MONASH University

SHARED MEMORY (OPENMP)

OBJECTIVES

- The purpose of this workshop is to introduce Parallel Computing on shared memory
- Understand the concept of OPENMP (OMP) thread

Note: Workshops are not assessed. Nevertheless, please attempt the questions to improve your unit comprehension in preparation for the lab assessments and assignments.

LEARNING OUTCOMES

- Explain the fundamental principles of parallel computing architectures and algorithms through practice-based learning (LO1)
- Design and develop parallel algorithms for various parallel computing architectures (LO3)

REMINDER: Before participating in the workshop, please check the pre-workshop lesson in Moodle. You should watch videos, and read supplementary notes before your scheduled workshop. This lesson is a necessary part of the preparation for the workshop this week, and should be completed before you come to the class.

MONASH University

WORKSHOP ACTIVITIES

Activity 1:

- Join the Jamboard and work in a small group.
- Discuss why the OpenMP provides a temporary view of the thread memory.

  All OpenMP threads have access to a place to store and to retrieve variables, called the memory. In addition, each thread is allowed to have its own temporary view of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called thread private memory.

  https://www.openmp.org/spec-html/5.0/openmpsu9.html

Activity 2:

With reference to the following code:

```
#include <stdio.h>
#include <omp.h>
#define N 20

int main()
{
    int a[N] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    int b[N] = {1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
    int c[N] = {0};
    int i;


    #pragma omp parallel for shared (a,b,c) private(i)
    for(i=0; i<N; i++) {
       if(i > 0){
             c[i] = c[i-1] + (a[i] * b[i]);
       }else{
             c[i] = (a[i] * b[i]);
       }
    }

       printf("Values of C:\n");
       for(i=0; i<N; i++) {
             printf("%d\t", c[i]);
       }
       printf("\n");

       return 0;
}
```

When compiling **without OpenMP**, the following output is observed:

```
mpi@mpi-DL01:~/FIT3143/Week_03$ gcc omp_0_Tut.c -o Out
mpi@mpi-DL01:~/FIT3143/Week_03$ ./Out
Values of C:
1       5       14      30      55      91      140     204     285     385     5
06      650     819     1015    1240    1496    1785    2109    2470    2870
```

However, when compiling **with OpenMP enabled**, the following output is observed:

```
mpi@mpi-DL01:~/FIT3143/Week_03$ gcc -Wall omp_0_Tut.c -fopenmp -o Out
mpi@mpi-DL01:~/FIT3143/Week_03$ ./Out
Values of C:
1       5       9       25      25      61      49      113     194     294     1
21      265     434     630     225     481     289     324     685     1085
mpi@mpi-DL01:~/FIT3143/Week_03$
```

a) Based on the screen shots above, it is apparent that the answers are different when compiling without and with OpenMP. The code which was compiled without OpenMP (i.e., serial code) provides a correct answer. However, this is not the case for the code which was compiled using OpenMP. Explain this discrepancy.

Hint: You should consider performing a dependency analysis using Bernstein's conditions. Click here for an example of Bernstein's conditions.

To verify that the process of calculating individual element in each loop iteration, Bernstein's Condition are used. Bernstein's conditions outline the conditions which are sufficient to determine whether the execution of two or more threads can be performed simultaneously. The three conditions which constitute the Bernstein's conditions are as follows:

- $I_1 \cap O_2 = \emptyset$ (Anti dependency)
- $I_2 \cap O_1 = \emptyset$ (Flow dependency)
- $O_1 \cap O_2 = \emptyset$ (Output dependency)

with I_1 and O_1 represent the input and output for the first thread T_1 whereas I_2 and O_2 represent the input and output for second thread T_2. If these three conditions are satisfied, the two threads can be concluded as independent to each other and therefore deemed parallelizable.

e.g., When i = 1;

I_1={c[0], a[1], b[1]}

O_1={c[1]}

When i = 2;

I_2={c[1], a[2], b[2]}

O_2={c[2]}

- $I_1 \cap O_2 = \emptyset$
- $I_2 \cap O_1 \neq \emptyset$ Does not fulfill condition, there is a flow dependency.
- $O_1 \cap O_2 = \emptyset$

Since there is a flow dependency between I_2 and O_1, if two threads were to read and write simultaneously from c[1], this would lead to a race condition and hence the content of the array will be corrupted.

b) In the context of data parallelism, is there a way to fix this discrepancy without changing the algorithm?

Based on the Bernstein analysis in (a), there is a flow dependency between I_2 and O_1., Bernstein's conditions were not met, and the data parallelism cannot be applied in this context. Therefore, there is no fix to this discrepancy unless the algorithm is modified to remove `c[i-1]` from the iteration.

Activity 3:

The following code attempts to parallelize a nested loop using OpenMP

```
int main()
{
     int a[M][N] =
{{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15},{16,17,18,19,20}};
     int b[M][N] =
{{1,2,3,4,5},{6,7,8,9,10},{11,12,13,14,15},{16,17,18,19,20}};
     int c[M][N] = {0};
     int i,j;


     #pragma omp parallel for shared (a,b,c)
     for(i=0; i<M; i++){
       for(j=0; j<N; j++){
             c[i][j] = a[i][j] * b[i][j];
       }
     }

     printf("Values of C:\n");
     for(i=0; i<M; i++){
       for(j=0; j<N; j++){
             printf("%d\t", c[i][j]);
       }
       printf("\n");
     }
     printf("\n");
     return 0;
}
```

When compiling the code above and executing it several times, the following inconsistent and incorrect results were obtained:

```
mpi@mpi-DL01:~/FIT3143/Week_03$ gcc -Wall -fopenmp omp_q3.c -o Out
mpi@mpi-DL01:~/FIT3143/Week_03$ ./Out
Values of C:
1       0       0       4       25
36      49      64      81      100
121     0       156     196     225
256     289     324     361     400

mpi@mpi-DL01:~/FIT3143/Week_03$ ./Out
Values of C:
1       0       4       16      0
0       0       64      81      100
121     144     169     196     0
256     289     340     361     0
```

a) These inconsistencies or incorrect results were caused by the usage of an incomplete OpenMP construct. Therefore, identify and correct the OpenMP construct in the code above.

```
#pragma omp parallel for shared (a,b,c) private (j)
```

Note: By default, OpenMP *parallel for* construct only privatizes the first (or outer most) loop counter variable (i.e., *i*). It does not privatize the second (or inner) loop counter variable (i.e., *j*). Therefore, variable *j* will be shared between multiple threads, leading to a race condition when attempting to increment the value of *j*. Consequently, this leads to the inconsistent and incorrect results as seen above.

b) Upon correcting the OpenMP construct, notice that the OpenMP construct only parallelizes the first or outer *for* loop. How do we apply OpenMP to parallelize both the outer and inner for loops? Hint: please read this site: https://ppc.cs.aalto.fi/ch3/nested/

```
#pragma omp parallel for collapse(2) shared (a,b,c) private(j)
     for(i=0; i<M; i++){
       for(j=0; j<N; j++){
             c[i][j] = a[i][j] * b[i][j];
       }
     }
```

c) Explain a possible weakness of using multiple nested parallel loops in an OpenMP fork-join model.

Each fork has to create a team of threads, likewise each join has to synchronize and destroy the team of threads. Each fork and join operation will have some non-zero overhead, so parallelizing each portion of a nested loop may result in more overhead than speed-up.

Activity 4:

The following code attempts to analyze the performance of the OpenMP schedule clause:

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

#define THREADS 6
#define N 18

int main ()
{
   int i;
   struct timespec start, end;
   double time_taken;

   clock_gettime(CLOCK_MONOTONIC, &start);
   #pragma omp parallel for schedule(static) num_threads(THREADS)
   for (i = 0; i < N; i++) {
       sleep(i);
       printf("Thread %d has completed iteration %d.\n",
omp_get_thread_num(), i);
   }
   clock_gettime(CLOCK_MONOTONIC, &end);
   time_taken = (end.tv_sec - start.tv_sec) * 1e9;
   time_taken = (time_taken + (end.tv_nsec - start.tv_nsec)) * 1e-9;
   printf("Overall time (s): %lf\n", time_taken);


    /* all threads done */
    printf("All done!\n");
    return 0;
}
```

When `schedule(static)` is specified within the OpenMP construct, the following outcome is observed (code compiled and executed on a computer with six cores):

```
mpi@mpi-DL01:~/FIT3143/Week_03$ gcc -Wall -fopenmp omp_q4.c -o Out
mpi@mpi-DL01:~/FIT3143/Week_03$ ./Out
Thread 0 has completed iteration 0.
Thread 0 has completed iteration 1.
Thread 1 has completed iteration 3.
Thread 0 has completed iteration 2.
Thread 2 has completed iteration 6.
Thread 1 has completed iteration 4.
Thread 3 has completed iteration 9.
Thread 4 has completed iteration 12.
Thread 1 has completed iteration 5.
Thread 2 has completed iteration 7.
Thread 5 has completed iteration 15.
Thread 3 has completed iteration 10.
Thread 2 has completed iteration 8.
Thread 4 has completed iteration 13.
Thread 3 has completed iteration 11.
Thread 5 has completed iteration 16.
Thread 4 has completed iteration 14.
Thread 5 has completed iteration 17.
Overall time (s): 48.000949
All done!
```

However, when `schedule(dynamic)` is specified within the OpenMP construct, the following outcome is observed (code compiled and executed on a computer with six cores):

```
mpi@mpi-DL01:~/FIT3143/Week_03$ gcc -Wall -fopenmp omp_q4.c -o Out
mpi@mpi-DL01:~/FIT3143/Week_03$ ./Out
Thread 2 has completed iteration 0.
Thread 3 has completed iteration 1.
Thread 1 has completed iteration 2.
Thread 0 has completed iteration 3.
Thread 4 has completed iteration 4.
Thread 5 has completed iteration 5.
Thread 2 has completed iteration 6.
Thread 3 has completed iteration 7.
Thread 1 has completed iteration 8.
Thread 0 has completed iteration 9.
Thread 4 has completed iteration 10.
Thread 5 has completed iteration 11.
Thread 2 has completed iteration 12.
Thread 3 has completed iteration 13.
Thread 1 has completed iteration 14.
Thread 0 has completed iteration 15.
Thread 4 has completed iteration 16.
Thread 5 has completed iteration 17.
Overall time (s): 33.000660
All done!
```

Based on the aforementioned code and observations:

a) Explain why the **`schedule(dynamic)`** construct performs better (i.e., lower computational time) than that of the **`schedule(static)`** construct. Hint: Refer to this link for an explanation of the OpenMP schedule clause.

Dynamic scheduling is better when the iterations may take very different amounts of time, which is the case in the code example above.

b) What would be the expected observation if **`schedule(guided)`** was used?

Answers from: http://jakascorner.com/blog/2016/06/omp-for-scheduling.html

The guided scheduling type is similar to the dynamic scheduling type. OpenMP again divides the iterations into chunks. Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.

The difference with the dynamic scheduling type is in the size of chunks. The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads. Therefore, the size of the chunks decreases.

In the context of the code above, if **`schedule(guided)`** was used, the performance would be similar to the dynamic scheduling type.