Report of Gaussian Elimination using Pthread

- **Algorithm Description & Correctness Argument**

The code below is the Gaussian Elimination using Pthread, while creating the threads, I return a thread id to this function, based on the thread id, I divide the myrow loop into several static chucks with size--(N-norm-1)/NUM_THREADS+1.

Since all the variables using by different threads is based on global variable norm and private variable thread_id, the private variables cannot be intersected. In another word, the loops contains no data dependence.

```
void *gauss(void *pID) {
  int col, myrow;
  int thread_id = (int)(long)pID; /*get the thread id*/
  int start, end;
  float multiplier;

  /* Gaussian elimination */
  /* Define a static way to parallelize the Gaussian elimination.
   * start represents the start of the static chunk, end represents the end of the static chunk.
   * As for min(), it is used to limit the end to N, else if N-norm-1 cann't
   * be exact divided by NUM_THREADS, the end may be larger than N, which causes core
dumped.
   * Since start, end, myrow is related to thread_id, and norm is synchronized,
   * the values in each thread won't be intersected.
   */
  while (norm < N-1) {
    start = norm + 1 + thread_id*((N-norm-1)/NUM_THREADS+1);
    end = min(N, norm + 1 + (thread_id+1)*((N-norm-1)/NUM_THREADS+1));
      for (myrow = start; myrow < end; myrow++) {
        multiplier = A[myrow][norm] / A[norm][norm];
        for (col = norm; col < N; col++) {
              A[myrow][col] -= A[norm][col] * multiplier;
        }
        B[myrow] -= B[norm] * multiplier;
      }
  barrier(&norm);
  }
}
```

The barrier function handles the synchronization problem, when a thread has finished the myrow loop in the code above, it will execute the barrier function, if the other threads have not finished, this thread will be locked.

Not until all the threads have finished, the norm will not increase. By executing the barrier, the threads can be synchronized perfectly.

```
/* Define a barrier to synchronize the thread.
 * When a thread is finished, count--, when count==0, it means all the threads have finished
 * And after NUM_THREADS threads have finished, increase norm by 1
 */
void barrier(int *norm){
  pthread_mutex_lock(&count_lock);

  if(count==0){
    (*norm)++;
    count = NUM_THREADS-1;
    pthread_cond_broadcast(&next);
  }
  else{
    count--;
    pthread_cond_wait(&next, &count_lock);
  }
  pthread_mutex_unlock(&count_lock);
}
```

- **Different Version's Performance & Performance Analyze**

My laptop has 4 cores, and according to tests, the optimal number of threads is equal to the number of cores.

If the number of threads is less than 4, definitely it can't reach the best performance since there are redundant cores .

And if the number of threads is growing, according to the test, the system CPU time for parent will increase, the performance will degrade rather than improve.

Here is the test result of different versions for 2000*2000 matrix.

Original serial program running time: 8483.16ms

Parallel performance:

	Running time	Speedup
2 threads	4726.52ms	1.79
4 threads	4518.91ms	1.87
6 threads	4847.34ms	1.75
10 threads	4908.89ms	1.73
100 threads	7243.35ms	1.17
200 threads	10001.6ms	0.84