ECE 420 Parallel and Distributed Programming Assignment 1 Solutions

Instructor: Di Niu
Email: dniu@ualberta.ca
Department of Electrical and Computer Engineering
University of Alberta

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

    SISD: Uniprocessors
    SIMD: Array processors, GPU
    MISD: N/A
    MIMD: Multi-core processors (most modern PCs), clusters, network of workstations

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

Shared Memory Programming
Pros: Provides a user-friendly programming perspective to memory
Data sharing between tasks is both fast and uniform
Cons: No scalability between memory and CPUs
Need to correctly access shared resources, and handle synchronization

Distributed Memory Programming
Pros: Memory is scalable with the number of processors.
Cost effective, can use commodity, off-the-shelf processors.
Cons: Data communication between processors.
Difficult to map existing data structures to the memory organization.
Non-uniform memory access times: remote vs. local

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.

See slides.

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?

No.
In order to maintain a fixed efficiency, we should have

$n/(n + p\log_2(p)) = n'/(n' + kp\log_2(kp))$

Thus, $n' = nk(\log pk)/\log p$.

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

No. The first one will output

```
The worker thread has returned the status 10
```

The second one will output the following on a Mac:

```
Segmentation fault
```

The output on Linux and Windows may be different than the above, but definitely not 10.

The reason is that in the second program, the int pointer b is pointing to a local variable "number" on the stack of the thread created. This variable is gone when the thread terminates.

```
/* 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) {

     pthread_mutex_lock(&(b -> count_lock));
     b -> count ++;
     if (b -> count == num_threads) {
          b -> count = 0;
          pthread_cond_broadcast(&(b -> ok_to_proceed));
     }
     else
          while (pthread_cond_wait(&(b -> ok_to_proceed),
               &(b -> count_lock)) != 0);
     pthread_mutex_unlock(&(b -> count_lock));

}
```

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++){

  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++];
  printf("%c\n",item);
  buffer.nextout %= BSIZE;
  buffer.occupied--;

  /* now: either buffer.occupied > 0 and buffer.nextout is the index
     of the next occupied slot in the buffer, or
     buffer.occupied == 0 and buffer.nextout is the index of the next
     (empty) slot that will be filled by a producer (such as
     buffer.nextout == buffer.nextin) */

  pthread_cond_signal(&(buffer.less));
  pthread_mutex_unlock(&(buffer.mutex));
  }
  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.

   8 × 8,000,000 will have the false sharing issue, since if y[0], …, y[7] lie on the same cache line, every time a thread updates a y[i], the cache controller will invalidate the whole cache line. When some other thread updates another variable y[j], although y[j] and y[i] are different variables, this thread will be forced to read the whole cache line from the main memory due to the invalidate signal.

   The other input sizes are unlikely to suffer from false sharing, because the number of rows is large. Suppose there are 4 threads, the only variables that may share a same cache line is on the boundaries. For example, for 8000 × 8000, thread 0 handles y[0], …, y[1999], and thread 1 handles y[2000], …, y[3999]. Here, y[1999] and y[2000] belonging to different threads may share the same cache line. However, when thread 0 computes its last element y[1999], thread 1 should be done with its first element y[2000] long ago. Therefore, false sharing is unlikely.

One method to avoid false sharing is to let each thread update its local variables first and then update the global variables in one pass in the end. See the following new code.

```
#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;
   double* y_local;
   y_local = malloc(m*sizeof(double));

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

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

Another method to avoid false sharing is to pad the vector y with dummy elements, so that each element of y is on a different cache line. This can be done by declaring the vector y as a 2-D array, but only using the first element of each row in this 2-D array to store a y[i].