Introduction to Parallel Programing Techniques

Deferred Assessment

Professor: Stylianos Sygletos

Nestor Edgar Sandoval Alaguna
200243856

Assignment 5

Exercise – 1 [3 points]
Write a Pthreads program that implements the trapezoidal rule. Use a shared variable for the sum of all the threads' computations. Create implementations based on busy-waiting, mutexes, and semaphores to enforce mutual exclusion in the critical section and compare them in terms of performance. What advantages and disadvantages do you see with each approach? Identify suitable performance metrics and running scenarios to support your answer.

Answer:

The busy-waiting is used mostly in order to avoid nondeterminism among the thread while the access critical sections. It makes other threads to wait for their turn to share according to how it is set, it can be set from the lowest to highest rank.

Mutexes helps to avoid wasting CPU cycles helping to avoid nondeterminism. This method blocks other threads from running until the process finished its computations. This differs from busy-waiting helping to not waste time while waiting for access, while it lets the order of the threads access to be set by the system.

For Semaphores, a signaling mechanism gives the signal to each process, the value of this signal is 1 or 0, it becomes 0 when the process accesses the critical section and changes to one once the process has finished its task.

Code:

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int thread_count;
char t;
long n;
double a, b, h;
double integral_estimation = 0.0;
pthread_mutex_t mutex;
sem_t semaphore;
int flag = 0;
char t;
double evaluated_function(double x) {
double return_val;
return_val = x * x;
return return_val;
}
void usage(char *prog_name) {
fprintf(stderr, "usage: %s <number of threads>.\n", prog_name);
exit(0);
}
void *thread_trap(void *rank) {
long current_rank = (long)rank;
long local_n = n / thread_count;
double local_a = current_rank * local_n * h;
double local_integral_estimation;
```

```
int i;
for (i = 1; i <= local_n; i++) {
if (local_a + i * h == b)
break;
local_integral_estimation += evaluated_function(local_a + i * h);
}
local_integral_estimation = local_integral_estimation * h;
switch (t) {
case 'm':
pthread_mutex_lock(&mutex);
integral_estimation += local_integral_estimation;
pthread_mutex_unlock(&mutex);
break;
case 's':
sem_wait(&semaphore);
integral_estimation += local_integral_estimation;
sem_post(&semaphore);
break;
case 'b':
while (flag != current_rank)
;
integral_estimation += local_integral_estimation;
flag++;
break;
}
return NULL;
}
int main(int argc, char *argv[]) {
long thread;
pthread_t *thread_handler;
struct timespec start, finish;
double elapsed_time;
thread_count = strtol(argv[1], NULL, 10);
printf("Enter A:\n");
scanf("%lf", &a);
printf("Enter B:\n");
scanf("%lf", &b);
printf("Enter N:\n");
scanf("%ld", &n);
h = (b - a) / (double)n;
thread_handler = malloc(thread_count * sizeof(pthread_t));
t = 'm';
pthread_mutex_init(&mutex, NULL);
clock_gettime(CLOCK_MONOTONIC, &start);
for (thread = 0; thread < thread_count; thread++)
pthread_create(&thread_handler[thread], NULL, thread_trap, (void *)thread);
for (thread = 0; thread < thread_count; thread++)
pthread_join(thread_handler[thread], NULL);
integral_estimation +=
h * (evaluated_function(a) + evaluated_function(b)) / 2.0;
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed_time = (finish.tv_sec - start.tv_sec);
elapsed_time += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
printf("A: %lf, B: %lf, N: %ld\n", a, b, n);
printf("Mutex: \n");
printf("\tIntegration: x^2 = %lf", integral_estimation);
printf("\tElapsed time: %.10f seconds.\n", elapsed_time);
t = 's';
integral_estimation = 0.0;
sem_init(&semaphore, 0, 1);
clock_gettime(CLOCK_MONOTONIC, &start);
for (thread = 0; thread < thread_count; thread++)
pthread_create(&thread_handler[thread], NULL, thread_trap, (void *)thread);
```

```c
for (thread = 0; thread < thread_count; thread++)
pthread_join(thread_handler[thread], NULL);
integral_estimation +=
h * (evaluated_function(a) + evaluated_function(b)) / 2.0;
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed_time = (finish.tv_sec - start.tv_sec);
elapsed_time += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
printf("Semaphores: \n");
printf("\tIntegration: x^2 = %lf", integral_estimation);
printf("\tElapsed time: %.10f seconds.\n", elapsed_time);
t = 'b';
integral_estimation = 0.0;
clock_gettime(CLOCK_MONOTONIC, &start);
for (thread = 0; thread < thread_count; thread++)
pthread_create(&thread_handler[thread], NULL, thread_trap, (void *)thread);
for (thread = 0; thread < thread_count; thread++)
pthread_join(thread_handler[thread], NULL);
integral_estimation +=
h * (evaluated_function(a) + evaluated_function(b)) / 2.0;
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed_time = (finish.tv_sec - start.tv_sec);
elapsed_time += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
printf("Busy Wating: \n");
printf("\tIntegration: x^2 = %lf", integral_estimation);
printf("\tElapsed time: %.10f seconds. \n", elapsed_time);
free(thread_handler);
pthread_mutex_destroy(&mutex);
sem_destroy(&semaphore);
return 0;
}
```

```
Enter A:
3
Enter B:
4
Enter N:
3
A: 3.000000, B: 4.000000, N: 3
Mutex:
        Integration: x^2 = 4.166667     Elapsed time: 0.0002183810 seconds.
Semaphores:
        Integration: x^2 = 4.166667     Elapsed time: 0.0001278820 seconds.
Busy Wating:
        Integration: x^2 = 4.166667     Elapsed time: 0.0141759750 seconds.
```

This program was running with 4 threads

Exercise – 2 [3 points]
Recall that in C a function that takes a two-dimensional array argument must specify the number of columns
in the argument list. This is quite common for C programmers to only use one-dimensional arrays, and to write explicit code for converting pairs of subscripts into a single dimension. Modify the Pthreads matrixvector
multiplication, which we examined during the lecture sessions, so that it uses a one-dimensional array for the matrix and calls a matrix-vector multiplication function. How does this change affect the run-time? Run your program and take measurable results to support your answer.

Answer:

The part that was modified from the code

```
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;
}
```

Exercise – 3 [4 points][Pthread Implementation]
a) Modify the matrix-vector multiplication program so that it pads the vector y when there's a possibility of
false sharing. The padding should be done so that if the threads execute in lock-step, there's no possibility that a single cache line containing an element of y will be shared by two or more threads. Suppose, for example, that a cache line stores eight doubles and we run the program with four threads. If we allocate storage for at least 48 doubles in y , then, on each pass through the for i loop, there's no possibility that two threads will simultaneously access the same cache line. [2 points]
b) Modify the matrix-vector multiplication so that each thread uses private storage for its part of y during the for i loop. When a thread is done computing its part of y, it should copy its private storage into the shared variable. [1 point]
c) How does the performance of these two alternatives compare to the original program? How do they compare to each other? [1 point]

Answer:

Code:

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


int thread_count;
int m, n;
double *M;
double *x;
double *y;

void generate_matrix(double M[], int m, int n) {
int i, j;
for (i = 0; i < m; i++)
```

```
for (j = 0; j < n; j++)
M[i * n + j] = random() / ((double)RAND_MAX);
}

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

void *path_matrix_vector(void *rank) {
long current_rank = (long)rank;
int i, j;
int local_m = m / thread_count;
int first_row = current_rank * local_m;
int last_row = (current_rank + 1) * local_m - 1;
for (i = first_row; i <= last_row; i++) {
y[i + (current_rank * 8)] = 0.0;
for (j = 0; j < n; j++)
y[i + (current_rank * 8)] += M[i * n + j] * x[j];
}
return NULL;
}

void show_matrix(char *title, double M[], int m, int n) {
int i, j;
printf("%s\n", title);
for (i = 0; i < m; i++) {
for (j = 0; j < n; j++)
printf("%4.1f\t", M[i * n + j]);
printf("\n");
}
}

void show_vector(char *title, double v[], double m) {
int i;
printf("%s\n", title);
for (i = 0; i < m; i++)
printf("%4.1f\t", v[i]);
printf("\n");
}

int main(int argc, char *argv[]) {
long thread;
pthread_t *thread_handles;
double elapsed_time = 0;
thread_count = atoi(argv[1]);
thread_handles = malloc(thread_count * sizeof(pthread_t));
printf("Enter m:\n");
scanf("%d", &m);
printf("Enter n:\n");
scanf("%d", &n);
M = malloc(m * n * sizeof(double));
x = malloc(n * sizeof(double));
y = malloc((m + 8 * thread_count) * sizeof(double));
generate_matrix(M, m, n);
show_matrix("Matirx:", M, m, n);
generate_vector(x, n);
show_vector("Vector: ", x, n);
struct timespec start, finish;
clock_gettime(CLOCK_MONOTONIC, &start);
for (thread = 0; thread < thread_count; thread++)
pthread_create(&thread_handles[thread], NULL, path_matrix_vector,
```

```c
(void *)thread);
for (thread = 0; thread < thread_count; thread++)
pthread_join(thread_handles[thread], NULL);
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed_time = (finish.tv_sec - start.tv_sec);
elapsed_time += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
printf("Elapsed time: %.5f seconds.\n", elapsed_time);
int i, j = 0;
printf("Result: \n");
for (i = 0; i < m + 8 * thread_count; i++) {
printf("%4.1f ", y[i]);
j++;
if (j == (m / thread_count)) {
j = 0;
i += 8;
}
}
printf("\n");
free(M);
free(x);
free(y);
return 0;
}
```

```
Enter m:
6
Enter n:
8
Matirx:
 0.8     0.4     0.8     0.8     0.9     0.2     0.3     0.8
 0.3     0.6     0.5     0.6     0.4     0.5     1.0     0.9
 0.6     0.7     0.1     0.6     0.0     0.2     0.1     0.8
 0.2     0.4     0.1     0.1     1.0     0.2     0.5     0.8
 0.6     0.3     0.6     0.5     0.5     1.0     0.3     0.8
 0.5     0.8     0.4     0.9     0.3     0.4     0.8     0.9
Vector:
 0.1     0.9     0.5     0.1     0.2     0.7     0.9     0.3
Elapsed time: 0.00018 seconds.
Result:
 1.8  2.4  1.4  1.6  0.0
```

This program was running with 4 threads.

```c
#include <math.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

int thread_count;
int m, n;
double *M;
double *x;
```

```
double *y;

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

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

void show_matrix(char *title, double M[], int m, int n) {
int i, j;
printf("%s\n", title);
for (i = 0; i < m; i++) {
for (j = 0; j < n; j++)
printf("%4.1f\t", M[i * n + j]);
printf("\n");
}
}

void show_vector(char *title, double v[], double m) {
int i;
printf("%s\n", title);
for (i = 0; i < m; i++)
printf("%4.1f\t", v[i]);
printf("\n");
}

void *path_matrix_vector(void *rank) {
long current_rank = (long)rank;
int i, j;
int local_m = m / thread_count;
int first_row = current_rank * local_m;
double *local_y = malloc(local_m * sizeof(double));
for (i = 0; i < local_m; i++) {
local_y[i] = 0.0;
for (j = 0; j < n; j++)
local_y[i] += M[(i + first_row) * n + j] * x[j];
}
memcpy(y + first_row, local_y, local_m * sizeof(double));
free(local_y);
return NULL;
}

int main(int argc, char *argv[]) {
long thread;
pthread_t *thread_handles;
double elapsed_time = 0;
thread_count = atoi(argv[1]);
thread_handles = malloc(thread_count * sizeof(pthread_t));
printf("Enter m:\n");
scanf("%d", &m);
printf("Enter n:\n");
scanf("%d", &n);
M = malloc(m * n * sizeof(double));
x = malloc(n * sizeof(double));
y = malloc(m * sizeof(double));
generate_matrix(M, m, n);
```

```c
show_matrix("Matirx:", M, m, n);
generate_vector(x, n);
show_vector("Vector: ", x, n);
struct timespec start, finish;
clock_gettime(CLOCK_MONOTONIC, &start);
for (thread = 0; thread < thread_count; thread++)
pthread_create(&thread_handles[thread], NULL, path_matrix_vector,
(void *)thread);
for (thread = 0; thread < thread_count; thread++)
pthread_join(thread_handles[thread], NULL);
clock_gettime(CLOCK_MONOTONIC, &finish);
elapsed_time = (finish.tv_sec - start.tv_sec);
elapsed_time += (finish.tv_nsec - start.tv_nsec) / 1000000000.0;
printf("Elapsed time: %.5f seconds.\n", elapsed_time);
show_vector("Result: ", y, m);
free(M);
free(x);
free(y);
return 0;
}
```

```
Enter m:
6
Enter n:
8
Matirx:
 0.8     0.4     0.8     0.8     0.9     0.2     0.3     0.8
 0.3     0.6     0.5     0.6     0.4     0.5     1.0     0.9
 0.6     0.7     0.1     0.6     0.0     0.2     0.1     0.8
 0.2     0.4     0.1     0.1     1.0     0.2     0.5     0.8
 0.6     0.3     0.6     0.5     0.5     1.0     0.3     0.8
 0.5     0.8     0.4     0.9     0.3     0.4     0.8     0.9
Vector:
 0.1     0.9     0.5     0.1     0.2     0.7     0.9     0.3
Elapsed time: 0.00020 seconds.
Result:
 1.8     2.4     1.4     1.6     0.0     0.0
```

This program was running with 4 threads.