coms527-hw3

Shengwei Mao

February 2023

Some answers are referred from document of openmp official website www.openmp.org.

1 Q1

Answer the following questions:

1.1 a

How do threads differ from processes?

1.2 b

What does thread-safe mean?

1.3 Answer-a

A process is a program in execution, while threads are sharing resources with other threads belonging to the same process.

1.4 Answer-b

Serializability. Functions without access to global data or read only access are trivially thread safe. Typical restrictions for thread unsafe functions.

2 Q2

What are the advantages and disadvantages of OpenMP?

2.1 answer

Advantage: Incremental parallelization, small increase in code, code readablity, single CPU supported, single memory address space .
Disadvantage: Limited scalability, special compiler, implicit communication hard to understand, danger of incorrect synchronization.

3 Q3

Illustrate the fork-join execution model through a figure and explain how this model works.

3.1 Answer

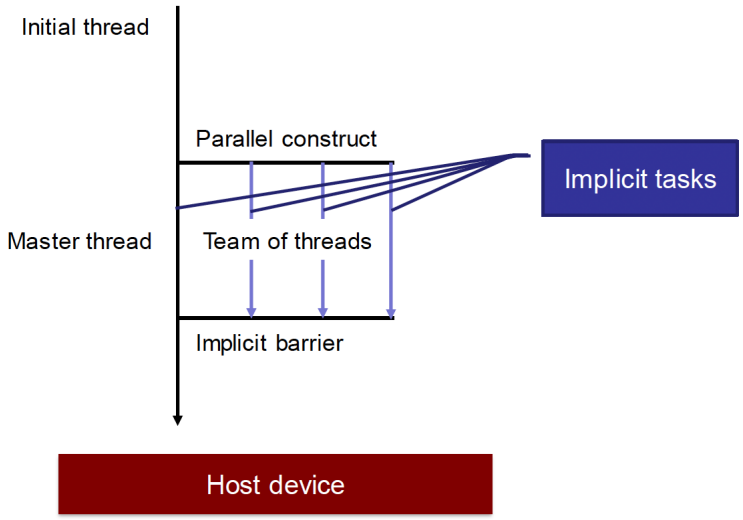

Figure 1:

From attached figure from class slice, it shows how fork-join execution happened. By parallel construct, initial thread is to split into several term of threads, every thread execute implicit tasks at almost same time. When met implicit barrier, the thread will stop and join together, and eventually join into master thread, then attached to host device.

4 Q4

4.1 a

count is shared
val is private
g is private
g is shared

4.2 b

count is shared
val is private
cnt is private
res is private
res depends, if execute "return *g", *res will be shared, else if execute "*return&val*", *res will be private
i is shared
j is private
a is shared

4.3 c

count have loop dependencies.

4.4 d

for j iteration, in line 23, read and write count.
for j+1 iteration, in line 18, read count;
So count could be flow dependency.

5 Q5

5.1 a

```
#pragma omp parallel for
for(i = 0; i < N; i++)
{
    y = sqrt(A[i]);
    D[i] = y + A[i] /(x*x);
}

y = y+2;
```

5.2 b

What is the difference between static scheduling and dynamic scheduling in OpenMP?
What are the advantages and disadvantages of dynamic scheduling compared to static scheduling?

5.2.1 Answer

Static scheduling is done(deterministically) at loop-entry time based on number of threads, total iterations, index of every iteration, while for dynamic scheduling, the distribution is done during execution of loop, each thread is assigned subset of iteration at loop entry, or after completing each thread asked for more iterations.
Static scheduling has low scheduling overhead but less flexible.
Dynamic scheduling has synchronization overhead but can be easily adjusted to load imbalance.

6 Q6

6.1 a

Briefly explain the three components (directives, runtime library, and environment variables) of the OpenMP API.

6.1.1 Answer

Directives:
In c/c++, directives are lines of code using "#pragma" as the beginning, it exploit shared memory parallelism by defining various types of parallel regions.
OpenMP runtime library are a collection of software programs used at openmp program run time to provide one or more native program functions or services.
OpenMP environment variables are store data that's used by the openmp and related programs.

6.2 b

How can a programmer enable conditional compilation with a preprocessor macro for OpenMP?

6.2.1 Answer

The _OPENMP macro name is defined to have the decimal value yyyymm where yyyy and mm are the year and month designations of the version of the OpenMP API that the implementation supports.
If a #define or a #undef preprocessing directive in user code defines or undefines the _OPENMP macro name, the behavior is unspecified.
Usually, programmer should write $\#include < omp.h >$ at the beginning of code to import openmp module, and use directives to indicate which parts of code should be implemented with openmp library. When compiling the program, in order to use openmp library, "-fopenmp" should also be used in command line.

6.3 c

If an OpenMP program is compiled with a compiler which doesn't support OpenMP, what will happen?

6.3.1 Answer

It highly depends on compiler, some compiler may show error or warning if it doesn't support openmp, some compiler may ignore directives code and compiler the rest of code without openmp.

7 Q7

7.1 a

What is an ICV in the OpenMP specification?

7.1.1 Answer

ICV is Internal Control Variables. It controls the behavior of an OpenMP program.

7.2 b

Lookup omp_get_num_threads. Is it a directive, a library function or an environment variable? What is its C-syntax? What does it do?

7.2.1 Answer

Library function, it returns the number of threads currently in the team executing the parallel region from which it's called.
C-syntax int omp_get_num_threads(void);

7.3 c

Lookup omp_get_thread_num. Is it a directive, a library function or an environment variable? What is its C-syntax? What does it do?

7.3.1 Answer

Library function, it returns the thread number, within its team, of the thread executing the function.
C-syntax: int omp_get_thread_num(void);

7.4 d

Lookup the barrier construct. Is it a directive, a library function or an environment variable? What is its C-syntax? What does it do?

7.4.1 Answer

Directive, it specifies an explicit barrier at the point at which the construct appears.
C-syntax: #pragma omp barrier new-line

7.5 e

What is an inactive parallel region?

7.5.1 Answer

A parallel region that is executed by a team of only one thread.

7.6 f

Lookup OMP_NUM_THREADS. Is it a directive, a library function or an environment variable? What does it do?

7.6.1 Answer

Environment variables, it is the maximum number of threads in the parallel region.

7.7 g

Lookup OMP_THREAD_LIMIT. Is it a directive, a library function or an environment variable? What does it do?

7.7.1 Answer

Environment variable, it sets the maximum number of OpenMP threads to use in a contention group.

8 Q8

Write a program to be implemented in OpenMP to add 2 arrays each of size 1000. Fill the arrays with random values. Also calculate the time taken by your parallel implementation. Compare the time taken with the sequential implementation and record the speedup. Please screenshot the time taken for both implementations.

8.0.1 Answer

The following is screenshot of runtime of program on MacBook Air M1(2020) and origin program.
From the result, it is obvious that if the iteration time is too low, parallel programming may show shortage, since it take time to load openmp library, but as the iteration time increase, the running time for parallel programming will have significant improvement comparing to seqential programming.

```
Iteration 100 times
Seqential runtime 000002 milliseconds
Parallel  runtime 000201 milliseconds
Speedup -00199 milliseconds

Iteration 1000 times
Seqential runtime 000003 milliseconds
Parallel  runtime 000104 milliseconds
Speedup -00101 milliseconds

Iteration 10000 times
Seqential runtime 000030 milliseconds
Parallel  runtime 000044 milliseconds
Speedup -00014 milliseconds

Iteration 100000 times
Seqential runtime 000288 milliseconds
Parallel  runtime 000135 milliseconds
Speedup 000153 milliseconds

Iteration 1000000 times
Seqential runtime 002743 milliseconds
Parallel  runtime 000761 milliseconds
Speedup 001982 milliseconds

Iteration 10000000 times
Seqential runtime 031997 milliseconds
Parallel  runtime 012818 milliseconds
Speedup 019179 milliseconds

Iteration 100000000 times
Seqential runtime 568069 milliseconds
Parallel  runtime 113002 milliseconds
Speedup 455067 milliseconds
```

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
struct timeval tval_before, tval_after, tval_result;
void init_random_array(int a[], int b[], long int length)
{
    int i = 0;
    srand(time(NULL));
    for(i = 0; i < length; i++){
        a[i] = rand();
        b[i] = rand();
    }
}

long int sequential_add(int a[], int b[], long int length)
{
    int* c = (int *)malloc(length*4);
    gettimeofday(&tval_before, NULL);
    for (int i = 0; i < length; i++)
    {
        c[i] = a[i] + b[i];
    }
    gettimeofday(&tval_after, NULL);
    timersub(&tval_after, &tval_before, &tval_result);
    return (long int)tval_result.tv_usec;
}

long int parallel_add(int a[], int b[], long int length)
{
    int* c = (int *)malloc(length*4);
    struct timeval tval_before, tval_after, tval_result;
    gettimeofday(&tval_before, NULL);
    #pragma omp parallel for
    for (int i = 0; i < length; i++)
    {
        c[i] = a[i] + b[i];
    }
    gettimeofday(&tval_after, NULL);
    timersub(&tval_after, &tval_before, &tval_result);
    return (long int)tval_result.tv_usec;
}

void exe(long int length){
    int* a = (int *)malloc(length*4);
```

```
    int* b = (int *)malloc(length*4);
    init_random_array(a, b, length);
    printf("Iteration %ld times\n", length);
    long int stime = sequential_add(a, b, length);
    long int ptime = parallel_add(a, b, length);
    printf("Seqential runtime %06ld milliseconds\n", stime);
    printf("Parallel  runtime %06ld milliseconds\n", ptime);
    printf("Speedup %06ld milliseconds\n\n", stime - ptime);
}

int main(){
    long int length;
    for(int i = 2; i < 9; i++){
        length = (long int)pow(10, (double)i);
        exe(length);
    }
}
```