# Hello, World!

Dr. Christopher Marcotte — *Durham University*

As with every programming course, the first thing we will do is compile and run a “Hello world” program… or *three*.

***Hint***

Take a look at the Hamilton guide under “HPC: Week 0” if you haven’t already.

## A serial version

Log in to Hamilton[1] and load the relevant compiler modules with `module load gcc`. To open and edit a text file from the command line, use `nano serial.c` to create a *C* file `serial.c` whose content matches mine:

**serial.c**

```c
#include <stdio.h>

int main(void)
{
  printf("Hello, World!\n");
  return 0;
}
```

Having done that you should compile the code using `gcc serial.c -o hello-serial` or `cc serial.c -o hello-serial`.

The outputs an executable named `hello-serial` from the `serial.c` file.

Run the compiled executable on the login node using `./hello-serial` which will output `Hello World!` in the terminal.

As you saw in the Hamilton guide, we should not run our code on the login nodes, and actually submit jobs to run on the compute nodes. To do this, we need to create a submission script – create a file `serial.slurm`:

**serial.slurm**

```bash
#!/bin/bash

# 1 core
#SBATCH -n 1
#SBATCH --job-name="hello-serial"
#SBATCH -o hello-serial.%J.out
#SBATCH -e hello-serial.%J.err
#SBATCH -t 00:01:00
#SBATCH -p test

module load gcc

./hello-serial
```

Since we only have a small job, we have requested the test queue with the line `#SBATCH -p test`. This has a fast turn-around, but does not allow you to submit long-running or large simulations; i.e. it is meant for testing.

Submit your job with `sbatch serial.slurm` at the command line. After running, this should create two files named `hello-serial.<NUMBERS>.out` and `hello-serial.<NUMBERS>.err` in your current directory, where

[1] Note: Hamilton 8 – not any other version.

`<NUMBERS>` corresponds to the job id. Verify these files exist by writting `ls hello-serial.*.{err,out}` at the command line. Despite not running our job on the login node, the files generated and written to the compute node home directory are synced back to the login node.

You can see the contents of these by opening them with `nano`, or `cat`-ing them to the screen. The error file should be empty: `cat hello-serial.*.err` will show no output, and the output file should contain the string `Hello, World!`.

***Exercise***

It's sometimes a good idea to compile your code in your batch script, to ensure the environment for compiling and running the executable is the same. Move the compilation step into the batch script `serial.slurm` and submit it with `sbatch`. Does it still work?

## An OpenMP version

One of the parallelisation paradigms we will see in this course is shared memory parallelism, for which we use OpenMP. OpenMP is a specification for compiler directives and library routines that can be used to specify parallelism at a reasonably high level for Fortran, C, and C++ programs.

To compile OpenMP programs, we need the same modules as for the serial programs above. Our code now looks a little different. Create a file `openmp.c` with the content below:

**openmp.c**

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

int main(void)
{
  int nthread = omp_get_max_threads();
  int thread;
#pragma omp parallel private(thread) shared(nthread)
  {
    thread = omp_get_thread_num();
    printf("Hello, World! I am thread %d of %d\n", thread, nthread);
  }
  return 0;
}
```

The line beginning with `#pragma` is likely mysterious to you for now – we will develop the tools to understand this line in the next few exercises.

If we try and compile this like we did before (`gcc -o hello-openmp openmp.c`), we will get an error:

```
/usr/bin/ld: /tmp/ccjUdH2z.o: in function `main':
openmp.c:(.text+0xd): undefined reference to `omp_get_max_threads'
/usr/bin/ld: openmp.c:(.text+0x15): undefined reference to `omp_get_thread_num'
collect2: error: ld returned 1 exit status
```

The linker complained that there were two undefined functions: `omp_get_max_threads()` and `omp_get_thread_num()`. OpenMP support is implemented in most modern compilers, but must be explicitly requested. To do so we must add an additional flag to our compile command:

`gcc -fopenmp -o hello-openmp openmp.c`.

Our submission script also looks a little different. Create a file `openmp.slurm`:

**openmp.slurm**

```
#!/bin/bash

# 1 node
#SBATCH --nodes=1
#SBATCH --cores=2
#SBATCH --job-name="hello-openmp"
#SBATCH -o hello-openmp.%J.out
#SBATCH -e hello-openmp.%J.err
#SBATCH -t 00:01:00
#SBATCH -p shared

module load gcc

# unnecessary, since we've specified --cores=2 above
export OMP_NUM_THREADS=2
./hello-openmp
```

We select the amount of parallelism by setting the `OMP_NUM_THREADS` environment variable before running the executable, though this is not strictly necessary since we have specified the number of cores with `#SBATCH --cores=2`.

After submitting our job with `sbatch openmp.slurm` we again get some output files (running `ls hello-openmp.*.{err,out}`) yields

```
hello-openmp.3186885.err  hello-openmp.3186885.out
```

We can inspect the contents as before `cat hello-openmp.*.out`:

```
Hello, World! I am thread 0 of 2
Hello, World! I am thread 1 of 2
```

Note how the individual threads are numbered from zero and live in the interval $[0, 2)$.

***Exercise***

Try changing the number of threads. What do you notice about the output? Can you change `openmp.c` and the submission script so that we use more threads and only the thread with the highest index says “hello”?

## An MPI version

The other parallelization paradigm we will use is for programming *distributed memory* systems. We will use MPI for this. MPI is a specification for a library-based programming model. The standard specifies Fortran and C/C++ interfaces, and there are wrappers for many popular programming languages including Python and Julia.

To compile MPI programs, we need to load, in addition to the previous compiler module, an MPI module. This can be done similarly to before, e.g. `module load gcc openmpi` to load both `gcc` and OpenMPI.

***Hint***

MPI modules require a compatible compiler to be loaded, first. You can verify this by clearing the loaded modules (`module purge`) and trying to load the `openmpi` module before the `gcc` module.

Our program code looks different from both the serial and OpenMP versions. Make a file `mpi.c`:

**mpi.c**

```c
#include <mpi.h>
#include <stdio.h>

int main(void){
  MPI_Init(NULL, NULL);
  int rank, size, len;
  MPI_Comm comm;
  char name[MPI_MAX_PROCESSOR_NAME];
  comm = MPI_COMM_WORLD;
  MPI_Comm_rank(comm, &rank);
  MPI_Comm_size(comm, &size);
  MPI_Get_processor_name(name, &len);
  printf("Hello, World! I am rank %d of %d. Running on node %s\n", rank, size, name);
  MPI_Finalize();
  return 0;
}
```

Notice how compared to the OpenMP version, there are no `#pragma`s and the parallelism is not explicitly annotated, there are just calls to some library functions from MPI.

If we try and compile with `gcc  -o hello-mpi mpi.c`, we get errors:

```
mpi.c:2:10: fatal error: mpi.h: No such file or directory
    2 | #include <mpi.h>
      |          ^~~~~~~
compilation terminated.
```

To compile this file, we need to tell the compiler about all the MPI-relevant include files and libraries to link. Since this is sometimes complicated, MPI library implementors typically ship with *compiler wrappers* that set the correct flags. On Hamilton these are named `mpicc` (for the MPI wrapper around the C compiler), `mpicxx` (for the wrapper around the C++ compiler), and `mpif90` (for the Fortran wrapper). Since we have a C source file, we should use `mpicc`. Our correct compilation command is `mpicc -o hello-mpi mpi.c`.

Running the executable is now also more complicated, as MPI executables use an explicit launcher. We need to use `mpirun` to run an MPI executable. This takes care of allocating parallel resources and setting up the processes so that they can communicate with one another.

You should create a submission file `mpi.slurm` containing:

**mpi.slurm**

```bash
#!/bin/bash

# 1 node
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=2
#SBATCH --job-name="hello-mpi"
#SBATCH -o hello-mpi.%J.out
#SBATCH -e hello-mpi.%J.err
#SBATCH -t 00:01:00
#SBATCH -p shared

module load gcc
module load openmpi

mpirun -np 2 ./hello-mpi
```

***Hint***

For larger jobs you may need to change which queues you submit to on Hamilton. For more than one node, use the shared queue `#SBATCH -p shared`. You can get information about all the available queues with the command `sinfo`. A summary of the queue utilization is also available with the command `sfree`.

On some systems you need to specify the number of processes you want to use when executing `mpirun`. However, on Hamilton, the metadata in the scheduler is used to determine the number of processes. Here we request 1 node (`#SBATCH --nodes=1`) and 2 tasks per node (`#SBATCH --ntasks-per-node=2`). Hence you should only explicitly specify the parallelism of your `mpirun` command if you want to run with parallelism different to that specified in your submission script.

***Exercise***

Compile and run the MPI program on Hamilton using 1 node and 2 tasks per node. Try running on two compute nodes, by changing the `#SBATCH --nodes=1` line to `#SBATCH --nodes=2`. How many total processes do you now have? What do you notice about the node names? How many nodes could you run on while maintaining the total number processes as in the original submission script above?

## Parallel Scaling

Throughout this course, we will compare the scaling of some code in either the *strong* (Amdahl) or *weak* (Gustafson) sense. You have two datasets associated with this exercise, one showing representative times for strong scaling of a program (`strong.dat`) and one showing representative times for weak scaling of a different program (`weak.dat`).

| **Threads** $p$ | **Runtime** $t(p)$ |
|---:|---:|
| 1 | 64.424242 |
| 2 | 33.901724 |
| 4 | 17.449995 |
| 8 | 8.734972 |
| 16 | 4.789075 |
| 32 | 2.749116 |
| 64 | 1.627157 |
| 128 | 1.017307 |
| 256 | 1.436728 |

Table 1: Strong scaling data.

| **Processes** $p$ | **Runtime** $t(p)$ |
|---:|---:|
| 1 | 0.582497 |
| 2 | 0.627571 |
| 4 | 0.724130 |
| 8 | 0.787381 |
| 16 | 0.893998 |
| 32 | 0.968348 |
| 64 | 1.030421 |
| 128 | 1.257982 |
| 256 | 2.408056 |

Table 2: Weak scaling data.

***Hint***

The values in Table 1 and Table 2 come from different programs.

### Strong Scaling

As discussed in the lecture, Strong Scaling concerns itself with the latency of a program – the time-to-solution for a fixed-size problem. Recall Amdahl's Law,

$$t(p) = t(1) \cdot f + t(1) \cdot (1 - f)/p, \tag{1}$$

which describes the time $t(p)$ it takes to run a fixed-size problem across $p$ processors, given it takes time $t(1)$ on one processor, with serial fraction $f$.

**Exercise**

Consider Table 1, and Equation 1. Fit the thread-timing pairs $(p, t(p))$ to Amdahl's Law to estimate the serial fraction $f$. Plot the speedup data $(p, t(1)/t(p))$ with the functional form of speedup, Equation 1, using your fit serial fraction. Is it a good fit – why? What is the serial fraction? If you had infinitely many threads, what is the maximal speedup achievable by this program according to Equation 1?

## Weak Scaling

Weak scaling concerns itself with the *throughput* of a program – the amount of computation achievable in a fixed time. Recall Gustafson's Law,

$$t(1) = t(p) \cdot f + t(p) \cdot (1 - f) \cdot p, \tag{2}$$

which describes the time it *would* take for a $p$-scaled program running on $p$ processors in time $t(p)$ with a *known* serial fraction $f$ to complete on a single process, $t(1)$.

Fitting Gustafson's Law is more subtle than fitting Amdahl's Law. For weak scaling, you again have a dataset labeling processes and runtimes, $(p, t(p))$, for a range of $p$ but the *meaning* of the runtime is different – the speedup implied by Equation 2 is comparing the runtime on $p$ processes to the *$p$-scaled problem runtime on a single process*. Thus, one can not fit a single value for $t(1)$ to the whole dataset – each $p$ implies a differently sized problem, and thus a different implicit $t(1)$.

In the simplest case, one simply runs each $p$-scaled problem on a single process, and now we have a dataset $(p, t(p), t(1))$ which can be used to fit Equation 2 with the unknown parameter $f$. However, because one is typically concerned with the solution of large problems in weak scaling experiments, you frequently do not (and, practically, *can* not) determine what the runtime for these $p$-scaled problems *is* on a single process. Instead, one must measure something else

**Hint**

If this is confusing, let's try to expand the notation using an interesting synthesis of these two scaling laws. Let $T(n, p)$ be the runtime of an $n$-scaled program on $p$ processes. Then $T_s(n) = \beta(n, p) \cdot T(n, p)$ is the serial time of the $n$-scaled program running on $p$ processors, i.e. $\beta(n, p)$ is the proportion of the total runtime of an $n$-scaled program running on $p$ processors which is essentially serial. Likewise, $T_p(n, p) = T(n, p) - T_s(n)$ is the runtime not accounted for by the serial time, i.e. the parallel time. We note that $T_s(n)$ is independent of $p$ – i.e., the time it takes to process the essentially serial portion of the program depends only on the size of the program ($n$) and not on the number of processors we are running it on ($p$), in principle.[2] In this notation, Amdahl's and Gustafson's Laws take the form,

$$\begin{aligned} T(n, p) &= T_s(n) + T_p(n, 1) \cdot p^{-1} && (3(1) \\ &= \beta(n, 1) \cdot T(n, 1) + (1 - \beta(n, 1)) \cdot T(n, 1) \cdot p^{-1}, && (3(2) \end{aligned}$$

and

$$T(n, 1) = T_s(n) + T_p(n, p) \cdot p \qquad (4(1)$$

[2] This is not strictly true – e.g. we could design a program which must handle some essentially serial work with the number of processors, like global adaptive load-balancing – but assuming $T_s(n)$ is independent of $p$ is certainly reasonable in most cases.

$$= \beta(n,p) \cdot T(n,p) + (1 - \beta(n,p)) \cdot T(n,p) \cdot p, \qquad \text{(4(2)}$$

respectively.

We see that Table 2 contains the values of $p$ and $T(p,p)$ – the runtime of the $p$-scaled problem running on $p$ processors. In the weak scaling context, what we normally seek is $f$, the serial fraction of the runtime for the $p$-scaled problem on $p$ processors $f \equiv \beta(p,p)$.

To determine $f$ without recourse to actually running each $p$-scaled problem on a single process and recording the time, one should measure the serial runtime $t_s(p)$ in the course of the $p$-scaled problem, as well as the total run time $t(p)$. Then determining $T_s(n) = \beta(n,p) \cdot T(n,p)$ gives us a way to estimate $\beta(p,p)$, and thus $f$.

## *Exercise*

Consider Table 2, and Gustafson's Law, Equation 2.

Where $t(p)$ is the run time for a process-scaled problem across $p$ processes, with an *a priori* unknown serial fraction $f$. Can you fit the process-timing pairs $(p, t(p))$ to Gustafson's Law to estimate the serial fraction $f$? Why, or why not?

Create an additional column representing the serial runtime, $t_s(p)$ for each $(p, t(p))$ pair according to the formula

$$t_s(p) = 0.4 \cdot t(1) + \xi \cdot p, \quad \xi \sim U\left[0, 10^{-2}\right]. \qquad (5)$$

The serial runtime $t_s(p)$ is something you can measure for a code you can modify. Use your generated serial runtimes to estimate the speedup according to Equation 2; what is the serial fraction? Plot the speedup data $(p, t(1)/t(p))$ with the functional form of speedup using your serial fraction. Is it a good fit – why? If you had infinitely many threads, what is the maximal speedup achievable by this program according to Equation 2?

## *Challenge*

Look at the "Universal Scaling Law"📗,

$$X(N) = \frac{\gamma N}{1 + \alpha(N-1) + \beta N(N-1)}$$

and try to fit the data in Table 1 and Table 2 to this model of computational scalability. Do you get a better or worse fit? Do the parameter meanings enable you to make a diagnosis for the performance of these unknown codes?

## *Aims*

- Introduction to compilation of *C* code.
- Introduction to usage of SLURM and the batch system.
- Introduction to library inclusion (`#include <omp.h>`, `#include <mpi.h>`).
- Introduction to different parallelism paradigms.
- Practice with estimating serial fractions from timing data using Amdahl's and Gustafson's scaling laws.

## *References*

[1] N. J. Gunther, "A general theory of computational scalability based on rational functions," *arXiv preprint arXiv:0808.1431*, 2008.

📗Gunther, Neil J. "A General Theory of Computational Scalability Based on Rational Functions." *Arxiv Preprint Arxiv:0808.1431*, 2008