Assignment 1 report

20191165 Eunyeong Sim

1. How to execute parallelization

Algorithm: Parallelized LU decomposition

```
0:  n as matrix size
1:  allocate memory for upper(n,n), lower(n,n), input(n,n), _input(n,n), pi(n)
2:  initialize matrices
3:          set 1 to pivot of lower(n,n)
4:          set random floating number to every entry of input(n,n)
5:          copy input(n,n) into _input(n,n)
6:  for k=0 to n-1
7:          max_value = 0; max_idx = 0;
8:          #pragma omp parallel default(none) firstprivate(k, matrix_size) shared(input, upper, lower, pi,
max_value, max_idx)
9:          {
10:                 #pragma omp for reduction(max: max_value) reduction(max: max_idx)
11:                 for i=k to n-1
12:                         If max_value < |input(i,k)|
13:                         max_value = |input(i,k)|
14:                         max_idx = i
15:                 #pragma omp single
16:                 {
17:                         if max_value == 0 then error(singular matrix)
18:                         swap pi[k] and pi[max_idx]
19:                         swap lower(k,0:k-1) and lower(max_idx, 0:k-1)
20:                         upper(k,k) = input(k,k)
21:                 }
22:                 #pragma omp for nowait
23:                 for i=k+1 to n-1
24:                         lower(i,k) = input(i,k) / upper(k,k)
25:                 #pragma omp for schedule(static, 8)
26:                 for i=k+1 to n-1
27:                         upper(k,i) = input(k,i)
28:                 #pragma omp for
29:                 for i=k+1 to n-1
30:                         for j=k+1 to n-1
31:                                 input(i,j) -= lower(i,k) * upper(k,j)
32:         }
33:  check_L21norm(_input, pi, lower, upper, matrix_size);
33:  deallocate memory
```

2. Description of parallelization strategy

To construct my algorithm, I first studied which parts should execute in order. There are two major parts where the execution should be in order.

1. **Outermost for-loop should be executed in order**
   Because previous step's computation output is used at next iteration's computation input. For example, at iteration k=0, *input* matrix is updated except first row and column (line 29-31). And at iteration k=1, that previously updated value is used to update *lower* and *upper* matrix (line 22-27). Due to this data dependency between each iteration, outermost for-loop should run in order.
2. **At every iteration, there are 4 computational streams which should be executed in order**
   To correctly compute LU decomposition, execution order in each iteration should be (A) finding maximum row at column k of *input* matrix, (B) swapping elements, (C) updating *lower* matrix, (D) *upper* matrix and (E) updating *input* matrix. Like #1 case, every stream has data dependency. so stream A,B,C and D should be in order.

Based on these findings, parallelization strategy is constructed.

Detail explanation of parallelization strategy

- **2-dimensional array allocated at heap memory**
  As a matrix structure, 2D array with dynamically allocated matrix using *new* keyword was selected. Vector was ignored because computation in LU decomposition has more scalar computation rather than vector computation. 1-dimensional single array was also considered. However, there occurred more branch instructions per iteration to update input matrix (For example, if-statement is needed to check current matrix index is target index to update because it is single contiguous array). it could be longer latency.

- Line 8, **forking the worker threads**
  To prevent multiple forks at each iteration, worker threads are forked at the top of iteration. *Default(none)* is set to avoid incorrect sharing directive. *firstprivate(k, matrix_size)* is set because those variables can be private to threads and should be initialized. Other variables are set to *shared*.

- Line 10, **finding maximum row**
  for-loop of this computation can be parallelized. However, max_idx and max_value should be reduced between each thread. So *reduction(max: max_idx)* and *reduxtion(max: max_value)* are set to achieve reduction operations. *nowait* predicate is not used because there is data dependency between stream A and B.

- Line 15, **swapping elements**
  This section B should only run once per iteration and should locate between stream A and C due to data dependency (output of stream A is input of stream B and output of stream B is input of stream C). Therefore, *single* predicate is used not master predicate. This choice is driven from line 8. Because I chose to avoid multiple forks, this synchronization with single predicate is unavoidable.

- line 22, **updating lower matrix**
  To parallelize this for-loop, block distribution is used. Because line 24 accesses rows with same column which may generate cache miss at every access. I tried to transpose it to address cache miss, but another cache miss will occur during transpose. And there will be no false sharing because every row accessing will fill different data at cache line.
  Moreover, *nowait* predicate is used. Because this stream C and next stream D have no data or control dependency.

- Line 25, **updating upper matrix**
  To parallelize this for-loop, block-cyclic distribution is used. Because line 27 accesses columns with same row which may generate false-sharing. To avoid it, chunk size is set to 8 which is same with (sizeof cacheline / sizeof double). By static scheduling, each worker thread can update upper matrix in round-robin manner with minimized false sharing.

- Line 28, **updating input matrix**
  This for-loop is most time-consuming part. To parallelize it, block distribution is used. Because outer-most loop is parallelized, false-sharing can be avoided at line 31. In this case, block-cyclic manner can degrade spatial locality. Alternatively, lower matrix can be transposed to reduce cache miss. But this is not same with semantic of lower matrix.

Above strategies are selected to parallelize LU decomposition. However, there are more parts which can be parallelized. Initializing input matrix and swapping elements are examples. They weren't parallelized because parallelization on them didn't dramatically enhance overall performance. It seemed unnecessary to generate additional synchronization due to minor parallelism in those insignificant parts.

3. Performance of parallelization strategy

   a. Execution time on different number of threads

# threads \ matrix size	1024	2048	4096
Baseline (serial)	0.495788	4.2751	33.3469
1 thread	0.483854	4.21668	32.6574
2 threads	0.283926	2.50486	21.0299
4 threads	0.195707	2.15892	18.0296

   b. Parallel efficiency

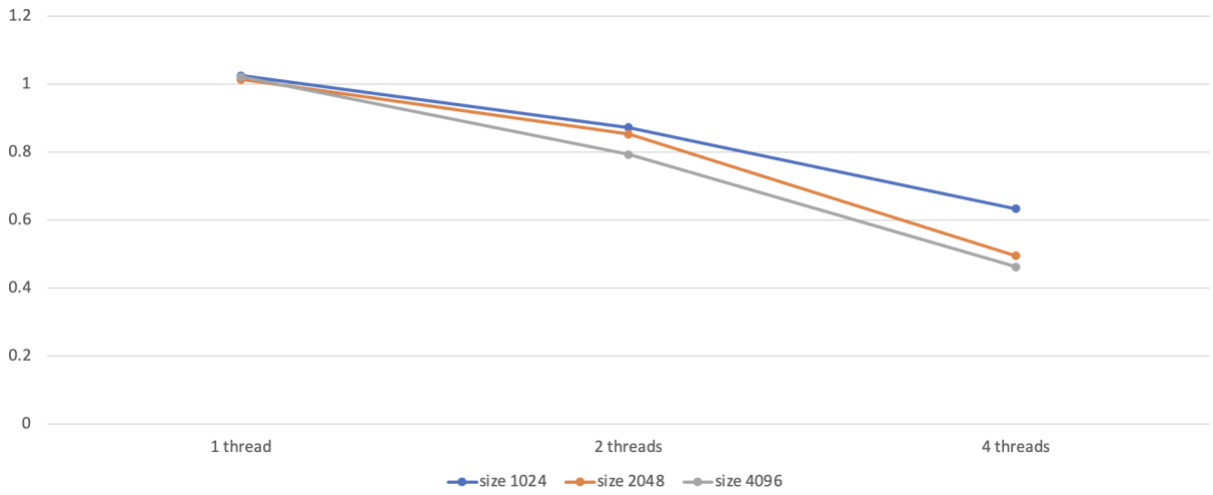

Table (a) shows that execution time of parallel algorithm with different size of matrix and number of threads. And graph (b) shows that efficiency (s/(p*T(p))) of each execution.
In table (a), we can see that execution time decreases as the number of threads increases. It implies that proposed algorithm did achieve speed-up. However, in graph (b), efficiency decreases as the number of threads increases. This means that proposed algorithm is not ideal to utilize multiple parallelized threads. There can be many factors for degrading performance. But major is fork that occurs repeatedly. Because outer-most loop should be in order, fork at every iteration is unavoidable. Although forking thread is lighter than forking process, It still needs significant amount of allocating and deallocating memory and managing context. Therefore, this repeated fork could be overhead.
We can observe that overall efficiency decreases as the size of matrix increases. Memory limit would affect this performance degrade. The larger the matrix size, the more the memory is required. But memory size is limited. L1, L2 and L3 cache which is on chip alone can't handle large matrix. Then memory access becomes DRAM access which takes longer time to data transfer. This explicitly degrades performance.