AME 60614: Numerical Methods
Fall 2021

Problem Set 0
Due: September 9, 2021

Prof. Jonathan F. MacArt

Submission guidelines: A hard copy of written work is due in class. Additionally, submit through Sakai an archive (`tar` or similar) of all files requested in this assignment. Place files from individual problems into separate folders within your archive, *i.e.*, `p1/`, `p2/`, etc. Please name this archive in the form `lastname_firstname_ps0.tar`.

Contact `jmacart@nd.edu` with any questions.

1 Code Documentation

1. Download the document `sample_ps0.tex` and `sample_figure.eps` from the course Sakai page. Rename the file `lastname_firstname_ps0.tex`, and edit the report to add your name. Compile it using `pdflatex` or the LaTeX editor of your choice.
2. Document your response to Problem 2 in your report. Use appropriately highlighted code listings for codes requested in Problem 2. Print a copy and bring it to class on the due date.

2 Computing in Parallel

The Mandelbrot set is defined as the collection of points $c = (x, iy)$ on $\Omega = [-2, 0.5] \times [-2i, 2i] \subset \mathbb{C}$ for which the iterative sequence

$$z_{n+1}(c) = z_n^2(c) + c$$

remains bounded, where $z_n(c) \in \Omega$ is the value of the sequence at a point c for iteration n, $n \in [0, \infty)$, and $z_0 = 0$.

It may be shown that the sequence becomes unbounded if

$$|z_n(c)| > 2$$

for any n. To determine if a point is a member of the Mandelbrot set, one must check each point in the domain under the infinite sequence and exclude those points for which $|z_n(c)| > 2$. In practice, this involves creating a grid of finite resolution and iterating on each point a maximum of `MAX_ITER` times.

2.1 Assignment

a. Using a scripting language of your choice, determine the points in a $10\,000 \times 10\,000$-point discretization of Ω that belong to the Mandelbrot set. Test your code's performance using `MAX_ITER` $= 10$, 100, and 1000, and report the runtime in your document. In terms of your algorithm, explain why a scripting language is ill-suited to this task. How could its performance be improved? Explain how you might go about this. Plot the computed Mandelbrot set at lower resolution.

Note: There are several ways to plot the Mandelbrot set; an Internet search should provide a good overview of how.

b. Translate your scripting-language program to a compiled language of your choice. Your program should save the computed set as binary data, which you should then read into a separate plotting script (e.g., in Python or Matlab). A tutorial on compiled-language programming may be found below.

c. Parallelize your compiled-language code using OpenMP. Refer to the lecture notes and the tutorial below. Numerous additional OpenMP tutorials are available on the web, for example, from Lawrence Livermore National Laboratory: `https://hpc.llnl.gov/tuts/openMP/`.

d. Test the serial and parallel performance of your compiled-language code on a personal machine or the CRC cluster. Compare its serial performance to that of your scripting-language code, and generate strong and weak parallel scaling plots the parallelized code up your machine's maximum number of threads.

e. Include in your `tar` archive your commented source code and a graphical rendering of your results.

f. **Optional bonus:** Parallelize your code using MPI. (An MPI tutorial can be found at `https://hpc-tutorials.llnl.gov/mpi/`.) Consider the following two work-distribution strategies:

- Static domain decomposition and blocking communication: processes divide the domain equally and iterate to problem completion.
- Manager/worker decomposition with nonblocking communication: one "manager" process assigns a small chunk of grid points to "worker" processes. When a "worker" has finished, the "manager" collects its results and assigns more work. This process continues until all grid points have been computed.

Which strategy is advantageous for computing the Mandelbrot set? Which would be more advantageous for, e.g., computational fluid dynamics?

Note: There are multiple ways to manage memory in MPI. In general, one should not allocate the global problem domain Ω on each process, as this memory usage does not scale.

2.2 Tutorial: Programming in C

This tutorial proceeds in three steps:

1. Setting up your programming environment
2. Writing a "hello world" program in C
3. Parallelizing your program using OpenMP

2.2.1 Setting up your programming environment

How you set up your programming environment will depend on your operating system and the type of machine in use. Here, we will focus on installing GCC (the GNU Compiler Collection), but other compilers (Intel, LLVM, PGI, etc.) *should* work if you already have them installed.

First, determine if you already have GCC in your system path. Open a terminal (MacOS/Linux), a command prompt (Windows), or Integrated Terminal (VS Code) and enter `gcc --version`. If it returns configuration information, then great! You already have a compiler installed.

However, you may wish to install a different compiler. For example, the default GCC version on Linux tends to be quite old, and the default compiler on MacOS is actually LLVM. How to do so depends on your system type.

- **MacOS:** I recommend installing/using the Spack package manager (instructions below). Note that your system will likely prompt you to install the XCode development tools.
- **Windows:** Microsoft's VS Code website[1] has some information on using C/C++ with their VS Code Integrated Development Environment (IDE). You may wish to check their instructions on installing and using GCC with Windows.
- **Linux clusters** will already have compilers installed. Use `module avail` to list available modules, and load an appropriate compiler, e.g., `module load gcc` to load the latest version.
- **Personal Linux machines** will already have GCC installed, although it could be old. Update using your system's package manager or Spack.

MacOS: Getting Started with Spack. See the Spack documentation for full details (`https://spack.readthedocs.io/en/latest/getting_started.html`). Open a Terminal and ensure that you are in your home directory (`cd` with no arguments). The basic steps are:

1. Clone the Github repository:

```
git clone https://github.com/spack/spack.git
```

2. Source the appropriate shell script. The default shell is either bash or zsh. For these, use:

```
. spack/share/spack/setup-env.sh
```

   Note that the leading "." is important. If you use a different shell (tcsh, csh, etc.), then look up the appropriate shell script in the documentation.

3. To see available GCC versions, enter `spack info gcc`. Pick a recent version and go with it. For example, to install version 9.2.0, use:

```
spack install gcc@9.2.0
```

4. Finally, load the newly installed version into your environment:

```
spack load gcc
```

   If this works, then the output of `gcc --version` should show the path to your Spack installation.

2.2.2 Writing a "hello world" program in C

The following "hello world" program demonstrates for-loops, print statements (with numerical data formatting), basic integer arithmetic, and array memory allocation.

1. Save the code as `hello.c`
2. Compile using `gcc hello.c -o hello`
3. Run using `./hello`

That's it! You should see several lines printed to the terminal.

[1] https://code.visualstudio.com/docs/languages/cpp

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

int main() {

  // Define an integer counter
  int counter = 0;
  counter = 0;

  // Allocate double-precision data on the heap
  //   Note: *data denotes a pointer to memory
  int N = 10;
  double *data = malloc(N*sizeof(double));

  // Loop to perform operations
  for (int i=0; i<N; i++) {
    data[i] = 2.5*i;
    printf("Hello world %d %4.3f\n", counter, data[i]);
    counter++;
  }

  // Read the saved data in reverse
  for (int i=N; i>0; i--) {
    printf("%d %4.3f\n", i, data[i-1]);
  }

  // Clean up
  free(data);
  data = NULL;

  return 0;
}
```

2.2.3 Parallelizing your code using OpenMP

Let's now focus parallelizing a single for-loop. Note the following in the program below:

- The OpenMP headers need to be included in C/C++: `#include <omp.h>`. In Fortran, this is usually done by the compiler, so an include statement is unnecessary.
- The code initializes an OpenMP parallel region with "data" shared, but the thread ID (`tid`) is declared as private, for it will be different for each thread in the team.
- The main additions are the `#pragma omp` statements; these direct the compiler that you want these sections to run in parallel. The Fortran syntax for OpenMP pragmas is slightly different, but the usage (shared/private data) is exactly the same.
- Note the intrinsic OpenMP functions `omp_get_thread_num()`, `omp_get_num_threads()`. These are useful if each thread needs to be aware of its position within the team. However, for simple for-loop partitioning, they are usually unnecessary.

Compile the code and run as follows.

1. Save the code as `hello_omp.c`
2. Compile using `gcc -fopenmp hello_omp.c -o hello_omp`. Note that the `-fopenmp` flag is necessary; GCC will complain if you omit it. (For Intel compilers, the flag is `-qopenmp`, and the compiler will not always complain if you omit it—your code will just run in serial!)

3. Run using `OMP_NUM_THREADS=4 ./hello_omp` for 4 threads (adjust to suit your system).

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

int main() {
  int nthreads,tid;

  // Allocate double-precision data on the heap
  int N = 10;
  double *data = malloc(N*sizeof(double));

  // Beginning of parallel region
#pragma omp parallel shared(data) private(tid)
  {
    // Each thread gets its thread ID
    tid = omp_get_thread_num();

    if (tid==0) {
      // Root thread writes the total number of threads to screen
      nthreads = omp_get_num_threads();
      printf("Number of threads is %d\n",nthreads);
    }

    // OMP partitions a loop among all threads
#pragma omp for
    for (int i=0; i<N; i++) {
      data[i] = 2.5*i;
      printf("Hello from thread %d, i=%d, data=%4.3f\n", tid, i, data[i]);
    }

  } // End of parallel region
  printf("End of parallel region\n");

  // Write the contents of 'data' from the root processor
  for (int i=0; i<N; i++) {
    printf("tid=%d, %d %4.3f\n", tid, i, data[i]);
  }

  // Clean up
  free(data);
  data = NULL;

  return 0;
}
```