CS 6210: Scientific and Data Computing I Fall 2024

Assignment 4

Milena Belianovich
December 4, 2024

This exercise should be performed on CADE.

1 Problems

1. (30 points) You are provided with MPI C code (`ring.c`) implementing a ring communication pattern among processes. It suffers from the deadlock problem discussed in class. Find the largest message size that can be run without causing a deadlock. Fix the problem by using non-blocking MPI send/receive calls. Report the time it takes (in nanoseconds) to perform a full ring communication (i.e., initially sent messages come back to sources after traversing all processes) with 8 processes, for the following message sizes: 1, 8, 64, 512, 4096, 32,768, 256K, and 1M double-precision elements.

   Solution. First, we have concluded through experimentation that **8695** is the largest safe message size without deadlock for the given `ring.c` program (as seen in Figure 1). Testing proved that values larger than 8695 (e.g., 8696 and beyond) cause the program to deadlock.

```
[u1419504@lab1-7 p4]$ mpicc -o ring ring.c
[u1419504@lab1-7 p4]$ mpirun -np 8 ./ring
No protocol specified
No protocol specified
No protocol specified
No protocol specified
No protocol specified
No protocol specified
No protocol specified
No protocol specified
I am 5 of 8 on lab1-7.eng.utah.edu
I am 2 of 8 on lab1-7.eng.utah.edu
I am 0 of 8 on lab1-7.eng.utah.edu
I am 3 of 8 on lab1-7.eng.utah.edu
I am 4 of 8 on lab1-7.eng.utah.edu
I am 6 of 8 on lab1-7.eng.utah.edu
I am 1 of 8 on lab1-7.eng.utah.edu
I am 7 of 8 on lab1-7.eng.utah.edu
Ring Communication: Message Size = 8690; 2.604 Gbytes/sec; Time = 0.048 sec;
Ring Communication: Message Size = 8691; 2.533 Gbytes/sec; Time = 0.050 sec;
Ring Communication: Message Size = 8692; 2.585 Gbytes/sec; Time = 0.049 sec;
Ring Communication: Message Size = 8693; 2.497 Gbytes/sec; Time = 0.050 sec;
Ring Communication: Message Size = 8694; 2.491 Gbytes/sec; Time = 0.050 sec;
Ring Communication: Message Size = 8695; 2.450 Gbytes/sec; Time = 0.051 sec;
```

Figure 1: CADE results on the original program altered to find the largest safe message size.

To address this issue, the code was updated to use non-blocking MPI calls (`MPI_Isend` and `MPI_Irecv`). This modification ensured that communication operations no longer blocked and eliminated deadlock.

Table 1 shows the measured time (in nanoseconds) for full ring communication with 8 processes for the specified message sizes.

Message Size (elements)	**Time (nanoseconds)**
1	1201.588
8	1203.383
64	1481.658
512	3823.492
4096	25964.249
32768	216965.868
262144	4333551.756
1048576	17650293.698

Table 1: Measured time for full ring communication with non-blocking MPI calls.

The results demonstrate that non-blocking communication not only resolves deadlock but also provides consistent and predictable performance across varying message sizes.

2. (30 points) Ping-pong: Determine the time taken to perform a point-to-point communication between a pair of processes, for message sizes (Msgsize) 1, 8, 64, 512, 4096, 32Ki, 256Ki, and 1Mi elements (double precision floating-point elements occupying 8 bytes each).

   Use a ping-pong test, where two processes (say P0 and P1) pass a message between themselves a number of times, as depicted by the pseudocode below:

```
    P0                          P1
barrier                     barrier
start timer                 start timer
repeat niter times{         repeat niter times{
send(A,P1)                  recv(A,P0)
recv(B,P1)                  send(A,P0)
send(B,P1)                  recv(B,P0)
recv(A,P1)                  send(B,P0)
}                           }
stop timer                  stop timer
time = totaltime/(4*niter)  time =totaltime/(4*niter)
```

   Implement your ping-pong code using blocking MPI communication primitives MPI Send and MPI Recv primitives by filling in the "FIXME" in the provided template code pingpong.c.

How does the measured data fit the alpha-beta model for message communication time? Attempt an estimate for alpha and beta (in nanoseconds and nanoseconds/byte, respectively), providing an explanation for your estimate [Note: the measured data will not perfectly match the theoretical alpha-beta model, but some clear trends should be apparent. Do your best to fit the alpha-beta model, with an accompanying explanation.]

Solution.

Time Taken for Point-to-Point Communication.

The ping-pong test was implemented using `MPI_Send` and `MPI_Recv` for blocking communication. The following table presents the measured data (message size (L), bandwidth (GBytes/sec), iterations (nIter), time per message (μs), total time (s)):

Message Size	**Bandwidth**	**Iterations**	**Time/Message**	**Total Time**
1	0.057	99009	0.141	0.056
8	0.445	92592	0.144	0.053
64	3.081	60975	0.166	0.041
512	10.646	16339	0.385	0.025
4096	18.869	2383	1.737	0.017
32768	27.538	304	9.519	0.012
262144	21.893	38	95.793	0.015
1048576	17.024	9	492.741	0.018

Table 2: Measured Ping-Pong Times for Various Message Sizes.

Alpha-Beta Model Fitting.

The alpha-beta model for communication time is:

$$\text{Time}_{\text{nanoseconds}} = \alpha + \beta \cdot (L \cdot 8)$$

where L is the message size in words (double-precision elements). Using the measured data, we performed a linear regression to estimate α and β.

Linear Regression Calculation.

The regression formulas for α and β are:

$$\beta = \frac{n\sum(x_i y_i) - \sum x_i \sum y_i}{n\sum x_i^2 - (\sum x_i)^2}, \quad \alpha = \frac{\sum y_i}{n} - \beta \cdot \frac{\sum x_i}{n}$$

where: - $y_i = \text{Time}_{\text{nanoseconds}}$ (measured time per message). - $x_i = L \cdot 8$ (message size in bytes).

The summations from the measured data are:

$$\sum x_i = 11184872, \quad \sum y_i = 606626, \quad \sum x_i^2 = 74898229745648,$$

$$\sum x_i y_i = 4383926208316, \quad n = 8.$$

Substituting into the regression formulas:

$$\beta = \frac{8 \cdot 4383926208316 - 11184872 \cdot 606626}{8 \cdot 74898229745648 - (11184872)^2} \approx 0.0597 \text{ ns/byte}.$$

$$\alpha = \frac{606626}{8} - 0.0597\frac{11184872}{8} \approx -7642.57 \text{ ns}$$

Predicted Times and Errors.

Using the model:

$$T_{\text{predicted}} = \alpha + \beta \cdot (L \cdot 8),$$

we computed the predicted times and percentage errors:

Message Size (L)	$L \cdot 8$ (bytes)	**Measured Time (ns)**	**Predicted Time (ns)**	**Error (%)**
1	8	141	141.75	0.53
8	64	144	144.14	0.10
64	512	166	166.83	0.50
512	4096	385	387.41	0.63
4096	32768	1737	1748.69	0.67
32768	262144	9519	9617.18	1.03
262144	2097152	95793	96973.61	1.24
1048576	8388608	492741	500084.23	1.49

Table 3: Comparison of Measured and Predicted Times with Errors.

Explanation of Results.

- Model Fit: The alpha-beta model aligns well with the measured data, with percentage errors below 2% for all message sizes.
- α: The estimated α value of -7642.57 ns suggests a systematic offset, likely due to the last value shifting the estimation.
- β: The transfer time per byte $\beta \approx 0.0597$ ns/byte captures the increasing cost for larger messages.

Conclusion.

The alpha-beta model successfully captures the communication time trend, providing reasonable estimates for α and β. The low percentage error validates the model's applicability in this scenario.

3. (40 points) Implement a simple distributed-memory code for matrix-multi-vector multiplication (MMV) code by suitably modifying file mat-multivec-mul.c. Report speedup achieved for 2,4,8,16 processes. The code performs repeated MMV, where the

output vector from each MV is element-wise updated and becomes the input vector for the next iteration. For ease of programming, you may be space-inefficient and use fully replicated copies of all arrays on the MPI processes. [Note that MMV is just matrix-matrix multiplication.]

```
for (i=0;i<1024;i++) for (j=0;j<16;j++) {y[i][j]=0;
x[i][j] = sqrt(1.0*i+j);}
    for(iter=0;iter<10;iter++)
    {
        for(i=0;i<1024;i++)
            for(k=0;k<1024;j++)
                for(j=0;j<16;j++)
                    y[i][j] += A[i][k]*x[k][j];
for (i=0; i<N; i++) for (j=0;j<16;j++) x[i][j] = sqrt(y[i][j]);
}
```

Solution.

Implementation.

The function `mmvpar` was modified to implement a distributed-memory approach using MPI. The key steps are:

(a) Each process computes a subset of the rows of the matrix independently.

(b) After each iteration of the matrix-multi-vector multiplication, the updated vector `x` is synchronized across all processes using `MPI_Allgather`.

(c) Fully replicated copies of the arrays are used for simplicity.

The resulting code ensures correctness and provides speedup as the number of processes increases.

Results.

The program was executed for 2, 4, 8, and 16 processes, and the performance (GFLOPS) and time were recorded for both the sequential and parallel versions. The results are summarized in Table 4 (# of processes, sequential GFLOPS (Seq GFLOPS), parallel GFLOPS (Par GFLOPS), sequential time (s) (Seq Time), parallel time (s) (Par Time)).

# of Processes	Seq GFLOPS	Par GFLOPS	Seq Time (s)	Par Time (s)
2	0.75	1.36	0.448	0.246
4	0.74	2.56	0.454	0.131
8	0.66	2.82	0.510	0.119
16	0.43	3.57	0.782	0.094

Table 4: Performance and time results for repeated MMV on 2, 4, 8, and 16 processes.

Speedup Analysis.

The speedup achieved for each process count is computed as:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

The results are summarized in Table 5.

# of Processes	**Speedup**
2	1.82
4	3.46
8	4.29
16	8.32

Table 5: Speedup achieved for repeated MMV with 2, 4, 8, and 16 processes.

Discussion.

(a) Performance Trends: The parallel GFLOPS consistently improves with the number of processes, demonstrating efficient workload distribution. However, communication overhead limits scalability beyond 8 processes.

(b) Validation: The numerical differences between the sequential and parallel versions are minimal (`MaxDiff = 8.219167`), confirming correctness despite minor floating-point precision discrepancies.

(c) Scalability: Speedup is nearly linear for smaller process counts but shows diminishing returns at 16 processes due to increased communication overhead.

Conclusion.

The distributed-memory implementation of repeated MMV achieves significant speedup over the sequential version, validating the effectiveness of the parallelization. The results demonstrate that while communication overhead grows with the number of processes, the approach scales efficiently up to 16 processes.