

Parallel Programming Project Notes



Oregon State
University

Mike Bailey

mjb@cs.oregonstate.edu



To a Better Grade



Oregon State
University
Computer Graphics

Why Are These Notes Here?

2

These notes are here to:

1. Help you setup and run your projects
2. Help you get the data you collect in the right format for submission
3. Help you get a **better grade** by doing all of this correctly!
better grade!
better grade!
better grade!
better grade!
better grade!

- Feel free to run your projects on whatever systems you have access to.
- If you don't have access to your own systems, then you can use what we have at OSU. **On-campus users** will have access to Windows and Linux systems here. **Ecampus users** will have remote access to our Linux systems, such as **flip**.
- Most of the projects will require timing to determine performance. Use the OpenMP timing functions. They give decent answers, and this will make the timing consistent across projects and across people. The OpenMP call:

```
double prec = omp_get_wtick( );
```

tells you the precision of the clock in seconds. I get 10^{-9} seconds on the systems I've been using. (I really doubt that this is true.) The OpenMP call:

```
double time0 = omp_get_wtime( );
```

samples the clock right now. It gives you wall-clock time in seconds. In parallel computing, memory latency and thread-idle time are part of the equation, so wall clock time is what you want.



How We Will Be Doing Timing

4

In this class, we don't want to just **implement** – we want to **characterize** performance. What speed-ups do we see, and why do we see them? How do we generalize that to other types of problems? What insights does this give us?

So, as part of your project assignments, you will be doing a lot of timing to determine program speed-ups.

```
#include <omp.h>
```

```
double time0 = omp_get_wtime( );           // seconds
```

```
...
```

```
double time1 = omp_get_wtime( );           // seconds
```

```
fprintf( stderr, "Elapsed time = %10.2lf microseconds\n", 1000000. * ( time1 – time0 ) );
```

%10.2**lf** is a good way to print doubles ("long float")

How Reliable is the Timing?

5

This way of timing measures **wall-clock time**, which is really what we want to know in a parallel environment, *not CPU time*.

However, this puts you at the mercy of the other users on the system. If you are on one of our public systems (e.g., flip), I advise you to check the system load to see how much off your wall-clock time measurement will be due to the competition from other users. Use the Linux **uptime** command::

flip01 34% uptime

11:13:37 up 96 days, 11:52, 23 users, load average: 3.56, 3.08, 2.82

These three numbers represent total CPU load averages for the last 1, 5, and 15 minutes respectively. If the CPU load average is greater than the number of CPUs, then each CPU is over-burdened.

Clearly you want these numbers, especially the 1-minute one, to be as small as possible when you run your test. If they are “big”, you might want to ssh to other systems (flip01, flip02, flip03, ...) to see if you can find a better place to run, or try again later.

How Reliable is the Timing? A Useful Trick!

I like to check the consistency of the timing by computing both peak speed and average speed and seeing how close they are:

```
double maxmflops = 0.;
double summflops = 0.;

for( int t = 0; t < NUMTRIES; t++ )
{
    double time0 = omp_get_wtime( );

    #pragma omp parallel for
    for( int i = 0; i < ARRAYSIZE; i++ )
    {
        C[i] = A[i] * B[i];
    }

    double time1 = omp_get_wtime( );
    double mflops = (double)ARRAYSIZE/(time1-time0)/1000000.;
    summflops += mflops;
    if( mflops > maxmflops )
        maxmflops = mflops;
}

printf( "    Peak Performance = %8.2lf MFLOPS\n", maxmflops );
printf( "Average Performance = %8.2lf MFLOPS\n", summflops/(double)NUMTRIES );
```

You should record the peak performance value. This gives you as close to the best answer that you will get. But, compare that with the average performance value. That will tell you how reliable that peak value is.



This is a reliable result:

Peak Performance = 1183.31 MFLOPS
Average Performance = 1141.41 MFLOPS

This is an unreliable result:

Peak Performance = 627.39 MFLOPS
Average Performance = 294.86 MFLOPS

If you are on Linux and have access to the Intel compiler, *icpc*, don't use it unless we tell you to! (*icpc* is so good that it often does optimizations that undermine the very things you are testing.)

Use *g++*. The compilation sequences are:

On Linux, the typical compile sequence for files that use OpenMP is:

```
g++ -o proj proj.cpp -O3 -lm -fopenmp
```

```
icpc -o proj proj.cpp -O3 -lm -openmp -align -qopt-report=3 -qopt-report-phase=vec
```

Note that OpenMP should always be included because we are using OpenMP calls for timing.

Note that the second character in the 3-character sequence “-lm” is an ell, i.e., a lower-case L. This is how you link in the **math** library.



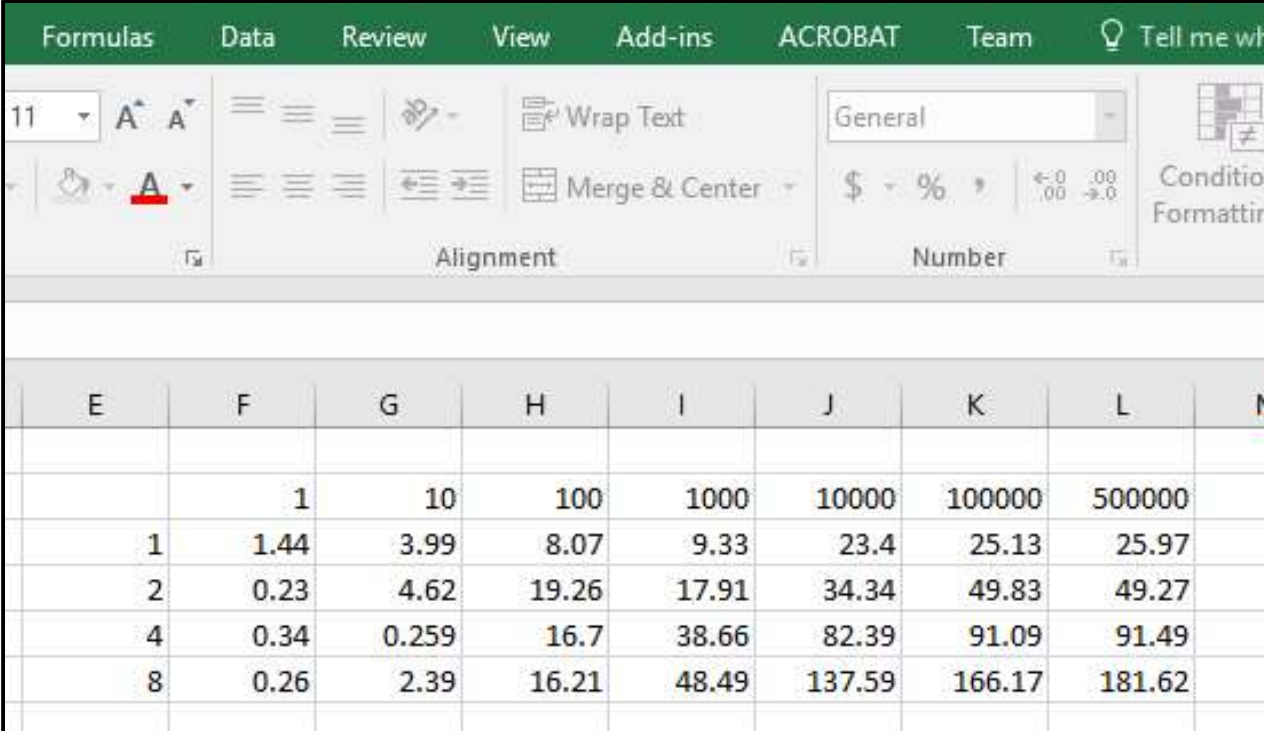
- Most of these projects will require you to submit graphs. You can prepare the graphs any way you want, except for drawing them by hand. (The Excel Scatter-with-Smooth-Lines-and-Markers works well.) So that we can easily look at each other's graphs, please follow the convention that **up is faster**. That is, do not plot seconds on the Y axis because then "up" would mean "slower". Instead, plot something like Speedup or MFLOPS or frames-per-second.
- I expect the graphs to show off your **scientific literacy** -- that is, I expect axes with numbers, labels, and units, If there are multiple curves on the same set of axes, I expect to be able to easily tell which curve goes with which quantity. After all, there is a reason this major is called Computer **Science**. Not doing this makes your project unacceptable for grading.

You lose points if you don't do it this way.



Making Graphs

In Excel, I have had the most success with creating tables that look like this:



	E	F	G	H	I	J	K	L	M
		1	10	100	1000	10000	100000	500000	
1	1.44	3.99	8.07	9.33	23.4	25.13	25.97		
2	0.23	4.62	19.26	17.91	34.34	49.83	49.27		
4	0.34	0.259	16.7	38.66	82.39	91.09	91.49		
8	0.26	2.39	16.21	48.49	137.59	166.17	181.62		

where the 1,2,4,8 rows are holding the number of threads constant, and the 1, 10, 100, 1000, etc. columns are holding the dataset size constant. The cells are holding performance numbers, with higher numbers representing faster performance.

Sweep over the entire table, select Copy, and then insert it into one of the scatterplot options.

Making Graphs

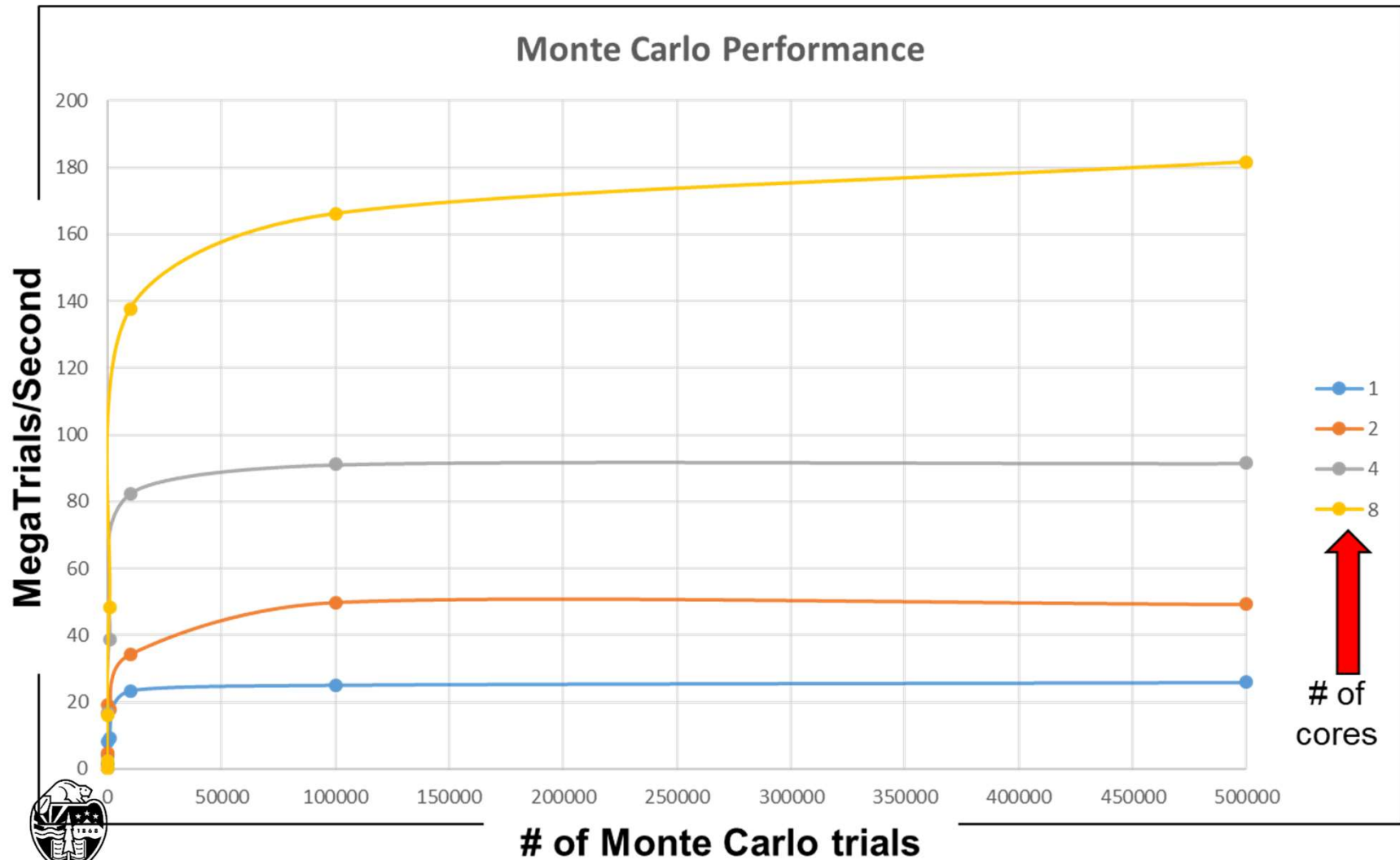
10

Sweep over the entire table, select Copy, and then insert it into one of the **scatterplot** options.

The screenshot shows the Microsoft Excel interface. The 'Insert' tab is selected in the ribbon. The 'Charts' group in the ribbon is expanded, showing various chart options. The 'Scatter' chart icon is highlighted with a red circle. A red arrow points from the 'Insert' tab to the 'Scatter' chart icon. Below the ribbon, a table of data is visible, with a red box highlighting the first column (B) and the first four rows (1-4).

	B	C	D	E	F	G	H	I	J	K	L
1	1.44										
10	3.99										
100	8.07										
1000	9.33										
10000	23.4										
100000	25.13										
1	0.23										

How Did this Monte Carlo Simulation Turn Out?

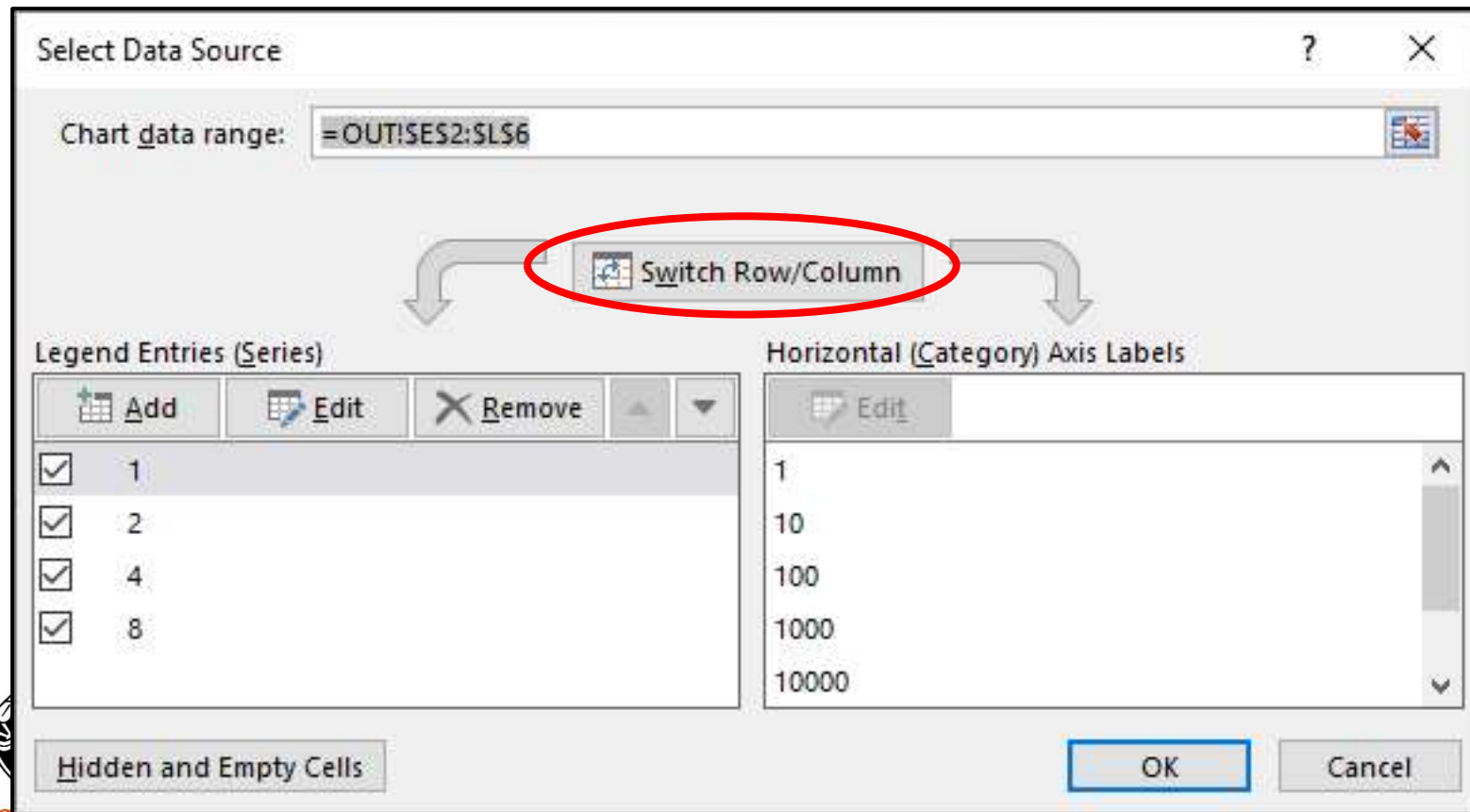


Transposing the Graph

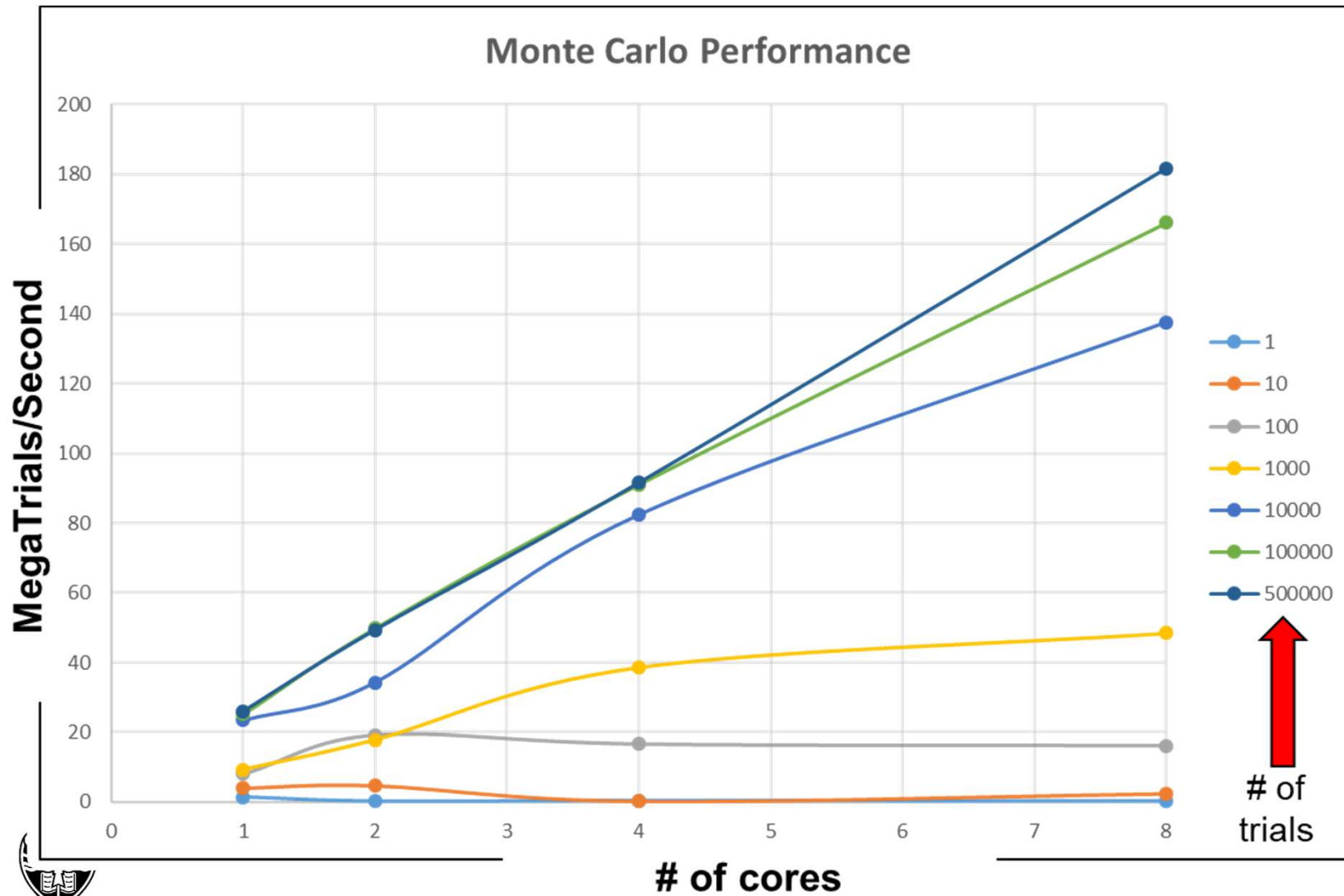
12

To transpose the sense of the graph (which you also need to do), right-click on the border of the graph and then click on "Select Data".

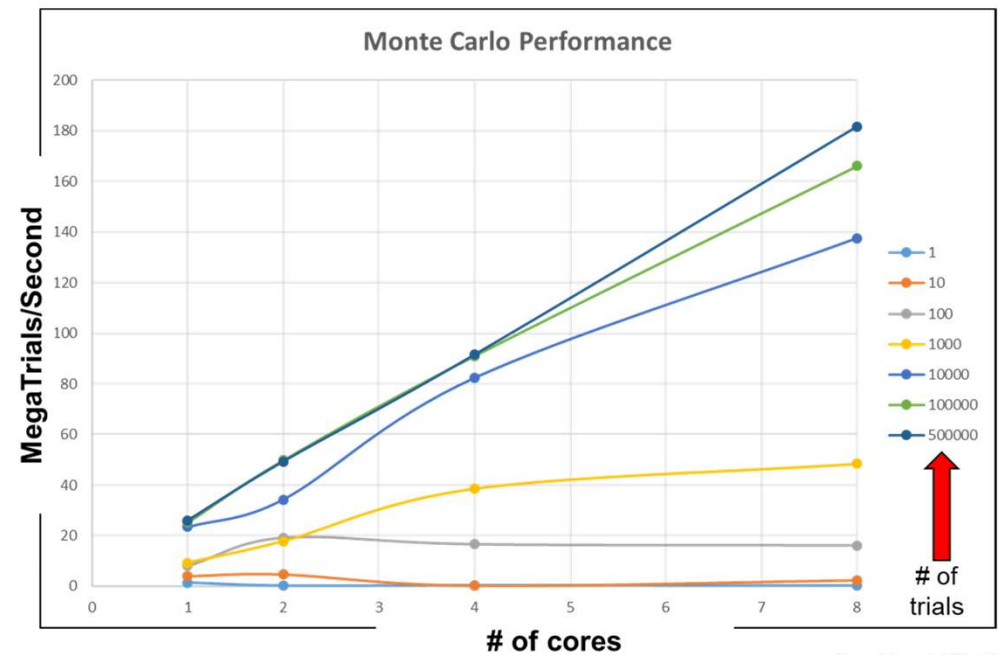
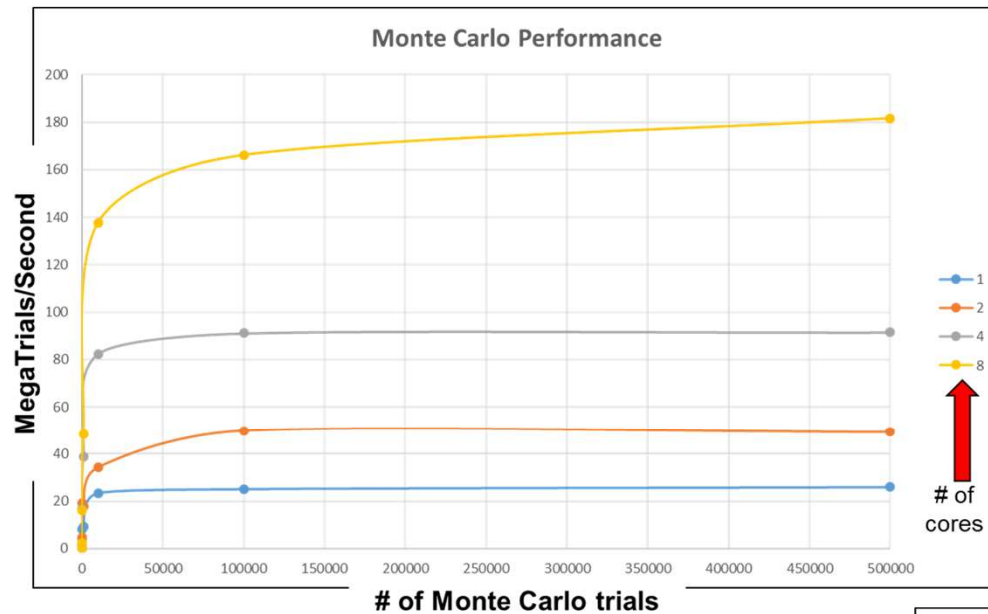
Then click on "Switch Row/Column".



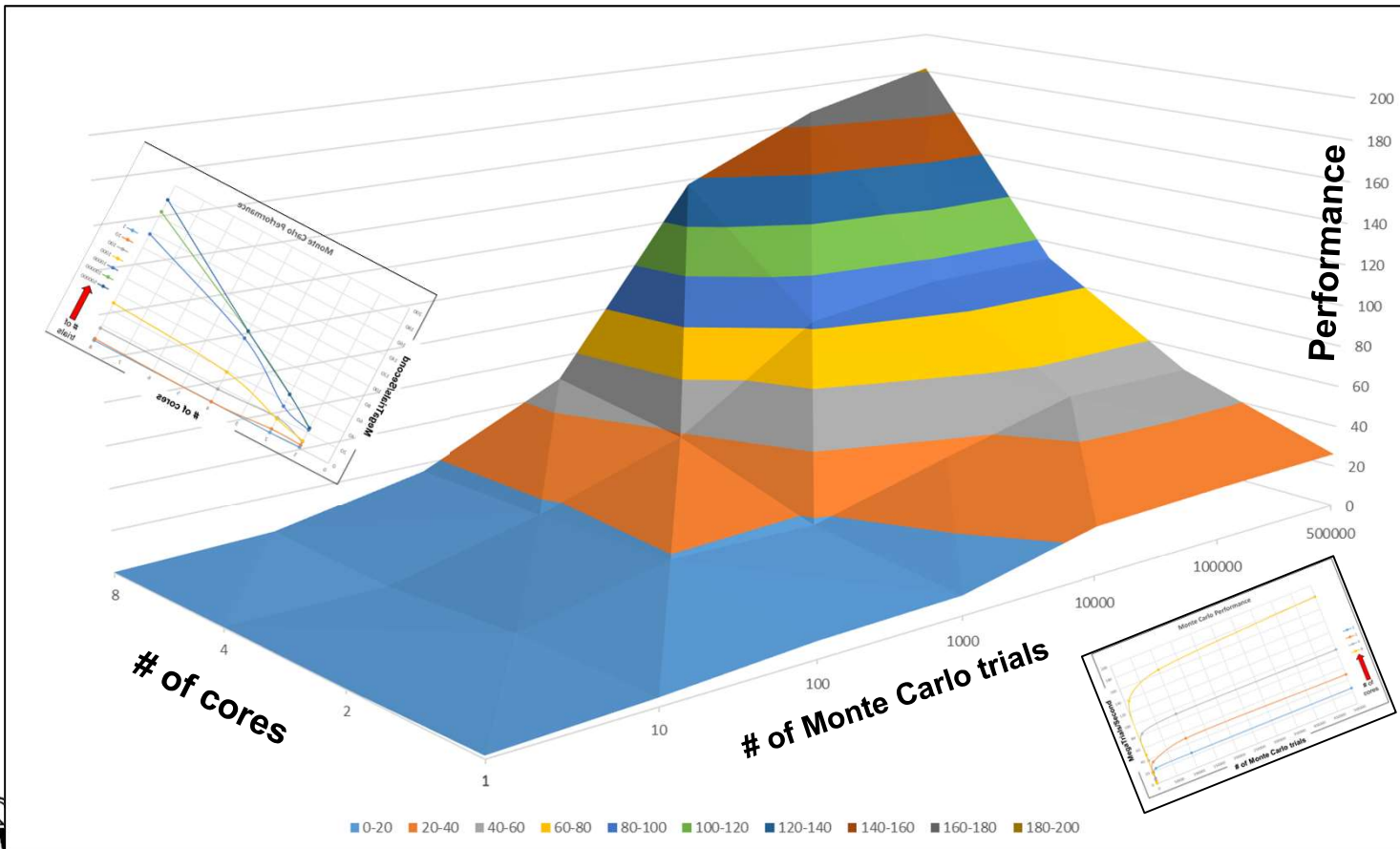
How Did this Monte Carlo Simulation Turn Out?



It's the Same Data, but Each Graph Gives You a Different Insight into what the Data is Telling You



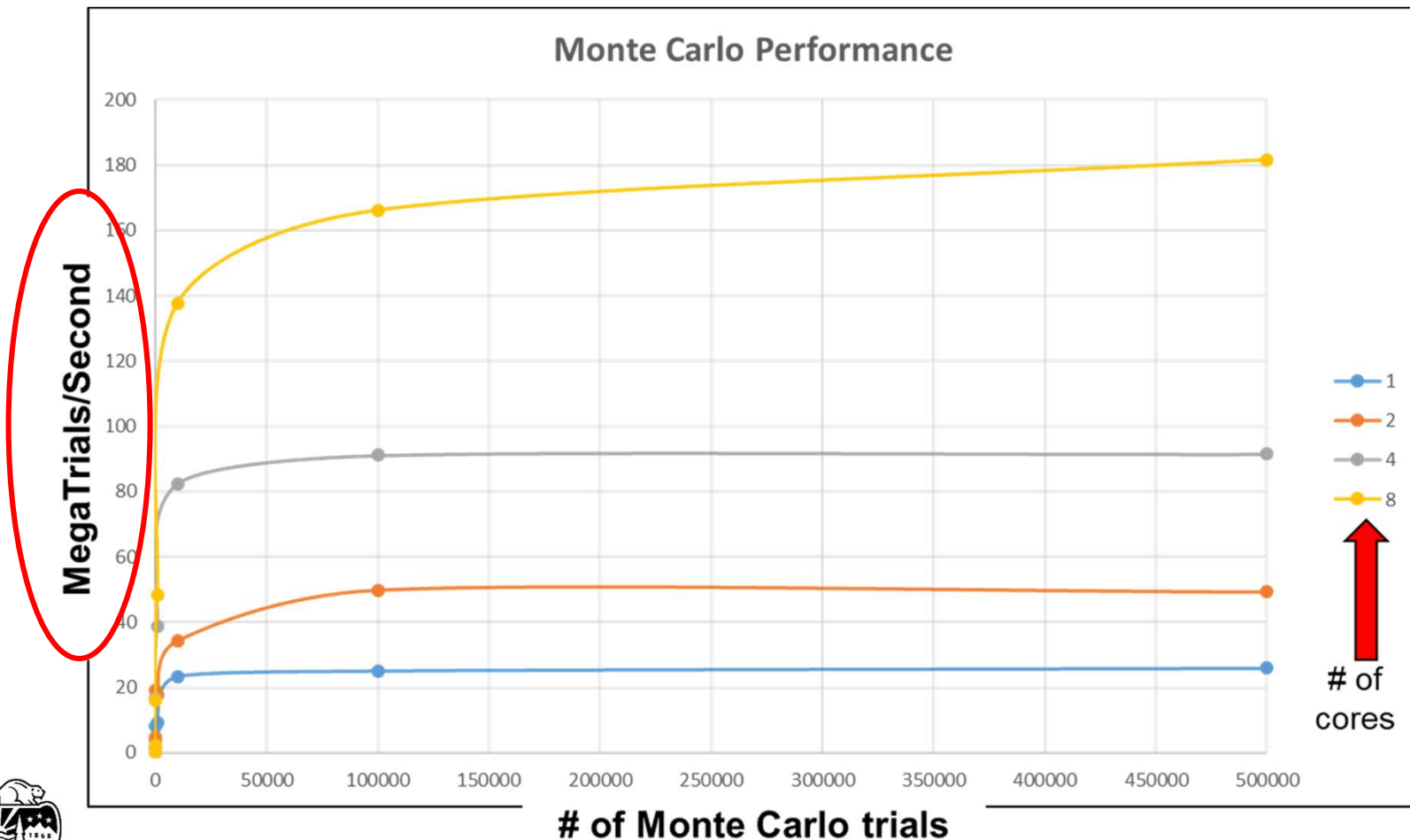
**This Data is actually a 3D Surface Plot –
The 2D Graphs are actually sets of 2D slices through the 3D Surface**



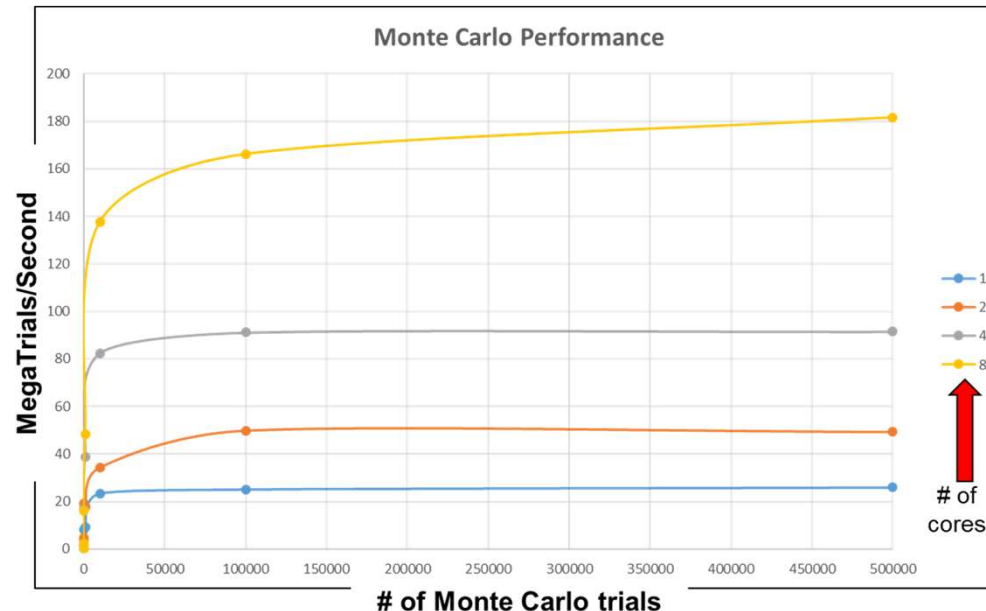
Making Graphs

16

When we plot, we will all put **execution performance on the Y axis** (as opposed to putting elapsed time on the Y axis). Thus, as far as performance goes, up will mean “good”. So, for example:



Making Graphs



As you can tell, these performance measurements will be far more intelligible when examined as a graph than as raw numbers. ***Thus, you are expected to have access to a good automated graphing package.*** If you don't have one, or can't get access to one – go get one!

Hand-drawn graphs, whether analog or digital, will not be accepted for your assignments.



You will also need a word processor, with a way to import your tables and graphs, and with a way to turn that document into PDF.

Setting up Your Benchmarks to run from Scripts: #1 -- the #define Approach

There are always advantages to not hardcoding constants into the middle of your program and, instead, setting them with a `#define` at the top where you can find that value and change it easily, like this:

```
#include <stdio.h>
#include <math.h>

#ifndef NUMT
#define NUMT      8
#endif

#ifndef NUMS
#define NUMS     32
#endif
```

Then, in the C or C++ program, all you have to do is use `NUMT` to set the number of threads, like this:

```
omp_set_num_threads( NUMT );
```

But, the use of the `#ifndef/#endif` construct has other advantages. It lets you either run this as a standalone program or run many occurrences of the program from a *script*.

Setting up Your Benchmarks to run from Scripts: #1 -- the #define Approach

In our project assignments, you will run benchmarks, that is, you will try your application using several different combinations of parameters. Setting these combinations by hand inside your program one-by-one is a time-consuming pain.

Your time is more valuable than that. Try doing it from a script.

In most C and C++ compilers, there is some mechanism to set a **#define** from outside the program. Many of them use the **-D** construct on the command line:

```
#!/bin/csh

#number of threads:
foreach t ( 1 2 4 6 8 )
    echo NUMT = $t
    g++ -DNUMT=$t prog.cpp -o prog -lm -fopenmp
    ./prog
end
```

Then, in the C or C++ program, all you have to do is use NUMT. For example:

```
omp_set_num_threads( NUMT );
```

This lets you automatically run your program 5 times with 1, 2, 4, 6, and 8 threads.

Setting up Your Benchmarks to run from Scripts: #1 -- the #define Approach

20

You can also test multiple parameters from the same script by nesting the loops. This one is done using **C Shell (csh)**:

```
#!/bin/csh

# number of threads:
foreach t ( 1 2 4 6 8 )
    echo NUMT = $t
    # number of subdivisions:
    foreach s ( 2 4 8 16 32 64 128 256 512 1024 2048 3072 4096 )
        echo NUMS = $s
        g++ -DNUMS=$s -DNUMT=$t prog.cpp -o prog -lm -fopenmp
        ./prog
    end
end
end
```

Or, in *bash* (Bourne-again Shell) ...

21

```
#!/bin/bash

# number of threads:
for t in 1 2 4 6 8
do
    echo NUMT = $t
    # number of subdivisions:
    for s in 2 4 8 16 32 64 128 256 512 1024 2048 3072 4096
    do
        echo NUMS = $s
        g++ -DNUMS=$s -DNUMT=$t prog.cpp -o prog -lm -fopenmp
        ./prog
    done
done
```



```
import os

for t in [ 1, 2, 4, 6, 8 ]:
    print "NUMT = %d" % t
    for s in [ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 3072, 4096 ]:
        print "NUMS = %d" % s
        cmd = "g++ -DNUMS=%d -DNUMT=%d prog.cpp -o prog -lm -fopenmp" % ( s, t )
        os.system( cmd )
        cmd = "./prog"
        os.system( cmd )
```



Setting up Your Benchmarks to run from Scripts: #2 -- the Command Line Arguments Approach

Instead of this:

```
#include <stdio.h>
#include <math.h>

#ifdef NUMT
#define NUMT      8
#endif

#ifdef NUMS
#define NUMS      32
#endif
```

Do this:

```
#include <stdio.h>
#include <math.h>

int NUMT = 8;

int NUMS = 32
```

Then, in the C or C++ program, all you have to do is use NUMT to set the number of threads, like this:

```
omp_set_num_threads( NUMT );
```

But, the use of the `#ifndef/#endif` construct has other advantages. It lets you either run this as a standalone program or run many occurrences of the program from a *script*.

argc and argv

When you write in C or C++, your *main* program, which is really a special function call, looks like this:

```
int main( int argc, char *argv[ ] )
{
    . . .
```

These arguments describe what was entered on the command line used to run the program.

The **argc** is the number of arguments (the arg **C**ount)

The **argv** is a list of argc character strings that were typed (the arg **V**ector).

The name of the program counts as the 0th argv (i.e., argv[0])

So, for example, when you type

ls -l

in a shell, the *ls* program sees argc and argv filled like this:

argc = 2

argv[0] = "ls"

argv[1] = "-l"



argc and argv

So, if NUMT and NUMS are global int variables:

```
int NUMT = 2;  
int NUMS = 32;
```

and you want to set them from the command line, like this:

```
./prog 1 64
```

Then, *inside your main program*, you would say this:

```
if( argc >= 2 )  
    NUMT = atoi( argv[1] );
```

```
if( argc >= 3 )  
    NUMS = atoi( argv[2] );
```

The if-statements guarantee that nothing bad happens if you forget to type values on the command line.

The *atoi* function converts a string into an integer (“ascii-to-integer”). If you ever need it, there is also an *atof* function for floating-point.



shared() in the *#pragma omp* Line

Also, remember, since NUMS is a variable, it needs to be declared as *shared* in the *#pragma omp* line:

```
#pragma omp parallel for default(none) shared(NUMS,xcs,ycs,rs,tn) reduction(+:numHits)
```

NUMT does not need to be declared in this way because it is not used in the for-loop that has the *#pragma omp* in front of it.

Setting up Your Benchmarks to run from Scripts: #2 -- the Command Line Arguments Approach

In our project assignments, you will run benchmarks, that is, you will try your application using several different combinations of parameters. Setting these combinations by hand inside your program one-by-one is a time-consuming pain.

Your time is more valuable than that. Try doing it from a script.

In most C and C++ compilers, there is some mechanism to set a **#define** from outside the program. Many of them use the **-D** construct on the command line:

```
#!/bin/csh

g++  prog.cpp -o prog -lm -fopenmp

#number of threads:
foreach t ( 1 2 4 6 8 )
    echo NUMT = $t
    ./prog $t
end
```

Then, in the C or C++ program, all you have to do is use NUMT. For example:

```
omp_set_num_threads( NUMT );
```

This lets you automatically run your program 5 times with 1, 2, 4, 6, and 8 threads.

Setting up Your Benchmarks to run from Scripts: #2 -- the Command Line Arguments Approach

You can also test multiple parameters from the same script by nesting the loops. This one is done using **C Shell (csh)**:

```
#!/bin/csh

g++  prog.cpp -o prog -lm -fopenmp

# number of threads:
foreach t ( 1 2 4 6 8 )
    echo NUMT = $t
    # number of subdivisions:
    foreach s ( 2 4 8 16 32 64 128 256 512 1024 2048 3072 4096 )
        echo NUMS = $s
        ./prog $t $s
    end
end
end
```

Or, in *bash* (Bourne-again Shell) ...

29

```
#!/bin/bash

g++ prog.cpp -o prog -lm -fopenmp

# number of threads:
for t in 1 2 4 6 8
do
    echo NUMT = $t
    # number of subdivisions:
    for s in 2 4 8 16 32 64 128 256 512 1024 2048 3072 4096
    do
        echo NUMS = $s
        ./prog $t $s
    done
done
```



```
import os

cmd = "g++ prog.cpp -o prog -lm -fopenmp"
os.system( cmd )

for t in [ 1, 2, 4, 6, 8 ]:
    print "NUMT = %d" % t
    for s in [ 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 3072, 4096 ]:
        print "NUMS = %d" % s
        cmd = "./prog %d %d" % ( s, t )
        os.system( cmd )
```



I Don't Recommend That You Put These Loops in the Program!

31

I know what you're thinking.

You're thinking:

Those scripts are a pain, and I've never done them before. So, I'll just build the iterations through all the parameters into for-loops in the program.

Don't! I have seen evidence that the first time OpenMP does anything, it also does some one-time setups. This will mess up your timing because your first test will seem slower than it should be and the others will seem faster than they should be.

I recommend you run the program *separately* for each combination of parameters. (The script code in the previous pages shows that.)

When computing performance, be sure that the **numerator is amount of work done** and the **denominator is the amount of time it took to do that work**. For example, in the Bezier surface example, computing one height is the work done at each node and you have NUMS*NUMS total nodes, so (NUMS*NUMS)/dt is one good way to measure performance.

NUMS, NUMS*NUMS, 1./dt, and NUMS/dt are not good ways to measure performance as they don't reflect the true amount of **work done per time**.

If you are using ridiculously high values for NUMS, the quantity NUMS*NUMS might overflow a normal 32-bit int. You can use a long int, or just float each one separately. Instead of (float)(NUMS*NUMS)/dt, you could say (float)NUMS*(float)NUMS/dt

If you are squaring a size number, and are using signed ints, the largest NUMS you can use is:

$$\sqrt{2,147,483,647} = 46,340$$

If you are squaring a size number, and are using unsigned ints, the largest NUMS you can use is:

$$\sqrt{4,294,967,295} = 65,535$$



Project Turn-in Procedures

33

Your project turnins will all be electronic.

Your project turnins will be done at <http://engr.oregonstate.edu/teach> and will consist of:

1. Source files of everything (.cpp, .cl, .cuda)
2. A Linux or Windows executable file, if needed.
3. A report in PDF format. ***You can .zip everything else if you want, but please leave the PDF as a separate file.***

Electronic submissions are due at 23:59:59 on the listed due date.

Your PDF report will include:

1. A title area, including your name, email, project number, and project name.
2. Any tables or graphs to show your results.
3. An explanation of what you did and *why it worked the way it did*. Your submission will not be considered valid unless you at least attempt to explain why it works the way it does.

Your project will be graded and the score posted to the class web page. Any loss of points will be explained in a Canvas grade comment.

Projects are due at 23:59:59 on the listed due date, with the following exception:

Each of you has been granted **five** Bonus Days, which are no-questions-asked one-day project extensions which may be applied to any project, subject to the following rules:

1. No more than **2** Bonus Days may be applied to any one project
2. Bonus Days cannot be applied to tests
3. Bonus Days cannot be applied such that they extend a project due date past the start of Test #2.

To use one or more Bonus Days on a given project:

- You don't need to let me know ahead of time.
- Turn-in promptness is measured **by date**. Don't worry if *teach* tells you it's late because it is between 23:30:01 and 23:59:59. But, *after* 23:59:59 on the posted due date, **it's late!**
- *teach* has been instructed to accept your turn-in, no matter when you do it.
- I will run a script to identify the projects that will have Bonus Days deducted
- You can see how many **Bonus Days** you have **Left** by looking in the **BDL** column of the grading table on the class web site.

A Warning About Virtual Machines

36

Virtual machines are, apparently, not automatically setup to do multithreading.

If you are running on your own virtual machine, and are getting performance numbers that make absolutely no sense, try using one of the OSU machines.

Some of you will end up having strange, unexplainable problems with your csh scripts or .cpp programs. This could be because you are typing your code in on Windows (using Notepad or Wordpad or Word) and then running it on Linux. Windows likes to insert an extra carriage return ('\r') at the end of each line, which Linux interprets as a garbage character.

You can test this by typing the Linux command:

od -c loop.csh

which will show you all the characters, even the '\r' (carriage returns, which you don't want) and the '\n' (newlines, which you do want).

To get rid of the carriage returns, enter the Linux command:

tr -d '\r' < loop.csh > loop1.csh

Then run loop1.csh

Or, on some systems, there is a utility called *dos2unix* which does this for you:

dos2unix < loop.csh > loop1.csh

Sorry about this. Unfortunately, this is a fact of life when you mix Windows and Linux.

