I OpenMP and MPI

Week of January 28, 2018

While plenty of interesting work can be done on a single core, a key part of scientific computing is expanding a calculation across multiple CPU cores within a single node as well as across multiple nodes connected by network fiber.

Outside of the context of scientific computing, CPUs having threads cores is good for MIMD—multiple instruction multiple data—tasks. In other words, each thread is doing its own thing. In general, though, there are far more threads (or, more appropriately, processes) running on a machine than cores. It's easy to check this on your own machine.

You can find out how many cores your machine has by going to the command line and typing:

```
cat /proc/cpuinfo
```

Amongst a much of information, you can count how many cores your machine has. Now, to check how many processes you have *running* on your machine, run the command:

```
top
```

Each line corresponds to a different process—you'll see there are a lot of them, many of which are idle, but your web browser (and, if you're me, Spotify) are taking up a good deal of resources. You can exit `top` by pressing `q`. Your computer handles there being far more processes than threads by *context switching*—basically storing the relevant memory contents, restoring the state of some other process, switching back, etc.

You can do the same thing in your VM. If you have your VM set up with the default settings, when you run `cat /proc/cpuinfo` you'll see a single core. If you close your VM and go to its settings, under Systems, then Processors, you can change the number of cores (note: if you have only two cores on your machine, don't give your VM both of them—it's bad news). After rebooting your VM, you'll see you have two cores.

Let's hop back to scientific computing. In general, your applications are going to be SIMD—single instruction, multiple data—or, in other words, you want each thread performing the *same* instruction on *multiple* data points in parallel.

We're going to look at three examples of this across three levels of parallelism. The three examples will be everyone's favorite "Hello world!", summing two vectors of numbers into a third (i.e., $\vec{z} = \vec{x} + \vec{y}$), and performing a reduction over vectors (or a dot product, i.e., $\alpha = \vec{x} \cdot \vec{y}$). We'll test this is serial code, then with parallel code using OpenMP on a single node with multiple cores, and a bit later across more than one node using MPI on BU's shared computing cluster SCC. (At some point, we'll also get MPI going on your VMs for simplicity during code development.) We'll also see how to time routines in `C++11`.

II Hello world! in serial and with OpenMP

Go on the class git and grab the files `scal_hello.cpp` and `omp_hello.cpp` from the directory `/ReferenceCode/n00HelloWorld`. For the purpose of following the text here, comment out line 17 (`#pragma omp barrier`) of `omp_hello.cpp`.

The scalar code in `scal_hello.cpp` shouldn't scare you. You can compile it with

```
gcc -O2 scal_hello.cpp -o scal_hello
```

where `-O2` refers to the optimization level (not relevant here), and the name after the `-o` refers to the filename for the executable.

Next, let's hop over to the OpenMP version of this file. There are a handful of changes relative to the original version. First off: we've included an extra header for OpenMP functions! Once we head into the `main` function, we declare two variables:

- `nthreads`: this will eventually contain how many parallel threads there are in total.
- `thread_id`: this will contain a thread's *individual, unique* id number.

We next enter a parallel block, which will always be denoted by

```
#pragma omp parallel [...]
```

OpenMP follows a *fork-join* paradigm for parallelization. At the start of the program, there's only one thread. When the compiler hits a code block that starts with a parallel pragma, it inserts routines to spawn the multiple threads work is parallelized over. At the end of the pragma, it kills the extra threads, and we're back to the one thread we started with. (This spawning and destruction of threads has a constant cost overhead! We'll get to that later, but in short, it means it's a waste to parallelize only a small amount of work.)

Carrying on—the prama contained two extra arguments, `shared(nthreads)` and `private(thread_id)`. These have straightforward interpretation. A *shared* variable is shared between each thread—in other words, the variable `nthreads` points to the same memory location in each thread. On the other hand, when a variable is *private*, each thread maintains its own copy of the variable and is free to modify it independent of any other thread. Any variable defined *within* a parallel OpenMP block is implicitly private.

The function `omp_get_thread_num()` returns each thread's unique id. We then have the *first* thread call `omp_get_num_threads()` to learn how many threads there are *in the current code block*. If you call this function outside of a parallel block, it'll return one! In any case, since the total number of threads is a constant independent of which thread we're in, we have just one thread learn the value, and since `nthreads` is a shared variable, every other thread can reap the benefits of that knowledge... in principle.

Last (since we've commented out the barrier line), we print each thread's number.

To give this program a whirl, compile it as

```
gcc -O2 omp_hello.cpp -o omp_hello -fopenmp
```

Where we note the key addition of `-fopenmp`. If you run this program as is, you'll probably get the output:

```
Hello world from thread 0 of 1!
```

But wait! I thought we're trying to write threaded code. With OpenMP, you specify the number of threads you want via an *environment variable* on the command line. For example, to spawn 4 threads, run:

```
export OMP_NUM_THREADS=4;
```

And run the program again. Now, you might see something like:

```
Hello world from thread 1 of 4!
Hello world from thread 2 of -1!
Hello world from thread 0 of 4!
Hello world from thread 3 of -1!
```

There are two things of note here!

1. The threads (most likely) printed out of order: the order that operations happen is, as determined by a stopwatch, generally is not guaranteed. This is a feature, not a bug.
2. Some of the threads see `nthreads` as its default value of `-1`! Why?

The issue here is, while the first thread stopped to "learn" about the total number of threads, other threads pressed on and may have hit the `printf` statement before the *shared* variable was updated with the value `4`. If you uncomment the barrier line and recompile, you'll see this problem goes away. That's because a `barrier` forces all of the threads to synchronize before proceeding, and in this case, this means that the shared variable `nthreads` gets updated before any thread prints anything.

III Timing

One of the most valuable tools you have while developing optimized code is timing routines—if you don't know how long your code takes to run, do you really know if you've meaningfully optimized it?

Reference timing code lives on a repository separate from the class repository:

```
https://github.com/weinbe2/utilities-esw/blob/master/timing-with-chrono/
timing_reference.cpp
```

The key points of note are:

- The `#include <chrono>` at the top.

- Lines 28 through 30, which explain and initialize the timing kernels.
- Lines 39 through 47, which finishes the timing and convert it to seconds.

Of course, all of the comments and spacing makes this verbose—everything related to timing fits into four lines total.

The `chrono` library is part of the `C++11` standard, which means you need to add some extra flags when compiling with older versions of `gcc`. (The version on your VM doesn't strictly require it, but for robustness you should add it to any scripts anyway.) In total, compilation proceeds with

```
gcc -std=c++11 -O2 timing_reference.cpp -o timing_reference
```

Let's combine these timing kernels with OpenMP. First, run this executable as is to get a timing baseline. Next, let's add OpenMP pragmas. Whenever you operate on an array of data *without* some type of reduction, optimization is relatively simple. The for loops get modified with:

```
for (int t = 0; t < 10; t++) {
#pragma omp parallel for
for (int i = 0; i < length; i++) {
sum[i] = arr1[i] + arr2[i];
}
}
```

The keyworks `parallel for` denote, unsurprisingly, that the for loop should be parallelized. Give this code a whirl, remembering to add `-fopenmp`. If you're *not* running on your VM (since, for some reason there it doesn't work), you should see a factor of two speed up for `export OMP_NUM_THREADS=2` relative to `export OMP_NUM_THREADS=1`.

A question you might ask—why is there this extra loop over `t`? This goes back to how OpenMP has a fork-join model. Try setting the loop over `t` to just go up to one iteration. You probably won't see a difference in time between one and two threads. This extra outer loop amoritizes the constant overhead of the fork-join.

IV Parallel Reductions

To motivate parallel reductions, we're first going to talk about rough approximations to integrals. As a reminder, the intergral of some function $f(x)$ from -1 to 1 is written as

$$\int_{-1}^{1} f(x)\, dx \tag{1}$$

Graphically, an integral is the area under a curve over an interval. A really sloppy way of approximating an integral (even worse than the trapezoidal rule that you probably learned in calculus) is to split the integral into N intervals, and then perform a weighted sum over the value at the center of the function in each interval. Since that's an incredibly unclear sentence, let's write it as an explicit function. Let's say we're integrating $f(x)$ from a to b with N intervals. Further, let's have each interval be the same width. In this case, each interval is of length $\frac{b-a}{N}$. In each case, we can access the center of the i'th interval (i going between 0 and $N-1$) with the expression $a + i\frac{b-a}{N} + \frac{1}{2}\frac{b-a}{N}$: the a shifts us to the start of the integral, the $i\frac{b-a}{N}$ shifts us to the left side of the ith interval, and the extra $\frac{1}{2}\frac{b-a}{N}$ shifts us to the center of the interval.

The approximate integral over an interval a to b is given by the sum

$$\int_a^b f(x)\, dx \approx \sum_{i=0}^{N-1} \frac{b-a}{N} f\left(a + i\ \frac{b-a}{N} + \frac{1}{2}\frac{b-a}{N}\right) \tag{2}$$

To cut to the chase, assuming a, b, and N have been set appropriately, and we wish to integrate the `sin` function, this gets implemented in C++ (without timing kernels) as

```
double sum = 0.0;
#pragma omp parallel for reduction (+:sum)
for(int i = 0; i < N; i++) {
sum = sum + sin(a+i*(b-a)/((double)N) + 0.5*(b-a)/((double)N));
}
```

Note that we didn't set any variables as `private` or `public`: we only have to worry about variables that might get *modified* in the parallel block. One variable that did get a modifier is `sum`, which got noted as being a `reduction`—this makes sense, we're performing a reduction in that we're summing over the integral.

For reference, other possibilities for reductions are `*`, as well as the `<math.h>` functions `min` and `max`, and some bitwise operators. (There's apparently some way to define a custom reduction, too, if you Google for it.)

Under the hood, when this program gets run (with `OMP_NUM_THREADS` set to something greater than 1), each thread performs its own accumulation, and then at the end only a few numbers have to be summed between the threads.

V Hello MPI! and Accessing SCC

For the time being, we haven't figured out how to install MPI on the VM (sorry), but we do have access to SCC! To access `SCC`, open a terminal and run:

```
ssh [usename]@scc1.bu.edu
```

Your username is your BU account name. Once you're on `SCC`, your `bash` command prompt will include your username:

```
[(username)@scc1 ~]$
```

If that's roughly what it looks like, you're all set. You can navigate to our class project's active directory by running

```
cd /projectnb/paralg/
```

If you haven't yet, create your own directory with your kerberos username. If you'd like to play around with compiling and running code, you can log into an interactive shell by running:

```
qrsh -l h_rt=1:00:00 -P paralg
```

The last argument, `-P paralg`, is very important if you're a part of other projects on SCC—the PI on the project probably doesn't want you billing core hours for this class to their research allocation! Even if you aren't a part of other projects, it's a good habit to get into.

That flag aside, there's a wonderful collection of information on BU's IS&T website for running jobs on SCC: `http://www.bu.edu/tech/support/research/system-usage/running-jobs/`. We'll look at some of the other arguments in due time. The only other relevant command for now is requesting an interactive shell with more than one core. In this case, there's one additional flag, `-pe omp 4`, in that case requesting four cores. Altogether, you'd request such a setup with

```
qrsh -l h_rt=1:00:00 -pe omp 4 -P paralg
```

In this case, though, we're going to focus on MPI. There's a reference MPI "Hello world!" program in the class repository in the same directory as the reference scalar and OpenMP versions of the code. There's more going on here, though, so let's go through the relevant lines.

```
int world_size; // number of MPI processes
int world_rank; // rank of MPI process
char processor_name[MPI_MAX_PROCESSOR_NAME]; // name of processor
int name_len; // length of name string.
```

These variables are related to the number of processes, the process number, as well as the name of the machine a process is running on. We'll go through them as we hit the functions that initialize their respective values. In any case, any MPI program begins with

```
// Initialize MPI
MPI_Init(&argc, &argv);
```

Which just initializes the program. `argc` and `argv` are just the variables that get filled with command line arguments as defined at the top of your `main` program—you're being a good programmer and including those, right?

Next up, we learn the total number of processes and store it in the variable `world_size`. This is analagous to a call to `omp_get_num_threads()` in an OpenMP program. There's a key distinction in it being *processes*, though—if you run `top` while an MPI program is running, you'll see a distinct line for each process, unlike when there are multiple threads in an OpenMP program! The relevant call here is:

```
// Get the number of processes
MPI_Comm_size(MPI_COMM_WORLD, &world_size);
```

There's a processor define here that we'll see multiple times, `MPI_COMM_WORLD`. This is the default "world" all of the processes live in. For some applications, you may want to define multiple "worlds", each of which containing a subset of the processes. For the applications in this class, though, we (probably) won't worry about that.

Next, we learn the current process' unique id and store it in the variable `world_rank`. This is analagous to a call to `omp_get_thread_num()` in OpenMP except, again, that we're talking about processes, not threads. The relevant call here is:

```
// Get the rank
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
```

Last, we can learn about the process' name. This is more a useful diagnostic tool than anything else.

```
// Get the name of the processor
MPI_Get_processor_name(processor_name, &name_len);
```

We can then do some work with this information, run the program, and at the end let MPI clean itself up with the following call:

```
// Clean up
MPI_Finalize();
```

Alright, cool! Now let's try to compile and run this program. As a note, the `C++11` timing routines that we're using in this class doesn't play nicely with the default version of OpenMPI (one implementation of the MPI standard) that's on SCC. To fix that, you'll need to run the commands

```
module load gcc/5.3.0
module load mpich/3.2
```

To compile the code, run

```
mpicxx -std=c++11 mpi_hello.cpp -o mpi_hello
```

Now, with this version of MPI, you can technically just run the program on the command line. This isn't generically the case, though. Instead, you invoke the program using, for example,

```
mpirun -np 2 ./mpi_hello
```

If you're on a login node (i.e., you didn't use `qrsh` as described above), don't abuse this–you really shouldn't be using a login node for anything more than compiling code, and even then you might want to use an interactive shell for a program that takes a long time to compile. That said, though, we'll describe what the above call might print out:

```
Hello world from rank 1 of 2!  My name is scc1
Hello world from rank 0 of 2!  My name is scc1
```

Again, this might be in the other order—it's that same old social contract of running parallel programs. We see here we're running two processes—this corresponds with the `-np 2` in the `mpirun` call—and because we're running on the `scc1` login node, we get a name we expect!

VI Submit Scripts

The more appropriate way to run MPI jobs, and more honestly, any (non-interactive node) job is through *submit scripts*. The reference submit script for the MPI hello world program lives in the github at `ReferenceCode/n00HelloWorld/submit_mpi_hello.sh`. The script is largely well self-documented, and there is further information available at

`http://www.bu.edu/tech/support/research/system-usage/running-jobs/submitting-jobs/`

There are a few specific things deserving some extra comments.

- Line 1, "`#! /bin/bash -l`": This tells the machine to execute the script using `bash`, and the `-l` is necessary to use `module load` commands in the script.
- Line 4, "`$$ -l h_rt=2:00:00`": The `#$` makes this a special type of comment—the script ignores it, but the job queueing system looks at it. In this case, this line tells the job queueing system that this job should run for a max of two hours.
- Line 7, "`#$ -N submit_hello`": This is the name that shows up in third column of `qstat`.
- Line 16, "`#$ -pe mpi_4_tasks_per_node 8`": This is the real magic. The use of `mpi_4_tasks_per_node` implies running four processes on a given node. (8, 12, 16, and apparently awkwardly 28 also work.) The later `8` corresponds to requesting 8 processes *total*, so in this case, 4 processes per node across 2 nodes. In general, the later number must be an integer multiple of the former.
- Line 26, "`exec > ${SGE_O_WORKDIR}/${JOB_NAME}-${JOB_ID}.scc.out 2>&1`"
  - This command lets you redirect your output to a file with a custom filename, which is useful relative to the default `.o[NUMBER]` at the end.
  - `${SGE_O_WORKDIR}`: A custom environment variable that points to the program working directory.
  - `${JOB_NAME}`: The job name specified on line 7.
  - `${JOB_ID}`: The job number given when you submit a job.
- Line 30, "`mpirun -np $NSLOTS ./mpi_hello`": This is where the job ultimately gets run. `$NSLOTS` is an environment variable that contains the total number of processes—8, in this case—and then `./mpi_hello` is the executable name, which you'd replace as appropriate.

Whew! With all that set, it becomes really easy to run your job on the command line. Simply run:

```
qsub submit_mpi_hello.sh
```

So it's ultimately easy! One last note—if you don't want to hard code some of those `#$` arguments directly into the submit script, you don't have to. For example, you could remove line 16 (related to setting the MPI nodes), and equivalently modify your submit command to:

```
qsub -pe mpi_4_tasks_per_node 8 submit_mpi_hello.sh
```

In other words, just put everything after the `#$` in the submit script on the command line instead.

VI.1 Subdiving Problems with MPI

With OpenMP, as noted earlier, your entire application still runs in one process. Further, the compiler does the work in getting the application to sub-divide work for you. This is a great simplification: when you want to parallelize a `for` loop, for example, you don't have to query the number of threads and explicitly subdivide parts of each loop to each thread.

In contrast, MPI runs multiple processes, which means you *do* need to sub-divide the work by hand. When we put MPI routines into the vector addition code and timed it. After putting MPI into the vector addition code, but without making any other changes, we timed running the code on four processes within a node (`-pe mpi_4_tasks_per_node 4`) vs eight processes within a node (`-pe mpi_8_tasks_per_node 8`). Up to the effect that the clock speed and memory bandwidth on the 8-core nodes might be different from that on the 4-core nodes, the timing was about the same.

In general, the different processes have only one way to figure out what portion of the work to take: their rank number, which is the result of the call to `MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);`.

As an explicit example, let's say a scalar code did some (unspecified) work on some array of length `N` where each element is set equal to its index. More specifically:

```
double arr1[N];
for (int i = 0; i < N; i++) {
arr1[i] = (double)i;
}
```

The OpenMP version of this code is a simple, straightforward modification:

```
double arr1[N];
#pragma omp parallel for
for (int i = 0; i < N; i++) {
arr1[i] = (double)i;
}
```

There are more factors in mind for the MPI version. First, when we split over multiple processes, we *subdivide* the memory requirements between the processes. Instead of one process having an array of length `N`, each process has its own array of length `N/world_size`. Further, each process needs to fill its segment of the array appropriately. In a four process setup, the zeroth process fills its array with 0 to `N/world_size-1`, the first fills its array with `N/world_size` to `2*N/world_size=1`, etc. These are the types of problems you have to think about with MPI!

Given this, the MPI version of the code (assuming `world_size` and `world_rank` are set appropriately) is:

```
double arr1[N/world_size];
for (int i = 0; i < N; i++) {
arr1[i] = (double)(i+world_rank*N/world_size);
}
```

Of course, this doesn't become relevant until you do something with the split array afterwards, like a reduction... so on that note, let's get back to integrals!

VII Homework Assignment 2

For the first real problem, we're going to numerically calculate, and time, a single- and multi-dimensional integral parallelized with MPI and, in the multi-dimensional case, OpenMP.

VII.1 MPI Integration

For your first assignment, you're going to parallelize the one-dimensional integral over `sin` from 1 to 2 using MPI. So everyone can start on the same page, we've posted a single core version of the integral code on the class github. **You do not have to base your entire MPI code on this version, only the output format.** As long as you do the assignment correctly and your code isn't that awful, we're happy. You can assume you only have to worry about the cases of 4, 8, 12, and 16 processes—we'll keep the number of sub-intervals N well behaved so you don't have to worry about integer division rounding.

Similar to the OpenMP case, we're going to speed up approximating this integral by splitting the work over multiple processes. Again, unlike the OpenMP case, we need to manually split the subintervals between each process: assuming four processes and an integral from $a = 1$ to $b = 2$, the zeroth process should perform the integral from 1 to 1.25, the first from 1.25 to 1.5, etc. The *total* number of sub-intervals should still be N. In other words, assuming infinite precision, the final result should be the same independent of if there is one MPI process, two MPI processes, four, and so on. (Of course, because we're working in finite precision, the results won't be exactly the same.)

There's one more piece you need: the analogue of `reduction(+:sum)` for MPI. In this case, it's the function `MPI_Allreduce`. (Alternatively, since only one process ultimately needs the result—the process printing the result—you can use the the function `MPI_Reduce`.)

Let's say each process accumulates its *own* contribution of the integral into the variable `sum`, and then we want to reduce the sum over each process into the variable `global_sum`. The relevant call is

```
MPI_Allreduce(&sum, &global_sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

where each argument corresponds to, in order,

1. `&sum`: The contribution of the integral from each individual process.
2. `&global_sum`: The variable in which to store the sum over each process.
3. `1`: How many variables to sum over. Technically, `sum` could be an array of 10 (or however many) numbers per process. This call would then put the reduction, element by element, into the array `global_sum`. The benefit of performing 10 sums over the network at once instead of 10 separately is to save *latency*: the time overhead of sending messages over the network. This is similar to the constant fork-join overhead in OpenMP.

4. `MPI_DOUBLE`: An indication that we're summing over `doubles`. There's also `MPI_FLOAT`, etc, as appropriate.

5. `MPI_SUM`: That our final reduction is a sum. There are other possibilities, too.

6. `MPI_COMM_WORLD`: As always, the global world including all of the processes.

The purpose of this assignment is to successfully implement code which integrates $\sin(x)$ from $a = 1$ to $b = 2$ using $N = 30000000$ (that's thirty million). As with the reference scalar code, define these as variables at the top of the program—I should be able to change a to, say, 34, and b to, say, 37, and get the correct (approximate) answer.

Your bottom two lines of your output, with the parts in square brackets appropriately replaced with the relevant numbers, should read (only once!):

```
Hello from [processor_name]
The integral from [a] to [b] of sin(x) using [N] intervals is [global_sum]
With [world_size] processes, this took [time_seconds] seconds.
```

We're only going to test this using multiple processes on a single node, so `processor_name` is unambiguous. For the next part of the assignment, we're going to plot "how long the integral took" as a function of the number of processes—I'll get to the quotes in a moment.

To keep every process within a single node, use the following tasks arguments in the submit script:

- 4 processes: `-pe mpi_4_tasks_per_node 4`
- 8 processes: `-pe mpi_8_tasks_per_node 8`
- 12 processes: `-pe mpi_12_tasks_per_node 12`
- 16 processes: `-pe mpi_16_tasks_per_node 16`

Next—how long the integral took. As we learned in class, the actual amount of time an integral takes depends on the processor speed. Further, the machines with more cores per node are likely newer and likely have a higher clockspeed (or lower, if it's a more energy efficient, lower clock speed processor), which could give you a bigger (or smaller) time than expected. To solve this, we need to divide the time by the clock speed of the processors you ran the code on—this will give an "effective" amount of clock cycles your code took to run as a function of the number of processes. You could find this info out by running `cat /proc/cpuinfo` as part of your submit script, *or* you could take advantage of the fact you printed out the `processor_name` as part of your output and look up the clock speed on this page:

`https://www.bu.edu/tech/support/research/computing-resources/tech-summary/`

Technically, we should be averaging these numbers over multiple runs and plotting with error bars, but we'll get to that a bit later. For now, just make sure your plot looks sane (and if you can't get it to look sane, send me a message on Piazza, or an e-mail, or send smoke signals), remember to set your axes appropriately, and use a log scale as appropriate! To make it clear, the x axis should be the number of processes, and the y axis should be the effective number of cycles—the time divided by the processor speed.

For this part of the assignment, you need to submit your source file, your submit script (don't worry about which number of processes you leave at the top), and your figure.

VII.2 Multi-dimensional Integration

So far we've worried about one-dimensional integrals—in particular, we've been obsessing over the integral

$$\int_a^b \sin(x)\ dx \tag{3}$$

This can be extended to multi-dimensional integrals. A geometric interpretation for a two-dimensional integration, for example, is the area under a *surface* as opposed to a curve. The integral of some function $g(x, y)$ over the rectangle defined by $a < x < b, c < y < d$ can be written as:

$$\int_{x\in[a,b]y\in[c,d]} g(x,y)\ dA \tag{4}$$

I'm sure this is different from how you may have seen it in a multivariable calculus class. It's actually a non-trivial proof that this is equivalent, but a much clearer formulation (and likely familiar, if not more verbose than you're used to) is:

$$\int_c^d \left[\int_a^b g(x,y)\ dx\right]\ dy \tag{5}$$

Now that it's written in this form, it becomes a bit easier to see how to numerically approximate it. Assuming we have the same number of sub-divisions, N, in each dimension, we first replace the inner sum, then the outer sum, following

$$\int_c^d \left[\int_a^b g(x,y)\ dx\right]\ dy \approx \int_c^d \left[\sum_{i=0}^{N-1} \frac{b-a}{N} g\left(a + i\ \frac{b-a}{N} + \frac{1}{2}\frac{b-a}{N}, y\right)\right]\ dy \tag{6}$$

$$\approx \sum_{j=0}^{N-1}\sum_{i=0}^{N-1} \left(\frac{d-c}{N}\right)\left(\frac{b-a}{N}\right) g\left(a + i\ \frac{b-a}{N} + \frac{1}{2}\frac{b-a}{N}, c + j\ \frac{d-c}{N} + \frac{1}{2}\frac{d-c}{N}\right) \tag{7}$$

This approximation generalizes to even higher dimensions. For the first part of this problem, we're going to have you compute the following two integrals using $N = 5000$ in both the x and y direction:

$$\int_{-1}^1 dy \int_{-1}^1 dx\ \left[x^2 + y^2 + (y-1)^3(x-3)\right] = ? \tag{8}$$

$$\int_{-1}^1 dy \int_{-1}^1 dx\ \left[e^{-x^2-\frac{y^2}{8}} \cos(\pi x)\sin(\frac{\pi}{8}x)\right] = ? \tag{9}$$

You should write two versions of these integrals: an OpenMP version and an MPI version. To get good timings (i.e., don't run on the login node), you'll need to use an OpenMP submit script. A reference submit script for the vector submission example lives in the class github in the file `/ReferenceCode/n01TimingOpenMP/submit_omp_add.sh`. In both cases, remember to present timings in terms of "effective clock cycles" as we noted above.

As a note: in the case of OpenMP, there isn't a function similar to `MPI_Get_processor_name`. Instead, we explicitly print the `$HOSTNAME` environment variable within the submit script `submit_omp_add.sh`, see line 30. (I'm not sure if it's guaranteed to be true, but my guess is `MPI_Get_processor_name` is probably referring to the `$HOSTNAME` environment variable under the hood, at least on Unix/Linux bases systems.)

The output from the first integral, with emphasis on the second and third line as far as grading is concerned (you just need the first line to learn the clock frequency of the node you're on) should look like:

```
Hello from [processor_name]
The integral of the first function is [global_sum]
With [world_size] processes, this took [time_seconds] seconds.
```

While for the second integral, it should look like:

```
Hello from [processor_name]
The integral of the second function is [global_sum]
With [world_size] processes, this took [time_seconds] seconds.
```

In total, you should have four curves worth of data—two functions times OpenMP vs MPI. Plot each of these curves on the *same* plot, labeling the legend appropriately. To make it clear, the x axis on this figure should be the number of processes, and the y axis should be the effective number of cycles—the time divided by the processor speed.

For this part of the assignment, you need to submit an OpenMP and an MPI source file (for either function), your submit script (don't worry about which number of processes you leave at the top), and the one figure with all four curves.

VIII Submitting Your Assignment

This assignment is due at 11:59 pm on Wednesday, February 14. Please e-mail a **tarball** containing the assignment to the class e-mail, one clearly labeled subdirectory per part of the problem, to bualghpc@gmail.com. Include your name in the tarball filename.

If you're not familiar with tar, here's a sample instruction that perhaps I would use:

```
tar -cvf evan_weinberg_asgn2.tar [directory with files]
```

You may want to include other files. **Do NOT include a compiled executable!** That's a dangerous, unsafe practice to get into.