:Student Cluster Competition Lectures
Topic: Performance

Mary Thomas

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

Last Update: 09/27/17

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?

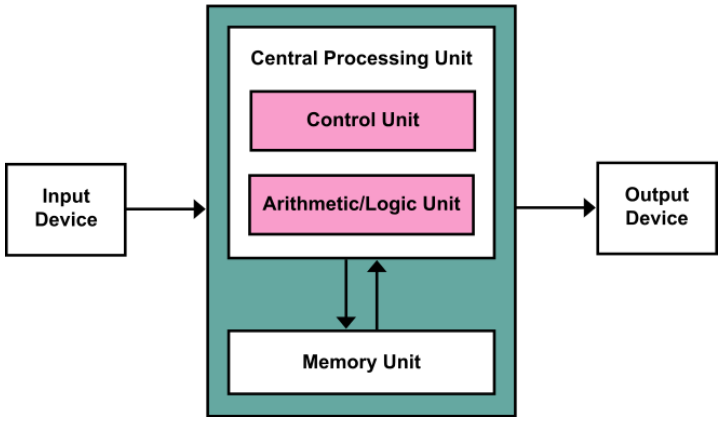
In general, performance is measured not calculated

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

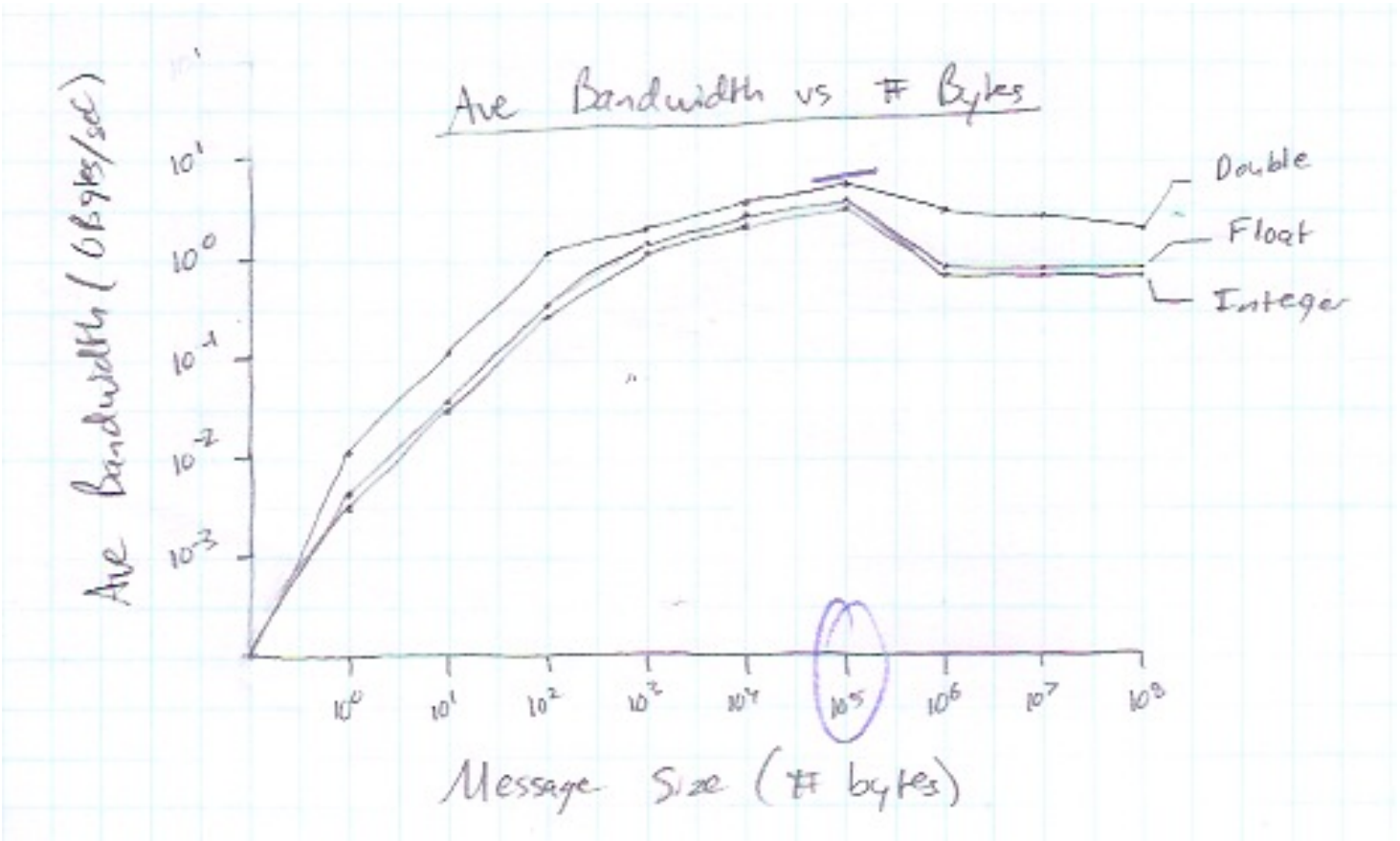
bin_count	data_count	T_wall	T_getargs	T_mem	T_gendat	T_genbins	T_whichbin
10	1.00E+00	1.34E-04	1.80E-05	1.10E-04	5.00E-06	0.00E+00	1.00E-06
10	1.00E+01	1.44E-04	2.70E-05	1.10E-04	5.00E-06	0.00E+00	2.00E-06
10	1.00E+02	1.52E-04	1.80E-05	1.17E-04	7.00E-06	0.00E+00	1.00E-05
10	1.00E+03	2.52E-04	1.60E-05	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

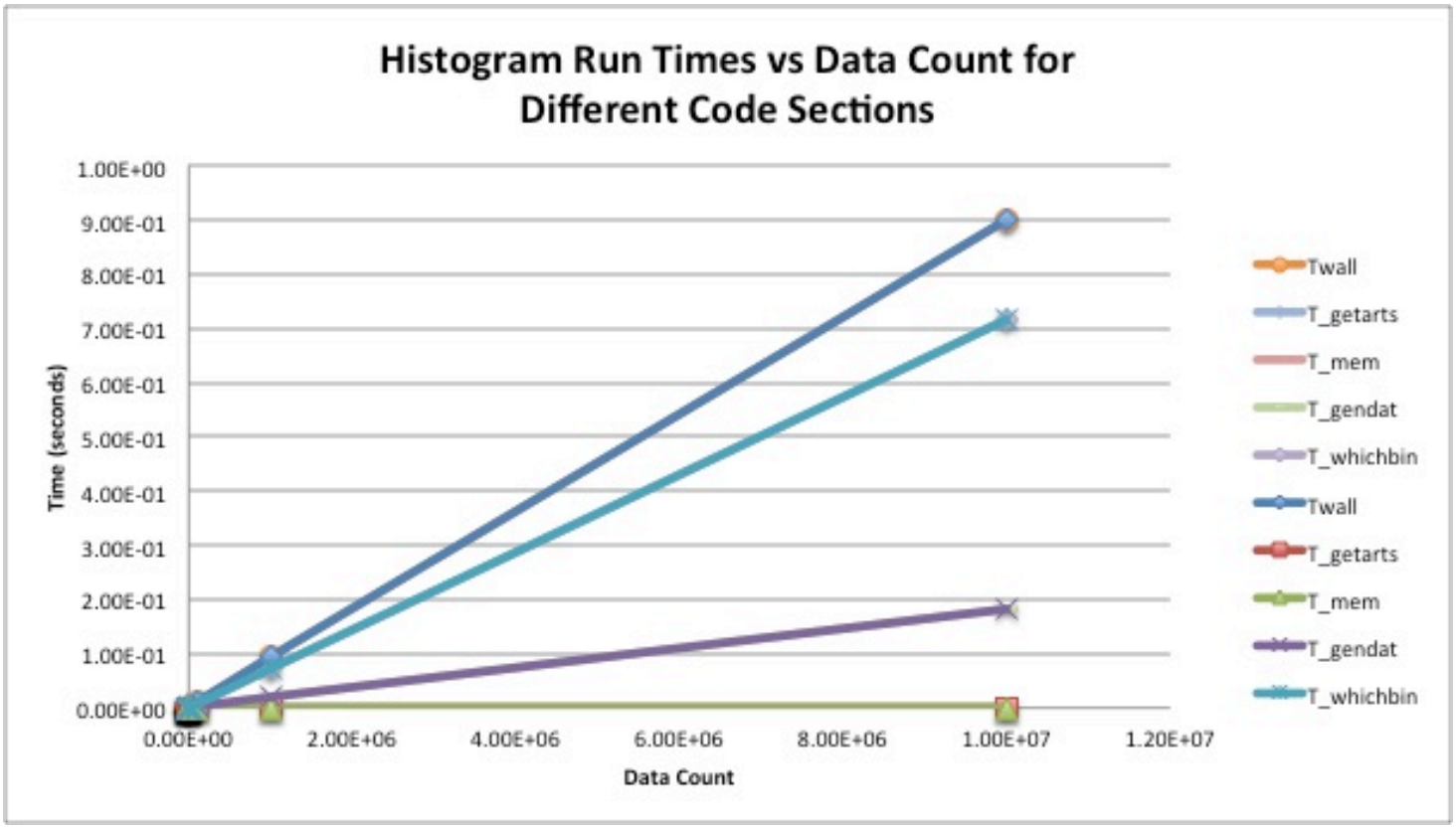
Double	Data Size	BW	Min	Max	Mean	Median	Variance	Standard deviation
	1,E+00	1,458254E+06	2,038479E-06	3,522754E-04	1,097202E-05	5,137920E-06	1,397227E-09	3,737949E-05
	1,E+01	9,906830E+06	3,945827E-06	2,325773E-05	5,178452E-06	5,137920E-06	3,636904E-12	1,907067E-06
	1,E+02	6,500278E+07	2,276897E-06	2,611876E-05	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

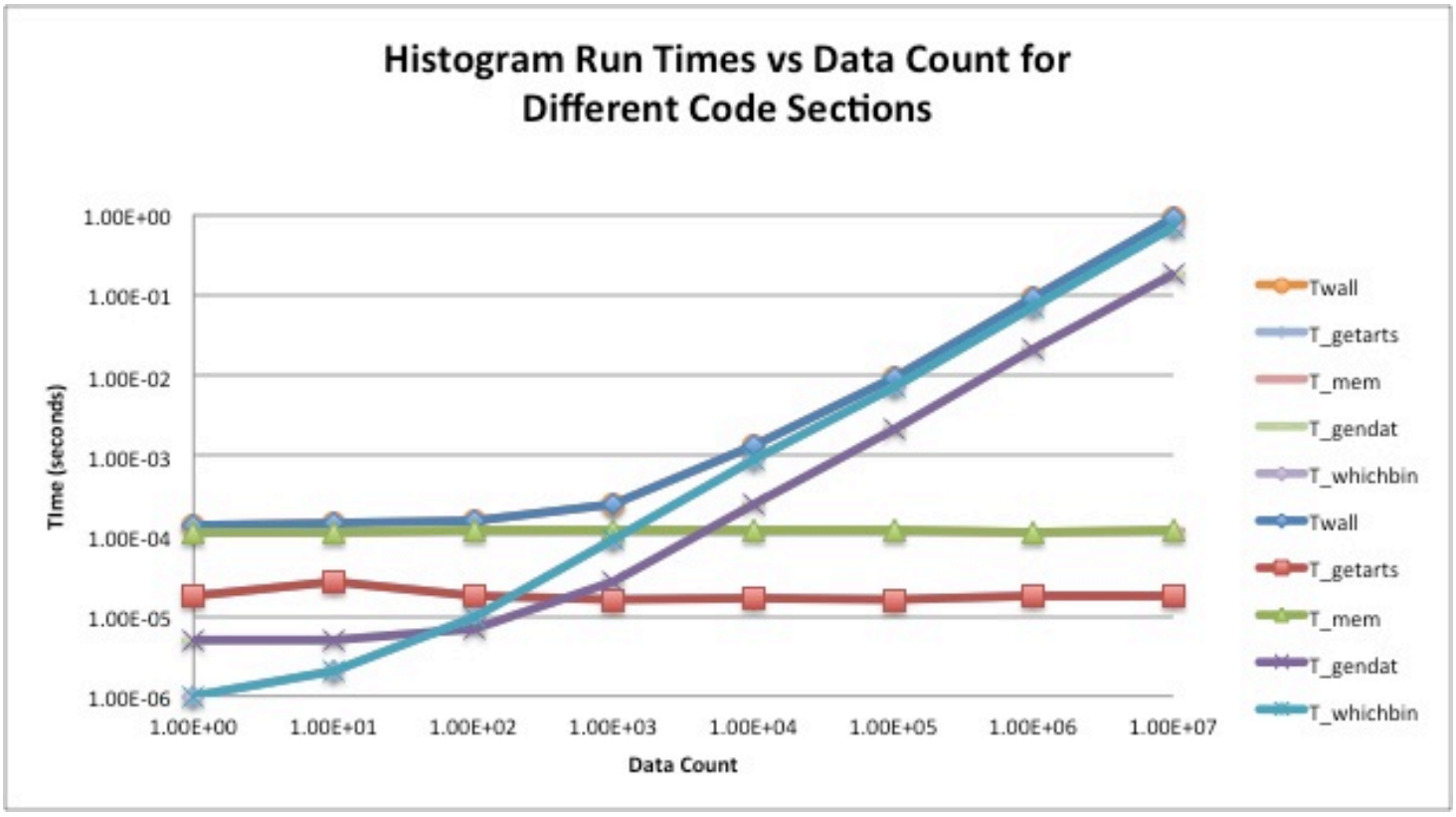
Plotting Results - Family of Curves (log-log)



Plotting Results - Family of Curves



Plotting Results - Family of Curves (log-log)



Timing Example: UCOAM Model

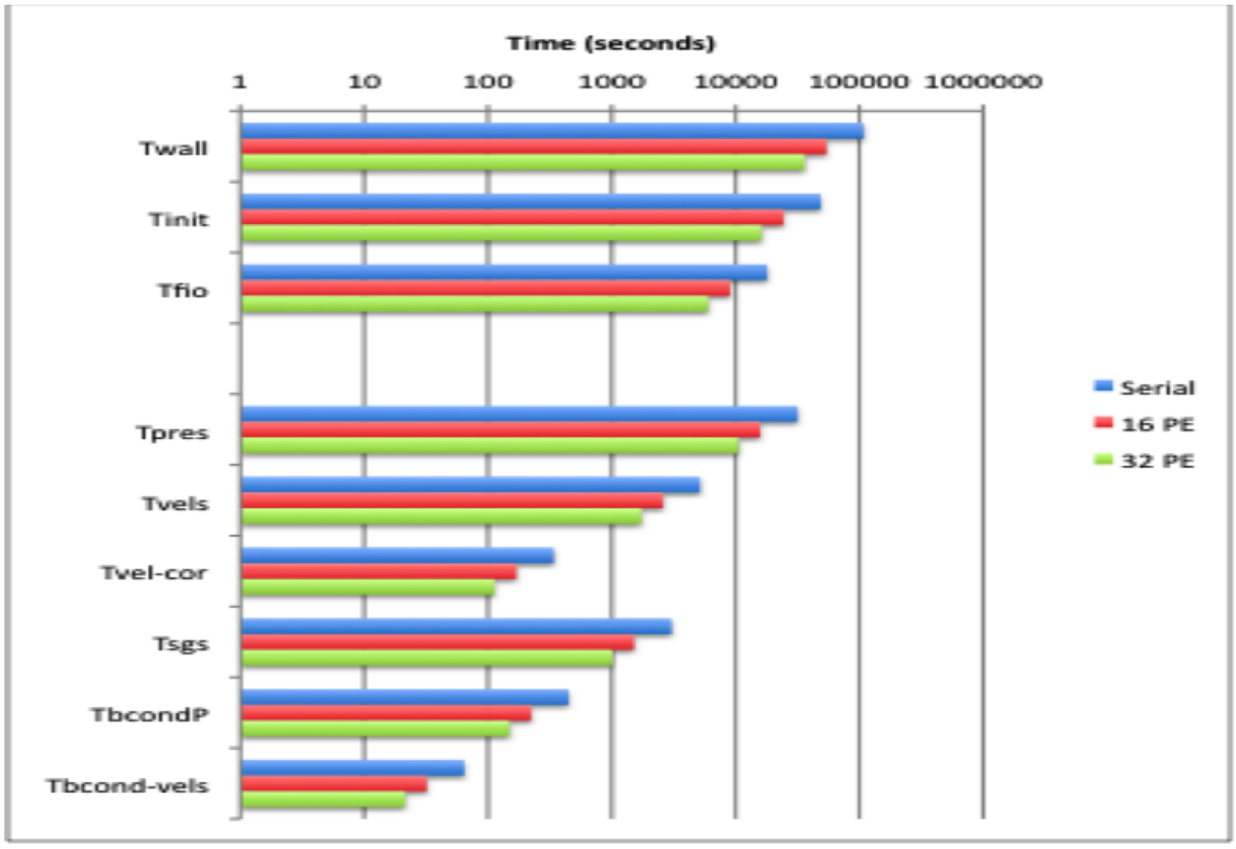
TABLE II. TIME SPENT IN MAIN SECTIONS OF THE SERIAL AND PARALLEL MODELS (16 AND 32 PROCESSOR ELEMENTS)

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

Timing Serial or Parallel Code

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

In general, performance is measured not calculated

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

Where to time the code?

- 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}

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.

Total Parallel Program Time

- The total parallel program run time is a function of a large number of variables: number of processing elements (PEs); communication; hardware (cpu, memory, software, network), and the program being run (algorithm, problem size, # Tasks, complexity, data distribution); parallel libraries:

$$T = \mathcal{F}\left(PEs, N, \ Tasks, \ I/O, \ Communication, \dots\right)$$

- The execution time required to run a problem of size N on processor i, is a function of the time spent in different parts of the program (computation, communication, I/O, idle):

$$T^i = T^i_{comp} + T^i_{comm} + T^i_{io} + T^i_{idle}$$

- The total time is the sum of the times over all processes averaged over the number of the processors: $T =$

$$\frac{1}{p}\left(\sum_{i=0}^{p-1} T_{comp} + \sum_{i=0}^{p-1} T_{comm} + \sum_{i=0}^{p-1} T_{io} + \sum_{i=0}^{p-1} T_{idel}\right)$$

Speedup

- Refers to how much faster the parallel algorithm runs than a corresponding sequential algorithm (non-MPI).
- T_{ser} = time between when *serial* program begins to when it completes its tasks.
- T_{par} = time between when *first* processor begins execution to when the *last* processor completes its tasks.
- The Speedup is defined to be: $S_p = \frac{T_{ser}}{T_{par}}$
- Where:
  - $p \equiv$ number of cores (processors, PE's)
  - $T_{ser} \equiv$ serial execution time
  - $T_{par} \equiv$ parallel execution time
- Linear speedup, or ideal speedup, is obtained when $S_p = p$, or

$$T_{par} = T_{ser} \,/\, p$$

Efficiency

- Estimation of how well the processors are used to solve the problem vs. effort is wasted in communication and synchronization.
- T_{elap} == time between when first processor begins execution to when the last processor completes its tasks

$$E = \frac{S}{p} = \frac{\left(\frac{T_{serial}}{T_{parallel}}\right)}{p} = \frac{T_{serial}}{p \cdot T_{parallel}}$$

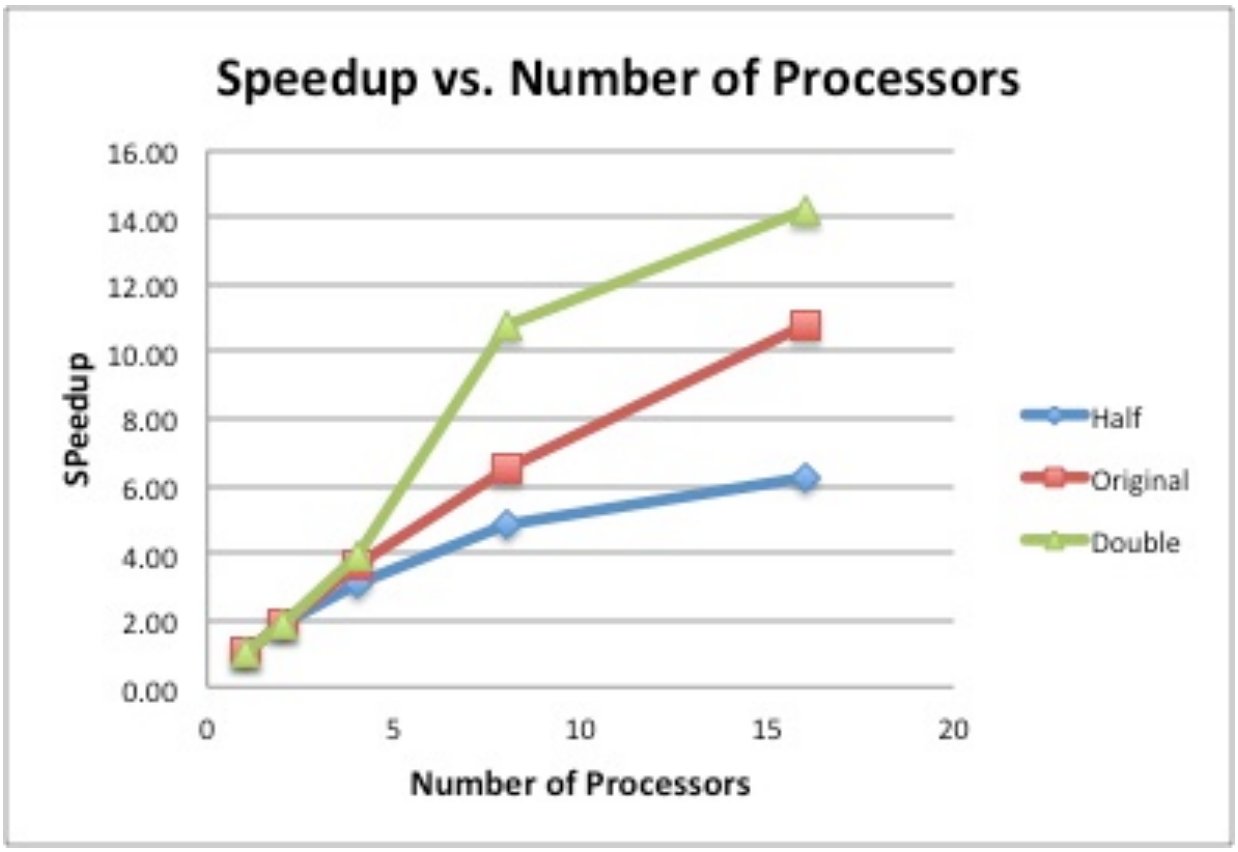
- Where:
  - p == number of cores (processors, PE's)
  - T_{ser} == serial execution time
  - T_{par} == parallel execution time
- Efficiency is ypically between *zero* and *one*

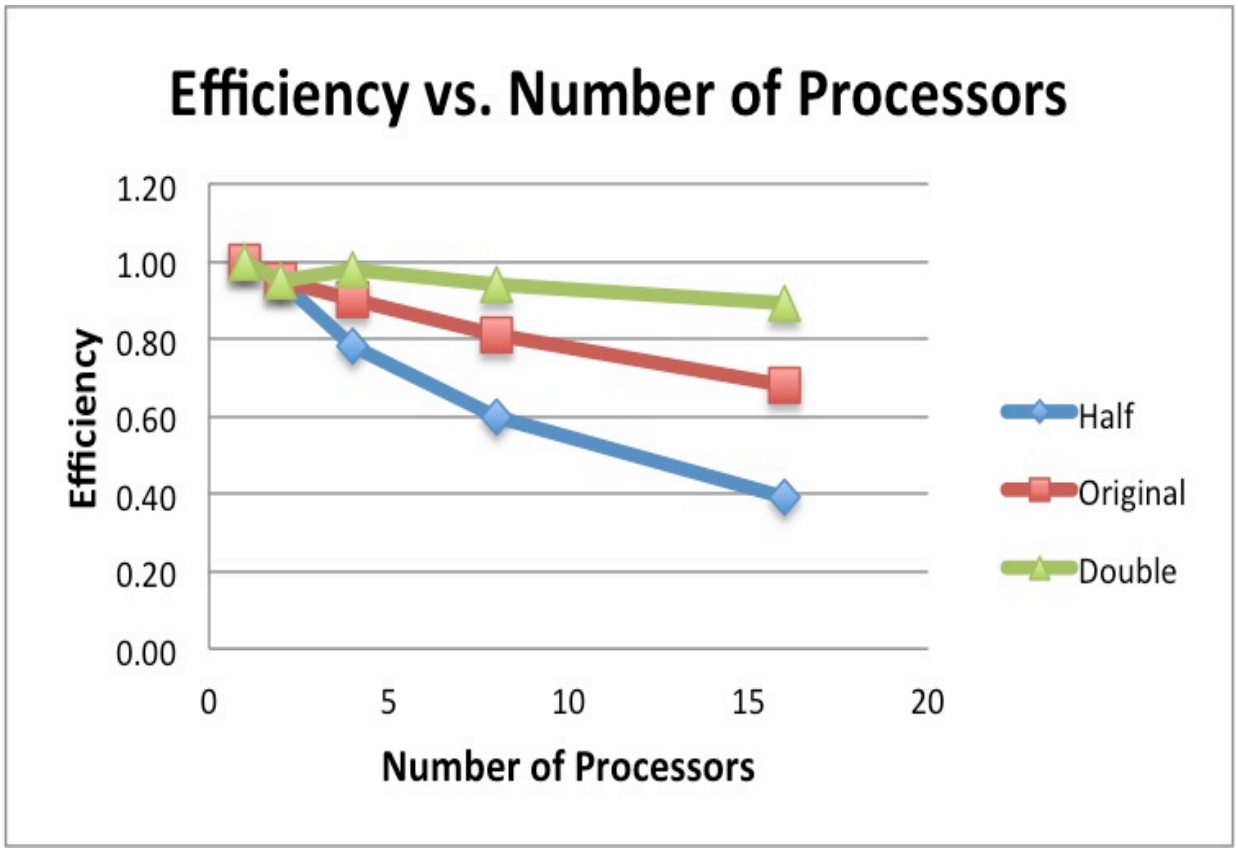
RunTimes	p	1	2	4	8	16
Half	RT	1.00	0.53	0.32	0.21	0.16
Original	RT	1.00	0.53	0.28	0.15	0.09
Double	RT	1.00	0.53	0.26	0.09	0.07

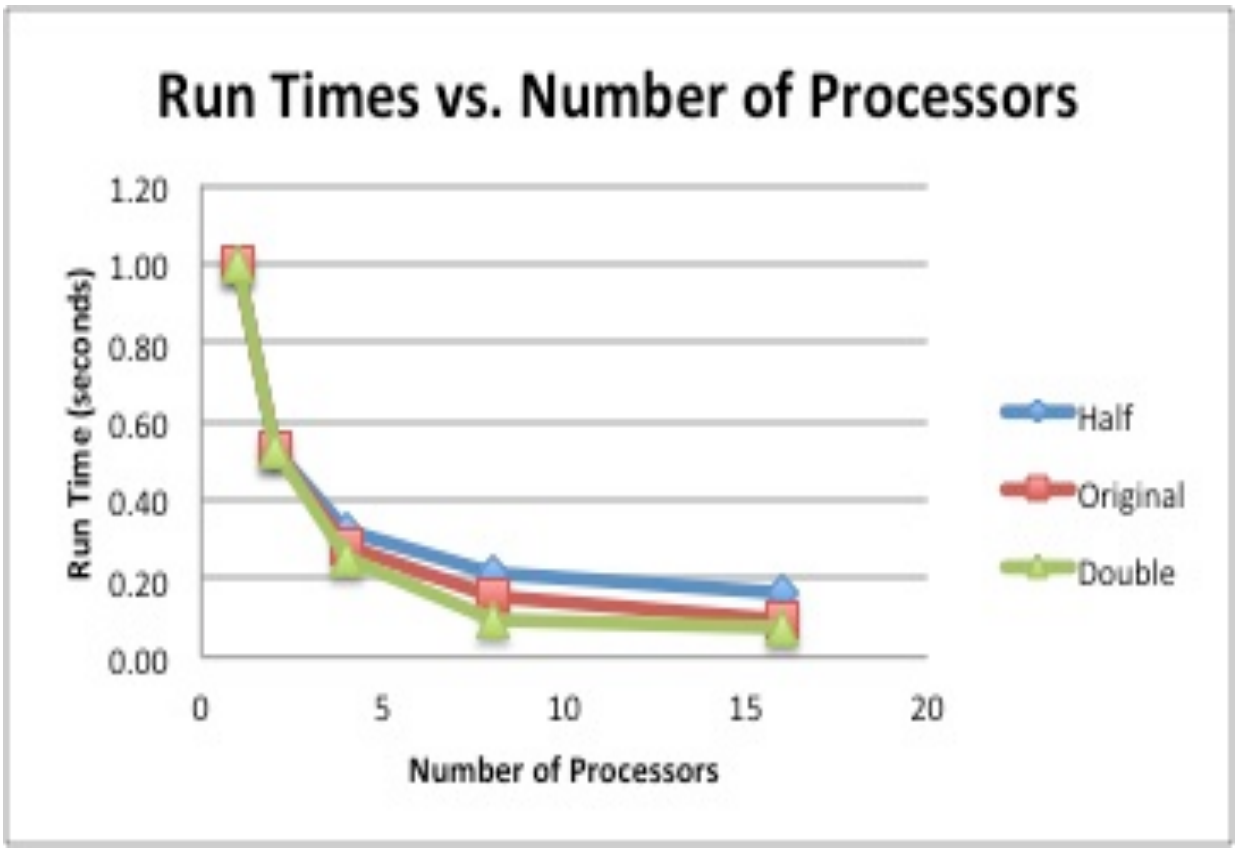
ProbSize	p	1	2	4	8	16
Half	S	1.00	1.90	3.10	4.80	6.20
	E	1.00	0.95	0.78	0.60	0.39
Original	S	1.00	1.90	3.60	6.50	10.80
	E	1.00	0.95	0.90	0.81	0.68
Double	S	1.00	1.90	3.90	10.80	14.20
	E	1.00	0.95	0.98	0.94	0.89

Test data from showing the effect of problem size on the run times (RT), speedup (S) and efficiency (E).

Source: Pacheco IPP (Ch 2)


Speedup vs. Number of Processors
SPeedup
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
0
5
10
15
20
Number of Processors
Half
Original
Double


Efficiency vs. Number of Processors
Efficiency
1.20
1.00
0.80
0.60
0.40
0.20
0.00
0
5
10
15
20
Number of Processors
Half
Original
Double


Run Times vs. Number of Processors
Run Time (seconds)
1.20
1.00
0.80
0.60
0.40
0.20
0.00
0
5
10
15
20
Number of Processors
Half
Original
Double

Effect of Overhead

- Overhead is associated with work done by program and system on non-computational activities
- These include process management, backend communications, page swapping and data access control, security, etc.

$$T_{par} = \frac{T_{ser}}{p} + T_{overhead}$$

Amdahl's Law

- Used to find the maximum expected improvement to an overall system when only part of the system is improved.
- Often used in parallel computing to predict the theoretical maximum speedup using multiple processors.

Definition: If B is the fraction of the algorithm that is strictly serial, and p is the number of processes (cores, threads, etc.), then the time $T(n)$ required for a program to execute can be written as:

$$\begin{aligned} T(n) &= T_{ser} + T_{par} \\ &= T(1)B + \frac{T(1)}{n}(1-B) \\ &= T(1)\left(B + \frac{1}{n}(1-B)\right) \end{aligned}$$

Example

- We can parallelize 90% of a serial program.
- Parallelization is "perfect" regardless of the number of cores *p* we use.
- T_{serial} = 20 seconds
- Runtime of parallelizable part is

$$0.9 \times T_{serial} / p = 18 / p$$

Example (cont.)

- Runtime of "unparallelizable" part is

$$0.1 \times T_{serial} = 2$$

- Overall parallel run-time is

$$T_{parallel} = 0.9 \times T_{serial} / p + 0.1 \times T_{serial} = 18 / p + 2$$

Example (cont.)

- Speed up

$$S = \frac{T_{serial}}{0.9 \times T_{serial} / p + 0.1 \times T_{serial}} = \frac{20}{18 / p + 2}$$

Scalability

- In general, a problem is *scalable* if it can handle ever increasing problem sizes.
- If we increase the number of processes/threads and keep the efficiency fixed without increasing problem size, the problem is *strongly scalable*.
- If we keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes/threads, the problem is *weakly scalable*.

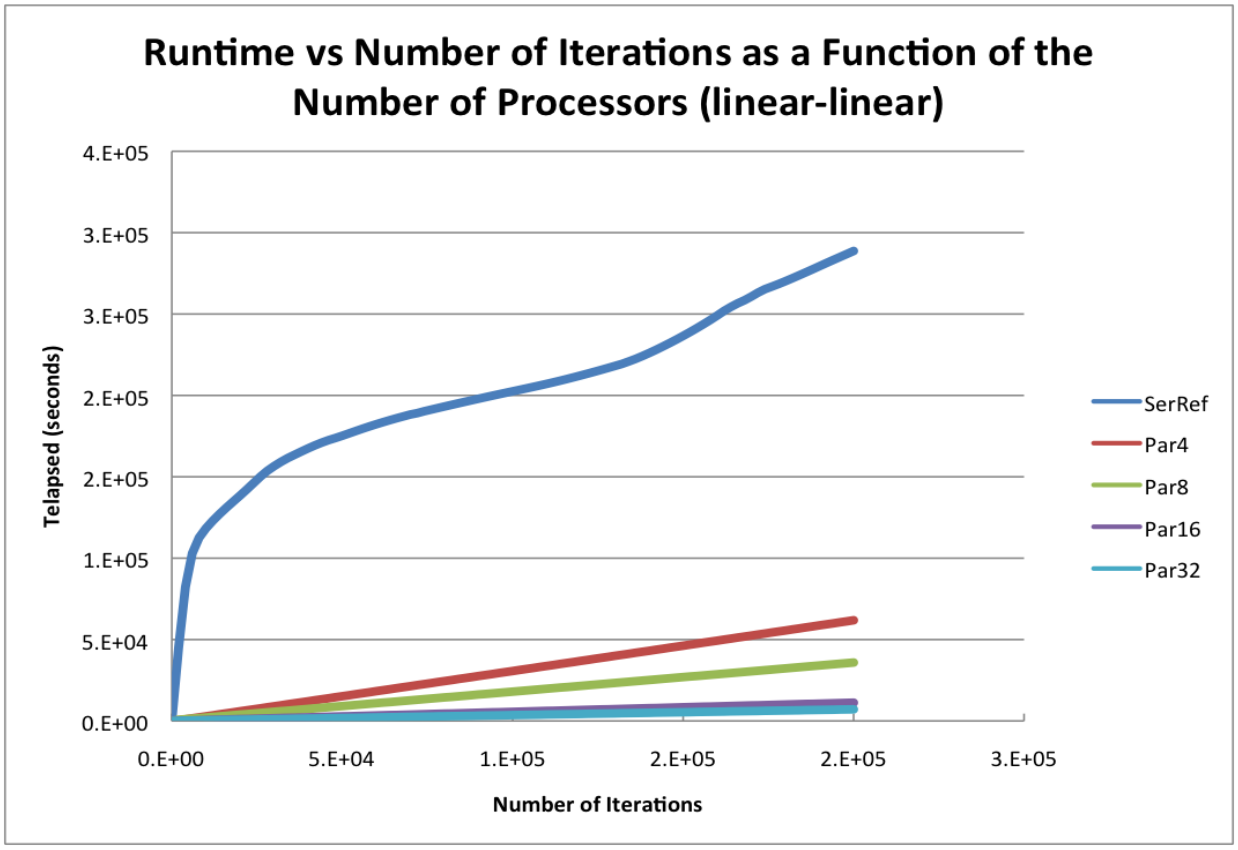
Customized Timings: Parallel Framework

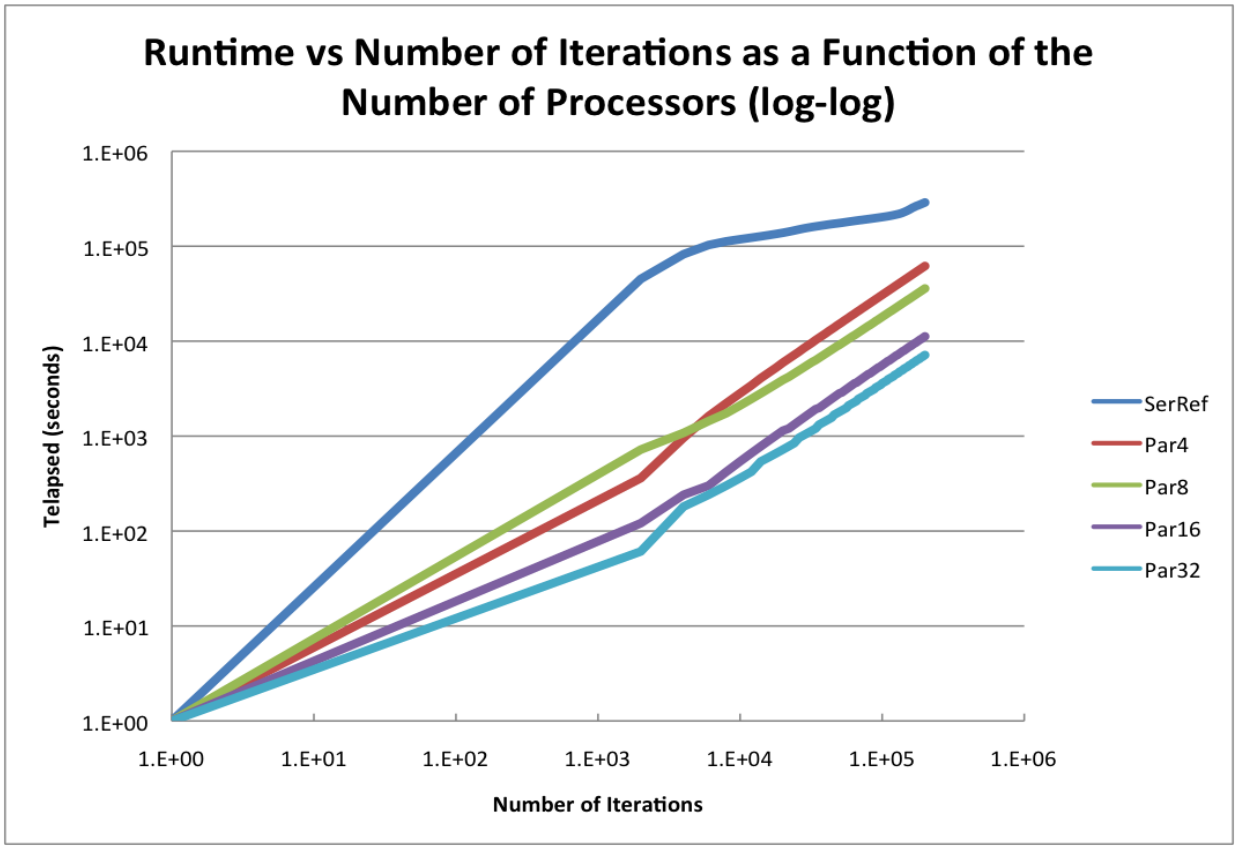
TABLE II. TIME SPENT IN MAIN SECTIONS OF THE SERIAL AND PARALLEL MODELS (16 AND 32 PROCESSOR ELEMENTS)

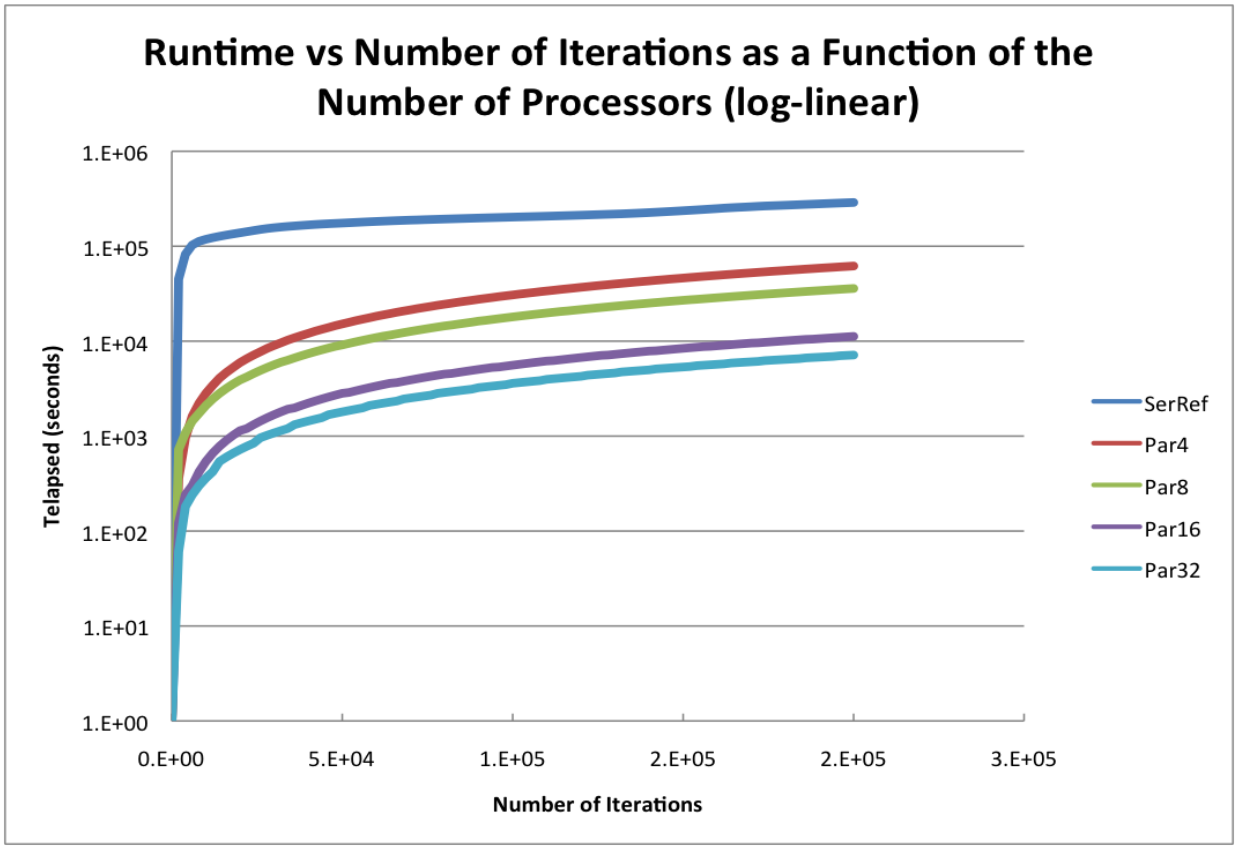
Section	Serial	16 Processors	32 Processors
Tinit	48571	24285	16190
Tloop	59451	29725	19817
Twall	108083	54041	36027

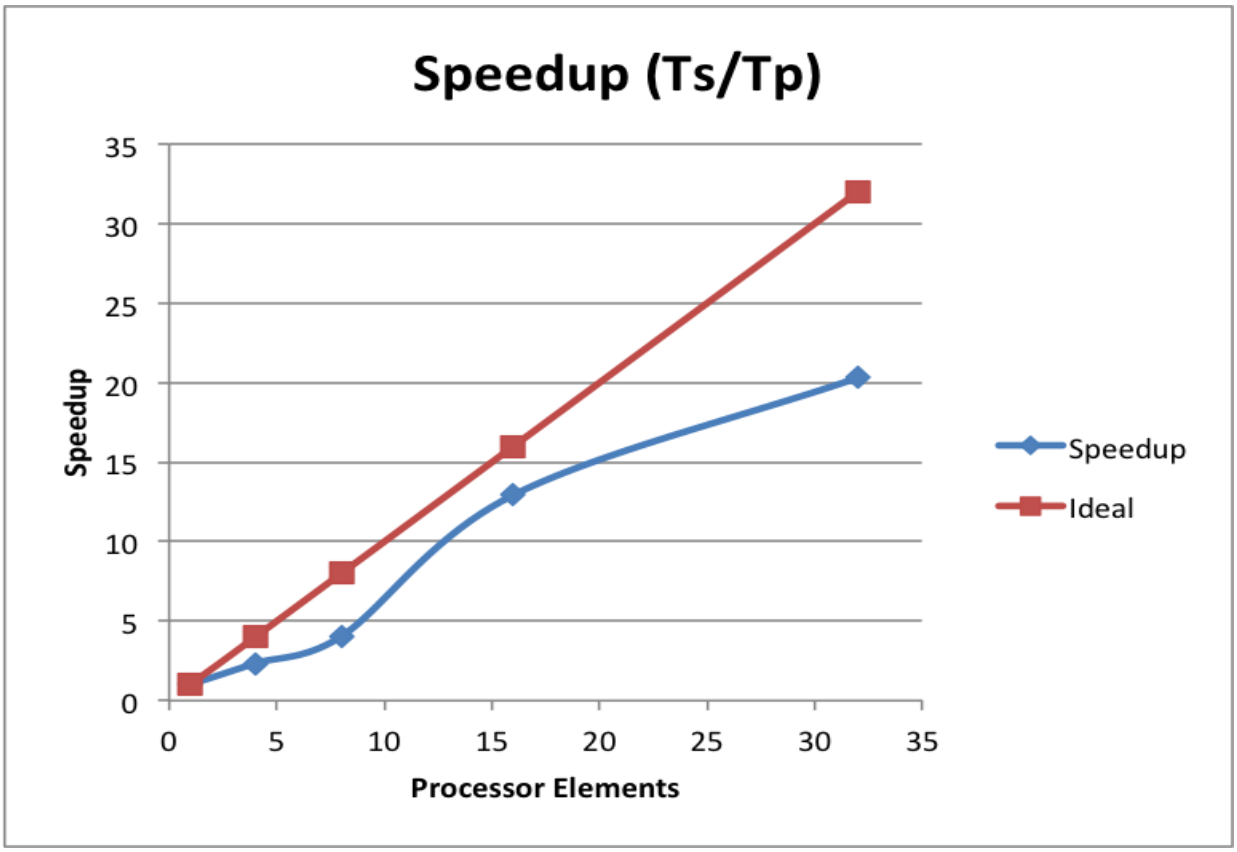
TABLE III. TIME SPENT IN DIFFERENT SUBMODULES EXECUTED DURING THE MAIN ITERATION LOOP

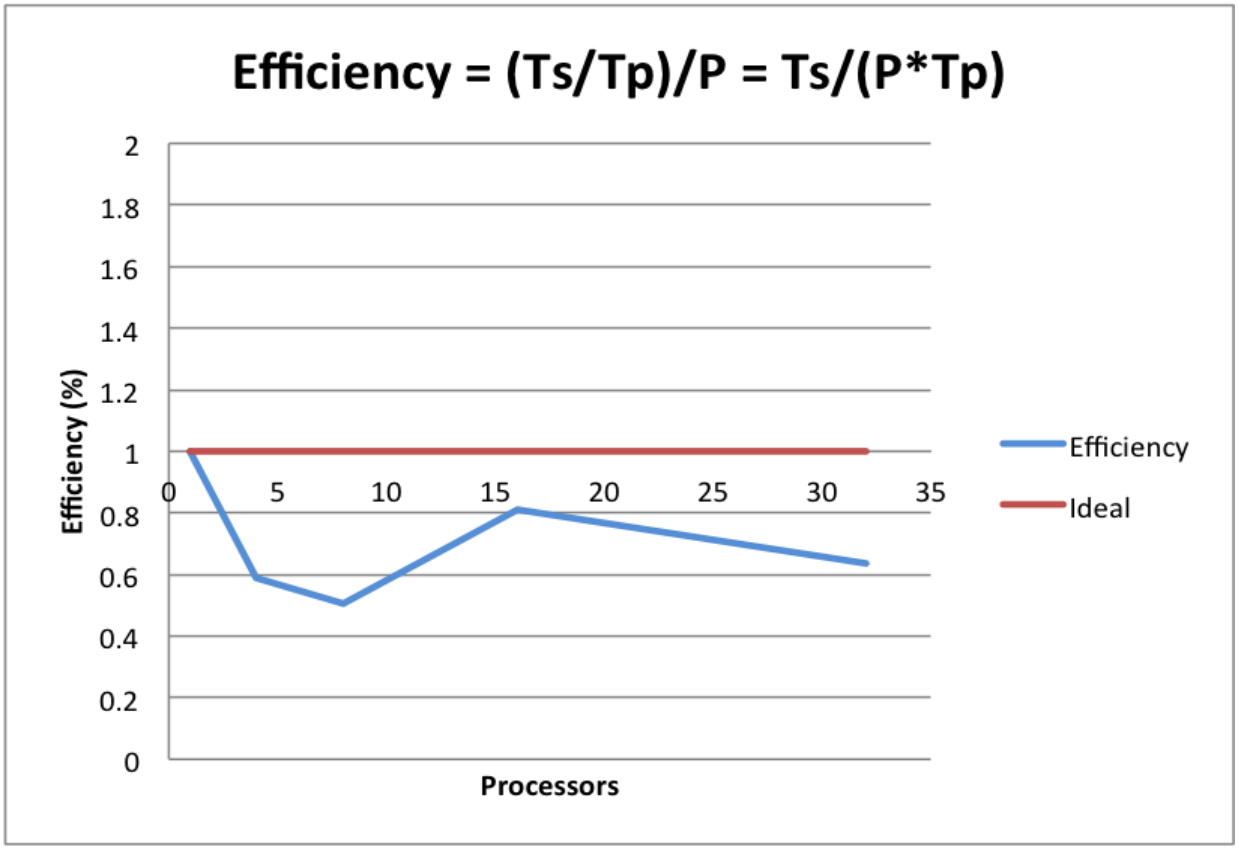
Section	Serial	16 Processors	32 Processors
Tpres	31619	15810	10540
Tfio	17961	8981	5987
Tsgs	3026	1513	1009
TVelw	1736	868	579
TVelu	1726	863	575
TVelv	1716	858	572
TbcondP	448	224	150
TvelcorV	120	61	40
TvelcorW	110	55	36
TvelcorU	109	54	367
TbcondW	22	11	7
TbcondU	22	11	7
TbcondV	20	11	67
Tloop (meas)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


Runtime vs Number of Iterations as a Function of the Number of Processors (linear-linear)
Telapsed (seconds)
4.E+05
3.E+05
3.E+05
2.E+05
2.E+05
1.E+05
5.E+04
0.E+00
0.E+00
5.E+04
1.E+05
2.E+05
2.E+05
3.E+05
Number of Iterations
SerRef
Par4
Par8
Par16
Par32


Runtime vs Number of Iterations as a Function of the Number of Processors (log-log)
Telapsed (seconds)
1.E+06
1.E+05
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
1.E+00
1.E+01
1.E+02
1.E+03
1.E+04
1.E+05
1.E+06
Number of Iterations
SerRef
Par4
Par8
Par16
Par32


Runtime vs Number of Iterations as a Function of the Number of Processors (log-linear)
Telapsed (seconds)
1.E+06
1.E+05
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
0.E+00
5.E+04
1.E+05
2.E+05
2.E+05
3.E+05
Number of Iterations
SerRef
Par4
Par8
Par16
Par32


Speedup (Ts/Tp)
Speedup
Processor Elements
0
5
10
15
20
25
30
35
Speedup
Ideal


Efficiency = (Ts/Tp)/P = Ts/(P*Tp)
Efficiency (%)
2
1.8
1.6
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
30
35
Processors
Efficiency
Ideal

```
/* hello.c by James Otto, 1/31/11
-- for running serial processes
   on a cluster... see batch.hello */
#include <stdio.h>
#include <unistd.h>
int main(void)
{
  char cptr[100];
  gethostname(cptr,100);
  printf("hello, world from %s\n", cptr);
  return 0;
}
```

COMPILE & RUN SERIAL PGM

```
[tuckoo]$ mpicc -o hello hello.c
[mthomas@tuckoo ex.2014]$ mpirun -np 5 ./hello
hello, world from tuckoo
hello, world from tuckoo
hello, world from tuckoo
hello, world from tuckoo
hello, world from tuckoo
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mpi.h"
int main (int argc, char* argv[])
{
  int rank, nprocs, ierr, i, error=0;
  MPI_Status status;

  ierr = MPI_Init(&argc, &argv);
  if (ierr != MPI_SUCCESS) {
    printf("MPI initialization error\n");}

  // processing element ID
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // ID of communicator connecting PE's
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  printf("Hello Processor: rank:
        %d, nprocs: %d\n", rank, nprocs);

  MPI_Finalize();
  return 0;
}
```

COMPILE & RUN PARALLEL PGM

```
[tuckoo]$ mpicc -o hello_mpi hello_mpi.c
[tuckoo]$ mpirun -np 5 ./hello_mpi
Hello Processor: rank: 0, nprocs: 5
Hello Processor: rank: 1, nprocs: 5
Hello Processor: rank: 3, nprocs: 5
Hello Processor: rank: 4, nprocs: 5
Hello Processor: rank: 2, nprocs: 5
```

Looptest demonstrates way to measure time app spends in subroutines

```
program looptest
! fortran 90 source code
implicit none
integer, parameter :: max=10000
integer            :: i, j
double precision   :: tws,twe, ts, te,
     a(max,max), x(max), y(max)
call cpu_time(tws)
!------ initialize arrays
a=0.0; x=0.0; y=0.0
do i=1,max
    x(i) = i;    y(i) = max-i
    do j=1,max
        a(i,j) = 10*j + i
enddo;   enddo
! -----compute  loop1
call cpu_time(ts)
call loop1(y,max)
call cpu_time(te)
print *,"Telap: loop 1 = ", (te - ts)
!------ compute  loop2
ts=0.0; te=0.0;
call cpu_time(ts)
call loop2(y,max)
call cpu_time(te)
print *,"Telap: loop 2 = ", (te - ts)
!------ compute  loop3
ts=0.0; te=0.0;
call cpu_time(ts)
call loop3(y,max)
call cpu_time(te)
print *,"Telap: loop 3 = ", (te - ts)
call cpu_time(twe)
print *,"Wallclock Time:  = ", (twe - tws)
```

```
contains
subroutine loop1(yloc,maxloc)
integer :: maxloc
double precision   :: yloc(maxloc)
do i=1,maxloc
    do j=1,maxloc
        yloc(i) = a(i,j) * x(j)
    enddo
enddo
end subroutine loop1

subroutine loop2(yloc,maxloc)
integer :: maxloc
double precision   :: yloc(maxloc)
do j=1,maxloc
    do i=1,maxloc
        yloc(i) = a(i,j) * x(j)
    enddo
enddo
end subroutine loop2

subroutine loop3(yloc,maxloc)
integer :: maxloc
double precision   :: yloc(maxloc)
do i=1,maxloc
    do j=1,maxloc
        yloc(i) = a(i,j) * sqrt(x(j))
    enddo
enddo
end subroutine loop3

end program looptest
```

Compile with *gprof* option *(-p)*, and run job from command line

```
[mthomas@tuckoo]$ cat makefile
=============================
MAKE FILE
=============================
MPIF90 = mpif90
MPICC = mpicc
CC     = gcc
all: looptst looptstp
looptst: looptst.f90
        $(MPIF90) -o looptst looptst.f90

looptstp: looptst.f90
        $(MPIF90) -p -o looptstp looptst.f90

clean:
        rm -rf *.o looptst looptstp
```

```
=============================
SERIAL JOB: FROM COMMAND LINE
=============================
[mthomas@tuckoo]$ ./looptstp
 Testing FORTRAN loops (column major):
 Telap: loop 1 =      960.8539      msec
 Telap: loop 2 =      580.9109      msec
 Telap: loop 3 =     1744.7349      msec
Wallclock Time: =     5861.1099      msec

=============================
PROFILING: using -p option in make
=============================
 [mthomas@tuckoo]$ gprof looptstp gmon.out
 Flat profile:
 Each sample counts as 0.01 seconds.
  cumulative   self              self     total
  time   seconds   seconds    calls   s/call   s/call  name
  37.58      1.39     1.39        1     1.39     3.67  MAIN__
  27.04      2.40     1.00        1     1.00     1.00  frame_dummy
  23.25      3.26     0.86        1     0.86     0.86  loop1.1529
  10.95      3.67     0.41        1     0.41     0.41  loop2.1523
```

Run Serial Job In Queue

```
==================
= SUBMIT SERIAL JOB TO QUEUE
==================
[mthomas@tuckoo looptst]$ cat batch.looptstp
#!/bin/sh
#PBS -V
#PBS -l nodes=2:ppn=4:core4
#PBS -N looptstp
#PBS -joe
#PBS -q batch
cd $PBS_O_WORKDIR
NCORES=`wc -w < $PBS_NODEFILE`
echo "looptstp-test using $NCORES cores..."
mpirun -np 4 -hostfile $PBS_NODEFILE
                    --nooversubscribe ./looptstp
[mthomas@tuckoo looptst]$ !qsub
qsub batch.mpi-looptstp
16478.tuckoo.sdsu.edu
==================
= OUTPUT (asynchronous)
==================
 Telap: loop 1 =   0.84287199999
 Telap: loop 2 =   0.4549309999
 Telap: loop 2 =   0.455931
 Telap: loop 2 =   0.449931
 Telap: loop 2 =   0.455931
 Telap: loop 3 =   0.9918490
 Wallclock Time:      =    5.028235
 Telap: loop 3 =   0.99084
 Wallclock Time:      =    5.02623
```

```
Telap: loop 1 =   0.8308729
Telap: loop 1 =   0.8308739
Telap: loop 1 =   0.8328739
Telap: loop 1 =   0.8428719
-----------------------------
Telap: loop 2 =   0.4499310
Telap: loop 2 =   0.4549309
Telap: loop 2 =   0.4559310
Telap: loop 2 =   0.4559310
-----------------------------
Telap: loop 3 =   0.9898489
Telap: loop 3 =   0.9908489
Telap: loop 3 =   0.9918490
Telap: loop 3 =    1.0078469
-----------------------------
Wallclock Time:      =    5.02523599
Wallclock Time:      =    5.0262349
Wallclock Time:      =    5.02823599
Wallclock Time:      =    5.049231
```

Note: no gain by using multiple PE's $-->$ no MPI calls in code

Add MPI Calls

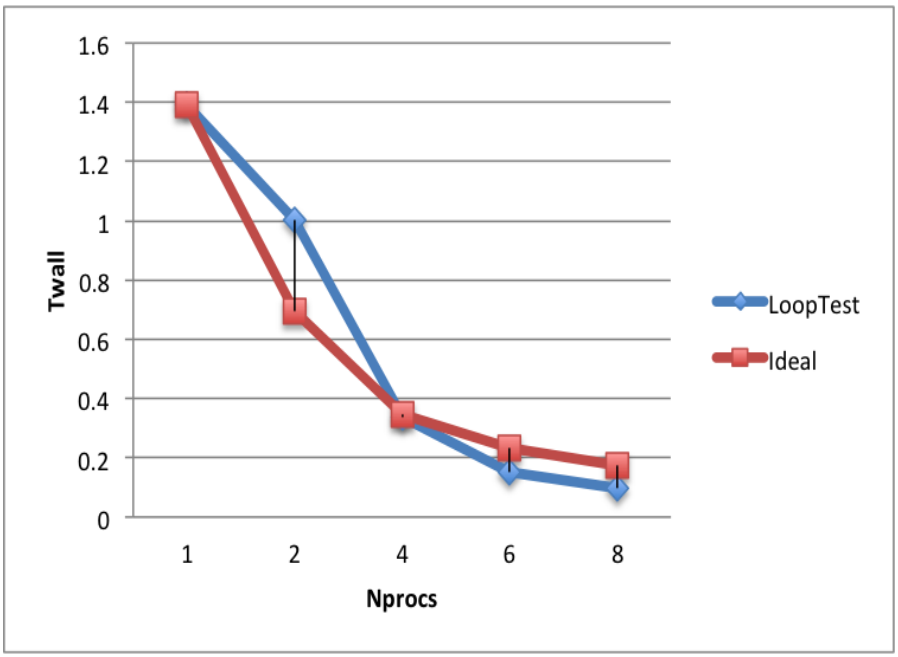
```
program looptest
!
implicit none
include "mpif.h"
integer, parameter :: max=10000
double precision, allocatable
:: a(:,:), x(:), y(:)
double precision    :: tws,twe, ts, te
integer
:: i,j, rank, nprocs, ierr, token
integer             :: status(MPI_STATUS_SIZE)

call cpu_time(tws)
call MPI_INIT(ierr)
if (ierr .ne. MPI_SUCCESS) then
    print *, "Error: initing in MPI_INIT()"
    stop
endif

!- find out how many processes \&
local process rank
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD, nprocs, ierr)

maxloc=gl_max/nprocs
allocate(a(maxloc,maxloc), x(maxloc), &
            y(maxloc), stat=ierr)

!initialize arrays
do i=1,max
    x(i) = i; y(i) = max-i
    do j=1,max
        a(i,j) = 10*j + i
    enddo
enddo

! compute loop1
call cpu_time(ts)
call loop1(y,max)
call cpu_time(te)
write( )

! compute loop2
ts=0.0; te=0.0;
call cpu_time(ts)
call loop2(y,max)
call cpu_time(te)
write( )

! compute loop3
ts=0.0; te=0.0;
call cpu_time(ts)
call loop3(y,max)
call cpu_time(te)
write( )
call cpu_time(twe)
write( )

call MPI_FINALIZE(ierr)

contains
 ..
```

Run MPI Job In Queue

```
==================
= SUBMIT JOB TO QUEUE
==================
[mthomas@tuckoo looptst]$ !qsub
qsub batch.mpi-looptstp
16478.tuckoo.sdsu.edu
==================
= OUTPUT (asynchronous)
==================
[mthomas@tuckoo looptst]$ cat mpi-looptstp.o16485
mpi-looptstp-test using 8 cores...
 LocaMAX:          2500
 LocaMAX:          2500
 LocaMAX:          2500
 LocaMAX:          2500
PE[ 0]: Telap, loop 1=       0.07698800
PE[ 3]: Telap, loop 1=       0.07698800
PE[ 2]: Telap, loop 1=       0.07598900
PE[ 2]: Telap, loop 2=       0.03799400
PE[ 1]: Telap, loop 3=       0.07998700
PE[ 1]: Telap, Twall=       0.33794800
PE[ 0]: Telap, loop 3=       0.07898800
PE[ 0]: Telap, Twall=       0.33594800
PE[ 3]: Telap, Twall=       0.34294600
PE[ 2]: Telap, loop 3=       0.07998800
PE[ 2]: Telap, Twall=       0.33294900
-----------------
PE[ 0]: Telap, loop 1=       0.07698800
PE[ 1]: Telap, loop 1=       0.07698800
PE[ 2]: Telap, loop 1=       0.07598900
PE[ 3]: Telap, loop 1=       0.07698800
-----------------
PE[ 0]: Telap, loop 2=       0.03799400
PE[ 1]: Telap, loop 2=       0.03799500
PE[ 2]: Telap, loop 2=       0.03799400
PE[ 3]: Telap, loop 2=       0.03699500
-----------------
PE[ 0]: Telap, loop 3=       0.07898800
PE[ 1]: Telap, loop 3=       0.07998700
PE[ 2]: Telap, loop 3=       0.07998800
PE[ 3]: Telap, loop 3=       0.08098700
-----------------
PE[ 0]: Telap, Twall=       0.33594800
PE[ 1]: Telap, Twall=       0.33794800
PE[ 2]: Telap, Twall=       0.33294900
PE[ 3]: Telap, Twall=       0.34294600
```

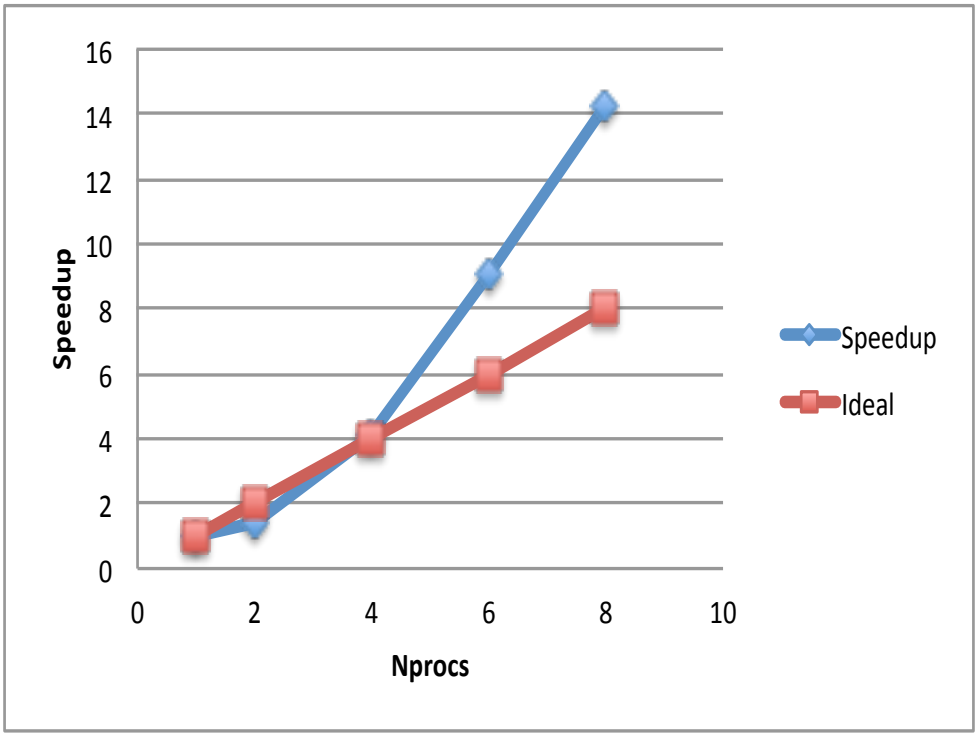
Note: T_{wall} reduced from 5+ seconds to 0.3

mpi-looptst RunTime (Twall)



Note: Ideal runtime computed using $T_{ideal} = \frac{T_{ser}}{p}$

mpi-looptst: Speedup

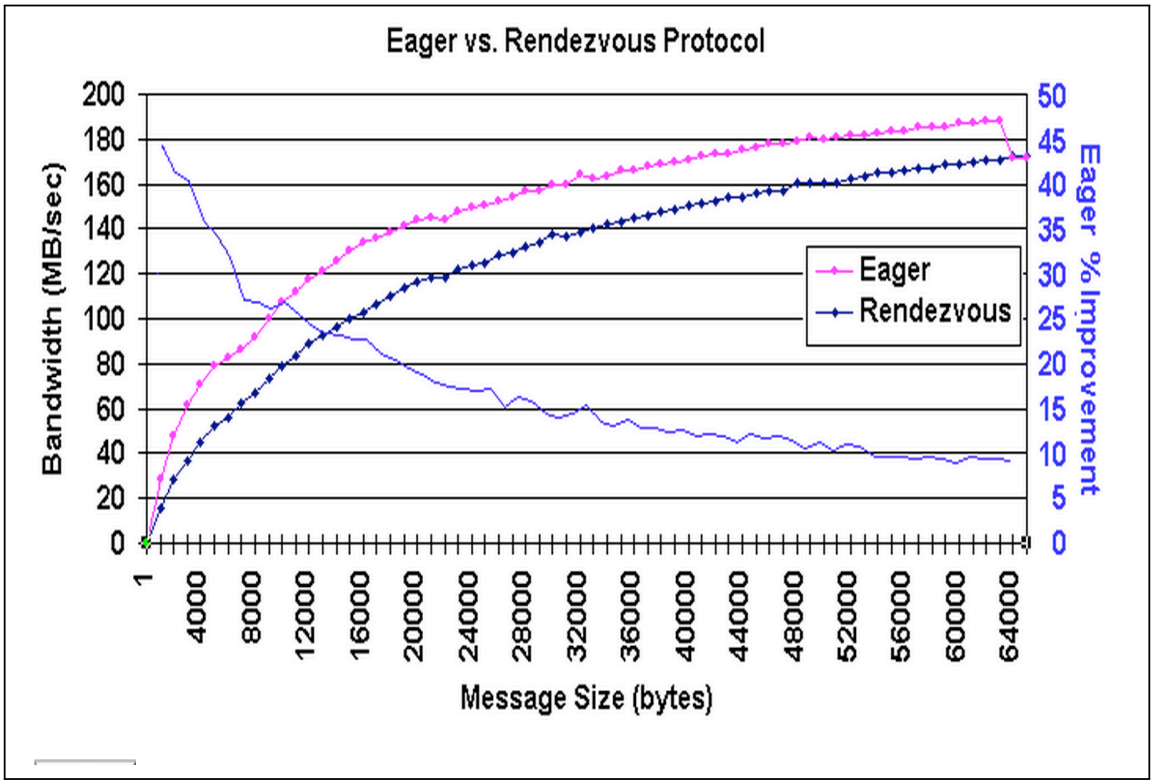


Factors Affecting MPI Communication Performance

- CPU/Processors:
  - Number of processors involved in the communication
  - Type of processor (speed, memory)
  - Software stack (including OS)
- Cluster Network Architecture:
  - Type/topology: `http://en.wikipedia.org/wiki/Network_topology`
  - Hardware design: Ethernet, Myrinet, WiFi
  - Protocols/Transport layer: TCP/IP, infiniband, `http://en.wikipedia.org/wiki/Lists_of_network_protocols`
- MPI Message Passing Protocols
- MPI Messages

MPI Message Passing Protocols

- MPI Protocol describes the internal methods and policies used to send messages.
- *Eager:* asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive
  - Sending process assumes receiving process can store message
  - Generally used for smaller message sizes (up to Kbytes).
  - Reduces synch. delays and simplifies programming.
  - not scalable: buffer "wastage"; program crash if data bigger than buffer
- *Rendezvous:* synchronous protocol; requires acknowledgement from a matching receive in order for the send operation to complete.
  - Requires some type of "handshaking" between the sender and the receiver processes
  - More scalable: robustness - prevents memory exhaustion and termination; only buffer small message envelopes; reduces data copy.
  - problem with synchronization delays; more programming complexity



Timings for Eager vs Rendevouz protocols

REF: https://computing.llnl.gov/tutorials/mpi_performance/

MPI Messages

- Characteristics
  - Message size (KBytes, MBytes, GBytes,) and buffering (GBytes/sec)
  - Number of other messages being sent
  - Where/how data is stored between the time a send operation begins and when the matching receive operation completes.
  - Larger messages tend to have better performance.
- Performance function of:
  - the number of words being sent
  - machine precision (32, 64 bit)
  - data type (int, long int, float, double)
- Performance measurement:
  - Calculate the time needed for a communication to start and send a message of known size.
  - Perform "warmup" events first: MPI implementation may use "lazy" semantics to setup and maintain streams of communications $\Rightarrow$ the first few events may take significantly longer than subsequent events.
- Speedup and Efficiency are relevant as well.

Total Parallel Run-Time

- The total parallel program run time is a function of a large number of variables: number of processing elements (PEs); communication; hardware (cpu, memory, software, network), and the program being run (algorithm, problem size, # Tasks, complexity, data distribution); parallel libraries:

$$T = \mathcal{F}\left(PEs, N, \ Tasks, \ I/O, \ Communication, \dots\right)$$

- The execution time required to run a problem of size N on processor i, is a function of the time spent in different parts of the program (computation, communication, I/O, idle):

$$T^i = T^i_{comp} + \ T^i_{comm} \ + T^i_{io} + T^i_{idle}$$

- The total time is the sum of the times over all processes averaged over the number of the processors: $T =$

$$\frac{1}{p}\left(\sum_{i=0}^{p-1} T_{comp} + \sum_{i=0}^{p-1} T_{comm} + \sum_{i=0}^{p-1} T_{io} + \sum_{i=0}^{p-1} T_{idel}\right)$$

The message passing communication time required to send N words (or Bytes):

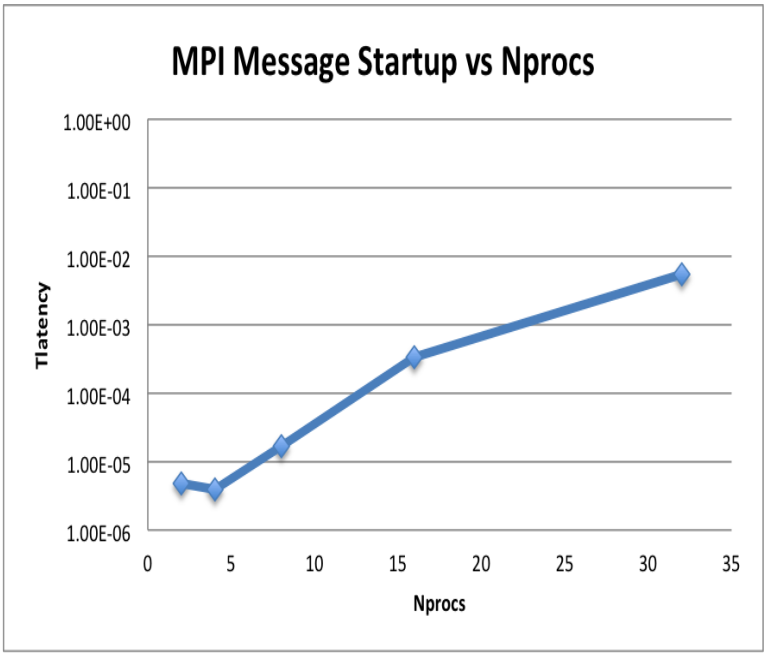
$$T_{comm} = t_{startup} + t_{bw}$$

Where:

- $t_{startup}$ is the message startup time (or latency)
  - Time required to set up communications on the nodes and to prepare them to send a message.
  - Estimated to be *half of the time* of a *ping-pong* operation with a message of size zero.
- t_{bw} is the message passing saturation bandwidth (BW).
  - Peak rate at which data packets can be sent across the network.
- Popular ways to measure:
  - *Ping-Pong*: measures communication between two PEs as function of message size.
  - *Ring*: measures communication between multiple PEs as a function of message size.
  - Can be used to test point-to-point or collective communications.

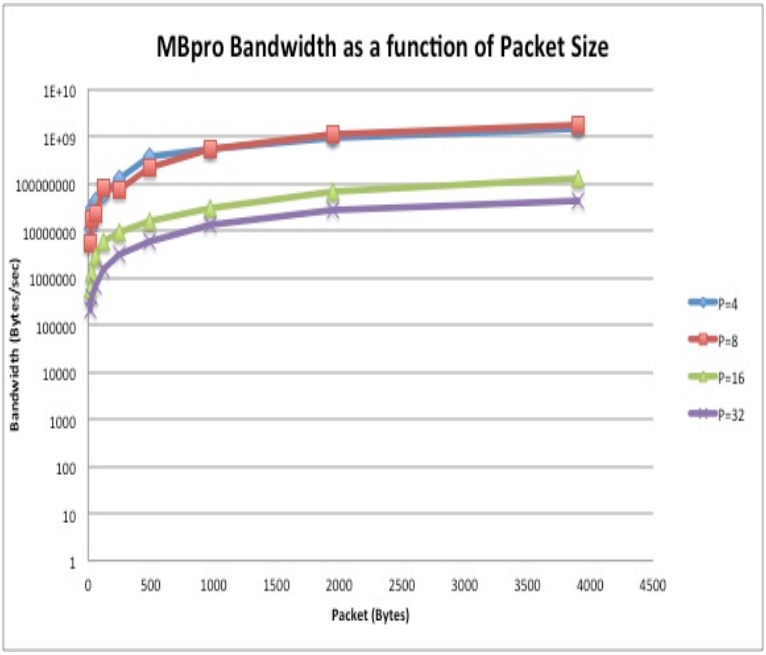
MPI Latency or Startup Time

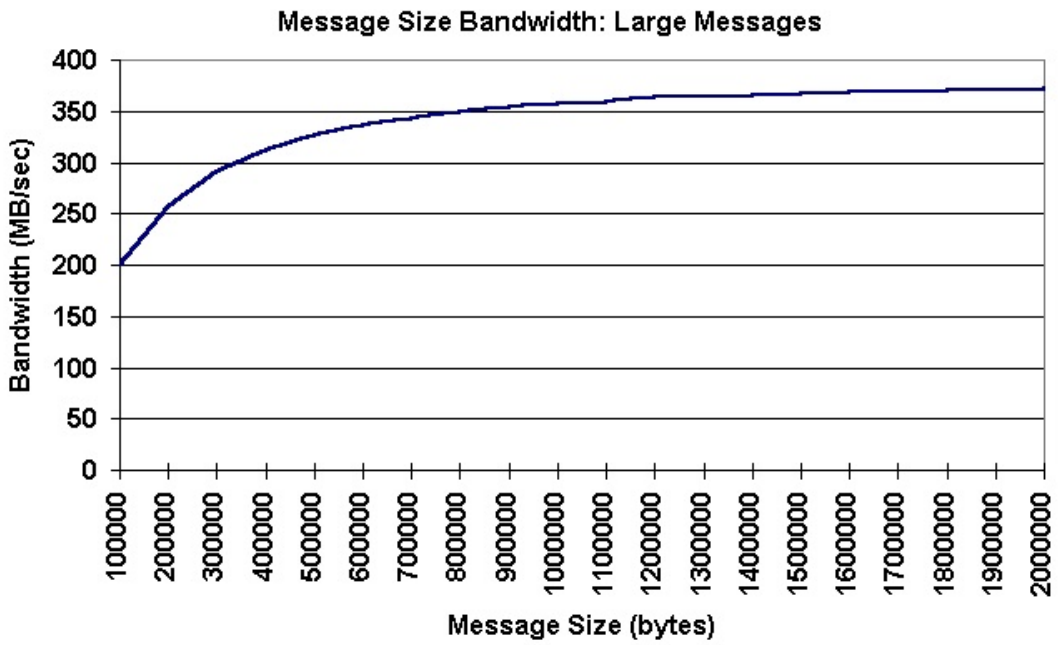
- **Message latency:** the time required to set up communications on the PEs and to prepare them to send a message.
- A function of the number and size of messages that need to be sent, and the number of PEs communicating.
- MPI latency is usually estimated to be 1/2 the time of a *"ping-pong"* operation with a message of size zero.
- In *ping-pong*, packets of information are exchanged between two PEs and the time required to do this is measured.
- Important when working with very fine-grained applications which have more frequent communication requirements.



MPI Message Bandwidth

- **Bandwidth:** Peak rate at which data packets can be sent across the network.
- Bandwidth is relevant for coarse-grained codes that send fewer messages, but typically need to communicate larger amounts of data.
- The bandwidth can be estimated using *ping-pong* and *ring* programs.
- Packets of information consist of an array of dummy integer or floating point numbers that vary in length.
- Code run-time is measured as a function of number of PE's (cores), and message size (number of Bytes).





Source: `https://computing.llnl.gov/tutorials/mpi_performance`

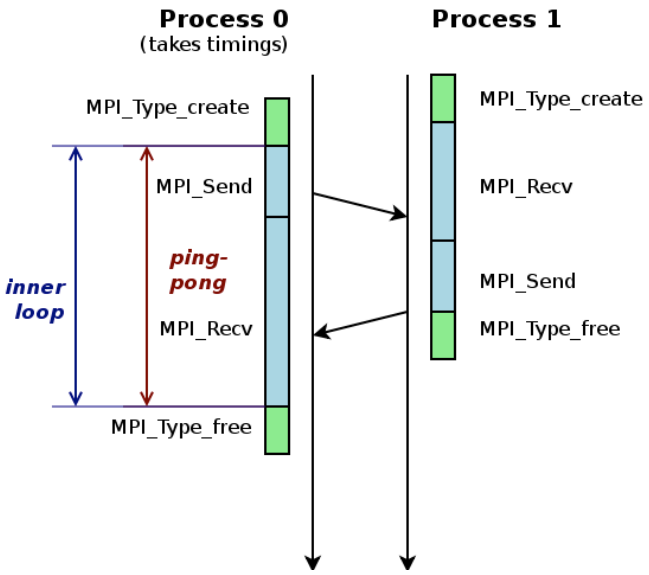
Communication Performance

- PingPong:
  - Two processes send packets of information back and forth a number of times
  - Compute average amount of time per message and transfer rate (bandwidth) as function of message size.
- Ring
  - Processes send packets of information to neighbor
  - Simple ordering: P0 to P1, P1-P2, ... Pn-1 to P0.
  - Measure time required to send message to all PE's as function of message size and the number of PEs.

Timing MPI Messages - Ping-Pong Algorithm

TimeStep	P_0	P_1
t_0	MPI_Send message to P_1	WAITS for message from P_0
t_1	WAITS for message from P_1	MPI_Recv message from P_0
t_2	WAITS for message from P_1	MPI_Send message to P_0
t_3	MPI_Recv message from P_0	

System has $sz = comm\_sz = 2$
Processors numbered $[P_1, P_2]$



Img source: `http://htor.inf.ethz.ch/research/datatypes/ddtbench/benchmark_expl.png`

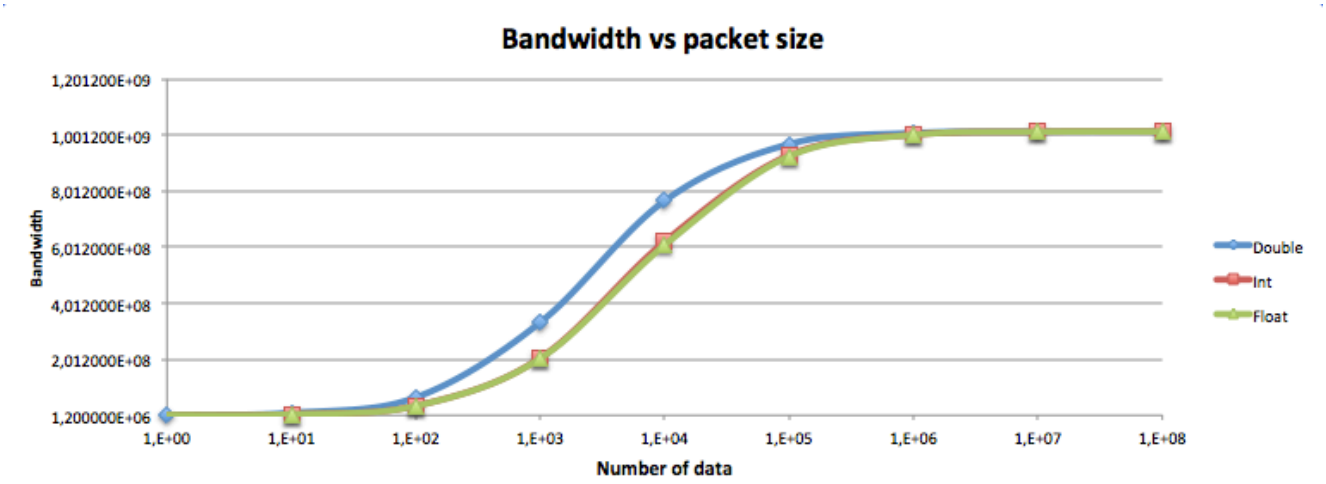
MPI Ping-Pong Code

```
/* ping_pong.c -- two-process ping-pong -- send from 0 to 1
 *  and send back   from 1 to 0
 * See Chap 12, pp. 267 & ff. in PPMPI */

#include <stdio.h>
#include "mpi.h"
#define MAX_ORDER 100
#define MAX 2
main(int argc, char* argv[]) {
int   p,my_rank, min_size = 0,max_size = 16;
int   incr = 8, size,pass;
float  x[MAX_ORDER];
int         i;
double      wtime_overhead;
double      start, finish;
double      raw_time;
MPI_Status  status;
MPI_Comm    comm;

/* startup the MPI environment */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_dup(MPI_COMM_WORLD, &comm);

wtime_overhead = 0.0;
for (i = 0; i < 100; i++) {
   start = MPI_Wtime();
   finish = MPI_Wtime();
   wtime_overhead = wtime_overhead + (start - finish);
}
wtime_overhead = wtime_overhead/100.0;

if (my_rank == 0) {
   for (size=min_size;size<=max_size; size=size+incr {
      for (pass = 0; pass < MAX; pass++) {
         MPI_Barrier(comm);
         start = MPI_Wtime();
         MPI_Send(x, size, MPI_FLOAT,1,0,comm);
         MPI_Recv(x, size, MPI_FLOAT,1,0,comm,&status);
         finish = MPI_Wtime();
         raw_time = finish - start - wtime_overhead;
         printf("%d %f\n", size, raw_time);
      }
   }
} else { /* my_rank == 1 */
   for (size=min_size;size<=max_size; size=size+incr {
      for (pass = 0; pass < MAX; pass++) {
         MPI_Barrier(comm);
         MPI_Recv(x, size, MPI_FLOAT,0,0,comm,&status);
         MPI_Send(x, size, MPI_FLOAT, 0, 0, comm);
      }
   }
}
MPI_Finalize();
}  /* main */
```

Timing MPI Messages: Ping-Pong Output

```
##############
# RUN USING MPICH on OS X
##############
[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=100
0 0.000005
0 0.000001
8 0.000009
8 0.000001
16 0.000001
16 0.000005

[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=10000
0 0.000007
0 0.000018
8 0.000002
8 0.000007
16 0.000001
16 0.000001

[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=1000000
0 0.000005
0 0.000011
8 0.000001
8 0.000001
16 0.000001
16 0.000006
```

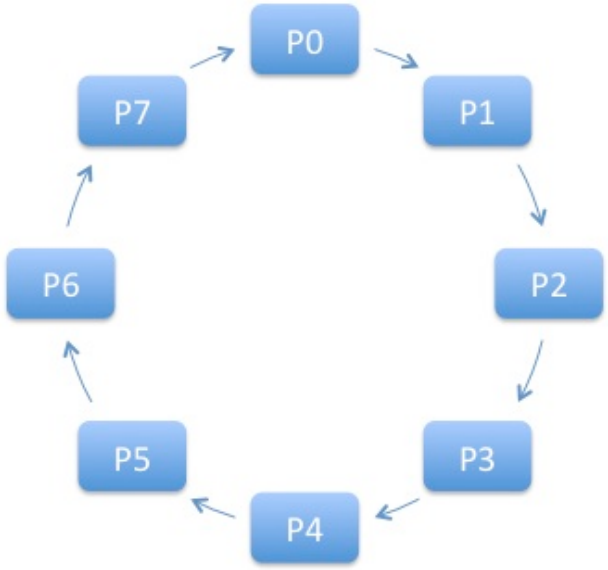
```
##############
# RUN USING %20.16f output
#############
[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=1000
0   0.0000049583311193
0   0.0000007883342914
8   0.0000138283637352
8   0.0000008103367873
16   0.0000007943296805
16   0.0000009803031571


[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=1000000
0   0.0000058855797397
0   0.0000010205834405
8   0.0000014185492182
8   0.0000012685480760
16   0.0000011545774760
16   0.0000009415956447
```



Source: COMP605 Student, J. Ayoub, Spring, 2014

Timing MPI Messages - Ring Algorithm

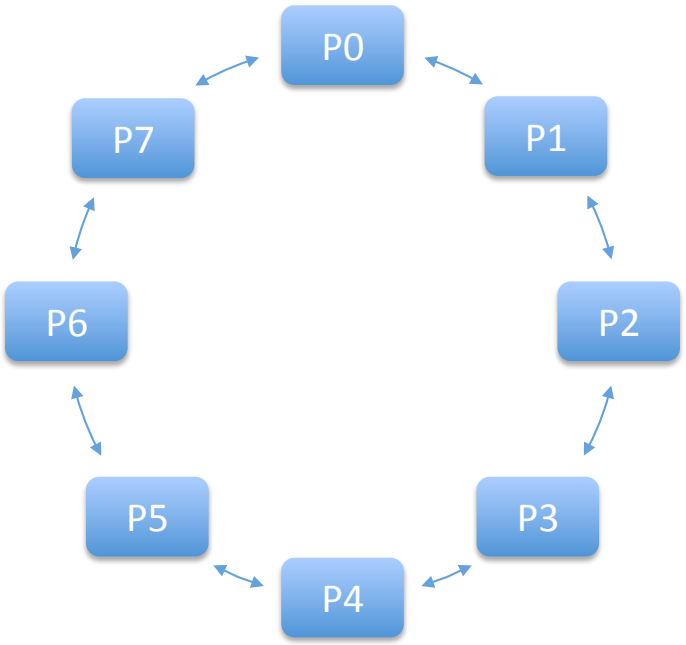
- System has $sz = comm\_sz$ processors numbered:
  $P_0, P_1, , P_{r-1}, P_r, P_{r+1}, ..P_{sz-1}$
- P_0 *sends* msg to P_1
  P_0 *waits* for msg from P_{sz-1}
  ...
  P_r *waits* for msg from P_{r-1}
  P_r *rcvs* msg, *sends* msg to P_{r+1}
  ...
  P_{sz-1} *sends* to P_0
  P_{sz-1} *waits* for msg from P_{sz-2}



8 Processors arranged in a ring

Timing MPI Messages - Ring Exchange

- System has $sz = comm_sz$ processors numbered
- **Step 0**: Each P_i creates unique msg.
- **Step 1**: P_i gets msg from lower nor, P_{i-1}, and sends its msg to upper nbr, P_{i+1}.
- **Step 2**: P_i gets msg from upper nbr, P_{i+1}, and sends its' msg to lower nbr, P_{i-1}.
- Code is done when all messages have been exchanged between each processor and its' neighbor.



Timing MPI Messages: pach_ring.c

```
 /*MPI ring message passing program
 * takes a single command line option: the maximum message
 * size in number of bytes
 * the program converts the number of bytes you specify
 * into numbers of doubles based on the byte size of a
 * double on that system. Then it starts with a message
 * of one double and scales by 2 until it reaches that
 * number, spitting out timing all along the way
 */

#include "stdlib.h"
#include "mpi.h"

/* if you want a larger number of runs to be averaged
#define ITERATIONS 1000
** together, increase INTERATIONS */
#define WARMUP 8

int main(int argc, char **argv)
{

   int i, j, rank, size, tag=96,bytesize, dblsize;
   int max_msg, min_msg, packetsize;
   int iterations;

   double *mess;
   double tend, tstart, tadd, bandwidth;
   MPI_Status status;

   MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD,&size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
/* get the message size from the command line */
 if(rank == 0)
 {
    printf("argcnt= %d\n",argc);
    dblsize = sizeof(double);

    if( argc >= 2 )
       max_msg = atoi(argv[1]);
    else
       max_msg = 4096;

    if( argc >= 3 )
       min_msg = atoi(argv[2]);
    else
       min_msg = 0;

   if( argc >= 4 )
       iterations = atoi(argv[3]);
    else
       iterations = 10;
```

Timing MPI Messages: pach_ring.c

```
 printf("ring size is %i nodes\n", size);
   printf("max message specified= %i\n", max_msg);
   printf("min message specified= %i\n", min_msg);
   printf("iterations =           %i\n", iterations);
   bytesize = max_msg;
   printf("double size is %i bytes\n", dblsize);
   max_msg = max_msg/dblsize;
   if(max_msg <= 0) max_msg = 1;
   printf("#of doubles being sent is %i\n", max_msg);

   printf("PacketLength\tBandwidth\tPacketTime\n");
   printf(" (MBytes)   \t (B/sec) \t(sec)\n");
   printf("------------ -------------- --------------\n");
}

/* pass out the size to the kids */
MPI_Bcast(&max_msg, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&min_msg, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&iterations, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* make the room for the largest sized message */
mess = (double*)malloc(max_msg * (sizeof(double)));
if(mess == NULL)
{
   printf("malloc prob, exiting\n");
   MPI_Finalize();
}
```

```
/* warmup lap */
 for(packetsize = 0; packetsize < WARMUP; packetsize++)
 {
    /* head node special case */
    if(rank == 0)
    {
       MPI_Send(mess, max_msg, MPI_DOUBLE, 1, tag, MPI_COMM_WORL
       MPI_Recv(mess, max_msg, MPI_DOUBLE, size-1,tag,
       MPI_COMM_WORLD, &status);
    }
    /* general case */
    if((rank != 0) && (rank != (size-1)))
    {
       MPI_Recv(mess, max_msg, MPI_DOUBLE, rank-1,tag,
       MPI_COMM_WORLD, &status);
       MPI_Send(mess, max_msg, MPI_DOUBLE, rank +1,tag,
       MPI_COMM_WORLD);
    }
    /* end node case */
    if(rank == size-1)
    {
       MPI_Recv(mess, max_msg, MPI_DOUBLE, rank-1,tag,
       MPI_COMM_WORLD, &status);
       MPI_Send(mess, max_msg, MPI_DOUBLE, 0,tag, MPI_COMM_WORLI
    }
 }
 /* end warmup lap */
 /*
 if(rank == 0)
 printf("warmup lap done\n");
 */
```

Timing MPI Messages: pach_ring.c

```
/* real timed stuff now */
for(packetsize = min_msg; packetsize <= max_msg; packetsize*=2)
{
   if(rank == 0)
      printf("Starting packetsize: %i\n",packetsize);
   /* init timing variables */
   tadd = 0.0;
   tend = 0.0;
   tstart = 0.0;

   for(j = 0; j < iterations; j++)
   {
      MPI_Barrier(MPI_COMM_WORLD);
      if(rank == 0)
      {
         tstart = MPI_Wtime(); /* timing call */
         MPI_Send(mess, packetsize, MPI_DOUBLE, 1, tag,
         MPI_COMM_WORLD);
         MPI_Recv(mess, packetsize, MPI_DOUBLE, size-1,tag,
         MPI_COMM_WORLD, &status);

         tend = MPI_Wtime();
         tadd += (tend - tstart);
      if( j%20 == 0 )
         printf("deltaT[%i]= %i\n",j,tend-tstart);
      }
               /* general case */
      if((rank != 0) && (rank != (size-1)))
      {
         MPI_Recv(mess, packetsize, MPI_DOUBLE, rank-1,tag,
         MPI_COMM_WORLD, &status);
         MPI_Send(mess, packetsize, MPI_DOUBLE, rank +1,tag,
         MPI_COMM_WORLD);
      }
      /* end node case */
      if(rank == size-1)
      {
         MPI_Recv(mess, packetsize, MPI_DOUBLE, rank-1,tag,
         MPI_COMM_WORLD, &status);
         MPI_Send(mess, packetsize, MPI_DOUBLE, 0,tag,
         MPI_COMM_WORLD);
      }

   }
   /* calc and print out the results */
   if(rank == 0)
   {
      bandwidth = ((size * packetsize *dblsize)/
                           (tadd/(double)iterations));
      printf("RESULTS: %16.12lf \t%20.8lf \t%16.14lf \n",
              (double)(packetsize * dblsize)/1048576.0,
              bandwidth,
              tadd/(double)iterations);
   }
   /* to make it possible to do a 0 size message */
   if (packetsize == 0) packetsize = 1;

}
/* end real timed stuff */

 if( rank == 0 ) printf("\nRing Test Complete\n\n");
MPI_Finalize();
exit(1);

} /* end ring.c */
```

Timing MPI Messages: pach_ring.c

```
[mthomas@tuckoo ring]$ mpirun -np 4 ./pach-ring
ring size is 4 nodes
max message specified= 4096,  min message specified= 0
iterations =           10
double size is 8 bytes,   #of doubles being sent is 512
PacketLength Bandwidth PacketTime
 (MBytes)     (B/sec)  (sec)
------------ -------------- --------------
Starting packetsize: 0  deltaT[0]= 0
RESULTS:   0.000000000000            0.00000000  0.00000300407410
Starting packetsize: 2  deltaT[0]= 0
RESULTS:   0.000015258789     13908572.84974093  0.00000460147858
Starting packetsize: 4  deltaT[0]= 0
RESULTS:   0.000030517578     14202934.17989418  0.00000901222229
Starting packetsize: 8  deltaT[0]= 0
RESULTS:   0.000061035156     61709300.22988506  0.00000414848328
Starting packetsize: 16  deltaT[0]= 0
RESULTS:   0.000122070312    138547332.12903225  0.00000369548798
Starting packetsize: 32  deltaT[0]= 0
RESULTS:   0.000244140625    258732969.63855419  0.00000395774841
Starting packetsize: 64  deltaT[0]= 0
RESULTS:   0.000488281250    445074331.19170982  0.00000460147858
Starting packetsize: 128  deltaT[0]= 0
RESULTS:   0.000976562500    885560267.21649492  0.00000462532043
Starting packetsize: 256  deltaT[0]= 0
RESULTS:   0.001953125000   1347440720.31372547  0.00000607967377
Starting packetsize: 512  deltaT[0]= 0
RESULTS:   0.003906250000   1391082525.02024293  0.00001177787781
Ring Test Complete
```

Comments: Calculating BW

Calculating BW:

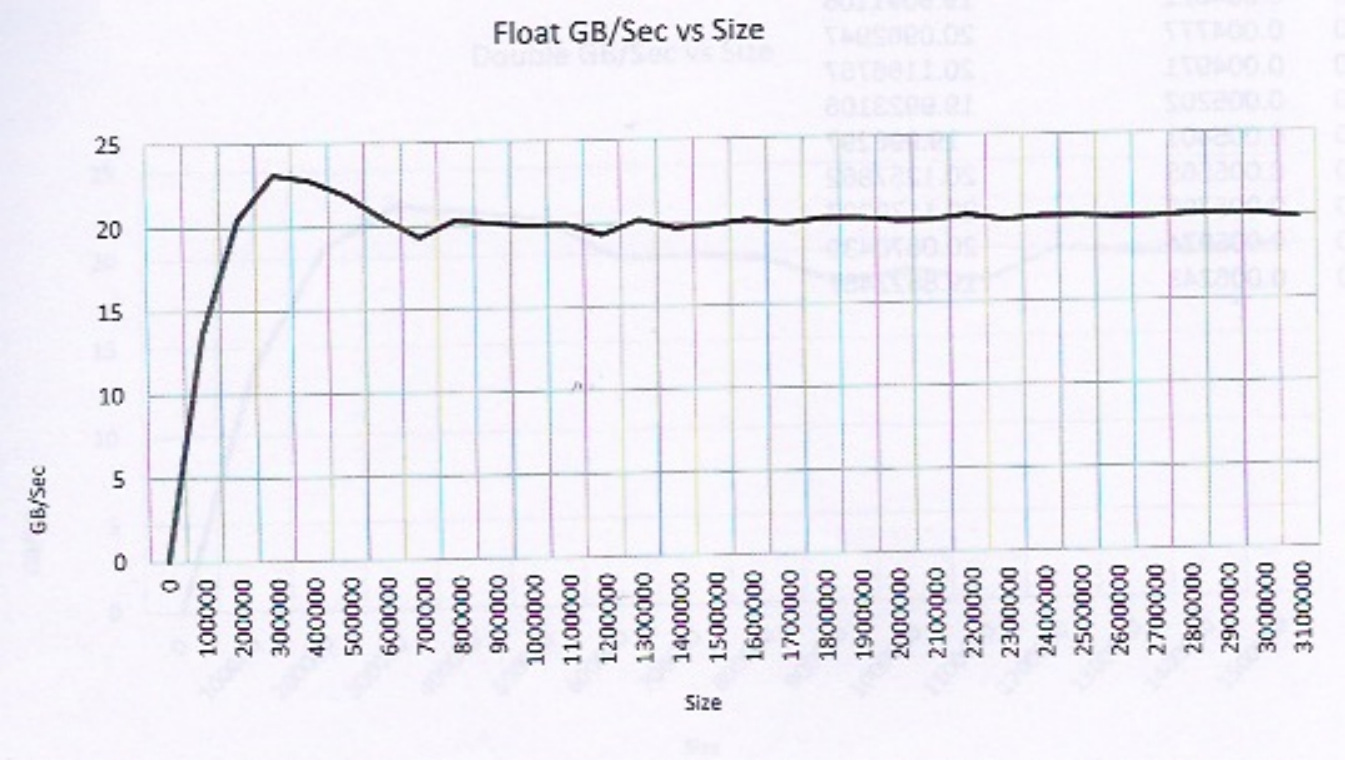
- BW units typically Mega or Giga Bytes per second, e.g., GByte/sec
- Estimate packet size per send or recv
- Count the number of sends or recvs you are using
- are you calculating BITS/sec, or BYTES/second? Convert packet size accordingly
- Example estimation: Ping-pong:

$$BW\left[\frac{a}{b}\right] \cong \frac{(\#exchanges) * packetSize[floats] * size[1float]}{rawTime[\mu sec]}$$

$$\cong \frac{[2] * 10^6[floats] * 32[bits/float]}{3x10^{-3}[seconds]}$$

$$\cong 21x10^9 \frac{bits}{second} * \frac{1Byte}{8bits}$$

$$\cong 2.67x10^9 \frac{GBytes}{second}$$



Source: COMP605 Student, D. Biscane, Spring, 2014