# MPI + OpenMP

Dr. Christopher Marcotte — *Durham University*

In this exercise we'll look at combining MPI & OpenMP for simultaneous shared and distributed memory parallelism. This parallelism model incorporates shared memory work coordinated by OpenMP threads, as well as distributed memory work coordinated by MPI processes. Generally, this approach yields significantly more complex programs than utilizing either MPI or OpenMP alone, as we will see; the benefit is the ability to solve *even larger* problems, while using all available resources.

The first step is to identify a problem worth parallelising, for which we will turn back to the calculation of $\pi$. But first, we need to get to terms with the newly complicated parallel heirarchy.

## *Exercise*

Consider the program code below:

***hybrid.c***

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

int main(int argc, char *argv[])
{
    int numprocs, rank, namelen, iam, np;
    char processor_name[MPI_MAX_PROCESSOR_NAME];





    {


        printf("Hybrid: Hello from thread %d out of %d from process %d out of %d on %s\n",
            iam, np, rank, numprocs, processor_name);
    }

    MPI_Finalize();

    return 0;
}
```

Add lines to `hybrid.c` to print the number of OpenMP threads, thread ID, number of MPI processes, and process ID (rank), and correctly tag each variable as `shared` or `private`.

Compile and run `hybrid.c` to investigate the coordination of the parallel hierarchy. You should have `nprocs * nthreads` reporting in your output. How does oversubscribing with MPI processes (i.e. using the command-line flag `-oversubscribe`) change the output?

### *Solution*

A correct implementation is given in `hybrid-solution.c`. Note the share attributes for all the variables: the OpenMP thread variables are private, while the MPI process variables are not. In this sense, MPI provides stronger guarantees about value correctness, in some sense, by sacrificing shared memory parallelism.

***hybrid-solution.c***

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

int main(int argc, char *argv[])
{
    int numprocs, rank, namelen, iam, np;
    char processor_name[MPI_MAX_PROCESSOR_NAME];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &namelen);

    #pragma omp parallel default(none) shared(rank, numprocs, processor_name) private(iam,
np)
    {
        np = omp_get_num_threads();
        iam = omp_get_thread_num();
        printf("Hybrid: Hello from thread %d out of %d from process %d out of %d on %s\n",
            iam, np, rank, numprocs, processor_name);
    }

    MPI_Finalize();

    return 0;
}
```

My output of the command `OMP_NUM_THREADS=4 mpirun -np 2 ./hybrid` is

```
Hybrid: Hello from thread 1 out of 4 from process 1 out of 2 on DA-NW72WVDH6H
Hybrid: Hello from thread 2 out of 4 from process 1 out of 2 on DA-NW72WVDH6H
Hybrid: Hello from thread 0 out of 4 from process 1 out of 2 on DA-NW72WVDH6H
Hybrid: Hello from thread 3 out of 4 from process 1 out of 2 on DA-NW72WVDH6H
Hybrid: Hello from thread 0 out of 4 from process 0 out of 2 on DA-NW72WVDH6H
Hybrid: Hello from thread 1 out of 4 from process 0 out of 2 on DA-NW72WVDH6H
Hybrid: Hello from thread 2 out of 4 from process 0 out of 2 on DA-NW72WVDH6H
Hybrid: Hello from thread 3 out of 4 from process 0 out of 2 on DA-NW72WVDH6H
```

If I oversubscribe, even by a single process more than cores available, the program will complete all the printing but exit with a

```
--------------------------------------------------------------------------
A system call failed during shared memory initialization that should
not have.  It is likely that your MPI job will now either abort or
experience performance degradation.
```

Such is the pleasure of working with MPI.

***Hint***

Compiling MPI + OpenMP programs is sometimes more complex than compiling either OpenMP or MPI programs. If we naively run

```
mpicc -fopenmp -o hybrid -Wall hybrid.c
```

(`-Wall` means "show all warnings") on our own machine, we might get a warning that that the pragma is ignored[*]. Running `mpicc -showme` will give you the complete compile command being invoked. This is it on my mac:[†]

```
gcc -I/opt/homebrew/Cellar/open-mpi/4.1.6/include \
    -L/opt/homebrew/Cellar/open-mpi/4.1.6/lib \
    -L/opt/homebrew/opt/libevent/lib -lmpi
```

Meaning the correct compile command on my mac is:

```
gcc -I/opt/homebrew/Cellar/open-mpi/4.1.6/include \
    -L/opt/homebrew/Cellar/open-mpi/4.1.6/lib \
    -L/opt/homebrew/opt/libevent/lib -lmpi \
    -fopenmp -lgomp -o hybrid hybrid.c
```

While on Hamilton, the usual `mpicc -fopenmp -o hybrid -Wall hybrid.c` should work fine[‡]. If you are interested in manually expanding the command, this is it on Hamilton:

```
gcc -I/apps/developers/libraries/openmpi/4.1.1/1/gcc-11.2/include \
    -pthread -Wl,-rpath \
    -Wl,/apps/developers/libraries/openmpi/4.1.1/1/gcc-11.2/lib \
    -Wl,--enable-new-dtags \
    -L/apps/developers/libraries/openmpi/4.1.1/1/gcc-11.2/lib \
    -lmpi -fopenmp -lgomp -o hybrid hybrid.c
```

up to version changes.

The executable can be run with (on either system) with the command:

```
OMP_NUM_THREADS=4 mpirun -np 2 ./hybrid
```

## Estimating $\pi$, again

In `pi.c` we have yet another estimation of $\pi$, this time using MPI+OpenMP for parallelism. The code uses the same technique as our OpenMP program, evaluating the integral $\int_0^1 4\left(1+x^2\right)^{-1}\,\mathrm{d}x$ using a large number of samples.

***Exercise***

Using different combinations of processes and threads, constrained by `nprocs * nthreads = 8`, find the *fastest* configuration and the configuration with the highest (estimated) *throughput* of the code for `NPTS = 10000000`.

[*] Or `clang: error: unsupported option '-fopenmp'`!
[†] Note the \ is just for a line break and not a part of the command.
[‡] Provided you've run `module purge; module load gcc openmpi`

Perform a weak and a strong scaling analysis of the code performance, using *transferable* quantification of performance over a serial program (feel free to use `OMP_NUM_THREADS=1 mpirun -np 1 ./hybrid` to estimate serial performance).

***Solution***

For my mac, I find the fastest combination with 8 MPI processes and 1 OpenMP threads (per-process) – I suspect this is a reflection of the high bandwidth of the M1 Pro rather than a truly transferable conclusion.

```
processes = 8, threads = 1, NPTS = 10000000, pi = 3.1415951647805969, error = 2.511191e-06
time = 0.004532, estimated MFlops = 88261.252337 (88261.252337)
```

## Calculating integrals in 3D

In previous exercises we've dealt with low-dimensional approximation of integrals for estimating $\pi$, if only because it is mathematically simple and pedagogically useful. But, of course, if your function of interest is $f : \mathbb{R} \to \mathbb{R}$, then only when the intermediate dimension is large do we need to think of high-performance computing. Frequently, we want to estimate integrals of functions which are much more general $f : \mathbb{R}^m \to \mathbb{R}^n$, where $m, n$ are input and output dimensions and at least one of them is quite large.

***Exercise***

Now you should consider how you must modify the program `pi.c` to estimate the integral of an unknown function in three dimensions, using both MPI and OpenMP, efficiently. That is, your new program should compute integrals of the form,

$$I = \int_\Omega f(\boldsymbol{x})\,\mathrm{d}\mu(\boldsymbol{x}),$$

where $\boldsymbol{x} \in \Omega \subset \mathbb{R}^d$ and $f : \mathbb{R}^d \to \mathbb{R}$; the volume element is given by $\mathrm{d}\mu(\boldsymbol{x})$ (i.e., it is $r\,\mathrm{d}r\,\mathrm{d}\varphi\,\mathrm{d}z$ in cylindrical and $\mathrm{d}x\,\mathrm{d}y\,\mathrm{d}z$ in Cartesian coordinates).

***Hint***

For simplicity, your code should assume that $\Omega = [0,1]^d$, i.e., the unit $d$-cube, and use Cartesian coordinates.

***Exercise***

Apply your adapted program to the function $f(\boldsymbol{x}) = r^{\frac{2}{3}} \sin\left(\frac{2}{3}\varphi\right) z$, using the transformation $r = \sqrt{\boldsymbol{x}^\top \boldsymbol{x}}$ and $\varphi = \operatorname{atan2}(y, x)$ to cylindrical coordinates from Cartesian $\boldsymbol{x} = (x, y, z)$.

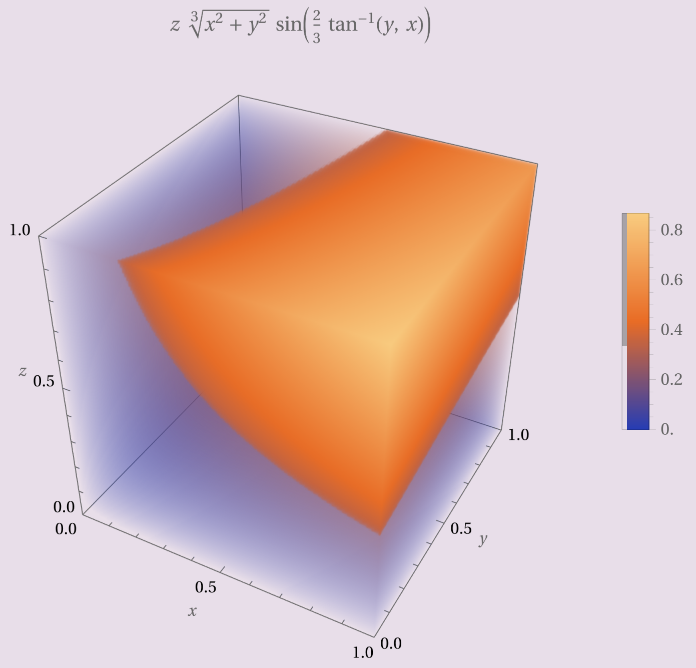

*Figure 1: Plot of the function $f(\boldsymbol{x})$ over the domain $\Omega = [0,1]^3$.*

How close is your numerical approximation? At what rate do you converge (with the number of samples)?

If you implemented this using Monte Carlo, how quickly would your integral converge? If you implement the Golden sampling scheme from an earlier exercise, should you expect to converge faster or slower than the strict Monte Carlo approach?

If we extend the region to $\Omega = [0,1] \times [-1,1] \times [1,2]$, how must your program adapt? Why would we still restrict $x \in [0,1]$ and $z \geq 0$? What about for implicitly defined regions, e.g., the unit-radius cylinder whose axis is the line segment connecting $\boldsymbol{x} = (0,0,0) \to (0,0,10)$?

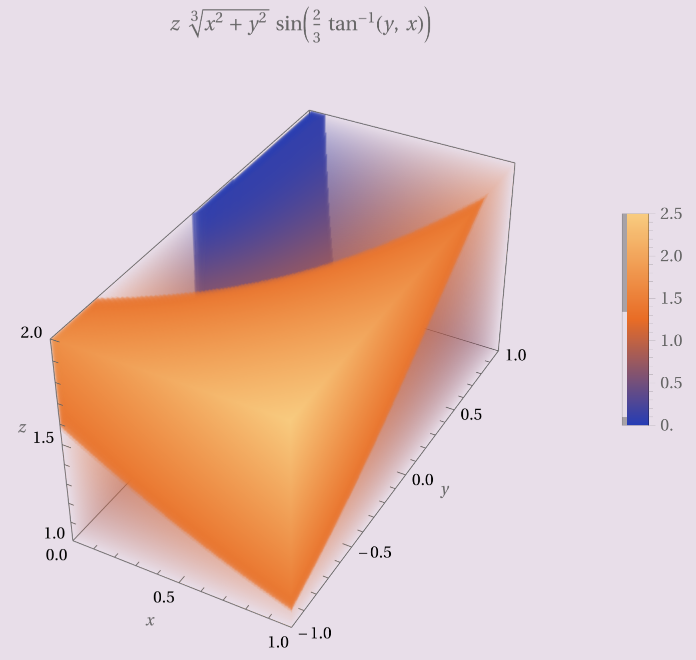

*Figure 2: Plot of the function $f(\boldsymbol{x})$ over the domain $\Omega = [0,1] \times [-1,1] \times [1,2]$.*

***Solution***

The analytic integral for the function $f$ over $\Omega = [0,1]^3$ is $I = \left(\frac{9}{160}\right)\left(1 + 2 \times 2^{\frac{1}{3}}\right) \approx 0.197991...$; How did you do?

The Monte Carlo approach should still converge with error as $O\left(N^{-\frac{1}{2}}\right)$. Golden Sampling does not have a proof for low-discrepancy in dimensions higher than $d = 2$ (as far as I know...), so you may or may not converge at all! If you actually implement this, then please share it with me... I will be *impressed*.

The extension to a different domain, you should include a function $g$ which scales the samples from the unit $d$ -cube to the function domain of $f$ so that $g : [0,1]^d \to [\lfloor \boldsymbol{x} \rfloor, \lceil \boldsymbol{x} \rceil]$, where $[\lfloor \boldsymbol{x} \rfloor, \lceil \boldsymbol{x} \rceil]$ designates the outer product of the lower and upper bounds of the integration domain, and also include the inverse for scaling the volume element, $d\mu(\boldsymbol{x})$. For implicit regions the simplest approach is to include a branch in your function code $f$, that tests for inclusion of the sample point in the integration region.

The analytic integral of $f$ over $\Omega = [0,1] \times [-1,1] \times [1,2]$ is $I = \left(\frac{81}{160}\right)\left(1 + 2 \times 2^{\frac{1}{3}}\right)$; how accurately did you estimate it?

We don't permit $z < 0$ because the function is antisymmetric, i.e. $f(r, \varphi, z) = -f(r, \varphi, -z)$ (and also antisymmetric in $x$), so if permit the domain to be symmetric, then the integral will be 0, identically; this is a good test of the accuracy of your program. I don't know of a reasonable way to test for symmtries in the input function rigorously... I suppose you could just test both $\boldsymbol{x}$ and $-\boldsymbol{x}$ for both inclusion in the integral domain, and also for negated function evaluation, and then after some exactly negated number of samples, but this is probably not worthwhile.

***Hint***

Remember that integration is a linear operator, so if $f$ expects inputs between $[\boldsymbol{x}_l, \boldsymbol{x}_u]$, then scaling your samples in $[0,1]^d$, $\boldsymbol{x} \to \boldsymbol{x} \cdot (\boldsymbol{x}_u - \boldsymbol{x}_l) + \boldsymbol{x}_l$ (i.e., to the expected domain) should be valid, so long as you account for it in the final weighting. You could even implement this in your code as a function composition, where if $f : [\boldsymbol{x}_l, \boldsymbol{x}_u] \to \mathbb{R}$, then $(f \circ g)(\boldsymbol{x}) : [0,1]^d \to \mathbb{R}$.