# Table of Contents

1. **Misc Information**
2. **Parallel Hardware Architectures (cont.)**
   - Agglomeration
   - Mapping
   - Histogram Example
3. **Performance**
   - Serial Histogram (Pacheco IPP)
4. **Next Time**
HW 1 due today
Next HW TBD
Agglomeration

- **Agglomeration**: Tasks are combined into larger tasks to improve performance or to reduce development costs.
Agglomeration

Examples of agglomeration.

- (a) the size of tasks is increased by reducing the dimension of the decomposition from three to two.
- (b) adjacent tasks are combined to yield a three-dimensional decomposition of higher granularity.
- (c) subtrees in a divide-and-conquer structure are coalesced.
- (d) nodes in a tree algorithm are combined.
Figure shows fine- and coarse-grained two-dimensional partitions. In each case, a single task is exploded to show its outgoing messages (dark shading) and incoming messages (light shading). In (a), a computation on a grid is partitioned into tasks; (b) the same computation is partitioned into tasks, each responsible for 16 points. In (a), communications are required, 4 per task; these transfer a total of 256 data values. In (b), only communications are required, and only data values are transferred.

Surface-to-Volume Effects. If the number of communication partners per task is small, we can often reduce both the number of communication operations and the total communication volume by increasing the granularity of our partition, that is, by agglomerating several tasks into one. This effect is illustrated in Figure 2.12. In this figure, the reduction in communication costs is due to a surface-to-volume effect. In other words, the communication requirements of a task are proportional to the surface of the...
Agglomeration Checklist

1. Has agglomeration reduced communication costs by increasing locality?
2. If agglomeration has replicated computation, have you verified that the benefits of this replication outweigh its costs, for a range of problem sizes and processor counts?
3. For data replication, verify that this does not compromise the scalability of your algorithm
4. Has agglomeration yielded tasks with similar computation and communication costs?
5. Does the number of tasks still scale with problem size?
6. If agglomeration eliminated opportunities for concurrent execution, verify that there is sufficient concurrency for current and future target computers
7. Can the number of tasks be reduced still further, without introducing load imbalances, increasing software engineering costs, or reducing scalability?
8. If you are parallelizing an existing sequential program, considered the cost of the modifications required to the sequential code
Mapping

- Maximize processor utilization and minimizing communication costs by distributing tasks to processors or threads.
- Specify where tasks will execute
- Not applicable to shared memory computers
- There are no general-purpose mapping solutions for distributed memory which have complex communication requirements
- Must be done manually. Main approaches:
  - Domain decomposition - fixed problem/tasks
  - Load balancing - dynamic task distribution
  - Task scheduling - many tasks with weak locality.
Domain Decomposition

- **Straightforward:**
  - Fixed number of equal sized tasks
  - Structured local/global communication.
  - Minimized communication

- **Complex Problems:**
  - Variable amounts of work per task
  - Unstructured communication (sometimes)

*Figure:* Block-block distribution. Each task does same work and communication. Dotted lines represent processor boundaries.
Load Balancing

Load Balancing Algorithms:
- Variable number of tasks.
- Variable communication.
- Performed multiple times.
- Often employ local load balancing
- Also called partitioning algorithms:
  - divide computational domain into specialized subdomains per processor

Figure: Irregular load balancing; each processor gets different data distribution and/or number of points
Types of Load Balancing Algorithms

- **Recursive Bisection:**
  - Partition domain into equal subdomains of equal computational costs.
  - Minimize communication costs.
  - "Divide and Conquer" – recursively cut domain

- **Local Algorithms**
  - Compensate for changes in computational load by getting information from a small number of neighbors.
  - Does not require global knowledge of program state

- **Probabilistic Methods**
  - Allocate tasks randomly to processors.
  - Assumes that if number of tasks is large, each processor will end up with about the same load.
  - Best when there are a large number of tasks and little communication.

- **Cyclic Mappings**
  - Computational load per grid varies and load is spatially dependent.
  - On average, each processor gets same load but communication costs may increase.
Task-Scheduling Algorithms:

- Used when functional decomposition yields many tasks
- Tasks have weak locality.
- Centralized task pool sent to/from processors
- Allocation of tasks to processors can be complex.

Manager-Worker:
- Centralized manager allocates tasks/problems
- Workers requests and executes tasks; may submit new tasks.

Hierarchical Manager-Worker:
- divides work into subsets which each have a manager

Decentralized Schemes:
- No centralized task manager - each processor has task pool.
- Idle workers request tasks from other processors
Mapping Checklist

1. For SPMD design for a complex problem, consider an algorithm based on dynamic task creation and deletion.
2. If considering design based on dynamic task creation and deletion, consider a SPMD algorithm.
3. For centralized load-balancing scheme, verify that manager will not become a bottleneck.
4. For dynamic load-balancing scheme, evaluate relative costs of different strategies.
5. For probabilistic or cyclic methods, load-balancing requires a large number of tasks.
Fosters Methodology Example: Histogram

1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3, 4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9

Image source: Pacheco 2011, Ch 2
Serial Histogram program - inputs

1. The number of measurements: $data_{count}$
2. An array of $data_{count}$ floats: $data$
3. The minimum value for the bin containing the smallest values: $min_{meas}$
4. The maximum value for the bin containing the largest values: $max_{meas}$
5. The number of bins: $bin_{count}$
Serial Histogram program - Outputs

1. *bin_maxes* : an array of *bin_count* floats; stores upper bound for each bin.

2. *bin_counts* : an array of *bin_count* ints; stores number of data elements in each bin.

3. assume *data_count* $\gg$ *bin_count*
Serial Histogram Pseudo-code

```c
/* Allocate arrays needed */
.
/* Generate the data */
.
/* Create bins for storing counts */
.
/* Count number of values in each bin */
for (i = 0; i < data_count; i++) {
    bin = Find_bin(data[i], bin_maxes, bin_count, min_meas);
    bin_counts[bin]++;
}
```

**Find_bin**: returns bin that data[i] belongs in - simple linear search function
Parallelizing Histogram program

Using Fosters methodology, identify the tasks and communication needed.

- **Tasks:**
  - Finding the bin for $data[i]$
  - Incrementing $bin\_count$ for that element

- **Communication:**
  - between identification/computation of the $bin$
  - incrementing the $bin\_count$
Problems occur when both data and bins are distributed.
What happens when $P_n$ needs to update $\text{bin\_count}$ on another Processor?
Solution: $\text{Task}_n$ updates local copy of $\text{bin\_count}$, then sum $\text{bin\_counts}$ at end
Fosters Methodology Example: Histogram

Tree structure gathering of \textit{bin\_count} data.
Performance measurements help determine how well a supercomputer runs.

HPC Hardware: Blue Gene/L Hardware
Top500.org TOP10 fastest machines in the world.

### 43rd List: The TOP10

<table>
<thead>
<tr>
<th>#</th>
<th>Site</th>
<th>Manufacturer</th>
<th>Computer</th>
<th>Country</th>
<th>Cores</th>
<th>Rmax (Pflops)</th>
<th>Power [MW]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>National University of Defense Technology</td>
<td>NUDT</td>
<td>Tianhe-2&lt;br&gt;NUDT TH-IVB-FEP, Xeon 12C 2.2GHz, IntelXeon Phi</td>
<td>China</td>
<td>3,120,000</td>
<td>33.9</td>
<td>17.8</td>
</tr>
<tr>
<td>2</td>
<td>Oak Ridge National Laboratory</td>
<td>Cray</td>
<td>Cray XK7, Opteron 12C 2.2GHz, Gemini, NVIDIA K20x</td>
<td>USA</td>
<td>560,640</td>
<td>17.6</td>
<td>8.21</td>
</tr>
<tr>
<td>3</td>
<td>Lawrence Livermore National Laboratory</td>
<td>IBM</td>
<td>Sequoia&lt;br&gt;BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>USA</td>
<td>1,572,864</td>
<td>17.2</td>
<td>7.89</td>
</tr>
<tr>
<td>4</td>
<td>RIKEN Advanced Institute for Computational Science</td>
<td>IBM</td>
<td>K Computer&lt;br&gt;SPARC64 V1fx 2.0GHz, Totu Interconnect</td>
<td>Japan</td>
<td>795,024</td>
<td>10.5</td>
<td>12.7</td>
</tr>
<tr>
<td>5</td>
<td>Argonne National Laboratory</td>
<td>IBM</td>
<td>Mira&lt;br&gt;BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>USA</td>
<td>786,432</td>
<td>8.59</td>
<td>3.95</td>
</tr>
<tr>
<td>6</td>
<td>Swiss National Supercomputing Centre (CSCS)</td>
<td>Cray</td>
<td>Piz Daint&lt;br&gt;Cray XC30, Xeon E5 8C 2.6GHz, Aries, NVIDIA K20x</td>
<td>Switzerland</td>
<td>115,984</td>
<td>6.27</td>
<td>2.33</td>
</tr>
<tr>
<td>7</td>
<td>Texas Advanced Computing Center/UT</td>
<td>Dell</td>
<td>Stampede&lt;br&gt;PowerEdge C8220, Xeon E5 8C 2.7GHz, Intel Xeon Phi</td>
<td>USA</td>
<td>462,462</td>
<td>5.17</td>
<td>4.51</td>
</tr>
<tr>
<td>8</td>
<td>Forschungszentrum Juelich (FZJ)</td>
<td>IBM</td>
<td>JuQUEEN&lt;br&gt;BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>Germany</td>
<td>458,752</td>
<td>5.01</td>
<td>2.30</td>
</tr>
<tr>
<td>9</td>
<td>Lawrence Livermore National Laboratory</td>
<td>IBM</td>
<td>Vulcan&lt;br&gt;BlueGene/Q, Power BQC 16C 1.6GHz, Custom</td>
<td>USA</td>
<td>393,216</td>
<td>4.29</td>
<td>1.97</td>
</tr>
<tr>
<td>10</td>
<td>Government</td>
<td>Cray</td>
<td>Cray XC30, Xeon E5 12C 2.7GHz, Aries</td>
<td>USA</td>
<td>225,984</td>
<td>3.14</td>
<td></td>
</tr>
</tbody>
</table>

Source: [http://www.top500.org/blog/slides-for-the-43rd-top500-list-now-available/](http://www.top500.org/blog/slides-for-the-43rd-top500-list-now-available/)
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)

\[ T = F(\text{ProbSize}, \text{Tasks}, I/O, \ldots) \]

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: $\mathcal{O}(n \times m)$, Binary Search Tree: $\mathcal{O}(\log n)$
- Input and Output statements:
  - STDIO/STDIN
  - File I/O
Wallclock Time: $T_{\text{wall}}$

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.

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

Source: http://en.wikipedia.org/wiki/Wall-clock_time
FLOPs: Floating Point Operations

\[ FLOPs \equiv \text{Floating Point Operations} \]
\[ FLOPS \equiv \text{Floating Point Operations Per Second} \]

The FLOPS of a CPU can be estimated using empirical data:

\[ FLOPS \approx \text{clock\_frequency} \left( \frac{\text{cycles}}{\text{second}} \right) \times \frac{\# \text{FLOPs}}{\text{cycle}} \]

For example, most microprocessors today can do 4\text{FLOPs} per clock cycle. The tuckoo cluster nodes have 1.6 GHz CPU processors, which gives a theoretical performance of

\[ FLOPS_{\text{theoretical}} \approx 1.6 \times 10^9 \frac{\text{cycles}}{\text{second}} \times \frac{4 \text{ FLOPs}}{\text{cycle}} \]

\[ \approx 6.4 \times 10^9 \text{ FLOPS or 6.4 GFLOPS} \]

/* 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_counts[], int bin_count);
void Print_histo(float bin_maxes[], int bin_counts[], int bin_count, float min_meas);
```c
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]++;
    }

    /* Print the histogram */
    Print_histo(bin_maxes, bin_counts, bin_count, min_meas);
    free(data);
    free(bin_maxes);
    free(bin_counts);
    return 0;
} /* main */
```
```c
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>
    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 */
```
/* Function: Gen_data
 * Purpose: Generate random floats in the range min_meas \leq 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()/(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 */
serial_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 measurement
* */

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 */
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);
/ * 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);

gmtimeofday(&tstart, whichbin, 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 and Running the Job

```
[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]$ ./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 Time in 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

[mthomas@tuckoo ch2]$ prof histodat gmon.out

% cumulative self
self total
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
```
Histogram outputs: What do the results mean?

[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 Time in seconds: 0.000130 seconds
Sum Times in seconds: 0.000140 seconds
Data: bin_count, data_count, T.wall, T.getargs, T.mem, T.gendat, T.genbins, T.whichbin
CSV Dat:10,100,0.000140,0.000017,0.000106,0.000007,0.000000
[mthomas@tuckoo ch2]$

[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 Time in seconds: 0.000156 seconds
Sum Times in seconds: 0.000247 seconds
Data: bin_count, data_count, T.wall, T.getargs, T.mem, T.gendat, T.genbins, T.whichbin
CSV Dat:10,1000,0.000247,0.000018,0.000111,0.000027,0.000000
[mthomas@tuckoo ch2]$

[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 Time in seconds: 0.000360 seconds
Sum Times in seconds: 0.001275 seconds
Data: bin_count, data_count, T.wall, T.getargs, T.mem, T.gendat, T.genbins, T.whichbin
CSV Dat:10,10000,0.001275,0.000016,0.000105,0.000239,0.000000
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 total self total      name
 time seconds  seconds  calls  ms/call  ms/call
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       1    10.03  10.03 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 0.086 seconds.
```

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

\[ \text{FLOPS} \approx \frac{\text{Total Number of Operations}}{\text{Total Time}} \]

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

\[ \text{TotalOps}_{\text{WhichBin}} = (\#\text{Ops in WhichBin}) \times (\#\text{Calls to WhichBin}) \]

Analysis of the function \textit{Whichbin}, shows that the number of operations is of order \( \vartheta(N) = \vartheta(10) \). The number of calls is determined by \textit{data\_count}. For \textit{data\_count} = 10^6 elements, we measured the time spent in \textit{WhichBin} to be \( T_{\text{whichbin}} = 0.085712 \) seconds. We estimate that the FLOPS for the histogram program to be

\[ \text{FLOPS}_{\text{measured}} \approx \frac{\vartheta(N) \times \text{data\_count}}{T_{\text{whichbin}}} \approx \frac{10 \times 10^6}{0.085712} \approx 1.2 \times 10^8 \ \text{FLOPS} \]

This is less than the theoretical performance we calculated earlier: \( \text{FLOPS}_{\text{theoretical}} \approx 6.4 \ \text{GFLOPS} \).
### Performance

**Serial Histogram (Pacheco IPP)**

#### Summarizing the Timing Data in a Table

<table>
<thead>
<tr>
<th>bin_count</th>
<th>data_count</th>
<th>T_wall</th>
<th>T_getargs</th>
<th>T_mem</th>
<th>T_gendat</th>
<th>T_genbins</th>
<th>T_whichbin</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1.00E+00</td>
<td>1.34E-04</td>
<td>1.80E-05</td>
<td>1.10E-04</td>
<td>5.00E-06</td>
<td>0.00E+00</td>
<td>1.00E-06</td>
</tr>
<tr>
<td>10</td>
<td>1.00E+01</td>
<td>1.44E-04</td>
<td>2.70E-05</td>
<td>1.10E-04</td>
<td>5.00E-06</td>
<td>0.00E+00</td>
<td>2.00E-06</td>
</tr>
<tr>
<td>10</td>
<td>1.00E+02</td>
<td>1.52E-04</td>
<td>1.80E-05</td>
<td>1.17E-04</td>
<td>7.00E-06</td>
<td>0.00E+00</td>
<td>1.00E-05</td>
</tr>
<tr>
<td>10</td>
<td>1.00E+03</td>
<td>2.52E-04</td>
<td>1.60E-05</td>
<td>1.19E-04</td>
<td>2.60E-05</td>
<td>0.00E+00</td>
<td>9.10E-05</td>
</tr>
<tr>
<td>10</td>
<td>1.00E+04</td>
<td>1.29E-03</td>
<td>1.70E-05</td>
<td>1.21E-04</td>
<td>2.40E-04</td>
<td>0.00E+00</td>
<td>9.13E-04</td>
</tr>
<tr>
<td>10</td>
<td>1.00E+05</td>
<td>9.60E-03</td>
<td>1.60E-05</td>
<td>1.19E-04</td>
<td>2.18E-03</td>
<td>0.00E+00</td>
<td>7.29E-03</td>
</tr>
<tr>
<td>10</td>
<td>1.00E+06</td>
<td>9.26E-02</td>
<td>1.80E-05</td>
<td>1.13E-04</td>
<td>2.08E-02</td>
<td>0.00E+00</td>
<td>7.18E-02</td>
</tr>
<tr>
<td>10</td>
<td>1.00E+07</td>
<td>9.00E-01</td>
<td>1.80E-05</td>
<td>1.21E-04</td>
<td>1.83E-01</td>
<td>1.00E-06</td>
<td>7.18E-01</td>
</tr>
</tbody>
</table>
Plotting Results - Family of Curves

Histogram Run Times vs Data Count for Different Code Sections

- T\text{wall}
- T\text{getarts}
- T\text{mem}
- T\text{gendat}
- T\text{whichbin}
Plotting Results - Family of Curves (log-log)
### Timing Example: UCOAM Model

#### TABLE II.

<table>
<thead>
<tr>
<th>Section</th>
<th>Serial</th>
<th>16 Processors</th>
<th>32 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tinit</td>
<td>48571</td>
<td>24285</td>
<td>16190</td>
</tr>
<tr>
<td>Tloop</td>
<td>59451</td>
<td>29725</td>
<td>19817</td>
</tr>
<tr>
<td>Twall</td>
<td>108083</td>
<td>54041</td>
<td>36027</td>
</tr>
</tbody>
</table>

#### TABLE III.

<table>
<thead>
<tr>
<th>Section</th>
<th>Serial</th>
<th>16 Processors</th>
<th>32 Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tpres</td>
<td>31619</td>
<td>15810</td>
<td>10540</td>
</tr>
<tr>
<td>Tfio</td>
<td>17961</td>
<td>8981</td>
<td>5987</td>
</tr>
<tr>
<td>Tsgs</td>
<td>3026</td>
<td>1513</td>
<td>1009</td>
</tr>
<tr>
<td>TVelw</td>
<td>1736</td>
<td>868</td>
<td>579</td>
</tr>
<tr>
<td>TVelu</td>
<td>1726</td>
<td>863</td>
<td>575</td>
</tr>
<tr>
<td>TVelv</td>
<td>1716</td>
<td>858</td>
<td>572</td>
</tr>
<tr>
<td>TbcondP</td>
<td>448</td>
<td>224</td>
<td>150</td>
</tr>
<tr>
<td>TvelcorV</td>
<td>120</td>
<td>61</td>
<td>40</td>
</tr>
<tr>
<td>TvelcorW</td>
<td>110</td>
<td>55</td>
<td>36</td>
</tr>
<tr>
<td>TvelcorU</td>
<td>109</td>
<td>54</td>
<td>367</td>
</tr>
<tr>
<td>TbcondW</td>
<td>22</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>TbcondU</td>
<td>22</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>TbcondV</td>
<td>20</td>
<td>11</td>
<td>67</td>
</tr>
<tr>
<td>Tloop (meas)</td>
<td>58635</td>
<td>29317</td>
<td>19545</td>
</tr>
</tbody>
</table>
Timing Code
Next class: 09/11/14