CIS 372 INTRODUCTION TO PARALLEL COMPUTING
Spring 2018
MPI Lab 3

Note: You may work with up to one other person on this lab if you choose.

Objectives

- Implement a parallel sorting algorithm in MPI
- Use MPI Collective communications (scatter, gather)
- Optimize the merging so that some of it is done in parallel

Instructions:

Clone the lab 3 github starter code from your respository
Complete each part. Submit a writeup that contains answers to discussion/questions in green.
Submit all code produced.

Part A

Review the code in sort.c. It implements the basic startup and testing for a parallel sorting algorithm. However, right now it does not take advantage of MPI to actually do any work in parallel. It has a working sequential implementation of merge sort.

1. Create your own Makefile that will compile your code and generate timings for the following number of processes (N) and data size (D) using merge sort. Note that the largest of these should take around 40 seconds.

Fix **N**=1 and vary **D** with 16777216, 33554432, 67108864, and 134217728 elements

What is the asymptotic growth rate of this merge sort according to your timings?

2. Implement a basic parallel merge sort. The steps should be:

- Distribute the vector of data to all processes using scatter
- At each process sort the local subarray of data using merge sort
- Gather all of the sorted subarrays to the root process using gather
- Merge all of the sorted subarrays at the root process sequentially
    - **Note:** ***I provided a working implementation of merge that takes two array pointers, sizes, and places the merged results in the temporary array R. Included is also a merge_all function that will solve this last step exactly if called with the correct inputs.***

Check to make sure your results are correct and then generate timings for this version:

Fix **D**=134217728 and vary **N** with 1, 2, 4, and 8 processes

Create a graph of the speedup (*Tseq / Tpar*) with x-axis of N and y-axis of speedup. Does the speedup grow linearly in the number of processes?

3. Amdahl's law states that if P is the proportion of a system or program that can be made parallel, and 1-P is the proportion that remains serial, then the maximum speedup that can be achieved using N number of processors is **1/((1-P)+(P/N)).** Place some extra timing code around Part A.2 from the distribution through the gathering (exclude the sequential merge of the sorted subarrays).

If this is the region we consider to be P, what is the maximum speedup we could achieve according to Amdahl's law? Do your best to extrapolate your results from A.2 to an unlimited number of processes. Is the speedup for this unlimited number of processes similar or different to the maximum speedup you computed using Amdahl's law?

SAVE your code from Part A by copying it to sort_partA.c

Part B

We would like to improve the program by parallelizing the merge step. We have specifically made the data sets and number of processes powers of 2 to help here. Replace the gather + sequential merge code with a version that will gather and merge in parallel recursively:

1. Half of the remaining processes send their data to the other half and no longer do any more work
2. The receiving processes merge their local data with the received data
3. Repeat 1-2 until there is only 1 active process left (process 0)
4. Process 0 should have a fully sorted data

Note: Do not change the sequential merge sort algorithm that is run to locally sort the data at each process.

Repeat the timings and graph creation from Part A.2 and A.3 for this version.

Does our new version exhibit weak scaling? To show this we can double both the data size and number of processes for each data point.

Generate the following timings:

N	1	2	4	8
D	16777216	33554432	67108864	134217728

Create a graph of efficiency for each of the above data sizes as defined as $Tseq / (N \times Tpar)$ using Part A.1 timings for *Tseq*. Your x-axis should be N and y-axis should be efficiency.

In your writeup, discuss weak scalability and efficiency in your implementation and any other performance trends you see.

Grading

If you work with someone else, indicate on a READ.ME file who you worked with. Submit your writeup (including embedded graphs) and code (Makefile, sort_partA.c, sort_partB.c) to

github.

(35 points) = Code for Part A
(20 points) = Optimized code for Part B
(20 points) = Graph of speedup and writeup discussion for Part A
(20 points) = Graphs of speedup, efficiency and writeup discussion for Part B
(5 points) = Makefile for parts A and B