HIGH PERFORMANCE SCIENTIFIC COMPUTING - LAB 1

Tyler Renken, Nick Rovito

September 4, 2024

Parallel Design

Diving computing tasks may drastically reduce computation time when using large datasets. In this lab, we parallelize a linear interpolation model of a known set of coordinate pairs (x, y). The inputs are an array of length $m = 20$ with x values to be interpolated from the set of known coordinate pairs. We are given an interpolation function that takes inputs of known x and y arrays, that make up our known coordinate pairs, a $xVal$ value with a $yVal$ value to be interpolated, and the length of the set of coordinate pairs. The function outputs the interpolated $yVal$ value. We are given four processors. The output of our program returns an array of interpolated $yVal$ values from a list of $xVal$ values.

Each processor is responsible for assembling the initial x and y arrays, which we will now refer to as "base arrays". This simplifies the program complexity while also meeting all functional requirements. We determine the number of elements each processor will need to interpolate using Eq. 1, where e is the number of elements in each process; m is the lengths of the base arrays; and P is the number of processes. In the case of four processors and 20 lookup tasks to be performed, each processor will handle five lookup tasks.

$$e = \frac{m}{P} \tag{1}$$

The lookup function should only be called a total of 20 times and each element in the $yVal$ array should only be computed once. To accomplish this, we give each process indices of the $xVal$ and $yVal$ arrays on which to perform the lookup function. For a 20 element array, processor 0 is assigned to slots 0-4, processor 1 is assigned to slot 5-9, etc. The start and end indices (i_{start} and i_{end})for each process are given as

$$\begin{aligned} i_{start} &= \frac{e}{P_i} \\ i_{end} &= e + i_start \end{aligned} \tag{2}$$

where e is the number of elements in each process, defined above, and P_i is the rank of the current process. Eq. 2 is valid assuming there is no remainder in the number of tasks divided by the number of processors (an acceptable simplification for this assignment, but would need to be changed for more robust computation).

After each processor calculates its assigned start and end indices, each then performs the lookup operation and stores the $yVal$ output into a processor-specific results ($yVal$) array. Upon completion of the subset of calculation, each processor waits until the lookup operations are completed on all processes. Then, the MPI_Gather message is sent between the processors to concatenate all of the data into the $yVal$ array on processor zero. A length 20 $yVal$ array is not populated on the other processors as only processor zero will be reporting the data. Processor zero then reports the data to stdout, concluding execution of the program.

Self Evaluation

The general program workflow was quickly designed and implemented; however, a collection of small errors made collecting on processor zero challenging. Further investigation into the usage of the MPI_Gather function was needed to complete the project. We were unable to fully finish the development within the given lab time, and each team member solved the collection process individually. We had different solutions to concatenation, and reported on the one we felt was more computationally efficient.

The development of the slurm script went without issues. Entries to the slurm file were added as needed by studying the output of a more immature script.

Appendix A - Parallel Code

```cpp
#include <iostream>
#include <mpi.h>
#include <cmath>

// lookupVal function provided for lab.
double lookupVal(int n, double *x, double *y, double xval){
  for ( int i = 0 ; i < n ; ++i)
      if ( xval >= x[i] && xval <= x[i+1] )
    return y[i] + (xval - x[i]) * (y[i+1]-y[i]) / (x[i+1]-x[i]);
  return 0.;
}

int main(int argc, char *argv[]){
  int numPE, myPE;

  MPI_Init(&argc, &argv);
  // get number of processors in use for distributing work
  MPI_Comm_size(MPI_COMM_WORLD, &numPE);
  // figure out which processor this is
  MPI_Comm_rank(MPI_COMM_WORLD, &myPE);

  // lab wants array of length m
  int m = 20;
  int n = 100;

  double x[n], y[n];  // data
  double xVal[m], yVal[m]; // interpolated data

  // initialize data (done on all processors)
  for(int i = 0 ; i < n ; ++i){
    x[i] = i;
    y[i] = i*i;
  }

  for(int i = 0; i<m; ++i) xVal[i] = 2.*i;

  // distribute work
  int num_per_process = round(m / numPE);
  int index_per_process = num_per_process * myPE; // start location in array for each process
  int limit_per_process = num_per_process + index_per_process; // finish location per process

  double *arr = new double[num_per_process]; // put the output here

  // run calculations
  for(int i = index_per_process; i < limit_per_process; ++i)
    arr[i - index_per_process] = lookupVal(n, x, y, xVal[i]);

  // gather all output into yVal array on processor zero
  MPI_Gather(arr, num_per_process, MPI_DOUBLE, yVal, num_per_process, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  MPI_Finalize();
  delete[] arr;

  // processor zero reports all results
  if(myPE == 0) for(int i = 0; i < m; ++i) std::cout << yVal[i] << std::endl;

  return 0;
}
```

Appendix B - Slurm Batch Script

```bash
#!/bin/bash

# -
# | Slurm directives , to make 4 cores available
# -

#SBATCH --nodes=1
#SBATCH --ntasks=4

# -
# | Load the modules necessary for running the program
# -

module purge
module load intel
module load impi

# -
# | Because of the way it was compiled , uses mpirun instead of srun
# -

echo "Start execution"
mpirun -np 4 ./a.out
echo "Execution finished"
```

Appendix C - Output

```
Start execution
0
4
16
36
64
100
144
196
256
324
400
484
576
676
784
900
1024
1156
1296
1444
Execution finished
```