Parallel Computing COP6616
Fall 2024
Student: Aaron Goldstein
Instructor: Scott Piersall
Homework 2
Due: 10/15/2024, 1:00 pm

1. **(25 points) Please implement an MPI program in C, C++, <u>OR</u> Python to calculate the multiplication of a matrix and a vector (Using MPI Scatter and Gather).** Specifically, the MPI program can be implemented in the following way:

   1. Implement a Serial code solution to compute the multiplication of a matrix and a vector. This is the basis for your computation of Speedup provided by parallelization. (You should lookup the definition of Speedup in parallel computing carefully)
   2. According to the input argument (the size of the vector) from main() function, generate a matrix and a vector with random integer values, where the column size of matrix should be equal to the size of the vector.
   3. **SCATTER**: According to the number of processes from the input argument, split the matrix into chunks (row-wise) with roughly equal size, then distribute chunks to all processes using "scatter". Additionally, the vector can be broadcasted to all processes.
   4. Conduct product for the chunk of matrix and vector.
   5. **GATHER**: The final result is collected on the master node using "gather".
   6. **VERIFY CORRECTNESS**: Make sure the result of your MPI code matches the results of your serial code.
   7. **SPEEDUP**: What is the speedup S of your approach? (Speedup is a specific measurement and you should report it correctly) Combine the answer to this with your experiments in the next step
   8. **EXPERIMENTS**: You should run experiments consisting of running problem set sizes over varying number of compute nodes. Discuss the relationship between increasing the number of nodes to Speedup S in your submitted PDF

   Note:
   - You should develop your code on the MPI cluster.
   - I have placed a PowerPoint Presentation into the files section in canvas which contains a walkthrough and connection information for the compute cluster
   - You can login to this cluster using SSH and your UNF N#
   - For assistance with setup/compilation, you may ask members of the MPI Introduction Chapter Team in Discord

<u>Problem 1 Experiments:</u>

The execution time of each setup is listed below, 2048 is vector size for each execution.

- Serial Execution – 1 Process:
    - Time: 0.016346 seconds
    - Speedup: N/A (baseline for comparison)
    - Notes: This is the serial execution time with no parallel processing. We will use this as the base from which to calculate the speedup of parallel executions.
- Parallel Execution – 2 Processes:
    - Time: 0.006625 seconds
    - Speedup: Approximately 2.467 times faster than serial execution.
    - Doubling the number of processes resulted in a parallel execution more than two times faster than serial execution. Using parallel execution leads to a clear benefit in time efficiency for this problem size.
- Parallel Execution – 4 Processes:
    - Time: 0.003341 seconds
    - Speedup: Approximately 4.893 times faster than serial execution.
    - Notes: Continuing to see a good increase in time efficiency as we move from 2 to 4 processes.
- Parallel Execution – 8 Processes:
    - Time: 0.001705 seconds
    - Speedup: Approximately 9.587 times faster than serial execution.
    - Notes: Promising results as we move from 4 processes to 8.
- Parallel Execution – 16 Processes:
    - Time: 0.000878 seconds
    - Speedup: Approximately 18.617 times faster than serial execution.
    - Notes: Same speedup pattern continues where the parallel execution is about as many times faster as how many times more processes there are compared to serial.
- Parallel Execution – 32 Processes:
    - Time: 0.003927 seconds
    - Speedup: Approximately 4.16 times faster than serial execution.
    - Notes: The execution time increased compared to 16 processes, suggesting that the increased overhead from the extra 16 processes is unnecessary for this problem size.

General Observations on the Relationship Between Number of Processes and Problem Size:

The program scales effectively up to 16 processes, where the speedup is nearly proportional to the number of processes. For a problem size of N = 2048, using more than 16 processes introduces significant overhead from communication and process management, which becomes a limiting factor. To fully utilize 32 processes, a larger problem size than 2048 should be considered. For a problem size of 2048, 16 processes appear to be optimal, providing a good balance between workload distribution and overhead.

2. **(4 points)** For each of the following code segments, use OpenMP pragmas to make the loop parallel, or explain why the code segment is not suitable for parallel execution. You do not need to write executable C programs, please just add valid openMP pragmas. The code can be modified with keeping the same semantics for adding openMP pragmas.

For this question, I have highlighted and bolded any pragma changes needed. I have added explanations below the code segments where necessary.

(1)

```
#pragma omp parallel for
for (i=0;i<(int)sqrt(x);i++){
    a[i] = 2.3 * i;
    if (i<10) b[i] = a[i];
}
```

(2)

```
flag = 0;
for (i=0; (i<n) && (!flag); i++){
   a[i] = 2.3*i;
   if (a[i] < b[i]) flag = 1;
}
```

Not suitable! There is a dependency between iterations caused by the flag variable which is shared across the different threads. For one example, if a higher index iteration sets the flag variable to 1 before a lower index iteration executes, the result may be different from the sequential execution because the lower index iteration will not meet the condition while it might have met it in serial execution.

(3)

```
#pragma omp parallel for
for (i=0; i<n; i++)
   a[i] = foo(i);
```

Suitable as long as foo(i) does not access or change shared resources.

(4)

```
#pragma omp parallel for
for (i=0; i<n; i++){
    a[i] = foo(i);
    if (a[i] < b[i]) a[i] = b[i];
}
```

Suitable as long as foo(i) does not access or change shared resources.

(5)

```
for (i=0; i<n; i++){
    a[i] = foo(i);
    if (a[i] < b[i]) break;
}
```

Not suitable! Each current iteration depends on previous iterations' having executed to determine whether the current iteration should execute and if the loop should continue. For example, in serial execution, if a[2] < b[2], then the loop would terminate early and the iteration where i = 3 would not execute. This may not be the case when the code is parallelized since order is not guaranteed.

(6)

```
p = 0;
#pragma omp parallel for reduction(+:p)
for (i=0; i<n; i++){
    p += a[i] * b[i];
}
```

Suitable as long as p, a, and b are all of the integer datatype to avoid precision errors.

(7)

```
#pragma omp parallel for
for (i=k; i<2*k; i++){
    a[i] = a[i] + a[i-k];
}
```

Suitable, each iteration updates a[i] using a[i - k], however a[i - k] never indexes a value in the range a[k] -> a[(2 * k) - 1] which is the range of values being modified by other iterations. Since the a[i - k] values used by the loop range k to ((2 * k) - 1) for their calculations are never modified, the loop is suitable for parallelization.

(8)

```
for (i=k; i<n; i++){
    a[i] = b * a[i-k];
}
```

Not suitable, the loop has a dependency on the order of execution. Since i is not bound to 2 * k as in the previous problem, a[i - k] may be modified in different iterations. For example, when i = 2 * k, the value stored in `a[2 * k] = b * a[(2 * k) - k] will be different depending whether or not a[k] = b * a[k - k] has already been executed in the iteration i = k.`

3. (35 points) Computing Euclidean Distance Using OpenMP or MPI

In N-dimensional space, the Euclidean distance between two points P(p_1, p_2, p_3,…, p_n) and Q(q_1, q_2, q_3,…, q_n) is calculated as follows,

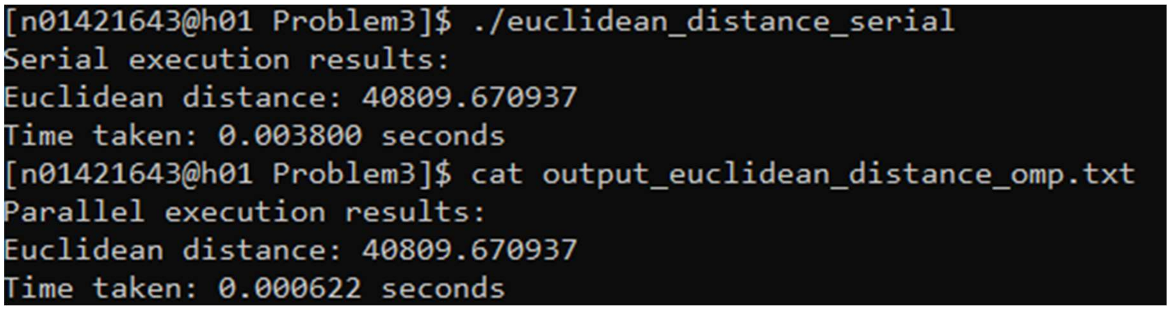
$$\sqrt{(p1-q1)^2+(p2-q2)^2+\cdots+(pn-qn)^2}$$

Please write **an OpenMP or MPI program** to compute the Euclidean distance between two *N*-dimensional points, where ***N is at least 1 million.*** Please initialize your vectors (denotes the two points) with random integers within a small range (0 to 99). This computation consists of a fully data-parallel phase (computing the square of the difference of the components for each dimension), a reduction (adding together all of these squares), and finally taking the square root of the reduced sum.

`Note: You may want to use gcc/g++ directly, since gcc/g++ already supports OpenMP. To use it, you can compile your code with a special flag "-fopenmp".`

Answer:

I solved this problem using OpenMP and display the results of both my serial and parallel execution below. I used a problem size of N = 1 million and 42 as the seed for the random number generator.



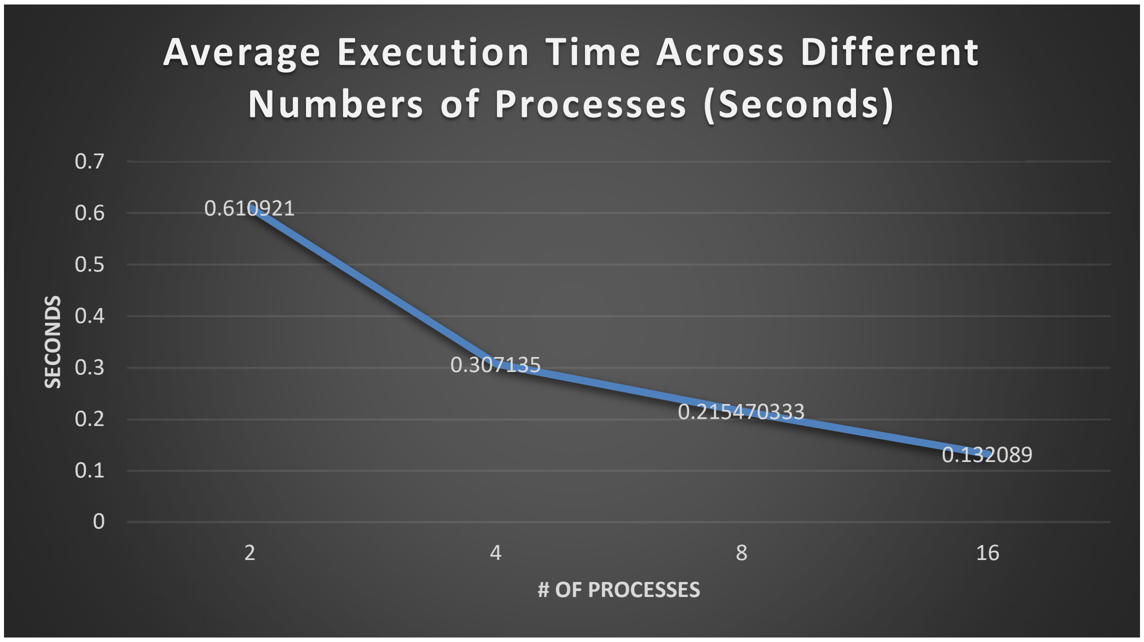
```
[n01421643@h01 Problem3]$ ./euclidean_distance_serial
Serial execution results:
Euclidean distance: 40809.670937
Time taken: 0.003800 seconds
[n01421643@h01 Problem3]$ cat output_euclidean_distance_omp.txt
Parallel execution results:
Euclidean distance: 40809.670937
Time taken: 0.000622 seconds
```

4. (35 points) Find prime numbers using MPI. Develop an MPI program to find and print the first n prime numbers in the range [0…n]. Run your code using 2,4,8, and 16 nodes. The MPI communication methods you choose are up to you, however….you MUST distribute the work evenly among the nodes, not perform redundant or unnecessary calculations AND you should automatically skip even numbers. For your experiment, use the value of 2,500,000 for n. Your program should print out each prime number found (and the node that found it), and the master node (rank 0) should output the time elapsed for your program to run. Run the experiment for 2,4,8, and 16 nodes. Graph your results. Include the graph and a paragraph interpreting and explaining your experimental results in the PDF file report. Include your source code in the ZIP file for turn in.

Hints: There are a variety of ways to implement a solution to question 4. My hints are

- Do not assign contiguous blocks of numbers to each node. That does NOT distribute the work evenly among the nodes. Think about this hint ***carefully***. If you do not understand why assigning a contiguous block of numbers to each node is incorrect for this problem, please come see me to discuss….seriously. Solutions submitted which just assign contiguous blocks of numbers to nodes will receive zero credit
- MPI_Reduce is your friend for this problem……..

My Problem 4 experiment results are displayed below in a graph, along with two small paragraphs which explain the results.



When finding every prime number between [0, N] where N = 2,500,000, we can observe that the program's performance improves as the number of processes increases from 2 to 16. The graph above, which is a line chart, shows the y-axis representing average execution time in seconds and the x-axis representing the number of processes. As we move to the right, indicating more processes, the execution time decreases. This behavior aligns with our expectations when parallelizing the program using MPI, as distributing the workload across more processes reduces computation time.

Further testing is needed to determine if increasing the number of processes to 32 would yield additional performance gains or result in diminished returns. This could occur if the overhead of managing more processes outweighs the benefits for the current problem size. Communication overhead and the efficiency of workload distribution are ultimately the key factors in this analysis.

What to submit:

1. For all code assignments, please write a README file explaining how to run and test your code. In your code, please add appropriate comments and make your code structured.
2. PDF File: You should create a report and convert its contents to PDF format. The report should answer the following questions:
    a. Answers to the Speedup portion of Question 1
    b. Answers to the 8 problems in Question 2
3. Zip all code and PDF files into a single file to upload.

NOTE: Because you are all sharing the compute nodes and cluster resources, each experiment should be run multiple times, and your runtimes averaged over each run for varying node sizes. This should reduce any large variances in your experimental results resulting from resource contention.