ECE 420 Parallel and Distributed Programming Assignment 1

Instructor: Di Niu
Email: dniu@ualberta.ca
Department of Electrical and Computer Engineering
University of Alberta
Due date: see the schedule posted on course website

Note: this assignment provides some sample questions that are representative of the questions to appear in exams, although questions in exams will be asked in a more formal and rigorous way. The marking of this assignment is largely based on efforts. Solutions will be posted after the due date.

1. Classify computer hardware architectures using Flynn's taxonomy. Name an example for each category.

2. What are the advantages and concerns of programming on shared-memory and distributed-memory systems, respectively?

3. (Amdahl's law) If y fraction of a serial program cannot be parallelized, prove that $1/y$ is an upper bound on the speedup of its parallel program, no matter how many processing elements are used.

4. Suppose in a parallel computing task, T_serial = n and T_parallel = $n/p + \log_2(p)$, where p is the number of processors and n is the problem size. If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes, then the program is said to be *scalable.* Is the above program scalable? If we increase p by a factor of k, by how much we'll need to increase n in order to maintain a fixed efficiency?

5. Do the following two programs have the same output? Why or why not?

```
/* Program 1 */
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* thread_function(void * arg)
{
    int * b = (int *) malloc(sizeof(int));
    *b = 10;
    pthread_exit((void*)b);
}
int main()
```

```
{
    pthread_t thread_id;
    int *status;

    pthread_create(&thread_id, NULL, &thread_function, NULL);

    pthread_join(thread_id, (void**)&status);

    printf("The worker thread has returned the status %d\n", *status);
    pthread_exit(NULL);
}


/* Program 2 */
#include <stdio.h>
#include <pthread.h>
#include <stdlib.h>

void* thread_function(void * arg)
{
    int number = 10;
    int * b = &number;
    pthread_exit((void*)b);
}
int main()
{
    pthread_t thread_id;
    int *status;

    pthread_create(&thread_id, NULL, &thread_function, NULL);

    pthread_join(thread_id, (void**)&status);

    printf("The worker thread has returned the status %d\n", *status);
    pthread_exit(NULL);
}
```

6. Complete the following code that implements a barrier using condition variables and mutex.

```
typedef struct {
     pthread_mutex_t count_lock;
     pthread_cond_t ok_to_proceed;
     int count;
} mylib_barrier_t;

void mylib_init_barrier(mylib_barrier_t *b) {
     b -> count = 0;
     pthread_mutex_init(&(b -> count_lock), NULL);
     pthread_cond_init(&(b -> ok_to_proceed), NULL);
}

void mylib_barrier (mylib_barrier_t *b, int num_threads) {
```

```
    /* Insert here */

}
```

7. In this question, you will be asked to complete the implementation of a producer/consumer buffer using mutex and condition variables. There is a single producer and a single consumer. The producer has 6 items, each being a character in the string `"HELLO."` it declares at the beginning of the producer thread. The producer inserts these 6 items one by one into the buffer which can only hold up to 3 items. These items will be inserted into the 3 buffer positions following the order 0, 1, 2, 0, 1, 2… If the buffer is full at some point, the producer will wait for the consumer to retrieve items until an empty position appears. Then the producer can continue. On the other hand, the consumer retrieves and prints items from the buffer positions 0, 1, 2, 0, 1, 2,… until there is nothing to be read, at which point, it will wait for a new item to be deposited by the producer to proceed.

   Read the following program partly provided to you. Complete it by inserting proper code at the place marked **`/* Insert here */`**. Note that the an `int` member of a user-defined `struct` is by default initialized to 0.

```
#include <pthread.h>
#include <stdio.h>

#define BSIZE 3
#define NUMITEMS 6

typedef struct {
    char buf[BSIZE];
    int occupied;
    int nextin, nextout;
    pthread_mutex_t mutex;
    pthread_cond_t more;
    pthread_cond_t less;
} buffer_t;

buffer_t buffer;

void * producer(void *);
void * consumer(void *);

#define NUM_THREADS 2
pthread_t tid[NUM_THREADS];      /* array of thread IDs */

main( int argc, char *argv[] )
{
    int i;

    pthread_cond_init(&(buffer.more), NULL);
    pthread_cond_init(&(buffer.less), NULL);
    pthread_mutex_init(&buffer.mutex, NULL);
```

```
    pthread_create(&tid[1], NULL, consumer, NULL);
    pthread_create(&tid[0], NULL, producer, NULL);
    for ( i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);

    printf("\nmain() reporting that all %d threads have terminated\n", i);

}  /* main */

void * producer(void * parm)
{
    char item[NUMITEMS]="HELLO.";
    int i;

    printf("producer started.\n");

    for(i=0;i<NUMITEMS;i++)
    { /* produce an item, one character from item[] */

        if (item[i] == '\0') break;  /* Quit if at end of string. */

        pthread_mutex_lock(&(buffer.mutex));

        if (buffer.occupied >= BSIZE) printf("producer waiting.\n");
        while (buffer.occupied >= BSIZE)
            pthread_cond_wait(&(buffer.less), &(buffer.mutex) );
        printf("producer executing.\n");

        buffer.buf[buffer.nextin] = item[i];
        buffer.nextin++;
        buffer.nextin %= BSIZE;
        buffer.occupied++;

        /* now: either buffer.occupied < BSIZE and buffer.nextin is the index
         of the next empty slot in the buffer, or
         buffer.occupied == BSIZE and buffer.nextin is the index of the
         next (occupied) slot that will be emptied by a consumer
         (such as buffer.nextin == buffer.nextout) */

        pthread_cond_signal(&(buffer.more));
        pthread_mutex_unlock(&(buffer.mutex));
    }
    printf("producer exiting.\n");
    pthread_exit(0);
}

void * consumer(void * parm)
{
    char item;
    int i;

    printf("consumer started.\n");
```

```
    for(i=0;i<NUMITEMS;i++){

        /*Insert here*/

    }
    printf("consumer exiting.\n");
    pthread_exit(0);
}
```

8. Suppose that in the following program, A and x are read from a file. For three matrices A with input sizes 8 × 8,000,000, 8000 × 8000, and 8,000,000 × 8, respectively, which is most likely to have the false sharing problem? Why? Why the other input sizes are unlikely to suffer from false sharing? Suggest two ways to modify the following program to avoid false sharing.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

/* Global variables */
int     thread_count;
int     m, n;
double* A;
double* x;
double* y;

/* Parallel function */
void *Pth_mat_vect(void* rank);

/*--------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   long       thread;
   pthread_t* thread_handles;

   thread_count = atoi(argv[1]);
   thread_handles = malloc(thread_count*sizeof(pthread_t));

   A = malloc(m*n*sizeof(double));
   x = malloc(n*sizeof(double));
   y = malloc(m*sizeof(double));

   /* Read A, x, m, n from a file (code omitted) */

   for (thread = 0; thread < thread_count; thread++)
      pthread_create(&thread_handles[thread], NULL,
         Pth_mat_vect, (void*) thread);

   for (thread = 0; thread < thread_count; thread++)
      pthread_join(thread_handles[thread], NULL);
```

```
   Print_vector("The product is", y, m);

   free(A);
   free(x);
   free(y);

   return 0;
}  /* main */

void *Pth_mat_vect(void* rank) {
   long my_rank = (long) rank;
   int i, j;
   int local_m = m/thread_count;
   int my_first_row = my_rank*local_m;
   int my_last_row = (my_rank+1)*local_m - 1;

   for (i = my_first_row; i <= my_last_row; i++) {
      y[i] = 0.0;
      for (j = 0; j < n; j++)
          y[i] += A[i*n+j]*x[j];
   }

   return NULL;
}  /* Pth_mat_vect */
```