

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	1
2	4.1 - Generalization of matrix-vector multiplication	1
3	4.2 - Physical data distribution	1
4	4.8 - Deadlock	1
5	4.11 - Linked list troubles	1
6	4.12 - Linked list insert and delete with read-write lock	3
7	4.17 - False sharing	3
8	A4.1 - Histogram	4
9	A4.3 - Trapezoidal rule	4
	9.1 Comparison between Mutex, Busy Waiting and Semaphore	5
10	A4.4 - Fork/join overhead	5
11	A4.5 - Task queue	5
	References	5
A	C Source Code for Exercise 4.2	5
B	C Source Code for Exercise A4.1	9
C	C Source Code for Exercise A4.3	16

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), generalization can be implemented rather easy, basically 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 access 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

- Both threads wait for the other to release the lock
- busy waiting with two flag variables would create the same situation as two mutex locks
- 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.

- 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 happen 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 freed. 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 redefined the pointer from element 2 to element 8, when the other thread will free element 8. Two delete operations will

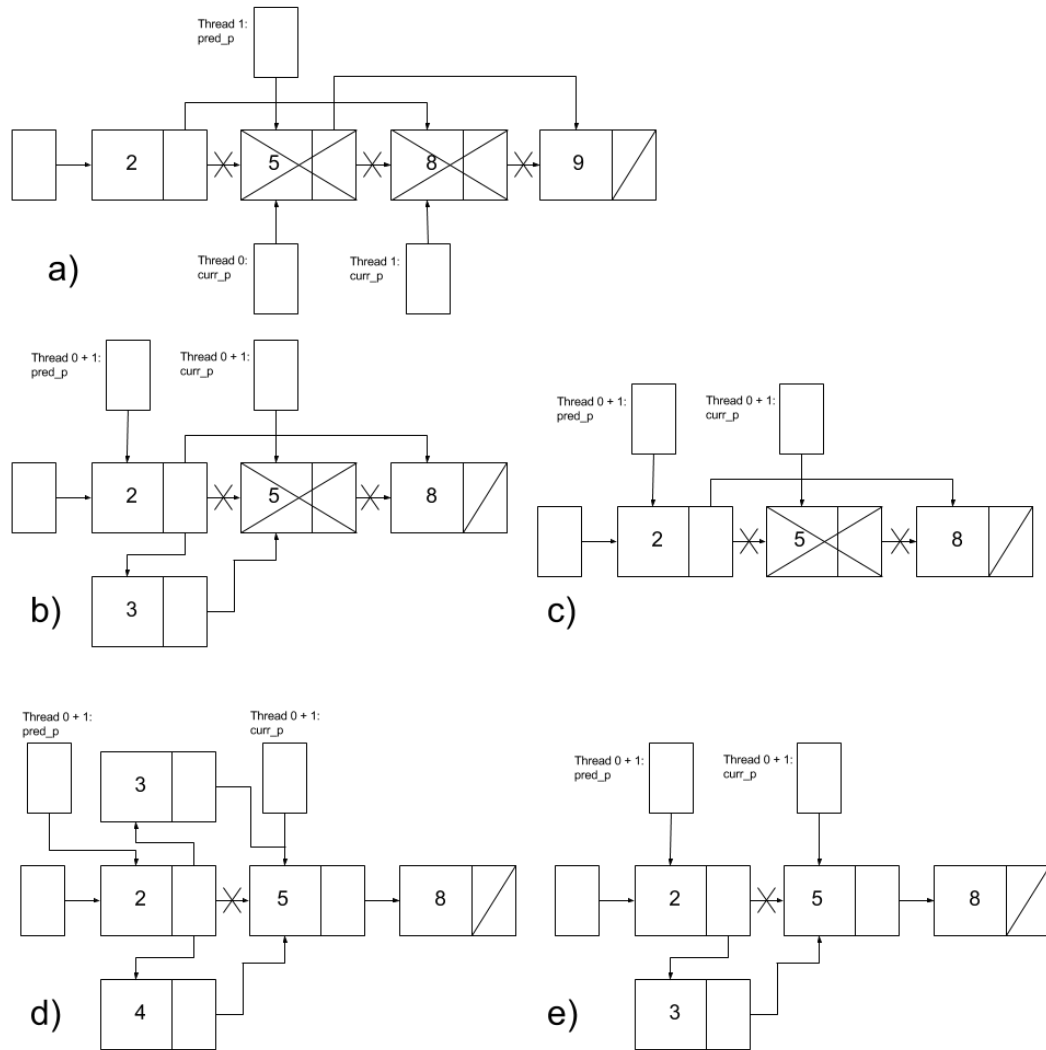


Figure 1: This graph shows the multithreaded linked list situations from a) to e) that are further described in the text.

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, its 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 member operation would then probably be executed on list element 8 instead of 5. Although, 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 redefinition 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

- a) The y vector's length is 8 doubles, hence 64 bytes. The minimum number of cache lines is therefore one.

- b) It is assumed that the vector is a consecutive memory address area. The 64 bytes long vector can therefore stretch over maximum two cache lines.
- c) Stretching over two cache lines, together of length 128 bytes, a 64 bytes long stretch can be divided in 8 bytes steps in 6 different ways.
- d) Four threads on two dual core processors results in choosing 2 from 4 permutation ($\frac{n!}{k!(n-k)!}$). The result is 6.
- e) Yes, if the vector is split half between two cache lines and if both threads from one processor access the vector positions from either the beginning to the middle or from the middle to the end.
- f) The number of such combinations is $6 \times 8 = 48$.
- g) There are six such combinations of thread assignments to the cores so that no false sharing occurs.

8 A4.1 - Histogram

The source code can be found in appendix B. Initially, the serial program from chapter 2 was copied and then modified for pthread support. Basically, the whole data structure is kept global. The data access is not critical. Increasing the count values is controlled by a mutex.

9 A4.3 - Trapezoidal rule

The program was redesigned from the serial trapezoidal-rule program contained in chapter 3 of the book. It was decided that only the loop within the *Trap* function shall be parallelized. Hence the first and last calculation: `integral = (f(a) + f(b))/2.0;` and `integral = integral*h` are executed just once by thread 0. The source code is shown in appendix ???. As the modifications for busy waiting and semaphore are minimal, only the critical sections with the respective code are shown below

Busy waiting

```
while(flag != my_rank);
integral += local_integral;
flag = (flag+1)%thread_count;
```

Additionally to the above code, the flag variable has to be set as a global.

Semaphore

```
sem_wait(&bin_sem);
integral += local_integral;
sem_post(&bin_sem);
```

Additionally, the header *semaphore.h* has to be loaded as well as the semaphore variable initialised and destroyed after usage.

9.1 Comparison between Mutex, Busy Waiting and Semaphore

In the presented implementation, there are some subtle differences between the three different protections for the critical section. Mutex can probably be seen as the standard way providing access to a critical section. Busy waiting in the current implementation allows access only in a predefined order, along ascending rank. This seems for the present application an acceptable limitation, however in many cases, it would be unsuitable. A disadvantage of busy waiting is that it eventually could break when it is forgotten to disable compiler optimization. The implementation of the semaphore was here in a very similar way as the mutex. In my case, I often develop on OSX, it is a disadvantage that *semaphore.h* library will not work here. According to the course book, there is another implementation, named semaphores, that will work on OSX.

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);

```

8(18)

```
# 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");
}

```

B C Source Code for Exercise A4.1

```

/* File:      pth_histogram.c
 * Purpose:   Build a histogram from some random data
 *
 * Compile:   gcc -g -Wall -o pth_histogram pth_histogram.c
 * Run:       ./pth_histogram <bin_count> <min_meas> <max_meas>
 *           <data_count> <thread_count>
 *
 * Input:     None
 * Output:    A histogram with X's showing the number of measurements
 *           in each bin
 *
 * Notes:
 * 1. Actual measurements y are in the range min_meas <= y < max_meas
 * 2. bin_counts[i] stores the number of measurements x in the range
 * 3. bin_maxes[i-1] <= x < bin_maxes[i] (bin_maxes[-1] = min_meas)
 * 4. DEBUG compile flag gives verbose output
 * 5. The program will terminate if either the number of command line
 *    arguments is incorrect or if the search for a bin for a
 *    measurement fails.
 *
 * IPP:  Section 2.7.1 (pp. 66 and ff.)
 */
#include <stdio.h>

```

```

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

/* Global variables */
int thread_count;
int bin_count;
float min_meas, max_meas;
float* bin_maxes;
int* bin_counts;
int data_count;
float* data;
pthread_mutex_t mutex;

/* Serial functions */
void Usage(char prog_name[]);

void Get_args(
    char*      argv[]      /* in */,
    int*       bin_count_p /* out */,
    float*     min_meas_p  /* out */,
    float*     max_meas_p  /* out */,
    int*       data_count_p /* out */,
    int*       thread_count /* out */);

void Gen_data(
    float min_meas /* in */,
    float max_meas /* in */,
    float data[]   /* out */,
    int data_count /* in */);

void Gen_bins(
    float min_meas /* in */,
    float max_meas /* in */,
    float bin_maxes[] /* out */,
    int bin_counts[] /* out */,
    int bin_count /* in */);

int Which_bin(
    float data /* in */,
    float bin_maxes[] /* in */,
    int bin_count /* in */,
    float min_meas /* in */);

void Print_histo(
    float bin_maxes[] /* in */,
    int bin_counts[] /* in */,
    int bin_count /* in */,
    float min_meas /* in */);

```

```

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

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

    long thread;
    pthread_t* thread_handles;

    /* Check and get command line args */
    if (argc != 6) Usage(argv[0]);
    Get_args(argv, &bin_count, &min_meas, &max_meas, &data_count, &thread_count);

    /* Allocate arrays needed */
    bin_maxes = malloc(bin_count*sizeof(float));
    bin_counts = malloc(bin_count*sizeof(int));
    data = malloc(data_count*sizeof(float));

    /* Generate the data */
    Gen_data(min_meas, max_meas, data, data_count);

    /* Create bins for storing counts */
    Gen_bins(min_meas, max_meas, bin_maxes, bin_counts, bin_count);

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

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

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

    # ifdef DEBUG
    printf("bin_counts = ");
    for (i = 0; i < bin_count; i++)
        printf("%d ", bin_counts[i]);
    printf("\n");
    # endif

    /* Print the histogram */
    Print_histo(bin_maxes, bin_counts, bin_count, min_meas);

```

```

    free(data);
    free(bin_maxes);
    free(bin_counts);
    return 0;

} /* main */

void *Pth_count(void* rank){
    /* which data to access */
    int i, bin;
    long my_rank = (long) rank;
    int first_i = data_count/thread_count* (int) my_rank;
    int last_i = data_count/thread_count* ((int) my_rank + 1)-1;

    /* Count number of values in each bin */
    for (i = first_i; i < last_i; i++) {
        bin = Which_bin(data[i], bin_maxes, bin_count, min_meas);
        pthread_mutex_lock(&mutex);
        bin_counts[bin]++;
        pthread_mutex_unlock(&mutex);
    }

    return NULL;
}

/*-----
* Function:  Usage
* Purpose:   Print a message showing how to run program and quit
* In arg:    prog_name:  the name of the program from the command line
*/
void Usage(char prog_name[] /* in */) {
    fprintf(stderr, "usage: %s ", prog_name);
    fprintf(stderr, "<bin_count> <min_meas> <max_meas> <data_count> <thread_count>\n");
    exit(0);
} /* Usage */

/*-----
* Function:  Get_args
* Purpose:   Get the command line arguments
* In arg:    argv:  strings from command line
* Out args:  bin_count_p:  number of bins
*            min_meas_p:   minimum measurement

```

```

*          max_meas_p:    maximum measurement
*          data_count_p:  number of measurements
*/
void Get_args(
    char*    argv[]        /* in */
    int*     bin_count_p   /* out */
    float*   min_meas_p    /* out */
    float*   max_meas_p    /* out */
    int*     data_count_p  /* out */
    int*     thread_count  /* out */) {

    *bin_count_p = strtol(argv[1], NULL, 10);
    *min_meas_p = strtol(argv[2], NULL);
    *max_meas_p = strtol(argv[3], NULL);
    *data_count_p = strtol(argv[4], NULL, 10);
    *thread_count = strtol(argv[5], NULL, 10);

# ifdef DEBUG
    printf("bin_count = %d\n", *bin_count_p);
    printf("min_meas = %f, max_meas = %f\n", *min_meas_p, *max_meas_p);
    printf("data_count = %d\n", *data_count_p);
    printf("thread_count = %d\n", *thread_count);
# endif
} /* Get_args */

/*-----
* Function:  Gen_data
* Purpose:   Generate random floats in the range min_meas <= x < max_meas
* In args:   min_meas:    the minimum possible value for the data
*            max_meas:    the maximum possible value for the data
*            data_count:  the number of measurements
* Out arg:   data:        the actual measurements
*/
void Gen_data(
    float    min_meas      /* in */
    float    max_meas      /* in */
    float    data[]        /* out */
    int      data_count    /* in */) {
    int i;

    srand(0);
    for (i = 0; i < data_count; i++)
        data[i] = min_meas + (max_meas - min_meas)*random()/((double) RAND_MAX);

# ifdef DEBUG
    printf("data = ");
    for (i = 0; i < data_count; i++)

```

```

        printf("%4.3f ", data[i]);
    printf("\n");
#   endif
}   /* Gen_data */

/*-----
* Function:   Gen_bins
* Purpose:    Compute max value for each bin, and store 0 as the
*             number of values in each bin
* In args:    min_meas:   the minimum possible measurement
*             max_meas:   the maximum possible measurement
*             bin_count:  the number of bins
* Out args:   bin_maxes:  the maximum possible value for each bin
*             bin_counts: the number of data values in each bin
*/
void Gen_bins(
    float min_meas    /* in */,
    float max_meas    /* in */,
    float bin_maxes[] /* out */,
    int   bin_counts[] /* out */,
    int   bin_count    /* in */) {
    float bin_width;
    int   i;

    bin_width = (max_meas - min_meas)/bin_count;

    for (i = 0; i < bin_count; i++) {
        bin_maxes[i] = min_meas + (i+1)*bin_width;
        bin_counts[i] = 0;
    }

#   ifdef DEBUG
    printf("bin_maxes = ");
    for (i = 0; i < bin_count; i++)
        printf("%4.3f ", bin_maxes[i]);
    printf("\n");
#   endif
}   /* Gen_bins */

/*-----
* Function:   Which_bin
* Purpose:    Use binary search to determine which bin a measurement
*             belongs to
* In args:    data:      the current measurement
*             bin_maxes: list of max bin values
*             bin_count: number of bins

```



```

*          min_meas:   the minimum possible measurement
* Return:   the number of the bin to which data belongs
* Notes:
* 1. The bin to which data belongs satisfies
*
*          bin_maxes[i-1] <= data < bin_maxes[i]
*
*       where, bin_maxes[-1] = min_meas
* 2. If the search fails, the function prints a message and exits
*/
int Which_bin(
    float  data          /* in */,
    float  bin_maxes[]   /* in */,
    int    bin_count     /* in */,
    float  min_meas      /* in */) {
    int bottom = 0, top = bin_count-1;
    int mid;
    float bin_max, bin_min;

    while (bottom <= top) {
        mid = (bottom + top)/2;
        bin_max = bin_maxes[mid];
        bin_min = (mid == 0) ? min_meas: bin_maxes[mid-1];
        if (data >= bin_max)
            bottom = mid+1;
        else if (data < bin_min)
            top = mid-1;
        else
            return mid;
    }

    /* Whoops! */
    fprintf(stderr, "Data = %f doesn't belong to a bin!\n", data);
    fprintf(stderr, "Quitting\n");
    exit(-1);
} /* Which_bin */

/*-----
* Function:  Print_histo
* Purpose:   Print a histogram. The number of elements in each
*           bin is shown by an array of X's.
* In args:   bin_maxes:   the max value for each bin
*           bin_counts:   the number of elements in each bin
*           bin_count:    the number of bins
*           min_meas:     the minimum possible measurment
*/
void Print_histo(

```

```

        float  bin_maxes[]  /* in */,
        int    bin_counts[] /* in */,
        int    bin_count   /* in */,
        float  min_meas    /* in */) {
    int i, j;
    float bin_max, bin_min;

    for (i = 0; i < bin_count; i++) {
        bin_max = bin_maxes[i];
        bin_min = (i == 0) ? min_meas: bin_maxes[i-1];
        printf("%.3f-%.3f:\t", bin_min, bin_max);
        for (j = 0; j < bin_counts[i]; j++)
            printf("X");
        printf("\n");
    }
} /* Print_histo */

```

C C Source Code for Exercise A4.3

```

/* File:      trap.c
 * Purpose:   Calculate definite integral using trapezoidal
 *           rule.
 *
 * Input:     a, b, n, thread_count
 * Output:    Estimate of integral from a to b of f(x)
 *           using n trapezoids.
 *
 * Compile:   gcc -g -Wall -o trap trap.c
 * Usage:     ./trap
 *
 * Note:      The function f(x) is hardwired. thread_count should
 *           the current version be integer divisible by n-1
 *
 * IPP:       Section 3.2.1 (pp. 94 and ff.) and 5.2 (p. 216)
 */

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

/* global variables */
int thread_count;
double integral; /* Store result in integral */
double a, b;     /* Left and right endpoints */
int n;           /* Number of trapezoids */
double h;        /* Height of trapezoids */

```

```

pthread_mutex_t mutex;

/* serial functions */
double f(double x);    /* Function we're integrating */
void Trap(double a, double b, int n, double h);

/* parallel function */
void *Pth_interpol(void* rank);

int main(void) {

    printf("Enter a, b, n and thread_count\n");
    scanf("%lf", &a);
    scanf("%lf", &b);
    scanf("%d", &n);
    scanf("%d", &thread_count);

    h = (b-a)/n;
    Trap(a, b, n, h);

    printf("With n = %d trapezoids, our estimate\n", n);
    printf("of the integral from %f to %f = %.15f\n",
        a, b, integral);

    return 0;
} /* main */

/*-----
 * Function:    Trap
 * Purpose:     Estimate integral from a to b of f using trap rule and
 *              n trapezoids
 * Input args:  a, b, n, h
 * Return val:  Estimate of the integral
 */
void Trap(double a, double b, int n, double h) {
    long thread;
    pthread_t* thread_handles;

    integral = (f(a) + f(b))/2.0;

    /* here comes the parallel part */

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

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

```

```

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

    integral = integral*h;

} /* Trap */

void *Pth_interpol(void* rank){
    /* not the first and not the last */
    long my_rank = (long) rank;
    int first_k = (n-1) / thread_count * my_rank + 1 ;
    int last_k= (n-1) / thread_count * (my_rank+1);
    double local_integral = 0;
    int k;

    //printf("first %d last %d\n", first_k, last_k);

    for (k = first_k; k <= last_k; k++) {
        local_integral += f(a+k*h);
    }

    /* updating global sum */
    pthread_mutex_lock(&mutex);
    integral += local_integral;
    pthread_mutex_unlock(&mutex);

    return NULL;
}

/*-----
* Function:    f
* Purpose:     Compute value of function to be integrated
* Input args:  x
*/
double f(double x) {
    double return_val;

    return_val = x*x;
    return return_val;
} /* f */

```