Umeå University
Department of Computing Science

Parallel Programming 7.5 p
5DV152

Exercises, Chapter/Topic 4

Submitted	2017-02-23
Author:	Lorenz Gerber (`dv15lgr@cs.umu.se` `lozger03@student.umu.se`)
Instructor:	Lars Karlsson / Mikael Ränner

Contents

1 Introduction

This report is part of the mandatory coursework. It describes the solutions for several chosen exercises from the course book [?].

2 4.1 - Generalization of matrix-vector multiplication

If we keep the same scheme of parallelization as mentioned in the book (outer for...loop), generalzation can be implemented rather easy, bascially in the same way as already shown in exercise 1.1:

```
my_first_i = k * m / p + ( k < m mod p ? k : m mod p )
my_last_i = (k + 1) * m / p + (k + 1 < m mod p ? k + 1 : m mod p)
```

It is not useful to parallelize into n as one thread needs to process as this would create a mutex for acces to shared variables.

3 4.2 - Physical data distribution

The source code for this exercise can be found in appendix **??**. Timing:

4 4.8 - Deadlock

a) Both thread wait for the other to release the lock

b) busy waiting with two flag variables would create the same situation as two mutex locks

c) if two semaphores are used, they act more or less in the same way as mutex locks

The main problem in the mentioned program is that it tries to acquire locks from within critical sections, interleaved.

5 4.11 - Linked list troubles

The linked list implementation from section 4.9.2 of the course book was assumed for this exercise. The situations to be described are shown in figure 1.

a) Here, first both threads have to find the element to be removed, 5 and 6. They will first probably first redefine the linked list pointer from their *pred_p* element to point to the next element of the *curr_p* element instead of the *curr_p* itself. Here it could happend that the thread that wants to delete element 8 will never arrive at its destined element as the other thread has already redefined the pointer. In the next step The pointer from *curr_p* will be set to *null* and the actual element free'd. Here again, one problem that could occur is that the thread to delete element 6 will not arrive at his destination because of a *null* pointer. Another obvious problem is that if the thread to delete element 5 will probably already have redifined the pointer from element 2 to element 8, when the other thread will free element 8. Two delete operations will

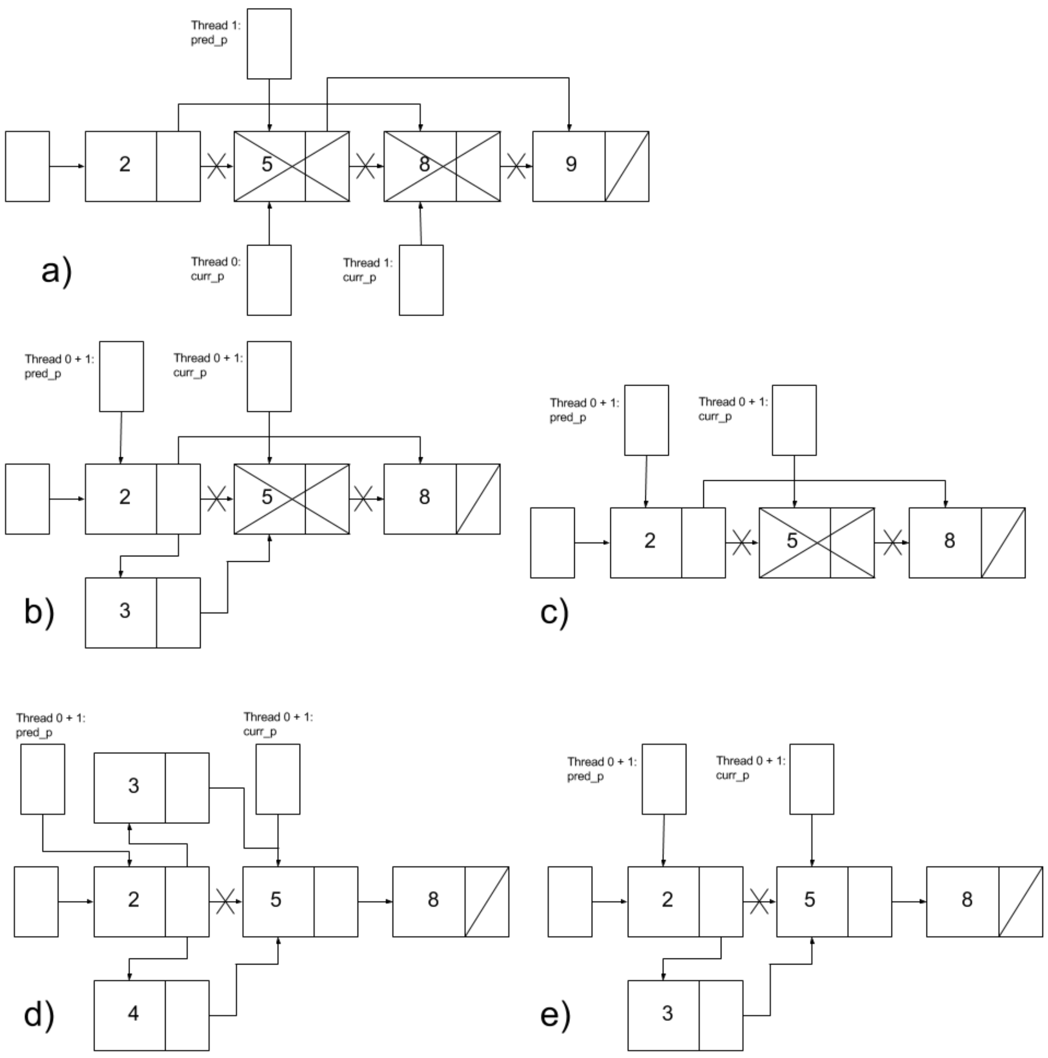

Figure 1: *This graph shows the multithreaded linked list situations from a) to e) that are further described in the text.*

certainly result in either a run time error immediately, or incorrect data structure that probably at a later point in time will lead to a run time error or worse, incorrect results.

b) If an insert and a delete are conducted on the same element, here element 5, the most obvious problem is that the new element 3 will point to a free'd element. Even if element 5 wouldn't be overwritten in the memory yet, it's pointer will be set to *null* anyway, hence the connection to the next list element is lost. Another problem is that both threads will try to redefine the pointer of element 2. Also this situation will probably result in run-time errors such as segmentation fault. If not immediately then probably at a later access. In a rare case when the program will not crash, it seems most likely that the data structure represented will be the linked list 2, 5 with 3 missing.

c) Here one thread wants to delete element 5 while another thread attempts a member operation on element 5. If thread doing a member operation is earlier, it will succeed in his operation, but in respect to the situation after the delete report a wrong value as element 5 does not exist anymore. In some cases, the thread to delete 5 could already have redefined the pointer in element 2 hence the memeber operation would then probably be executed on list element 8 instead of 5. Altough, note, that this could be the intended action. Basically, it will represent the current situation and the data structure is still sane.

d) When two threads try to insert a new element (3 and 4) at the same place (before element 5), it is obvious that the redefination of pointer from element 2 will happen sequential (or else a segmentation fault will happen already here), hence the linked list will incorporate either element 3 or 4. Probably here the chance is big that the program will not crash but that it will produce wrong results.

e) An insert and a member operation from two separate threads on the same list element 5, will most likely result in a wrong result from the member operation in respect to the new data structure. However, in most cases the program should not crash. Depending on timing, it could also happen some times that the member operation will be conducted already at on the new element 3. Note that it is not defined what is correct here, a member function reporting element 3 or 5. This could also be seen as a race condition with undetermined behaviour.

6 4.12 - Linked list insert and delete with read-write lock

No, it's not safe. Another thread could also have the read-lock and request the write lock at the same time. Then only one of two threads will initially get the write-lock and be able to modify the list. When the second thread finally get's the write lock, the list is probably already modified from the first write access and the program will crash.

7 4.17 - False sharing

8 A4.1 - Histogram

9 A4.3 - Trapezoidal rule

10 A4.4 - Fork/join overhead

11 A4.5 - Task queue

A C Source Code for Exercise 4.2

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

/* Global variables */
int     thread_count;
int     m, n;
int     input_counter = 0;
int     output_counter = 0;
pthread_mutex_t mutex_input;
pthread_mutex_t mutex_output;
pthread_cond_t cond_var_input;
pthread_cond_t cond_var_output;

/* Serial functions */
void Usage(char* prog_name);
void Gen_matrix(double A[], int m, int n);
void Read_matrix(char* prompt, double A[], int m, int n);
void Gen_vector(double x[], int n);
void Read_vector(char* prompt, double x[], int n);
void Print_matrix(char* title, double A[], int m, int n);
void Print_vector(char* title, double y[], double m);

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


/*------------------------------------------------------------------*/
int main(int argc, char* argv[]) {

  long       thread;
  pthread_t* thread_handles;

  if (argc != 4) Usage(argv[0]);
  thread_count = strtol(argv[1], NULL, 10);
  m = strtol(argv[2], NULL, 10);
  n = strtol(argv[3], NULL, 10);
```

```
# ifdef DEBUG
  printf("thread_count =  %d, m = %d, n = %d\n", thread_count, m, n);
# endif

  thread_handles = malloc(thread_count*sizeof(pthread_t));

  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);

   return 0;
}  /* main */


void Usage (char* prog_name) {
   fprintf(stderr, "usage: %s <thread_count> <m> <n>\n", prog_name);
   exit(0);
}  /* Usage */

void Read_matrix(char* prompt, double A[], int m, int n) {
   int             i, j;

   printf("%s\n", prompt);
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         scanf("%lf", &A[i*n+j]);
}  /* Read_matrix */


void Gen_matrix(double A[], int m, int n) {
   int i, j;
   for (i = 0; i < m; i++)
      for (j = 0; j < n; j++)
         A[i*n+j] = random()/((double) RAND_MAX);
}  /* Gen_matrix */


void Gen_vector(double x[], int n) {
   int i;
   for (i = 0; i < n; i++)
      x[i] = random()/((double) RAND_MAX);
}

void Read_vector(char* prompt, double x[], int n) {
   int   i;
   printf("%s\n", prompt);
   for (i = 0; i < n; i++)
      scanf("%lf", &x[i]);
}


void *Pth_mat_vect(void* rank) {
  double* A;
  double* x;
  double* y;
  long my_rank = (long) rank;
  int i;
  int j;
  int local_m = m/thread_count;
  int my_first_row = 0; //my_rank*local_m;
  int my_last_row = m/thread_count; //my_first_row + local_m;
  register int sub = my_first_row*n;
  double start, finish;
  double temp;

# ifdef DEBUG
  printf("Thread %ld > local_m = %d, sub = %d\n",
        my_rank, local_m, sub);
# endif
  // scheduling data input
  pthread_mutex_lock(&mutex_input);

  while(input_counter != my_rank){
    pthread_cond_wait(&cond_var_input, &mutex_input);
  }

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

  Gen_matrix(A, local_m, n);
# ifdef DEBUG
  Print_matrix("We generated", A, local_m, n);
# endif

  Gen_vector(x, n);
# ifdef DEBUG
  Print_vector("We generated", x, n);
# endif


  input_counter++;
  pthread_cond_broadcast(&cond_var_input);
  pthread_mutex_unlock(&mutex_input);

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

  pthread_mutex_lock(&mutex_output);

  while(output_counter != my_rank){
    pthread_cond_wait(&cond_var_output, &mutex_output);
  }
  printf("Thread %ld > Elapsed time = %e seconds\n",
     my_rank, finish - start);

# ifdef DEBUG
  Print_vector("The Product is", y, m/thread_count);
# endif

  output_counter++;
  pthread_cond_broadcast(&cond_var_output);
  pthread_mutex_unlock(&mutex_input);


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

  return NULL;
}


void Print_matrix( char* title, double A[], int m, int n) {
   int   i, j;

   printf("%s\n", title);
   for (i = 0; i < m; i++) {
      for (j = 0; j < n; j++)
         printf("%6.3f ", A[i*n + j]);
      printf("\n");
   }
```

```
}


void Print_vector(char* title, double y[], double m) {
   int   i;

   printf("%s\n", title);
   for (i = 0; i < m; i++)
      printf("%6.3f ", y[i]);
   printf("\n");
}
```