CS140 W2026 PA1: Distributed Memory MPI and SIMD Programming

Please refer to the function comments for their specifications in every problem described below. Do not change the provided function signatures or .h files. The autograding tool on GradeScope will use these signatures and .h files for testing. For MPI programming, compilation should keep –O3 option so code is optimized by C compiler.

Notice that to submit a job in Expanse, you need to modify the job script. For example to use 8 cores, change the allocation as:

```
#SBATCH --partition=shared
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
```

1. Write an MPI program that computes a tree-structured global sum. Notice that the number of processes participating in the global sum may not be a power of two. Do not change the function signature of treesum_mpi.c. Your task is to write function global_sum() which runs on each process and performs a tree-structure summation only using MPI_Send() and MPI_Recv(). You are NOT allowed to use any MPI collective function such as MPI_Reduce(). For this problem, you need to write your own test file in treesum_test_mpi.c following the sample given as mv_mult_test_mpi.c and should include the execution of some correctness tests. Our grading will run your treesum_mpi.c code using the included Makefile and our own test code.

Function to be tested by our grading script: global_sum

Files to submit: treesum_test_mpi.c, treesum_mpi.c

2. The sample code released includes a matrix vector multiplication in mv_mult_mpi.c with a test file called mv_mult_test_mpi.c. Your task is to leverage this code and parallelize the following iterative matrix multiplication algorithm using block mapping. Matrix A is of size n x n. Column vectors x , y, and d are of size nx1.

```
For k =  0 to t-1
      y = d+ Ax
      x=y
EndFor
```

Functions to be tested by our grading script: itmv_mult, init_matrix

Files to submit: itmv_mult_mpi.c, itmv_mult_test_mpi.c

Report to include: README.txt

Q 2.a Function itmv_mult_seq() in itmv_multi_mpi.c is the sequential code for this algorithm when argument matrix_type is 0. When matrix_type=1, that means the input matrix is an upper triangular matrix which will be handled in Problem 2.b.

The test file itmv_multi_test_mpi.c should use the following matrix:

The diagonal elements of matrix A are 0. Namely. A[i][i]= 0. Non-diagonal elements are A[i][j]= -1/n. Every element in vector d is (2n-1)/n. Initially each element of vector x is 0. You need to complete function init_matrix for such initialization at every process. Your test program should check the correctness of data initialization in every process and execute the corresponding tests in run_all_tests() successfully.

Part 2.a of README.txt should include a performance report when n=4096 and t=1024 for the above test case. Report the parallel time, gigaflops, speedup, and efficiency when the number of cores used is 2 and 4 where each core runs one MPI process. We expect your mapping solutions delivers decent parallel speedups. If not, check your code and evaluating setup.

Q2.b Revise your parallel code for Problem 2.a to handle an upper triangular matrix A where all of its lower triangular elements are 0. The input parameter matrix_type is 1 in this case. For such a matrix, A[i][j]=0 if i>j. Because of this zero pattern, the inner most loop j of the sequential code for computing Ax is modified with the start position as i instead of 0 to skip unnecessary computation. See itmv_mult_seq() defined in itmv_multi_mpi.c for more details:

```
for i=0 to n-1
  for j=i to n-1
    y[i]=d[i] + A[i][j]*x[j]
```

Your parallel code in function itmv_mult should work with such a matrix and should avoid perform computation which yields 0 (namely should not multiply the lower triangular part of matrix A). The test matrix case added to itmv_mult_test_mpi.c for 2.b is the same as one for Problem 2a, except that the lower triangular portion is 0. Namely A[i][j]= -1/n for $0 \leq j<i<n$ otherwise 0 for other elements. All elements in vector d are (2n-1)/n. Initially each element of vector x is 0. Your init_matrix function should accomplish this. Your test code needs to execute the 8 tests in run_all_tests() successfully.

Part 2b of README.txt should include a performance summary when n=4096 and t=1024 for the above test case. Report the parallel time, gigaflops, speedup, and efficiency when the number of cores used is 2 and 4 where each core runs one MPI process. Explain why the efficiency of the Problem 2.b solution for the above matrix is lower than that we have observed for Problem 2.a and describe a strategy that can be used to address this issue.

3. Modify the released sum code simd_sum.c in simd.tar.gz to complete a function called sum_avx() using AVX2 SIMD intrinsics. The startup code contains a function (sum_sse) that uses Intel SSE intrinsics with 128 bits to vectorize the sum of *n* integers. Following the code structure of this function, use AVX2 intrinsics with 256 bits registers to complete sum_avx().

   Conduct a latency and gigaflops comparison of 3 sum implementations in this code when *n*=1,000,000 in one core of an AMD CPU server of the Expanse cluster using -O gcc flag and -O3 flag separately.
   1) The naive sum of n integers (function sum_naive);
   2) The vectorized sum with SSE intrinsics (function sum_sse);
   3) The vectorized sum with AVX-256 intrinsics (function sum_avx).

   For -O compilation flag, is the naïve sum is substantially slower than others?

   For -O3 compilation flag, is the naïve sum is substantially slower than others?

   List the latency and GFLOPs of the above 3 version of implementation in Q3 of README.txt. Explain the reason for above raised questions in your report.

   If you write your report Q1 and Q2 with CSIL instead of Expanse, then complete Q3 report using CSIL.