CSE412 Parallel Computing (Fall 2022)
Instructor: Prof. Myeongjae Jeon
Due date: Nov 16, 2022, 11:59 pm. KST

Assignment 1: Parallel LU Decomposition

LU Decomposition (where 'LU' stands for 'lower upper') is a classical method for transforming an *N* x *N* matrix **A** into the product of a lower-triangular matrix **L** and an upper-triangular matrix **U**,

$$\mathbf{A} = \mathbf{LU}.$$

You will use a process known as Gaussian elimination to create an **LU** decomposition of a square matrix. By performing elementary row operations, Gaussian elimination transforms a square matrix **A** into an equivalent upper-triangular matrix **U**. The lower-triangular matrix **L** consists of the row multipliers used in Gaussian elimination. A description of LU decomposition can be found on MathWorld.

In this assignment, you will develop an OpenMP parallel implementation of LU decomposition that use Gaussian elimination to factor a dense *N* x *N* matrix into an upper-triangular one and a lower-triangular one. In matrix computations, *pivoting* involves finding the largest magnitude value in a row, column, or both and then interchanging rows and/or columns in the matrix for the next step in the algorithm. The purpose of pivoting is to reduce round-off error, which enhances numerical stability. In your assignment, you will use row pivoting, a form of pivoting involves interchanging rows of a trailing submatrix based on the largest value in the current column. To perform LU decomposition with row pivoting, you will compute a permutation matrix **P** such that **PA** = **LU**. The permutation matrix keeps track of row exchanges performed. Below is pseudocode for a sequential implementation of LU decomposition with row pivoting.

```
    inputs: a(n,n)
    outputs: π(n), l(n,n), and u(n,n)

    initialize π as a vector of length n
    initialize u as an n x n matrix with 0s below the diagonal
    initialize l as an n x n matrix with 1s on the diagonal and
0s above the diagonal

    for i = 1 to n
      π[i] = i
```

```
for k = 1 to n
  max = 0
  for i = k to n
    if max < |a(i,k)|
      max = |a(i,k)|
      k' = i
  if max == 0
    error (singular matrix)
  swap π[k] and π[k']
  swap a(k,:) and a(k',:)
  swap l(k,1:k-1) and l(k',1:k-1)
  u(k,k) = a(k,k)
  for i = k+1 to n
    l(i,k) = a(i,k)/u(k,k)
    u(k,i) = a(k,i)
  for i = k+1 to n
    for j = k+1 to n
      a(i,j) = a(i,j) - l(i,k)*u(k,j)
```

Here, the vector π is a compact representation of a permutation matrix p(n,n), which is very sparse. For the *i*th row of p, π(i) stores the column index of the sole position that contains a 1.

You will write a shared-memory parallel program using OpenMP that perform LU decomposition using row pivoting. Each LU decomposition implementation should accept two arguments: *n* - the size of a matrix, followed by t - the number of threads. Your programs will allocate an `n x n` matrix `a` of double precision (64-bit) floating point variables. You should initialize the matrix with uniform random numbers computed using a suitable random number generator, such as drand48, drand48_r, or the C++11 facilities for pseudo-random number generation. (Note: if you are generating random numbers in parallel, you will need to use a reentrant random number generator and seed the random number generator for each thread differently.) Apply LU decomposition with partial pivoting to factor the matrix into an upper-triangular one and a lower-triangular one. To check your answer, compute the sum of Euclidean norms of the columns of the residual matrix (this sum is known as the L2,1 norm) computed as **PA**-**LU**. Print the value of the L2,1 norm of the residual. (It should be *very* small.)

The verification step needs not be parallelized. Have your program time the LU decomposition phase by reading the real-time clock before and after and printing the difference.

The formal components of the assignment are listed below:

- Write a shared-memory parallel program that uses OpenMP to perform LU decomposition with partial pivoting.
- Write a document that describes how your programs work. This document should *not* include your programs, though it may include figures containing pseudo-code that sketch the key elements of your parallelization strategy for your implementation. Explain how your program partitions the data and work and exploits parallelism. Justify your implementation choices. Explain how the parallel work is synchronized.

  Use problem size *n > 1000* to evaluate the performance of your implementation. Prepare a table that includes your timing measurements for the LU decomposition phase of your implementation on 1, 2, and 4 threads. Graph of the parallel efficiency of your program executions. Plot a point for each of the executions. The x axis should show the number of threads. The Y axis should show your measured parallel efficiency for the execution. Construct your plot so that the X axis of the graph intersects the Y axis at Y=0.

  * Parallel efficiency is computed as *S/(p * T(p))*, where *S* represents the wall clock time of a sequential execution of your program, *p* is a number of processors and *T(p)* is the wall clock time of an execution on *p* processors. Don't make the mistake of using the execution time of a one thread version of your parallel implementation as the time of a sequential execution. The execution time of the one-thread parallel implementation is typically larger than that of the original sequential implementation. When you compute parallel efficiency, always use the performance of the original sequential code as a baseline; otherwise, you will overestimate the value of your parallelization.

In this assignment, reading and writing shared data will account for much of the execution cost. Accordingly, you should pay attention to how you lay out the data and how your parallelizations interact with your data layout. You should consider whether you want to use a contiguous layout for the array, or whether you want to represent the array as a vector, of **n** pointers to n-element data vectors. You should explicitly consider how false sharing might arise and take appropriate steps to minimize its impact on performance.

Submitting your Assignment

You should submit a **assignment1_[studentID].tar.gz** that contains:

- a directory containing the code for your OpenMP program and a Makefile to build the code, and
- a writeup about your programs in PDF format.

You will submit your assignment by uploading the compressed file on BlackBoard. If the submission comes with a wrong format or the code will not build, you will not receive points. Your assignment should be submitted prior to the assignment deadline noted at the top of this handout, or it will be subject to the policy on late work. If you have any issue on Blackboard submission, please send an email to the TA (tykim8191@unist.ac.kr).

Grading Criteria

- 40% Program correctness.
- 10% Program clarity, elegance and parallelization. The programs should be well-documented internally with comments.
- 50% Writeup. Your grade on the writeup includes your approaches on parallelizations, and an analysis on the program performance and scalability. We care about the quality of the writing (grammar, sentence structure), whether all of the required elements are included, and the clarity of your explanations.

Notes

- Make sure your code is compiled and works on Linux.
- Make sure your code is not generating segmentation fault.
- Do not change a given skeleton at all.
- You can discuss with others on the provided files (e.g., Makefile and README). Other than that, direct discussion is strictly prohibited.
- For any questions, please use a piazza.
- For collaborating with others, follow the rules from the slides for the first lecture on Blackboard. Please write down the student who collaborates with you at the first of your report.

OpenMP

The top-level directory contains README and Makefile. Please read them first and fully understand.

You can find information about OpenMP in several places. Lawrence Livermore National Laboratory has posted a good OpenMP tutorial. Complete information about OpenMP can be found in the OpenMP 4.5 specification. The OpenMP 4.5 standard was not intended to be read by application developers. A better reference for developers is the OpenMP Application Programming Interface Examples Version 4.5.0.

When you write OpenMP parallel regions and worksharing directives, be careful to specify the data scope attribute clauses for variables used within each directive's scope. I recommend using the data scoping directive `default(none)` to make sure that the compiler doesn't implicitly add an incorrect sharing directive.

Managing Threads and Data

On your machine, memory may be partitioned among the 2 sockets. Memory attached to a remote socket is farther away (slower to access) than memory attached to a local socket. You can manage the Non-Uniform Memory Access (NUMA) characteristics of your machine using the NUMA control library. See "man numactl" for more information. Some brief information is included below.

As an alternative to relying on defaults provided by "numactl --localalloc" for data placement, you can use primitives in libnuma to programmatically control where data is allocated. Before threads are created, you can allocate data in particular sockets before using libnuma's

```
void *numa_alloc_onnode(size_t size, int node);
```

After you create threads, you can have each thread allocate its own data in memory attached to the socket in which it is running using libnuma's

```
void *numa_alloc_local(size_t size);
```

For info about the libnuma interface, documentation is available both on a web page, or in PDF.

Atomic Operations

If you want to use atomic operations for synchronization, I suggest using the C++11 atomic operations. Using the C++ atomics requires some understanding of weak ordering and the C++ memory model.