Programming Excercises - Chapter 5

José Rodrigo Vilca Vargas

Ciencia de la computación
Universidad Católica San Pablo

May 4, 2018

1 Use OpenMP to implement the parallel histogram program discussed in Chapter 2

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <ctime>

/*--------------------------------------------------------------------*/


int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    int n;
    scanf("%d",&n);
    double A[n];
    for (int i=0;i<n;i++){
        scanf("%lf",A+i);
    }
    int H[5];
    #pragma omp parallel for num_threads(thread_count) schedule(runtime)\
    shared(H)
    for (int i=0;i<5;i++){
        H[i]=0;
    }
    #pragma omp parallel for num_threads(thread_count) schedule(runtime)\
    shared(H,A,n)
    for (int i=0;i<n;i++){
        #pragma omp critical
        H[int(A[i])]++;
    }
    for (int i=0;i<5;i++){
        printf("H[%d] = %d\n", i, H[i]);
    }
    return 0;
}  /* main */
```

2 **Write an OpenMP program that uses a Monte Carlo method to estimate π. Read in the total number of tosses before forking any threads. Use a reduction clause to find the total number of darts hitting inside the circle. Print the result after joining all the threads. You may want to use long long ints for the number of hits in the circle and the number of tosses, since both may have to be very large to get a reasonable estimate of π**

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

int main(int argc, char* argv[])
{
    int niter = 100000;            //number of iterations per FOR loop
    double x,y;                 //x,y value for the random coordinate
    int i;                      //loop counter
            int count=0;         //Count holds all the number of how many good coordinates
    double z;                 //Used to check if x^2+y^2<=1
    double pi;                 //holds approx value of pi
    int numthreads = 16;

    #pragma omp parallel firstprivate(x, y, z, i) reduction(+:count) num_threads(numthreads)
    {
        srand48((int)time(NULL) ^ omp_get_thread_num());    //Give random() a seed value
        for (i=0; i<niter; ++i)                  //main loop
        {
            x = (double)drand48();               //gets a random x coordinate
            y = (double)drand48();               //gets a random y coordinate
            z = ((x*x)+(y*y));               //Checks to see if number is inside unit circle
            if (z<=1)
            {
                ++count;               //if it is, consider it a valid random point
            }
        }
    }
    pi = ((double)count/(double)(niter*numthreads))*4.0;
    printf("Pi: %f\n", pi);
    return 0;
}
```

3 Count Sort

3.1 If we try to parallelize the for i loop (the outer loop), which variables should be private and which should be shared?

a,n y temp deberán ser variables globales
i y j deberán ser privadas

3.2 If we parallelize the for i loop using the scoping you specified in the previous part, are there any loop-carried dependences? Explain your answer

El proceso de cada iteración es diferente de los demás

3.3 Can we parallelize the call to memcpy? Can we modify the code so that this part of the function will be parallelizable?

'mempcy' podría ser paralelizable, teniendo el tamaño como variable compartida además de los punteros temp y a

3.4 Write a C program that includes a parallel implementation of Count sort

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <omp.h>

#ifndef LOOP
#define LOOP 0
#endif

#ifndef MIC
#define MIC 0
#endif

void omp_count_sort(int a[], int n, int P);

struct timespec diff(struct timespec start, struct timespec end)
{
    struct timespec temp;
    if ((end.tv_nsec-start.tv_nsec)<0) {
    temp.tv_sec = end.tv_sec-start.tv_sec-1;
    temp.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
    } else {
    temp.tv_sec = end.tv_sec-start.tv_sec;
    temp.tv_nsec = end.tv_nsec-start.tv_nsec;
    }
    return temp;
}

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

    struct timespec ts;
    clock_gettime(CLOCK_MONOTONIC, &ts);

    int N = atoi(argv[1]);
    int * A = (int *) malloc (N * sizeof(int));
    int P = atoi(argv[2]);

    srand(time(NULL));
    int i;
    for (i=0; i<N; ++i) {
        A[i] = (int) rand();
    }
    omp_count_sort(A, N, P);
    for (i=1; i<N; ++i) {
        if (A[i] < A[i-1]) {
        printf("ERROR: Sort failed\n");
            //cout << "ERROR: Sort failed" << endl;
            //cout << A[i-1] << " " << A[i] << endl;
            exit(1);
        }
    }

    struct timespec endts;
    clock_gettime(CLOCK_MONOTONIC, &endts);
    printf("%d\t%d\t%f\n", N, P, diff(ts,endts).tv_sec + (float) diff(ts,endts).tv_nsec / 1000000000.0);

}
```

```
60 void omp_count_sort(int a[], int n, int P) {
61
62     int * temp;
63     #pragma offload target(mic) if (MIC==1) inout(a:length(n)) nocopy(temp:length(n))
64     {
65     int i, j;
66     int count = 0;
67     temp = (int *) malloc(n * sizeof(int));
68     omp_set_num_threads(P);
69
70
71     #pragma omp parallel for if (LOOP==0) private(count)
72     for (i=0; i < n; i++) {
73         count = 0;
74         #pragma omp parallel for if (LOOP==1) shared(count)
75         for (j=0; j < n; j++)
76             if (a[j] < a[i])
77                 __sync_add_and_fetch(&count, 1);
78             else if (a[j] == a[i] && j < i)
79                 __sync_add_and_fetch(&count, 1);
80         temp[count] = a[i];
81     }
82     int k = 0;
83     #pragma omp parallel for
84     for (k=0; k < n; ++k)
85         memcpy(&a[k], &temp[k], sizeof(int));
86     free(temp);
87     }
88 } /* Count sort */
```

4 Gaussian elimination

4.1 Determine whether the outer loop of the row-oriented algorithm can be parallelized.

Es necesario mantener el orden del calculo de las filas, por lo tanto no es posible paralelizar el bucle externo.

4.2 Determine whether the inner loop of the row-oriented algorithm can be parallelized.

Si es paralelizable, pero se debe tener cuidado en el cálculo de la variable de la fila, al restar el resultado de la ecuación con los resultados calculados.

4.3 Determine whether the (second) outer loop of the column-oriented algorithm can be parallelized.

El bucle externo no es paralelizable, debido a que para realizar el cálculo de una columna es necesario que la anterior columna haya sido calculada por completo.

4.4 Determine whether the inner loop of the column-oriented algorithm can be parallelized

Si es paralelizable, y la principal ventaja es que no existe sección crítica.

	Row Oriented	Col Oriented
Default	0.118399	0.548665
Estatic	0.136892	0.549943
Dynamic	0.125099	0.540122
Guied	0.137736	0.529266

4.5 Write one OpenMP program for each of the loops that you determined could be parallelized.

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

int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    int n;
    scanf("%d",&n);
    double x[n];double A[n][n];double b[n];
    for (int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            scanf("%d", &A[i][j]);
        }
        scanf("%d", &b[i]);
    }
    for (int row=n-1;row>=0;row--){
        x[row]=b[row];
        int col,eq;
        #  pragma omp parallel for num_threads(thread_count) schedule(runtime) \
        shared(n,x,A,b,row)
        for (int col = row+1;col<n;col++){
            #  pragma omp critical
            x[row]-=A[row][col]*x[col];
        }
        x[row]/=A[row][row];
    }
    for (int row=0;row<n;row++){
        x[row]=b[row];
    }
    for (int col=n-1;col>=0;col--){
        x[col] /= A[col][col];
        #  pragma omp parallel for num_threads(thread_count) schedule(runtime)\
        shared(n,x,A,b,col)
        for (int row=0;row<col;row++){
            x[row] -= A[row][col] * x[col];
        }
    }
    return 0;
}  /* main */
```

4.6 Modify your parallel loop with a schedule (runtime) clause and test the program with various schedules. If your upper triangular system has 10,000 variables, which schedule gives the best performance?

El schedule con mejor performance es 'default' y 'dynamic' para "Row Oriented" debido a la sección crítica el balanceo de los índices afecta, y 'guied' para "Col Oriented" debió a que tiene mejor equilibrio en la asignación de los índices.

5 Use OpenMP to implement a program that does Gaussian elimination. (See the preceding problem.) You can assume that the input system doesn't need any row-swapping.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#include <ctime>

template<int t>
void gauss_elimination(double (&A)[t][t], double *b,int thread_count=1){
    int n=t;
    for (int i=0;i<n;i++){
        for (int j=i+1;j<n;j++){
            double temp=A[j][i];
            #pragma omp parallel for num_threads(thread_count) schedule(runtime)\
            shared(n,A,b,i,j,temp)
            for (int k=i;k<n;k++){
                A[j][k]=(A[j][k]/temp)*A[i][i]-A[i][k];
            }
            b[j]=(b[j]/temp)*A[i][i]-b[i];
        }
    }
    double x[10000];
    for (int col=n-1;col>=0;col--){
        x[col] /= A[col][col];
        #  pragma omp parallel for num_threads(thread_count) schedule(runtime)\
        shared(n,x,A,b,col)
        for (int row=0;row<col;row++){
            x[row] -= A[row][col] * x[col];
        }
    }
    for (int row=0;row<n;row++){
        printf("row[%d]=%7.f\n",row,x[row]);
    }
}

int main(int argc, char* argv[]) {
    int thread_count = strtol(argv[1], NULL, 10);
    int n;
    scanf("%d",&n);
    double A[10000][10000];
    double b[n];
    for (int i=0;i<n;i++){
        for (int j=0;j<n;j++){
            scanf("%lf", &A[i][j]);
        }
        scanf("%lf", &b[i]);
    }
    gauss_elimination(A,b,thread_count);
    return 0;
}  /* main */
```

6 Use OpenMP to implement a producer-consumer program in which some of the threads are producers and others are consumers.

```
int main(int  argc, char* argv[]) {
    int prod_count, cons_count;
    FILE* files[MAX_FILES];
    int file_count;

    if (argc != 3) Usage(argv[0]);
    prod_count = strtol(argv[1], NULL, 10);
    cons_count = strtol(argv[2], NULL, 10);

    /* Read in list of filenames and open files */
    Get_files(files, &file_count);

    # ifdef DEBUG
    printf("prod_count = %d, cons_count = %d, file_count = %d\n", prod_count, cons_count, file_count);
    # endif

    /* Producer-consumer */
    Prod_cons(prod_count, cons_count, files, file_count);
    return 0;
} /* main */
/* Function: Prod_cons * objetivo: Dividir tareas entre threads */
void Prod_cons(int  prod_count, int cons_count, FILE* files[ ], int file_count) {
    int thread_count = prod_count + cons_count;
    struct list_node_s* queue_head = NULL;
    struct list_node_s* queue_tail = NULL;
    int prod_done_count = 0;

    # pragma omp parallel
    num_threads(thread_count) default(none) \
    shared(file_count, queue_head, queue_tail, files, prod_count, \
    cons_count, prod_done_count){
        int my_rank = omp_get_thread_num(), f;
        if  (my_rank < prod_count) {  /* Producer code */
             /* A cyclic partition of the files among the producers */
            for (f = my_rank; f < file_count; f += prod_count) {
                Read_file(files[f], &queue_head, &queue_tail, my_rank);
            }
            # pragma omp atomic
             prod_done_count++;
        } else { /* Consumer code */
            struct list_node_s* tmp_node;
            while (prod_done_count < prod_count) {
                tmp_node = Dequeue(&queue_head, &queue_tail, my_rank);
                if (tmp_node != NULL) {
                    Tokenize(tmp_node->data, my_rank); free(tmp_node);
                }
            }
            while (queue_head != NULL) {
            tmp_node = Dequeue(&queue_head, &queue_tail, my_rank);

             if (tmp_node != NULL) {
            Tokenize(tmp_node->data, my_rank);
            free(tmp_node); } } }
    } /* pragma omp parallel */
}
```

```
59 /* Enqueue */
60 void Enqueue(
61 char* line, struct list_node_s** queue_head,
62 struct list_node_s** queue_tail) {
63      struct list_node_s* tmp_node = NULL;
64     tmp_node = malloc(sizeof(struct list_node_s));
65     tmp_node->data = line;
66     tmp_node->next = NULL;
67     # pragma omp critical
68     if (*queue_tail == NULL) { // list is empty
69     *queue_head = tmp_node;
70     *queue_tail = tmp_node;
71     } else {
72     (*queue_tail)->next = tmp_node;
73     *queue_tail = tmp_node;
74     }
75 }
76
77  /* Dequeue */
78 struct list_node_s* Dequeue(
79 struct list_node_s** queue_head,
80 struct list_node_s** queue_tail,
81 int my_rank) {
82     struct list_node_s* tmp_node = NULL;
83
84     if (*queue_head == NULL) // empty
85     return NULL;
86      # pragma omp critical
87     {
88         if (*queue_head == *queue_tail) // last node
89         *queue_tail = (*queue_tail)->next;
90         tmp_node = *queue_head;
91         *queue_head = (*queue_head)->next;
92     }
93     return tmp_node;
94 }
```