12/4/2023

LAB 5 – CS 380P – JOHN PETERSON - JLP5729

1. OVERALL APPROACH

My implementation uses a class-based, C++-heavy style of programming. I found a resource through a Princeton CS course website that had some sample classes in Java and detailed mathematic explanations, and this made it easier to adapt the functions and categorize functions into separate classes and files for ease of use.

For the parallelization using Message Passing Interface (MPI), I decided to focus on force parallelization, both due to time constraints and because I overlooked an overall issue with the tree recursing or building out too deeply in any of the test cases and figured that force calculation was the best way to get a speedup for this purpose. I took multiple approaches for MPI methods, which I will detail later in this writeup. Overall, the program achieved speedup on larger input sizes when parallelizing using MPI, and there are several paths to investigate further.

The following sections will detail the attempts/mistakes I made, the lessons learned, performance measurements, and the conclusion.

2. ATTEMPTS AND ISSUES ENCOUNTERED IN SEQUENTIAL PROGRAMMING AND ON CODIO

The class-based approach provided a sensible way to get started. However, I ran into an issue early on, needing to become more familiar with the class-based and extensibility features provided with C++, primarily having focused on C programming in the past. My program crashed before completing one step/pass-through on the 100,000-point sample test case. This led me on a “wild goose chase” to find all the pointers and objects I had left unintentionally dangling; it even led me to attempt overhauling my entire program to minimize the use of pointers. This led me down a path that spent about 12 hours rewriting the program's core functionality to remove pointers to the maximum extent; ultimately, I abandoned it when my test cases were not outputting consistent mathematical results. I had lost enough sleep and sanity and decided to revert to the class-based model to continue optimizing.

I used the debugger in Codio to find exactly where the program was crashing or getting killed. Each time, I reduced unnecessary function calls and dynamically allocated

objects. However, these steps were insufficient to ensure that my sequential implementation could run for more than 8-12 steps on the 100,000-point file.

My sequential program version ran fine on my Macbook Pro with an M1 Pro processor and 16GB of RAM; it only used approximately one CPU core and 80MB of RAM. The 100,000-test case file could complete 100 iterations in about 80 seconds. However, I did not take detailed measurements, as I did not successfully configure MPI on my machine. I would have only been able to graph the sequential implementation (giving no frame of reference for potential speed-up).

This led me to believe that Codio was implementing strict usage rules on their platform and not necessarily that I wrote extremely buggy code (although the latter is still possible and certainly likely). After spending about 15-20 hours trying to write, rewrite, and debug my sequential implementation, I cleaned up my print statements and moved on to parallelization.

3. PART 3: PARALLELIZATION APPROACH

The first step was understanding the MPI interface and programming model; the MPICH and other MPI implementation docs (like those from Microsoft) were very helpful and surprisingly straightforward, even if the programming model still took time to understand.

Reading again through the provided N-body paper resources led me to understand two aspects of the algorithm that could be parallelized. Due to peers suggesting that tree parallelization did not add to overall performance and, in some cases, detracted, I chose to focus on force parallelization.

This meant that each node kept track of its list of bodies, built its tree for each iteration, and moved all the bodies in their list based on the calculated force for each iteration but shared responsibilities for calculating the force for each body based on the existing tree. This required an understanding that the tree building was deterministic and that each core could successfully calculate the same force regardless of which individual bodies it was assigned.

The first attempt I chose to pursue was sending a basic MPIBcast for each body calculated. This would benefit from being easy to read and understand; it would also be easy to ensure that each node receives one copy of the force on each body.

However, when running this implementation, the overhead of broadcasting and syncing the processes using an MPIBarrier led to increased overhead for each core added from 2 through 8. Below, I have included graphs to demonstrate this performance result for individual message broadcasts for each body's force vector.

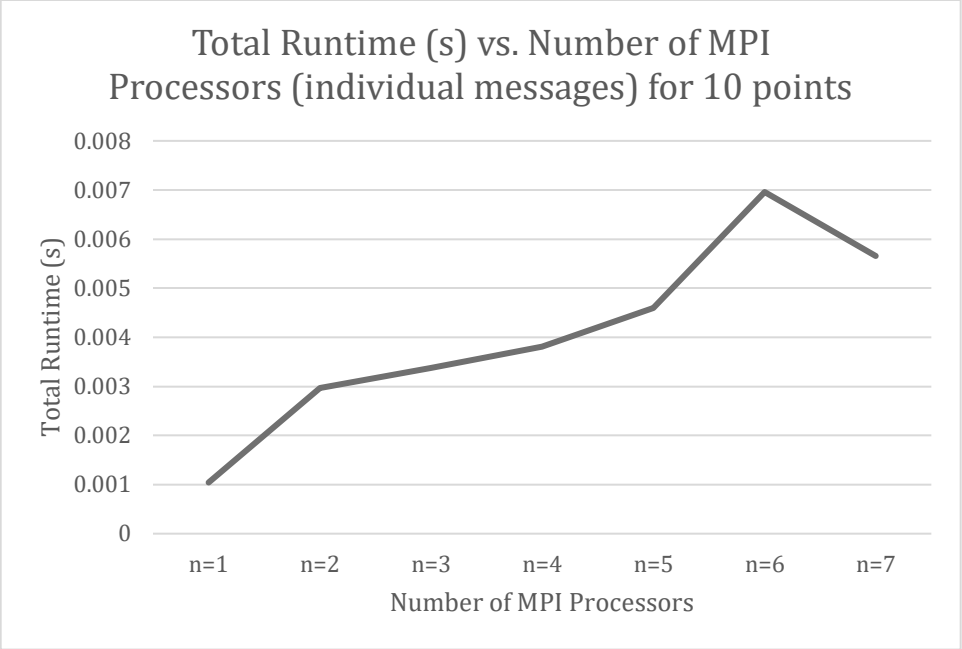

Figure 1. Runtime for individual message broadcast implementation for 10 points

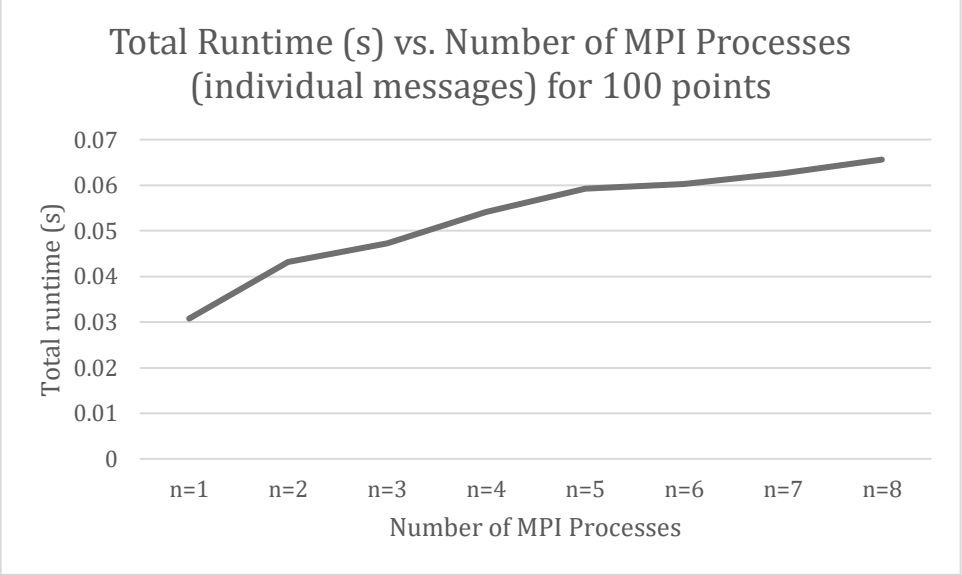

Figure 2. Runtime for individual message broadcast, 100 points

Figures 1 and 2 detail performance hit to adding MPI without optimizing or attempting to reduce the amount of work done per processor. Yes, each core only computed a subset of the overall forces. Still, the overhead of broadcasting, receiving a broadcast, and syncing for each force calculation did not make the program more efficient. Figure 3 also shows this impact for 10,000 points.

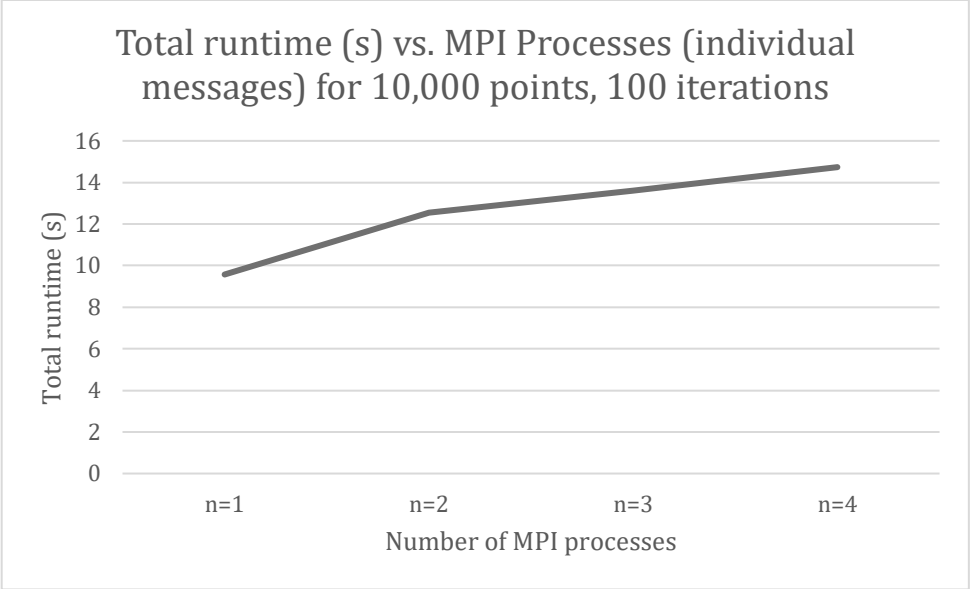

Figure 3. Runtime for 10,000 data points with individual messages

After successfully debugging and running the various nodes with a performance hit, the following step explored how the workload could be reduced using parallelism.

In the previous step, I realized I only needed to share three values: the index of which body the force would affect and the x- and y-components of that force vector. MPICH allows users to create a custom data type, which I used to make a struct to hold these values. This allowed for better clarity on data types and fields being transmitted rather than transmitting as a string or byte value.

My first attempt at improvement was due to the insight that each core could keep track of the forces calculated in a vector or array and then send one message to the other processes at the end of each iteration. I attempted it at first with an array of structs using broadcasts, but due to various factors, I could not debug it successfully.

I then found the MPIAllgatherv API, an explicit method in which all nodes can share a subset of a larger dataset with all other nodes. This allowed the designating of a transmission and receipt array and vectors with the sizes and indices of the values being stored. Once basic implementation and troubleshooting were completed, this provided an apparent speedup for all test files except for 10 points, likely due to the overhead of MPI and MPICH. These details are shown in Figures 4 - 7 below.

I tested the code on four inputs: three provided samples and a 10,000-point subset of the provided 100,000-point sample case to show more performance/scalability impact.

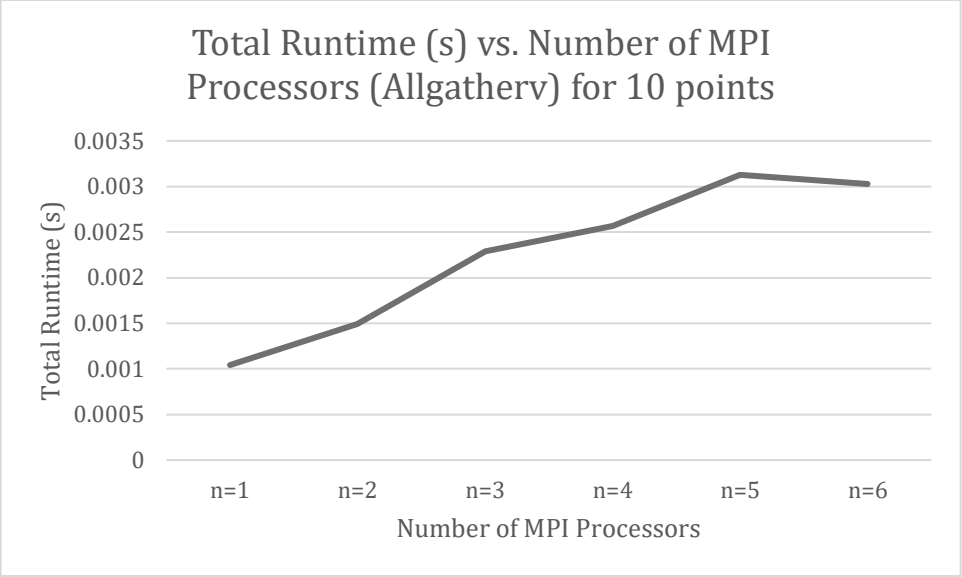

Figure 4. Runtime for Allgatherv method parallel 10 points

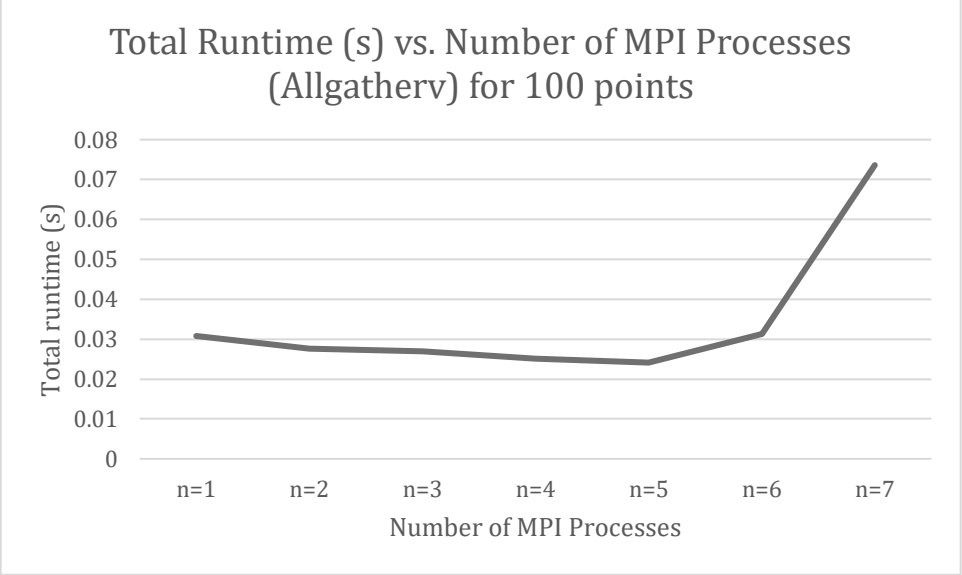

Figure 5. Runtime for Allgatherv for 100 points (Slight speedup for lower number of MPI nodes)

The implementation of MPICH or hardware limitations of CODIO combined to show a sharp increase in overhead on higher numbers of cores. These graphs omit 8-12, where performance ballooned so that graphs would not be legible on a linear scale to see the speedups observed. As might be expected, the most significant speedups were shown on larger input spaces when larger numbers of cores were assigned to the problem. We note a 20% speedup at n=5 for 100 points, a 40% speedup at n=4 at 10,000 points, and an estimated speedup of about 50% for the 100,000-point input sample.

While these speedups are less drastic than those listed in the papers provided, they show clearly that it is a problem that benefits from parallelization. With the right approach and hardware, the situation could scale considerably. This will undoubtedly help future astrophysics research as humankind reaches toward the stars.

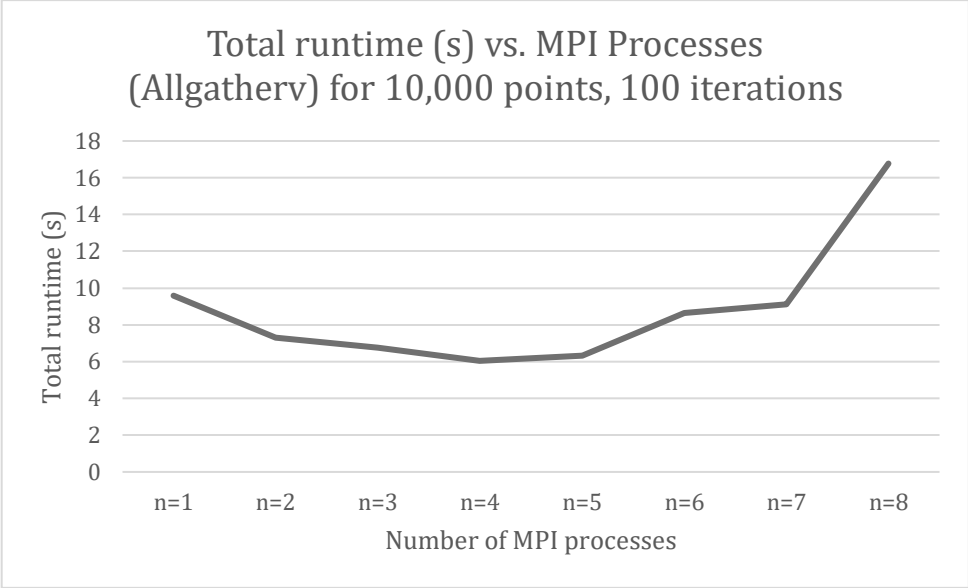

Figure 6. Runtime for 10,000 points using Allgatherv API; ~40% speedup demonstrated for n=4

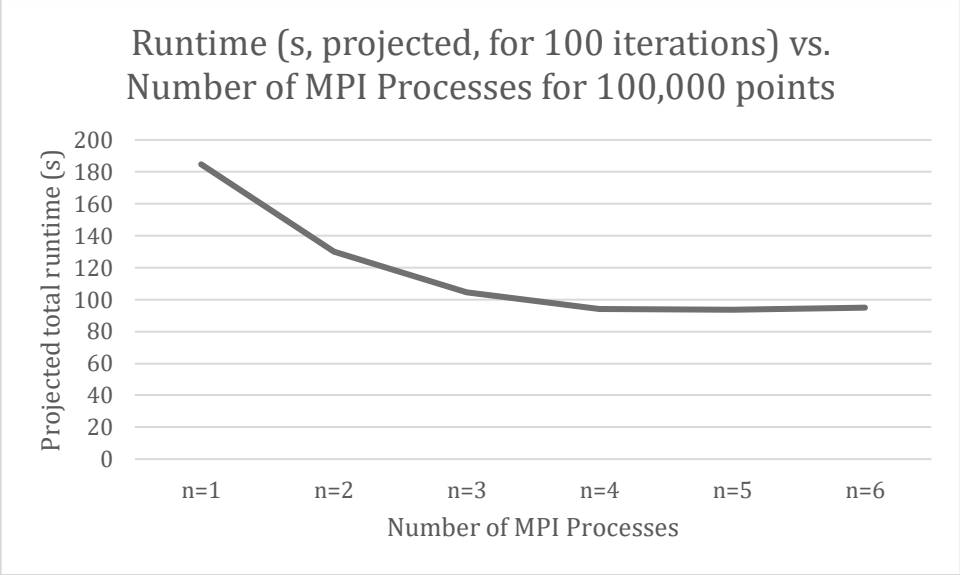

Figure 7. Runtime for Allgatherv method with 100,000 points (projected based on 8-12 steps per run)

The following section will detail the environments used, hours spent, and lessons learned.

4. DETAILS AND LESSONS LEARNED

The biggest mistake I made by far was focusing too much on sequential implementation for high input values when it became clear later on, both from TA/student posting and personal experience of rewriting entire classes/methods and chasing down memory issues that the Codio platform does not provide an adequate experience for testing and developing this assignment.

If I had known the extent of the issues, I would have focused on a C-based approach, which would be more explicit with pointers and data structures, as well as spent more time investigating an alternate environment, potentially configuring my machine more thoroughly to work with MPICH or renting a cloud server as necessary.

The majority of time spent in this lab was spent troubleshooting, debugging, and rewriting my sequential implementation. This lab could have been better spent focusing more on parallel improvements and a full parallelization (paper style) approach if I had either started earlier or the course staff had provided more sample codes (sequential code and test cases).

The time I spent breaks down as follows:

- 8 hours - initial coding and debugging of a sequential algorithm
- 10 hours - further debugging, rewriting, and reverting code to try to optimize for memory usage when getting SIGTERM on Codio
- 30 minutes - compiling and running sequentially on my Macbook to see that my implementation should not be an issue
- 3 hours - researching and deciding what strategy to take for parallelization
- 2 hours - implementing basic MPICH and troubleshooting (individual messages for each body)
- 2 hours - implementing vector message, then changing to Allgatherv method.
- 2 hours - measuring, plotting, and writing report

Overall time spent adds up to about 27.5 hours, of which a smaller portion than I would have liked was spent on parallel/interesting issues (primarily due to Codio limiting their platform performance).

I used Codio for all measurements and performance comparisons; I only took one on my laptop to confirm that my implementation could handle 100,000 points without getting killed.

Overall, the MPI APIs provide a vibrant way to handle inter-process communication. I would like to investigate their use in a distributed system, potentially with machines not physically collocated or worldwide. I have not explored current software utilizing MPI but would not be surprised to find popular software using it of this nature already.

I want to thank the course staff and fellow students for their hard work and input to teach and participate in an enriching course. I am pursuing a career transition into software development and graduating this semester; the skills I built in this class and program will help propel my career to the next level.