ECE 420 Assignment 1

Jiannan Lu 157761

1. SISD: Uniprocessor

   SIMD: Array processors

   MISD: Fault tolerance, but mostly nonsense

   MIMD: multi-core processor (most modern PCs)

2. Shared-memory:

   adv -> Provides a user-friendly programming perspective to memory

   Data sharing between tasks is both fast and uniform

   con -> No scalability between memory and CPUs

   Distributed-memory:

   adv -> Memory is scalable with the number of processors.

   Cost effective, can use commodity, off-the-shelf processors.

   con -> Data communication between processors.

   Difficult to map existing data structures to the memory

   Non-uniform memory access times: remote vs. local

3. If y fraction of a serial program cannot be parallelized, then 1-y fraction of a serial program can be parallelized, according to the law, $\lim_{p\to\infty} S(p) \le \frac{1}{1-(1-y)} \le \frac{1}{y}$, so that 1/y is the upper bound on the speedup of its parallel program, no matter how many processing elements are used.

4. No.

$$\frac{n1}{n1 + p\log_2 p} = \frac{n2}{n2 + kp\log_2 kp}$$

$$1 + \frac{p\log_2 p}{n1} = 1 + \frac{kp\log_2 kp}{n2}$$

$$n2 = n1 * \frac{k\log_2 kp}{\log_2 p}$$

5. No. The first program outputs “The worker thread has returned the status 10”

   The second program outputs “Segmentation fault”, because the second one has b points to a local variable that will be eliminated after thread exits.

6.
```
pthread_mutex_lock(&(b->count_lock));
b->count += 1;
if (b->count == num_threads) {
    // if all threads is here, reset count and broadcast
    b->count = 0;
    pthread_cond_broadcast(&(b->ok_to_proceed));
} else {
    // wait until receive the signal
    while (pthread_cond_wait(&(b->ok_to_proceed));
)
pthread_mutex_unlock(&(b->count_lock));
```

7.
```
pthread_mutex_lock(&(buffer.mutex));
if (buffer.occupied == 0) printf(“consumer waiting.\n”);
```

```
while (buffer.occupied == 0)
        pthread_cond_wait(&(buffer.more), &(buffer.mutex));
printf("consumer executing.\n");
item = buffer.buf[buffer.nextout];
buffer.nextout++;
buffer.nextout %= BSIZE;
buffer.occupied--;
pthread_cond_signal(&(buffer.less));
pthread_mutex_unlock(&(buffer.mutex));
```

8. 8*8000000 is most likely to have the false sharing problem than others. Because number of rows is small, suppose they are on the same cache line, each time other thread want to update any of the variables, it has to read the whole cache line. Which is most likely to have the false sharing problem.
   The other two, because of the large row size, it takes more time for all the way down to the last element of thread1, thread2 has already done with most of it's elements at that time, unlikely false sharing problem.
   way1: pad column vector with 0s, elements will be forced to different cache lines then.
   way2: without sharing a global matrix A, make a copy for each thread, and combine the changes at the end.