COSC3500/7502 Assignment 2 - Parallel Programming Techniques

On the COSC git server, you will find a directory for **`Assignment2`**. This contains a driver program that calculates some eigenvalues of an $N \times N$ matrix, where N is specified on the command line.

The eigenvalues are obtained using the popular ARPACK library. Instead of acting directly on a matrix, this library uses a `reverse communication interface', which is a primitive form of callback function. Thus it is required that the user of the library supplies a function that will execute the matrix-vector multiply and pass the result back to the ARPACK library. The driver program handles all of the details of calling ARPACK, and implements the matrix-vector multiply via a function

```
void MatrixVectorMultiply ( double * Y, const double * X)
```

which the user must supply. This function implements the operation

$$y = Mx$$

where x and y are N-dimensional vectors, and M is an $M \times M$ matrix.

The program `Assignment2_serial.cpp` implements a serial version of the matrix-vector multiply. This is a very simple function:

```
// global variables to store the matrix
double * M;
int N;
// implementation of the matrix - vector multiply function
void MatrixVectorMultiply ( double * Y, const double * X)
{
        for (int i = 0; i < N; ++i)
        {
                Y[i] = 0;
                for (int j = 0; j < N; ++j)
                {
                        Y[i] += M[i*N+j] * X[j];
                }
        }
}
```

Your task is to:

(a) Parallelize the matrix-vector multiplication in multiple ways; using
- (i) AVX
- (ii) OpenMP
- (iii) CUDA
- (iv) MPI

(b) Calculate some benchmarks and write a brief report that discusses the speedup and relates this to Amdahl's law and Gustafson's law.

Write three out of four implementations to receive a grade of 75% or below.
Implement all four to receive a grade of 75% and above.

The `Makefile` is set up to correctly compile files named `Assignment2_openmp.cpp` using the appropriate OpenMP flags, `Assignment2_cuda.cu` using the CUDA compiler, and `Assignment2_mpi.cpp` using the MPI compiler. If you use a different build system, then you must include a README _le with your submission that describes how to compile and run your programs.

You can copy the `Assignment2 serial.cpp` program and use this as a template for your parallel programs. It is heavily instrumented to calculate some timing benchmarks. The key timings are the time spent in the multiply function inside the eigensolver (this is the function that you are responsible for), compared with the `Total Serial (initialization + solver)' (you have no control over this, and it is inherently serial code). You should ensure that the matrix is *exactly* the same for each version of your code.

Note: The MPI version will be the most difficult. You should have an attempt at writing an MPI program, but you will be able to pass this assignment even if the MPI program doesn't work correctly, as long as your report and other programs are of high quality.

Additional hints

- The computational cost of the matrix-vector multiply is $O(N^2)$ for an input vector size of N. The number of iterations required by the eigensolver will also increase with N, leading to an overall cost that is closer to $O(N^3)$.
- OpenMP: The OpenMP version of the code should be the simplest to implement. You might be able to extract some additional performance by careful consideration of memory layout, but this isn't so easy the problem is intrinsically memory bandwidth limited.
- AVX: This won't be the prettiest code you've written, but don't be surprised if you get similar performance to OpenMP, even when operating single-thread. Again, because you'll be memory bandwidth limited.
- CUDA: Since the ARPACK library does all of the computations using the CPU, the vectors that you get in the `MatrixVectorMultiply()` function are in CPU memory, so you will need to copy them to the GPU. Where do you want to store the matrix? You might need to modify the initialization code in the `main()` function if you want to copy the matrix somewhere else.
- MPI: The MPI version of the code will need some substantial changes to the template code. Consider how you will construct the matrix, and how you will distribute it among the nodes. There are three main styles; split the matrix horizontally (distribute complete rows to each node), vertically (distribute complete columns to each node), or tiled (distribute a square or rectangular sub-matrix to each node). But make sure that the matrix elements are correct! Although the matrix is random, it is deterministic, and should produce the same eigenvalue spectrum no matter how you parallelize the code. The template code displays the largest eigenvalue, you might want to check some others. (The eigenvalues from ARPACK are sorted by magnitude, so the largest eigenvalues are at the end of the array.) Also for debugging purposes, you could display the individual matrix elements. A key advantage of distributed memory programs is that they can handle larger program sizes; you should strive, if possible, to allow your MPI program to handle matrices that are much larger than the memory allocated for each process. For serial components of the program that you cannot parallelize using MPI, you have a choice of either running it on a single process (e.g. the master process) and distributing the results to other processes as required, or running the serial component on every node. You could do the initialization of the matrix in a similar way on every process because the random number generator is designed to be deterministic (as long as you seed it in exactly the same way on every process!). You cannot assume that the ARPACK library will be deterministic though, so you should *only run the* ***`eigenvalues_arpack`*** *function on one node.*

Background

(This background to the problem is not necessary to understand for the assignment)
The problem that is solved by the code is connected to the theory of random matrices, which has many applications, for example in nuclear physics. A quantum mechanical system is described by an eigenvalue problem

$$H\psi_n = E_n\psi_n$$

where E_n is the n^{th} eigenvalue, and n is the corresponding eigenvector. The operator H describes the physical system, and is a Hermitian operator. Because H is Hermitian, the eigenvalues are real. This operator can be represented by a Hermitian matrix, or, if the operator is entirely real-valued (which generally means it is time-reversal symmetric), by a real-symmetric matrix. Most nuclei have thousands of states and energy levels, and are too complex to be described exactly. Instead, one must settle for a model that captures the statistical properties of the energy spectrum. Instead of dealing with the actual operator H, one can consider a family of random matrices, and compute the distribution of the eigenvalues of these matrices.

The theory of random matrices was pioneered by Eugene Wigner, and one of the key results that he postulated is that the spacing between the energy levels E_n in the eigenvalue spectrum should depend only on the symmetry class of the underlying system. That is, real-symmetric random matrices have a characteristic distribution of the eigenvalues, and this is different to the distribution of eigenvalues of a complex Hermitian random matrix.

In this assignment, the matrix is constructed to be real-symmetric, with diagonal elements drawn from the Gaussian (normal) distribution with mean 0 and standard deviation sqrt(2)), while the off-diagonal elements have a standard deviation of 1. This particular choice is unique because the probability distribution of such matrices is invariant under orthogonal rotations. That is, given a matrix H chosen randomly from this procedure, the matrix OHO^T (where O is any orthogonal matrix) has an equal probability of occurring.