int   i;
   bin_width = (max_meas - min_meas)/bin_count;
   for (i = 0; i < bin_count; i++) {
      bin_maxes[i] = min_meas + (i+1)*bin_width;
      bin_counts[i] = 0;
   }

#  ifdef DEBUG
   printf("bin_maxes = ");
   for (i = 0; i < bin_count; i++)
      printf("%4.3f ", bin_maxes[i]);
   printf("\n");
#  endif
}  /* Gen_bins */
```

ser_hist.c (cont)

```
/*-------------------------------------------------------------------
 * Function:  Which_bin
 * Purpose:   Use binary search to determine which bin a measurement belongs to
 * In args:   data:       the current measurement
 *            bin_maxes:  list of max bin values
 *            bin_count:  number of bins
 *            min_meas:   the minimum possible measurement
 * Return:    the number of the bin to which data belongs
 * Notes:
 * 1.  The bin to which data belongs satisfies
 *            bin_maxes[i-1] <= data < bin_maxes[i]
 *     where, bin_maxes[-1] = min_meas
 * 2.  If the search fails, the function prints a message and exits
 */
int Which_bin(float data, float bin_maxes[], int bin_count, float min_meas) {
   int bottom = 0, top =  bin_count-1;
   int mid;
   float bin_max, bin_min;

   while (bottom <= top) {
      mid = (bottom + top)/2;
      bin_max = bin_maxes[mid];
      bin_min = (mid == 0) ? min_meas: bin_maxes[mid-1];
      if (data >= bin_max)
         bottom = mid+1;
      else if (data < bin_min)
         top = mid-1;
      else
         return mid;
   }
   /* Whoops! */
   fprintf(stderr, "Data = %f doesn't belong to a bin!\n", data);
   fprintf(stderr, "Quitting\n");
   exit(-1);
}  /* Which_bin */
```

ser_hist.c (cont)

```
/*---------------------------------------------------------------------
 * Function:  Print_histo
 * Purpose:   Print a histogram.  The number of elements in each
 *            bin is shown by an array of X's.
 * In args:   bin_maxes:   the max value for each bin
 *            bin_counts:  the number of elements in each bin
 *            bin_count:   the number of bins
 *            min_meas:    the minimum possible measurment
 */
void Print_histo(float bin_maxes[], int bin_counts[],  int bin_count, float min_meas)   {
   int i, j;
   float bin_max, bin_min;

   for (i = 0; i < bin_count; i++) {
      bin_max = bin_maxes[i];
      bin_min = (i == 0) ? min_meas: bin_maxes[i-1];
      printf("%.3f-%.3f:\t", bin_min, bin_max);
      for (j = 0; j < bin_counts[i]; j++)
         printf("X");
      printf("\n");
   }
}  /* Print_histo */
```

Timing serial histogram code: gettimeofday()

```
int main(int argc, char* argv[]) {
   int bin_count, i, bin;
   float min_meas, max_meas;
   float* bin_maxes;
   int* bin_counts;
   int data_count;
   float* data;

   struct timeval tstart_wall,tstart_mem, tstart_getargs, tstart_gendat, tstart_genbins, tstart_whichbin
   struct timeval tstop_wall,tstop_mem, tstop_getargs, tstop_gendat, tstop_genbins, tstop_whichbin ;
   double T_wall,T_mem, T_gendat, T_genbins, T_whichbin ;

   gettimeofday (&tstart_wall, NULL);
   /* Check and get command line args */
   if (argc != 5) Usage(argv[0]);
   gettimeofday (&tstart_getargs, NULL);
   Get_args(argv, &bin_count, &min_meas, &max_meas, &data_count);
   gettimeofday (&tstop_getargs, NULL);

   /* Allocate arrays needed */
   gettimeofday (&tstart_mem, NULL);
   bin_maxes = malloc(bin_count*sizeof(float));
   bin_counts = malloc(bin_count*sizeof(int));
   data = malloc(data_count*sizeof(float));
   gettimeofday (&tstop_mem, NULL);

   /* Generate the data */
   gettimeofday (&tstart_gendat, NULL);
   Gen_data(min_meas, max_meas, data, data_count);
   gettimeofday (&tstop_gendat, NULL);

   /* Create bins for storing counts */
   gettimeofday (&tstart_genbins, NULL);
   Gen_bins(min_meas, max_meas, bin_maxes, bin_counts, bin_count);
   gettimeofday (&tstop_genbins, NULL);
```

Timing serial histogram code: gettimeofday()

```
/* Count number of values in each bin */
  gettimeofday (&tstart_whichbin, NULL);
  for (i = 0; i < data_count; i++) {
     bin = Which_bin(data[i], bin_maxes, bin_count, min_meas);
     bin_counts[bin]++;
  }
  gettimeofday (&tstop_whichbin, NULL);
  /* Print the histogram */
  //Print_histo(bin_maxes, bin_counts, bin_count, min_meas);
  //Print_histo_dat(bin_maxes, bin_counts, bin_count, min_meas);

  gettimeofday (&tstop_wall, NULL);
  ///T_wall,T_init, T_gendat, T_genbins, T_whichbin ;
  T_mem= (double)( (tstop_mem.tv_sec - tstart_mem.tv_sec)*1.0E6
         +tstop_mem.tv_usec - tstart_mem.tv_usec ) / 1.0E6;
  T_wall= (double)( (tstop_wall.tv_sec - tstart_wall.tv_sec)*1.0E6
         +tstop_wall.tv_usec - tstart_wall.tv_usec ) / 1.0E6;
  T_gendat= (double)( (tstop_gendat.tv_sec - tstart_gendat.tv_sec)*1.0E6
         +tstop_gendat.tv_usec - tstart_gendat.tv_usec ) / 1.0E6;
  T_genbins= (double)( (tstop_genbins.tv_sec - tstart_genbins.tv_sec)*1.0E6
         +tstop_genbins.tv_usec - tstart_genbins.tv_usec ) / 1.0E6;
  T_whichbin= (double)( (tstop_whichbin.tv_sec - tstart_whichbin.tv_sec)*1.0E6
         +tstop_whichbin.tv_usec - tstart_whichbin.tv_usec ) / 1.0E6;

  printf("T_mem in seconds: %f seconds\n", T_mem);
  printf("T_gendat in seconds: %f seconds\n", T_gendat);
  printf("T_genbins in seconds: %f seconds\n", T_genbins);
  printf("T_whichbin in seconds: %f seconds\n", T_whichbin);
  printf("Time sum in seconds: %f seconds\n",T_gendat+T_genbins+T_whichbin+T_mem);
  printf("T_wall in seconds: %f seconds\n", T_wall);

  free(data);
  free(bin_maxes);
  free(bin_counts);
  return 0;
```

Compiling code

```
[mthomas@tuckoo looptst]$ cat makefile
=====================================
MAKE FILE
=====================================
MPIF90 = mpif90
MPICC = mpicc
CC     = gcc
all: histogram, histodat
histogram: histogram.c
        $(MPICC) -o histogram histogram.c

histodat: histodat.c
        $(MPICC) -p -o histodat histodat.c

clean:
        rm -rf *.o histogram, histodat
[mthomas@tuckoo ch2]%
[mthomas@tuckoo ch2]%
[mthomas@tuckoo ch2]$ make histodat
make: 'histodat' is up to date.
[mthomas@tuckoo ch2]$ rm histodat
[mthomas@tuckoo ch2]$ make histodat
cc      histodat.c   -o histodat
[mthomas@tuckoo ch2]$ ls histodat
-rwxrwxr-x 1 mthomas mthomas 11724 Jan 29 13:58 histodat
```

Running the Job

```
[mthomas@tuckoo ch2]$ ./histodat 10 1 1000 1000000
T_wall in seconds: 0.107674 seconds
T_getargs in seconds: 0.000018 seconds
T_mem in seconds: 0.000010 seconds
T_gendat in seconds: 0.021932 seconds
T_genbins in seconds: 0.000001 seconds
T_whichbin in seconds: 0.085712 seconds
Initialization Timein seconds: 0.021961 seconds
Sum Times in seconds: 0.107673 seconds
Data: bin_count,data_count,T_wall,T_getargs,
                T_mem,T_gendat,T_genbins,T_whichbin
CSV Dat:10,1000000,0.107674,0.000018,0.000010,
                0.021932,0.000001,0.085712
```

Histogram outputs: Observations

- Job runtimes will vary
- Need to run multiple times to gather statistics
- What causes the variations?
  - other processes running on the system
  - other users
  - local/remote data or services dependency
  - other .

```
[mthomas@tuckoo ch2]$ ./histodat 10 1 100 100
T_wall in seconds: 0.000140 seconds
T_getargs in seconds: 0.000017 seconds
T_mem in seconds: 0.000106 seconds
T_gendat in seconds: 0.000007 seconds
T_genbins in seconds: 0.000000 seconds
T_whichbin in seconds: 0.000010 seconds
Initialization Timein seconds: 0.000130 seconds
Sum Times in seconds: 0.000140 seconds
Data: bin_count,data_count,T_wall,T_getargs,T_mem, ...
CSV Dat:10,100,0.000140,0.000017, ...
[mthomas@tuckoo ch2]$
[mthomas@tuckoo ch2]$ ./histodat 10 1 100 1000
T_wall in seconds: 0.000247 seconds
T_getargs in seconds: 0.000018 seconds
T_mem in seconds: 0.000111 seconds
T_gendat in seconds: 0.000027 seconds
T_genbins in seconds: 0.000000 seconds
T_whichbin in seconds: 0.000091 seconds
Initialization Timein seconds: 0.000156 seconds
Sum Times in seconds: 0.000247 seconds
Data: bin_count,data_count,T_wall, ...
CSV Dat:10,1000,0.000247, ...
[mthomas@tuckoo ch2]$
[mthomas@tuckoo ch2]$ ./histodat 10 1 100 10000
T_wall in seconds: 0.001275 seconds
T_getargs in seconds: 0.000016 seconds
T_mem in seconds: 0.000105 seconds
T_gendat in seconds: 0.000239 seconds
T_genbins in seconds: 0.000000 seconds
T_whichbin in seconds: 0.000915 seconds
Initialization Timein seconds: 0.000360 seconds
Sum Times in seconds: 0.001275 seconds
Data: bin_count,data_count,T_wall,T_getargs,...
CSV Dat:10,10000,0.001275,0.000016,  .
```

```
top - 16:38:31 up 5 days,  8:05,  7 users,  load average: 0.28, 0.31, 0.20
Tasks: 176 total,   2 running, 174 sleeping,   0 stopped,   0 zombie
Cpu(s): 24.2%us,  0.8%sy,  0.0%ni, 73.5%id,  1.4%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:  12188132k total,  4513528k used,  7674604k free,    29736k buffers
Swap: 33409020k total,    21692k used, 33387328k free,  1665928k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
16744 [redacted]  20   0 4664m 2.4g  896 R 99.7 20.9   0:11.82 histogram_mod
15234 [redacted]  20   0  105m 1812 1440 S  0.3  0.0   0:00.15 bash
16721 mthomas   20   0 15040 1292  940 R  0.3  0.0   0:00.05 top
    1 root      20   0 19364  696  480 S  0.0  0.0   0:02.61 init
    2 root      20   0     0    0    0 S  0.0  0.0   0:00.01 kthreadd
```

Profiling the programs using GPROF

We can use profiling applications to analyze the program call tree and obtain some timings. How closely do our results agree?

```
==================================
PROFILING: using -p option in make
==================================
[mthomas@tuckoo ch2]$ mpicc -p  -o histodat histodat.c
[mthomas@tuckoo ch2]$ ./histodat 10 1 1000 1000000
T_wall in seconds: 0.107674 seconds
 .
T_whichbin in seconds: 0.085712 seconds
 .
[mthomas@tuckoo ch2]$ gprof histodat gmon.out
 %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 75.19      0.06     0.06  1000000     0.00     0.00  Which_bin
 12.53      0.07     0.01        1    10.03    10.03  Gen_data
 12.53      0.08     0.01                             main
  0.00      0.08     0.00        1     0.00     0.00  Gen_bins
  0.00      0.08     0.00        1     0.00     0.00  Get_args

GPROF says that 75% of the time is spent in Which_bin, for 0.06 seconds.
Using our Twall, we measured .086 seconds
```

Which approach is correct? GNU Profile: `https://www.cs.utah.edu/dept/old/texinfo/as/gprof.html`

Statistical Methods

Run times on any computer are not reproducible, hence, it is important to analyze the distribution of the code run times.

- Standard statistical variables used to describe the distribution of the data include:
  - Max/Min (maximum/minium values)
  - Mean (average value)
  - Median (central value)
  - Variance (variance)
  - StandardDeviation (σ) of the timings.
- To test your codes:
  - Run and time critical blocks
  - Vary key parameters (packet or problem sizes, number of processors, etc.).
  - Calculate the statistics at run-time.
  - Summarize in a table
- Refs:
  `http://reference.wolfram.com/language/tutorial/BasicStatistics.html`
  `http://edl.nova.edu/secure/stats/`

Estimating the performance of the student cluster

As mentioned above, we can estimate the performance of the cluster using our timing data to solve the following equation:

$$FLOPS \simeq Total\ Number\ of\ Operations / Total\ Time$$

For the histogram program, the function *Whichbin* dominates the run-time, so we will use this function to estimate the FLOPS on tuckoo:

$$TotalOps_{WhichBin} = (\#Ops\ in\ WhichBin) * (\#Calls\ to\ WhichBin)$$

Analysis of the function *Whichbin*, shows that the number of operations is of order $\vartheta(N) = \vartheta(10)$. The number of calls is determined by *data_count*. For *data_count* $= 10^6$ elements, we measured the time spent in *WhichBin* to be $T_{whichbin} = 7.18x10^{-2}$ seconds. We estimate that the FLOPS for the histogram program to be

$$FLOPS_{measured} \simeq \frac{\vartheta(N) * data\_count}{T_{whichbin}} \simeq \frac{10 * 10^6}{7.18x10^{-2}} \simeq 1.4x10^8\ FLOPS$$

This is less than the theoretical performance we calculated earlier: $FLOPS_{theoretical} \simeq 6.4\ GFLOPS$.

Summarizing the Timing Data in a Table

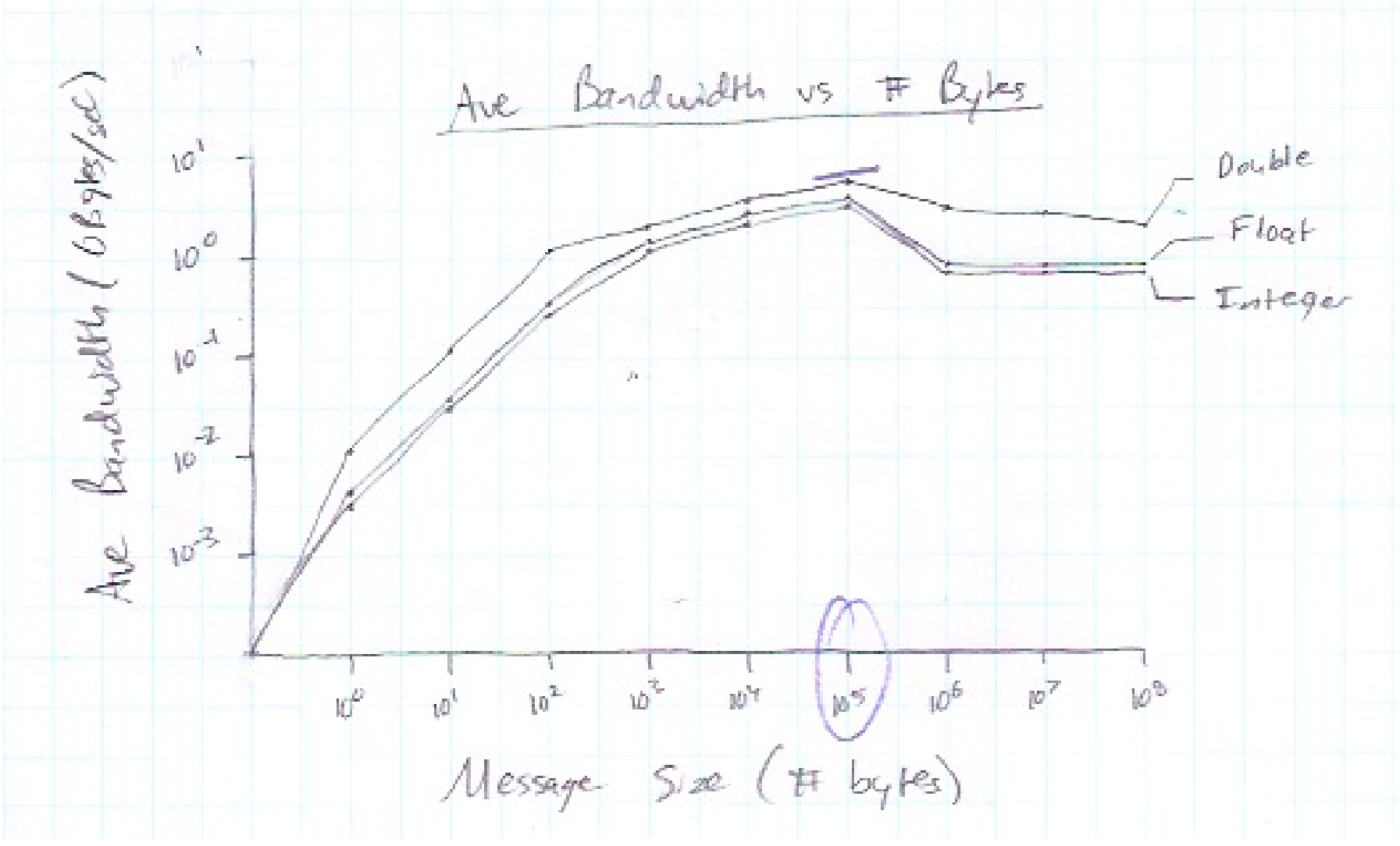
bin_count	data_count	T_wall	T_getargs	T_mem	T_gendat	T_genbins	T_whichbin
10	1.00E+00	1.34E-04	1.80E-05	1.10E-04	5.00E-06	0.00E+00	1.00E-06
10	1.00E+01	1.44E-04	2.70E-05	1.10E-04	5.00E-06	0.00E+00	2.00E-06
10	1.00E+02	1.52E-04	1.80E-05	1.17E-04	7.00E-06	0.00E+00	1.00E-05
10	1.00E+03	2.52E-04	1.60E-05	1.19E-04	2.60E-05	0.00E+00	9.10E-05
10	1.00E+04	1.29E-03	1.70E-05	1.21E-04	2.40E-04	0.00E+00	9.13E-04
10	1.00E+05	9.60E-03	1.60E-05	1.19E-04	2.18E-03	0.00E+00	7.29E-03
10	1.00E+06	9.26E-02	1.80E-05	1.13E-04	2.08E-02	0.00E+00	7.18E-02
10	1.00E+07	9.00E-01	1.80E-05	1.21E-04	1.83E-01	1.00E-06	7.18E-01

Summarizing the Timing Data in a Table: Using Statistics

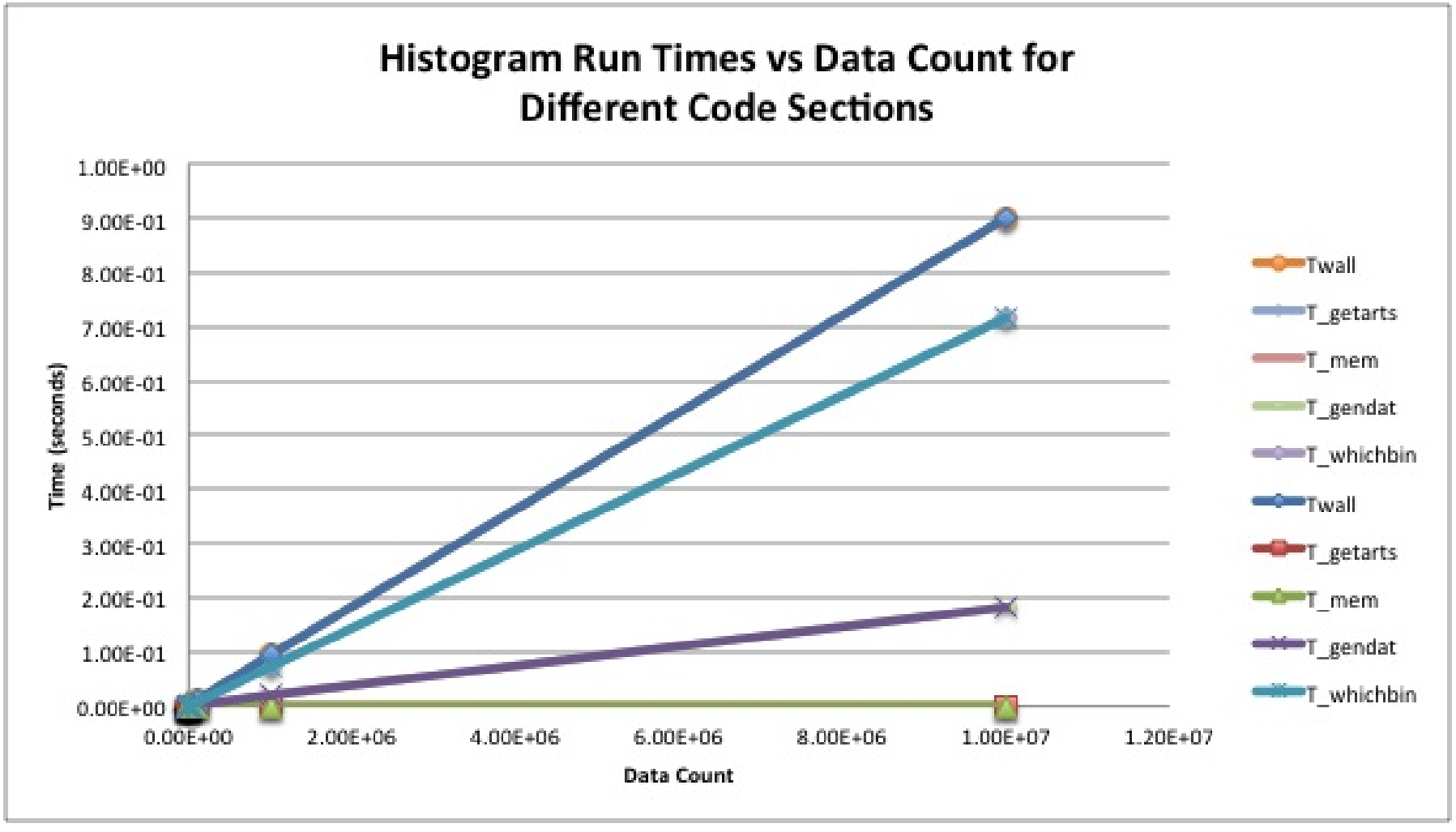
Double	Data Size	BW	Min	Max	Mean	Median	Variance	Standard deviation
	1,E+00	1,458254E+06	2,038479E-06	3,522754E-04	1,097202E-05	5,137920E-06	1,397227E-09	3,737949E-05
	1,E+01	9,906830E+06	3,945827E-06	2,325773E-05	5,178452E-06	5,137920E-06	3,636904E-12	1,907067E-06
	1,E+02	6,500278E+07	2,276897E-06	2,611876E-05	8,463860E-06	7,998943E-06	8,945307E-12	2,990871E-06
	1,E+03	3,326997E+08	1,395941E-05	4,113913E-05	2,347708E-05	2,409220E-05	6,910077E-11	8,312687E-06
	1,E+04	7,664854E+08	1,560569E-04	3,103137E-04	1,606536E-04	1,582026E-04	2,327512E-10	1,525618E-05
	1,E+05	9,686726E+08	1,433980E-03	1,631153E-03	1,443000E-03	1,441133E-03	3,653623E-10	1,911445E-05
	1,E+06	1,008201E+09	1,419009E-02	1,552309E-02	1,421810E-02	1,420463E-02	1,724856E-08	1,313338E-04

Source: J. Ayoub, CS596, Spring 2014

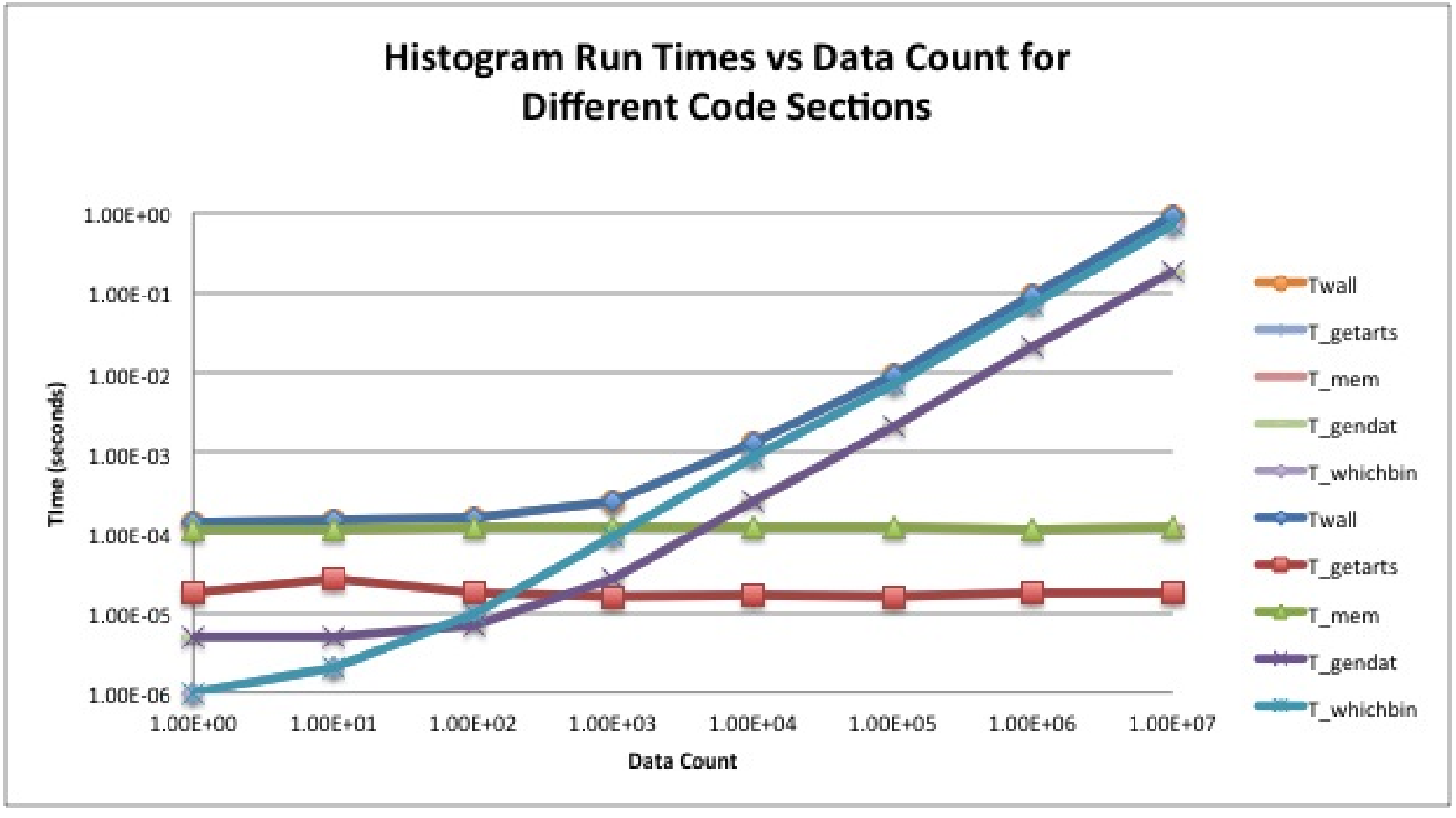
Plotting Results - Family of Curves (log-log)



Plotting Results - Family of Curves



Plotting Results - Family of Curves (log-log)



Timing Example: UCOAM Model

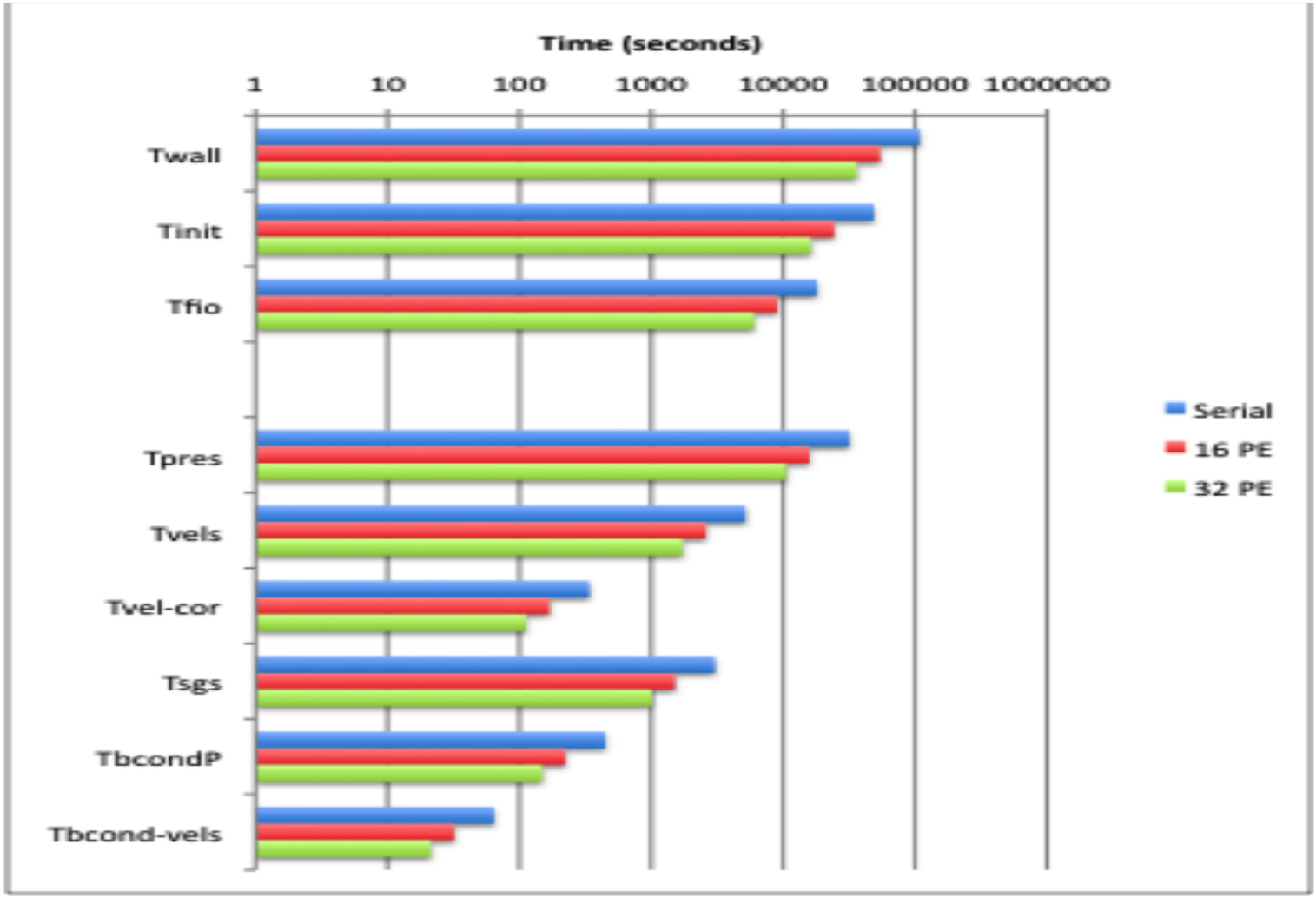
TABLE II. TIME SPENT IN MAIN SECTIONS OF THE SERIAL AND PARALLEL MODELS (16 AND 32 PROCESSOR ELEMENTS)

Section	Serial	16 Processors	32 Processors
Tinit	48571	24285	16190
Tloop	59451	29725	19817
Twall	108083	54041	36027

TABLE III. TIME SPENT IN DIFFERENT SUBMODULES EXECUTED DURING THE MAIN ITERATION LOOP

Section	Serial	16 Processors	32 Processors
Tpres	31619	15810	10540
Tfio	17961	8981	5987
Tsgs	3026	1513	1009
TVelw	1736	868	579
TVelu	1726	863	575
TVelv	1716	858	572
TbcondP	448	224	150
TvelcorV	120	61	40
TvelcorW	110	55	36
TvelcorU	109	54	367
TbcondW	22	11	7
TbcondU	22	11	7
TbcondV	20	11	67
Tloop (meas)	58635	29317	19545

Timing Code


Time (seconds)
1
10
100
1000
10000
100000
1000000
Twall
Tinit
Tfio
Tpres
Tvels
Tvel-cor
Tsgs
TbcondP
Tbcond-vels
Serial
16 PE
32 PE