Report of Assignment 2

Hanzhang Song studentID:hs50

October 20, 2018

1 Introduction of the Program's Organization and Some Configuration

There is an out iteration in the LU-decomposition algorithm. I do not consider parallelise this out iteration, for each iteration I have the following four tasks to do:

- Find the max absolute value of a column in A.
- Swap two whole rows of A and two partial rows of L.
- Update a row of U and a column of L.
- Using the updated row of U and column of L to update the submatrix of A.

In the OMP version, the four tasks is executed sequantially in my program. In the Pthread version, I tried to early lanch and combine the last two tasks. Detail performance and analysis will be illustrated in later sections.

I modified the Makefile, added a new compile object ${OBJ}–noverify. After verifying the results are right, I only this version to measure the performance.

2 OMP Part

After several experiments and with the help of hpctools, I find most of the time is spent on the last task. So I do nothing special for the first three tasks in my OMP version of code, but only set a reasonable schedule routine. After several tries, I set the schedule method to static and the trunc paramete is 100. The following content will only introduce how I parallelise the last task.

2.1 Memory Allocation and Thread Binding

Different ways of memory allocation will affect the performance a lot in this experiments. For OMP, given the configuration of #SBATCH –exclusive, OMP_PROC_BIND=spread, we can see the thread binding info like this:

tid 3613 thread 0 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}
tid 3614 thread 1 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}
tid 3615 thread 2 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}
tid 3616 thread 3 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}
tid 3617 thread 4 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}

tid 3618 thread 5 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}
tid 3619 thread 6 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}
tid 3620 thread 7 bound to OS proc set {0,1,2,3,4,5,6,7,16,17,18,19,20,21,22,23}
tid 3621 thread 8 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}
tid 3622 thread 9 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}
tid 3623 thread 10 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}
tid 3624 thread 11 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}
tid 3625 thread 12 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}
tid 3626 thread 13 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}
tid 3627 thread 14 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}
tid 3628 thread 15 bound to OS proc set {8,9,10,11,12,13,14,15,24,25,26,27,28,29,30,31}

I find that there are always half of the total threads running on the first CPU whose socket number is 0 and the other half running on the second CPU whose socket number is 1.

So I divide matrix A, U, L by row, in other words store an array of vectors for each of them. I allocate the memory for each row parallelly. In each thread it will use numa to allocate the memory like this:

```
#pragma omp for schedule(static, 1)
for (...)
  A[row] = (double*)numa_alloc_local(sizeof(double)*matrix_size);
```

In this way the I can know who is the allocator of this row given the row number. For example the thread id of the nth row's allocator is $n\%$ total_thread_num. For now, I only care about matrix A. If I use 16 threads to work, the memory layout of matrix A will be like this:

Figure 1: memory allocation of matrix A

The advantages of this way of memory allocation are:

- I can always use two sockets in the whole calculation.
- The relation between row number and thread id is simple, which means the calculation of index is simple and the code is clear.

Give a row I will let its allocator thread to do the calculation because the memory is local for the thread. To gurantee this point, I should manually manage the parallalism instead of simply using **for schedule**. Given a thread id, the rows this thread is responsible to calculate is: {thread_id, thread_id + 1*total_thread_num, thread_id + 2*total_thread_num, ... }. In each iteration I only need to do the calucation on a submatrix of A (from k+1 to the end), so the code of each thread's for is like:

```
int row = k+1 + (thread_id-(k+1)%nworkers);
if (row < k+1) row += nworkers;
for (; row < matrix_size; row+=nworkers)
  ...
```

The non-local version of code simply allocate all memory on one socket, the local version of code use the manual method above to do the task. Their performance comparison is like this:

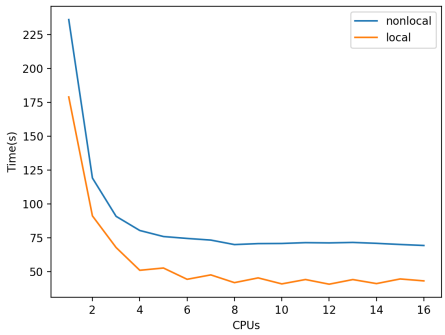

Figure 2: Local vs Nonlocal

2.2 Some Improvements about Matrix U and L

During the computation, I find that I always need to access some column of L. So instead of storing each row of L, it is better to store each column of L to avoid cache miss. A easy way to do this is to store the transposed L. Besides, matrix U and L are both triangle matrix, which means I only need to store half of them. So cache space will be saved for more useful data. The performance comparison between adapting these improvements and without them is show in figure 3. Notice

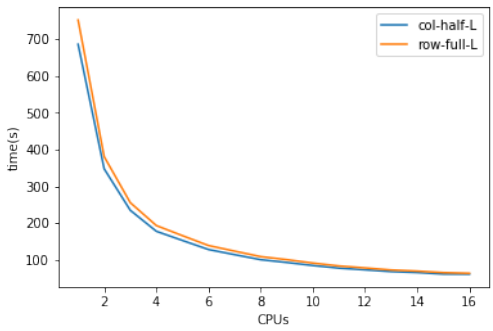

Figure 3: Column half vs Row full

that the performance data here is not good because I did this experiments before I move to solve the memory allocation problem. But I keep all the settings same except the ways I store matrix L and U. So this still can prove the improvement.

2.3 Parallel Efficiency Analysis

The parallel efficiency is show in figure 4 We can see that the parallel efficiency decrease quickly after 4 CPUs. Actually after 8 CPUs the performance goes bad, in other words the running time

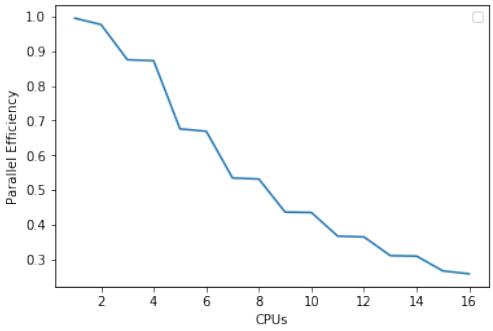

Figure 4: Parallel efficiency

becomes longer. This means that the overhead dominates the performance now. Another disadvantage about adding too many CPU into computation is that this will incur more cache miss. Because each CPU is responsible for calculating different rows and the memory of these rows are not continuously allocated, memories cached by one CPU will not hit the memory address needed by another CPU. So it is reasonable to use less CPU to do the job when the submatrix of A shrink to a small one. I implement it by using a large enough trunc parameter for schedule. Such that most of the threads just do nothing but fork and join. The performance comparison is shown in figure 5.

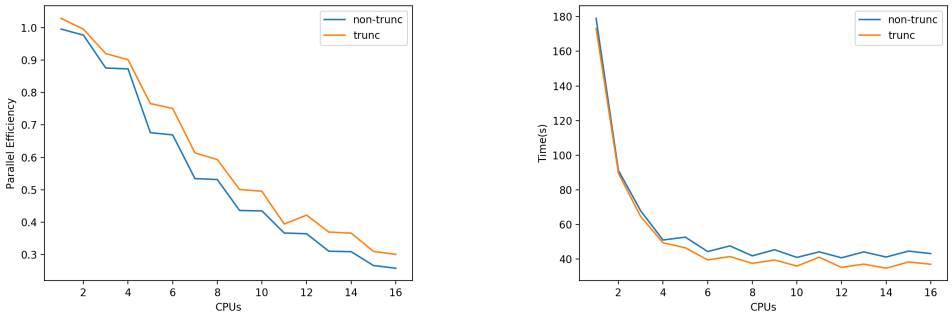

Figure 5: Performance of avoiding overuse CPUs

The parallel efficiency is still not good. I think the reason is that there is no work for the superfluous CPUs. I do not give them jobs because that once I asign work for these CPUs, it will cause large overhead.

3 Pthread Part

Most of the strategy of the Pthread version is as same as the OMP version's. It is a little trivial to bind threads with CPUs by using pthread. I achieve this by:

```
cpu_set_t cpuset_socket_0;
cpu_set_t cpuset_socket_1;
CPU_ZERO(&cpuset_socket_0);
CPU_ZERO(&cpuset_socket_0);
for (int i = 0; i < 8; ++i) {
  CPU_SET(i, &cpuset_socket_0);
  CPU_SET(i+16, &cpuset_socket_0);
```

```
  CPU_SET(i+8, &cpuset_socket_1);
  CPU_SET(i+24, &cpuset_socket_1);
}
...
if(tid <= n/2) {
  s = pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), s0);
 }
else {
  s = pthread_setaffinity_np(pthread_self(), sizeof(cpu_set_t), s1);
}
```

In the code, I use macro function to implement it.

3.1 Performance of Pthread

Pthread allows me to control the threads subtly. There are some tries I did with pthread.

- Allocate the memory of A in a different way like figure 7.
- Explicitly truncate the calculation by using less threads to work.
- Combine the last two tasks together.

Although I believe by using pthread I can get a better performance, I did not achieve it. The performance of pthread is shown in figure 6

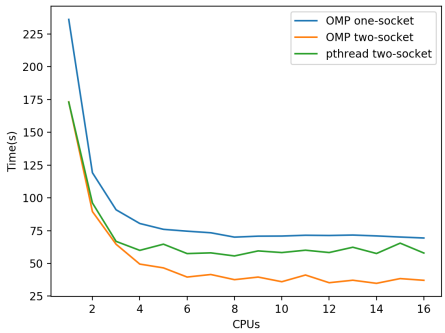

Figure 6: Pthread performance

3.2 A Failed Try and A Possible Improvement

With pthread I can do some subtle work, so I want to combine the last two tasks together by using condition wait between threads. My algorithm is as following:

```
// each thread
update_parts_U();
update_parts_L();

// tell other threads the data is ready
lock();
```

Figure 7: Another memory allocation of matrix A

```
pthread_cond_broadcast();
unlock();
update_parts_A();

while(there is work to do) {
  lock();
  while (data is not ready) pthread_cond_wait();
  unlock();
  // do the work
  ...
}
```

Although the result is right, I failed to achieve a better performance. A possible reason is that this algorithm divide each row of A into small fragments, which breaks the memory's continuity and will cause a lot of cache miss. Another reason I can come up with is maybe I use too many mutex lock. I still leave the code and some detail comments there, waiting for future solution.

From the performance data, it is clear that using more than 8 CPUs will not improve the performance. So it is reasonable to free some CPU and let them to be 'masters' to control other 'slave' CPU. However the control flow will be more complicated.