Introduction to Parallel Programing Techniques

Deferred Assessment

Professor: Stylianos Sygletos

Nestor Edgar Sandoval Alaguna
200243856

Assignment 7

Exercise – 1 [2 points] [OpenMP 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:

In order to avoid false sharing of cache, in this case padding is used. In the case of padding each line of cache stores more line of doubles, each thread will e executed in lock-step. For the second case uses private storage, it allocates private y to each thread and each of them copy the value to the global y.

It is possible to observe that the second method takes longer and it uses extra memory space due to the allocation tasks. For the first method, the padded version runs faster and performs better than the original method.

Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
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 omp_matrix_vector(double M[], double x[], double y[], int m, int n, int
thread_count) {
int i, j;
double start, finish, elapsed;
start = omp_get_wtime();
//int current_rank;
#pragma omp parallel num_threads(thread_count) \
default(none) private(i, j) shared(M, x, y, m, n)
{
int current_rank = omp_get_thread_num();
```

```
#pragma omp for
for (i = 0; i < m; 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];
}
}
finish = omp_get_wtime();
elapsed = finish - start;
printf("Elapsed time: %e seconds.\n", elapsed);
}
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[]) {
int thread_count;
int m, n;
double *M;
double *x;
double *y;
thread_count = strtol(argv[1], NULL, 10);
m = strtol(argv[2], NULL, 10);
n = strtol(argv[3], NULL, 10);
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("Matrix: ", M, m, n);
generate_vector(x, n);
show_vector("Vector: ", x, n);
omp_matrix_vector(M, x, y, m, n, thread_count);
{
int i, j = 0;
printf("Result:\n");
for (i = 0; i < m + 8 * thread_count; i++) {
printf("%f\t", y[i]);
j++;
if (j == (m / thread_count)) {
j = 0;
i += 8;
}
}
printf("\n");
}
free(M);
free(x);
free(y);
return 0;
```

```
Matrix:
 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     0.1     0.9
 0.5     0.1     0.2     0.7     0.9     0.3     0.1     0.0     0.5     0.1
 0.2     1.0     0.9     0.9     0.3     0.5     0.4     0.8     0.5     0.7
 0.5     0.0     0.4     0.9     0.9     0.7     0.3     0.7     0.6     0.4
 0.7     0.2     0.4     0.9     0.8     0.3     0.2     0.9     0.4     0.7
 1.0     0.6     0.7     0.9     0.4     0.9     0.4     0.8     0.7     0.9
Vector:
 0.5     0.2     1.0     0.9     0.1     0.9     0.6     0.4     0.6     0.3
Elapsed time: 2.319330e-04 seconds.
Result:
3.152529        3.108641        0.000000        2.973866        0.000000        0.000000        0.000
000     0.000000        0.000000        0.000000
```

The program was run with 4 threads and a matrix of 10 x 10

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
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 omp_matrix_vector(double M[], double x[], double y[], int m, int n,
int thread_count, int id, int chunk) {
int i, j;
double start, finish, elapsed;
start = omp_get_wtime();
omp_set_dynamic(0);
omp_set_num_threads(thread_count);
#pragma omp parallel private(i, j, id) shared(M, x, y, m, n)
{
id = omp_get_thread_num();
int st = id * chunk;
int ed = (id + 1) * chunk;
if (id == thread_count - 1) {
ed = m;
}
printf("Thread: %d.\n From row %d to < row %d.\n", id + 1, st, ed);
for (i = st; i < ed; i++) {
y[i] = 0.0;
for (j = 0; j < n; j++)
y[i] += M[i * n + j] * x[j];
}
}
finish = omp_get_wtime();
elapsed = finish - start;
printf("Elapsed: %e seconds.\n", elapsed);
}
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[]) {
int thread_count;
int chunk, m, n;
int id = 0;
double *M;
double *x;
double *y;
thread_count = strtol(argv[1], NULL, 10);
m = strtol(argv[2], NULL, 10);
n = strtol(argv[3], NULL, 10);
chunk = m / thread_count;
M = malloc(m * n * sizeof(double));
x = malloc(n * sizeof(double));
y = malloc(m * sizeof(double));
generate_matrix(M, m, n);
show_matrix("Matrix: ", M, m, n);
generate_vector(x, n);
show_vector("Vector:", x, n);
omp_matrix_vector(M, x, y, m, n, thread_count, id, chunk);
show_vector("Result: ", y, m);
free(M);
free(x);
free(y);
return 0;
}
```

```
Matrix:
 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     0.1     0.9
 0.5     0.1     0.2     0.7     0.9     0.3     0.1     0.0     0.5     0.1
 0.2     1.0     0.9     0.9     0.3     0.5     0.4     0.8     0.5     0.7
 0.5     0.0     0.4     0.9     0.9     0.7     0.3     0.7     0.6     0.4
 0.7     0.2     0.4     0.9     0.8     0.3     0.2     0.9     0.4     0.7
 1.0     0.6     0.7     0.9     0.4     0.9     0.4     0.8     0.7     0.9
Vector:
 0.5     0.2     1.0     0.9     0.1     0.9     0.6     0.4     0.6     0.3
Thread: 1.
 From row 0 to < row 2.
Thread: 3.
 From row 4 to < row 6.
Thread: 2.
 From row 2 to < row 4.
Thread: 4.
 From row 6 to < row 10.
Elapsed: 3.598970e-04 seconds.
Result:
 3.2     3.1     2.1     3.0     3.2     1.9     3.6     3.3     3.0     4.2
```

The program was run with 4 threads and a matrix of 10 x 10

Exercise – 2 [4 points]

a) After the algorithm has completed, we overwrite the original array with the temporary array using the string library function memcpy.
b) If we try to parallelize the for i loop (the outer loop), which variables should be private and which should be shared?
c) 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.
2020-2021 EE4107: Introduction to Parallel Programming Techniques (Deferred Assessment)
d) Can we parallelize the call to memcpy? Can we modify the code so that this part of the function will be parallelizable?
e) Write a C program that includes a parallel implementation of Count_sort.

Answer:

a) The array a[] should be accessible from all the threads, the array of size n should be shared to all. The temporary array temp[] must be shared so individual threads can insert elements inside of it. The iteration variables i, j, count must be private since individual elements are held by different threads.

b) Between iterations, there is no reliance. Only the elements of temp are written by each thread, and count is already private.

c) Yes, in order to make memcpy parallelizable the arrays should b copied locally. Each thread is able to identify and copy the local fraction of their temp[] to the main a[] array. Using a for loop to copy temp[] into a[] might be a better solution.

d)
Code:

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, char *argv[]) {
int thread_count, n, i, j;
int count;
thread_count = strtol(argv[1], NULL, 10);
n = strtol(argv[2], NULL, 10);
int *a = malloc(n * sizeof(int));
int *temp = malloc(n * sizeof(int));
printf("Input array:\n");
for (int i = 0; i < n; i++) {
scanf("%d", &a[i]);
}
#pragma omp parallel num_threads(thread_count) default(none) private(
i, j, count) shared(n, a, temp) {
#pragma omp for
for (i = 0; i < n; i++) {
count = 0;
for (j = 0; j < n; j++)
if (a[j] < a[i])
count++;
else if (a[j] == a[i] && j < i)
count++;
temp[count] = a[i];
}
```

```
#pragma omp for
for (i = 0; i < n; i++)
a[i] = temp[i];
}
}
```

The code was run with 4 threads and 8 numbers

```
Input array:
2 1 4 6 8 4 6 3
sorted array: 1 2 3 4 4 6 6 8
```

Exercise – 3 [4 points]

a) Determine whether the outer loop of the row-oriented algorithm can be parallelized.
b) Determine whether the inner loop of the row-oriented algorithm can be parallelized.
c) Determine whether the (second) outer loop of the column-oriented algorithm can be parallelized.
d) Determine whether the inner loop of the column-oriented algorithm can be parallelized.
e) Write one OpenMP program for each of the loops that you determined could be parallelized. You may find the single directive useful—when a block of code is being executed in parallel and a sub-block should be executed by only one thread, the sub-block can be modified by a #pragma omp single directive. The threads in the executing team will block at the end of the directive until all of the threads have completed it.

Answer:

a) The outer loop of the row-oriented algorithm cannot be parallelized due to the dependency across the iterations. x[row] depends on the value x [row+1]

b) In the case of the inner loop of the row-oriented, this can be parallelized due to non-data dependency.

c) The second outer loop of the column-oriented algorithm cannot be parallelized due to the data dependency, similar as the previous row-oriented one.

d)The inner loop of the column-oriented algorithm does not have any data dependency so it can be parallelized.

e) Code:

Row-oriented

```
double temp_val;
for (int row = n - 1; row >= 0; row--) {
       x[row] = b[row];
#pragma omp parallel for num_threads(thread_count) default(none) \
       private(col) shared (x, b, a, n, row)
              for (col = row + 1; col < n; col++) {
                     double value = a[row * n + col] * x[col];
#pragma omp critical
                     x[row] -= temp_val;
              }
       x[row] /= a[row * n + row];
}
```

Column-oriented

```
# pragma omp for
       for (row = 0; row < n; row++) {
       x[row] = b[row];
       for (col = n - 1; col >= 0; col--) {
#        pragma omp single
              x[col] /= a[col * n + col];
              private(row) shared(x, b, a, n, col)
#      pragma omp for schedule(runtime)
              for (row = 0; row < col; row++) {
                     x[row] += -a[row * n + col] * x[col];
              }
       }
}
```