DTS202TC Foundation of Parallel Computing

Lecture 4 OpenMP

Hongbin Liu

Xi'an Jiaotong-Liverpool University

Nov 30, 2023

Administrative

- Week 5 CUDA
  - 5 hours face-to-face lectures + lab from Nvidia Guest Lecturer
  - Same schedule as normal teaching, tutorial and lab for you to do exercise and test to get the CUDA certificate
  - No recording due to the copyright
- Individual Assessment 2 due on Saturday **Dec. 23rd**, 2023 @ 11:59pm
  - You must at least choose MPI

Outline

- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive
  - Scheduling Loops
- Synchronization
- Thread-safety

Recall from last lecture


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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

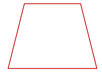
```c
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

Parallel for

- Forks a team of threads to execute the following structured block.
- However, the structured block following the parallel for directive must be a for loop.
- With the parallel for directive, the system parallelizes the for loop by dividing the iterations of the loop among the threads.
- With just a parallel directive, in general, the work must be divided among the threads by the threads themselves.

Example

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```



```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```


Scope

- The default scope for all variables in a parallel directive is shared.
- In a loop that is parallelized with a parallel for directive, the default scope of the loop variable is private.

Legal Forms for Parallelizable for Statements

- OpenMP will only parallelize for loops for which the number of iterations can be determined.
  - From the for statement itself
  - Prior to execution of the loop

Examples of Illegal Forms

- The "infinite loop" cannot be parallelized.

```
for ( ; ; ) {
     . . .
  }
```

- The following loop cannot be parallelized.

```
for (i = 0; i < n; i++) {
    if ( . . . ) break;
    . . .
 }
```

Since the number of iterations can't be determined from the `for` statement alone. This `for` loop is also not a structured block, since the `break` adds another point of exit from the loop.

Legal Forms

- Legal forms for parallelizable for statements

```
                                        index++
                                        ++index
                         index < end    index--
                         index <= end   --index
for ( index = start ;    index >= end ; index += incr               )
                         index > end    index -= incr
                                        index = index + incr
                                        index = incr + index
                                        index = index - incr
```


Caveats

- The variable index must have integer or pointer type (e.g., it can't be a float).
- The expressions start, end, and incr must have a compatible type. For example, if index is a pointer, then incr must have integer type.
- The expressions start, end, and incr must not change during execution of the loop.
- During execution of the loop, the variable index can only be modified by the "increment expression" in the for statement.

Data Dependencies

```
fibo[ 0 ] = fibo[ 1 ] = 1;
for (i = 2; i < n; i++)
   fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

note 2 threads

```
   fibo[ 0 ] = fibo[ 1 ] = 1;
#  pragma omp parallel for num_threads(2)
   for (i = 2; i < n; i++)
      fibo[ i ] = fibo[ i – 1 ] + fibo[ i – 2 ];
```

1 1 2 3 5 8 13 21 34 55

this is correct

but sometimes we get this

1 1 2 3 5 8 0 0 0 0

Possible Case

thread 0: fibo[2] fibo[3] fibo[4] fibo[5]
thread 1: fibo[6] fibo[7] fibo[8] fibo[9]

• Correct: thread 0 finishes its computations before thread 1 starts.

• Incorrect: thread 0 has not computed fibo[4] and fibo[5], when thread 1 computes fibo[6]. It appears that the system has initialized the entries in fibo to 0, and thread 1 is using the values fibo[4] = 0 and fibo[5] = 0 to compute fibo[6] and so on.

What happened?

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a parallel for directive.

2. A loop in which the results of one or more iterations depend on other iterations cannot, in general, be correctly parallelized by OpenMP.

Estimating π

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}$$

```
double factor = 1.0;
double sum = 0.0;
for (k = 0; k < n; k++) {
   sum += factor/(2*k+1);
   factor = -factor;
}
pi_approx = 4.0*sum;
```


OpenMP solution #1

loop dependency

```
    double factor = 1.0;
    double sum = 0.0;
#   pragma omp parallel for num_threads(thread_count) \
       reduction(+:sum)
    for (k = 0; k < n; k++) {
       sum += factor/(2*k+1);
       factor = -factor;
    }
    pi_approx = 4.0*sum;
```

OpenMP solution #2

Shared scope.

```
        double factor = 1.0;
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
        pi_approx = 4.0*sum;
```


OpenMP solution #3

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           reduction(+:sum) private(factor)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Insures factor has private scope.

More on Scope

- Let the programmer specify the scope of each variable in a block.

  The default clause:

- With this clause the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block.

The default clause

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           default(none) reduction(+:sum) private(k, factor) \
           shared(n)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

Outline

- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive
  - Scheduling Loops
- Synchronization
- Thread-safety

An Example Cyclic Partition

We want to parallelize this loop.

```
sum = 0.0;
for (i = 0; i <= n; i++)
   sum += f(i);
```

Thread	Iterations
0	$0,\ n/t,\ 2n/t,\ \ldots$
1	$1,\ n/t+1,\ 2n/t+1,\ \ldots$
$\vdots$	$\vdots$
$t-1$	$t-1,\ n/t+t-1,\ 2n/t+t-1,\ \ldots$

Assignment of work using cyclic partitioning.

Example of Cyclic Assignment

```
double f(int i) {
   int j, start = i*(i+1)/2, finish = start + i;
   double return_val = 0.0;

   for (j = start; j <= finish; j++) {
      return_val += sin(j);
   }
   return return_val;
} /* f */
```

Our definition of function f.

Results

- f(i) calls the sin function i times. The time to execute f(2i) requires approximately twice as much time as the time to execute f(i).
- n = 10,000
  - one thread
  - run-time = 3.67 seconds.
- n = 10,000
  - two threads
  - default assignment
  - run-time = 2.76 seconds
  - speedup = 1.33
- n = 10,000
  - two threads
  - cyclic assignment
  - run-time = 1.84 seconds
  - speedup = 1.99

The Schedule Clause

In OpenMP, assigning iterations to threads is called scheduling, and the schedule clause can be used to assign iterations in either a *parallel for* or a *for* directive.

- Default schedule:

```
   sum = 0.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+:sum)
   for (i = 0; i <= n; i++)
      sum += f(i);
```

- Cyclic schedule:

```
   sum = 0.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+:sum) schedule(static,1)
   for (i = 0; i <= n; i++)
      sum += f(i);
```


Schedule (type , chunksize)

- Schedule Clause:

```
schedule(<type> [, <chunksize>])
```

- Type can be:
  - static: the iterations can be assigned to the threads before the loop is executed.
  - dynamic or guided: the iterations are assigned to the threads while the loop is executing.
  - auto: the compiler and/or the run-time system determine the schedule.
  - runtime: the schedule is determined at run-time.
- The chunksize is a positive integer.
- In OpenMP, a chunk of iterations is a block of iterations that would be executed consecutively in the serial loop. The number of iterations in the block is the chunksize.
- Only static, dynamic, and guided schedules can have a chunksize .

The Static Schedule Type

- For a static schedule, the system assigns chunks of chunksize iterations to each thread in a round-robin fashion.
- The chunksize can be omitted. If it is omitted, the chunksize is approximately total_iterations/thread_count.

Example

- 12 iterations, 0, 1, . . . , 11, and 3 threads

`schedule(static,1)`	Thread 0 :	0, 3, 6, 9
	Thread 1 :	1, 4, 7, 10
	Thread 2 :	2, 5, 8, 11
`schedule(static,2)`	Thread 0 :	0, 1, 6, 7
	Thread 1 :	2, 3, 8, 9
	Thread 2 :	4, 5, 10, 11
`schedule(static,4)`	Thread 0 :	0, 1, 2, 3
	Thread 1 :	4, 5, 6, 7
	Thread 2 :	8, 9, 10, 11

The Dynamic Schedule Type

- The iterations are also broken up into chunks of chunksize consecutive iterations.
- Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system.
- This continues until all the iterations are completed.
- The chunksize can be omitted. When it is omitted, a chunksize of 1 is used.

The Guided Schedule Type

- Each thread also executes a chunk, and when a thread finishes a chunk, it requests another one.
- However, in a guided schedule, the chunksize is reduced exponentially as each chunk is dispatched to a thread.
- The chunksize refers to the smallest chunk that should be dispatched. When the number of iterations left is less than chunksize, the entire set of iterations is dispatched at once.
  - If chunksize is specified, it decreases down to chunksize, with the exception that the very last chunk can be smaller than chunksize.
  - If no chunksize is specified, the size of the chunks decreases down to 1.

Example

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

Assignment of trapezoidal rule iterations 1–9999 using a guided schedule(schedule(guided)) with two threads.

The Runtime Schedule Type

- The system uses the environment variable OMP_SCHEDULE to determine at run-time how to schedule the loop.
- The OMP_SCHEDULE environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule.

Ideas to Explore Schedules

- There is some overhead associated with the use of a schedule clause.
- The overhead is greater for dynamic schedules than static schedules, and the overhead associated with guided schedules is the greatest of the three.
- Thus, if we're getting satisfactory performance without a schedule clause, we should go no further.
- However, if we suspect that the performance of the default schedule can be substantially improved, we should probably experiment with some different schedules.

Ideas to Explore Schedules

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a static schedule with small chunksizes will probably give the best performance
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The schedule(runtime) clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable OMP_SCHEDULE .

Outline

- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive
  - Scheduling Loops
- Synchronization
- Thread-safety

The Atomic Directive (1)

- Unlike the critical directive, it can only protect critical sections that consist of a single C assignment statement.

```
# pragma omp atomic
```

- The statement must have one of the following forms:

```
x <op>= <expression>;
x++;
++x;
x--;
--x;
```

- Here <op> can be one of the binary operators

  +, *, −, /, &, ^, |, <<, or >>

- <expression> must not reference x.

The Atomic Directive (2)

- Only the load and store of x are guaranteed to be protected.
- E.g:

```
#    pragma omp atomic
     x += y++;
```

  - A thread's update to x will be completed before any other thread can begin updating x .
  - The update to y may be unprotected.
- Many processors provide a special load-modify-store instruction.
- A critical section that only does a load-modify-store can be protected much more efficiently by using this special instruction rather than the constructs that are used to protect more general critical sections.

Critical Sections

- OpenMP provides the option of adding a name to a critical directive:

```
# pragma omp critical(name)
```

- When we do this, two blocks protected with critical directives with different names can be executed simultaneously.
- However, the names are set during compilation, and we want a different critical section for each thread's queue. So need to set the names at run-time.
- When we want to allow simultaneous access to the same block of code by threads accessing different queues, the named critical directive isn't sufficient.

Locks

- A lock consists of a data structure and functions that allow the programmer to explicitly enforce mutual exclusion in a critical section.

```
/* Executed by one thread */
Initialize the lock data structure;
. . .
/* Executed by multiple threads */
Attempt to lock or set the lock data structure;
Critical section;
Unlock or unset the lock data structure;
. . .
/* Executed by one thread */
Destroy the lock data structure;
```


Types of Locks

- A simple lock can only be set once before it is unset.
- A nested lock can be set multiple times by the same thread before it is unset.
- Simple lock functions:

```
void omp_init_lock(omp_lock_t*    lock_p   /* out */);
void omp_set_lock(omp_lock_t*     lock_p   /* in/out */);
void omp_unset_lock(omp_lock_t*   lock_p   /* in/out */);
void omp_destroy_lock(omp_lock_t* lock_p   /* in/out */);
```

Using Locks

```
# pragma omp critical
    /* q_p = msg_queues[dest] */
    Enqueue(q_p, my_rank, mesg);
```

```
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, mesg);
omp_unset_lock(&q_p->lock);
```


Critical, Atomic or Locks?

- If the critical section consists of an assignment statement having the required form, it will probably perform at least as well with the atomic directive as the other methods.
- The atomic directive to enforce mutual exclusion across all atomic directives in the program—this is the way the unnamed critical directive behaves.
- If multiple different critical sections are protected by atomic directives—you should use named critical directives or locks.
- The use of locks should probably be reserved for situations in which mutual exclusion is needed for a data structure rather than a block of code.

Outline

- OpenMP Basics
  - Our First OpenMP Program
  - Fundamental Concepts and Library Functions
- The Trapezoidal Rule
- Scope of Variables
- Task Parallelism
  - The Reduction Clause
  - The `parallel for` Directive
  - Scheduling Loops
- Synchronization
- Thread-safety

Thread-Safety

- A block of code is thread-safe if it can be simultaneously executed by multiple threads without causing problems.

Example

- Suppose we want to use multiple threads to “tokenize” a file that consists of ordinary English text.
- The tokens are just contiguous sequences of characters separated from the rest of the text by white-space — a space, a tab, or a newline.

Simple Approach

- Divide the input file into lines of text and assign the lines to the threads in a round-robin fashion.
- The first line goes to thread 0, the second goes to thread 1, . . . , the tth goes to thread t, the t +1st goes to thread 0, etc.
- We can serialize access to the lines of input using semaphores.
- After a thread has read a single line of input, it can tokenize the line using the strtok function.

The Strtok Function

```
char* strtok(
      char*        string       /* in/out */,
      const char*  separators   /* in     */);
```

- The idea is that in the first call, strtok caches a pointer to string, and for subsequent calls it returns successive tokens taken from the cached copy.
- The first time it’s called the string argument should be the text to be tokenized.(One line of input.)
- For subsequent calls, the first argument should be NULL.

Multi-threaded Tokenizer

```
void Tokenize(
      char*  lines[]       /* in/out */,
      int    line_count    /* in     */,
      int    thread_count  /* in     */) {
   int my_rank, i, j;
   char *my_token;

#  pragma omp parallel num_threads(thread_count) \
      default(none) private(my_rank, i, j, my_token) shared(lines, line_count)
   {
      my_rank = omp_get_thread_num();
#     pragma omp for schedule(static, 1)
      for (i = 0; i < line_count; i++) {
         printf("Thread %d > line %d = %s", my_rank, i, lines[i]);
         j = 0;
         my_token = strtok( str: lines[i],  sep: " \t\n");
         while ( my_token != NULL ) {
            printf("Thread %d > token %d = %s\n", my_rank, j, my_token);
            my_token = strtok( str: NULL,  sep: " \t\n");
            j++;
         }
      if (lines[i] != NULL)
         printf("Thread %d > After tokenizing, my line = %s\n",
            my_rank, lines[i]);
      } /* for i */
   }  /* omp parallel */

}  /* Tokenize */
```

Running with One Thread

- It correctly tokenizes the input stream.

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

Running with Two Threads

Pease porridge hot.
Pease porridge cold.
Pease porridge in the pot
Nine days old.

```
Thread 0 > my line = Pease porridge hot.
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = hot.
Thread 1 > my line = Pease porridge cold.
Thread 0 > my line = Pease porridge in the pot
Thread 0 > string 1 = Pease
Thread 0 > string 2 = porridge
Thread 0 > string 3 = in
Thread 0 > string 4 = the
Thread 0 > string 5 = pot
Thread 1 > string 1 = Pease
Thread 1 > my line = Nine days old.
Thread 1 > string 1 = Nine
Thread 1 > string 2 = days
Thread 1 > string 3 = old.
```

Oops!

What happened?

- strtok caches the input line by declaring a variable to have static storage class.
- This causes the value stored in this variable to persist from one call to the next.
- Unfortunately for us, this cached string is shared, not private.
- Thus, thread 0's call to strtok with the third line of the input has apparently overwritten the contents of thread 1's call with the second line.
- So the strtok function is not thread-safe. If multiple threads call it simultaneously, the output may not be correct.

Other Unsafe C Library Functions

- Regrettably, it's not uncommon for C library functions to fail to be thread-safe.
- The random number generator random in stdlib.h.

“re-entrant” (Thread Safe) Functions

- In some cases, the C standard specifies an alternate, thread-safe, version of a function.

```
char* strtok_r(
      char*         string       /* in/out */,
      const char*   separators,  /* in     */
      char**        saveptr_p    /* in/out */);
```

Multi-threaded Tokenizer

```
void Tokenize(
      char*  lines[]       /* in/out */,
      int    line_count    /* in     */,
      int    thread_count  /* in     */) {
   int my_rank, i, j;
   char *my_token, *saveptr;

#  pragma omp parallel num_threads(thread_count) \
      default(none) private(my_rank, i, j, my_token, saveptr) \
      shared(lines, line_count)
   {
      my_rank = omp_get_thread_num();
#     pragma omp for schedule(static, 1)
      for (i = 0; i < line_count; i++) {
         printf("Thread %d > line %d = %s", my_rank, i, lines[i]);
         j = 0;
         my_token = strtok_r( str: lines[i],  sep: " \t\n",  lasts: &saveptr);
         while ( my_token != NULL ) {
            printf("Thread %d > token %d = %s\n", my_rank, j, my_token);
            my_token = strtok_r( str: NULL,  sep: " \t\n",  lasts: &saveptr);
            j++;
         }
      if (lines[i] != NULL)
         printf("Thread %d > After tokenizing, my line = %s\n",
            my_rank, lines[i]);
      } /* for i */
   }  /* omp parallel */

}  /* Tokenize */
```


Caveat

Incorrect programs can produce correct output!

One way to check if your parallel program is correct, the result/output of your parallel version should be always the same as your serial version (thread/process number = 1).

Summary

- Start working on your individual assessment 2
- Don’t forget to bring your laptop for the CUDA lectures.