DTS202TC Foundation of Parallel Computing

Lecture 4 OpenMP

Hongbin Liu

Xi'an Jiaotong-Liverpool University

Nov 27, 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.
  - Make sure you attend the lecture, you will be given a redeem code to redeem the course for free.
  - No recording due to the copyright.
- Individual Assessment 2 due on Saturday **Dec. 23rd**, 2023 @ 11:59pm
  - You must at least choose MPI.
- A2 FAQ
  - Can we use Gprof to get the elapsed time?
    - No, Gprof does not support multi-threading programs, please get the time manually by print it out in your code.
  - Can we make modifications of the serial code?
    - Yes, if you find problems of your A1 implementation, you can enhance it, and implement the parallel version base on the serial code.
  - Do we use the same pgm file for testing?
    - Yes, you can use the same pgm as A1 for your testing and evalution.

A1 Common Problems

- Did not close the file after finishing the job (part of the coding quality)
- Did not use struct (part of the coding quality)
- One main function() contains everything
  - Very bad coding practice
- Hard-coding image array size

```
int data[200000][200000];
```

- Output file name incorrect
- Makefile does not work, some students used cmake
- The parallel design should be in general, do not put it into one of the implementations.

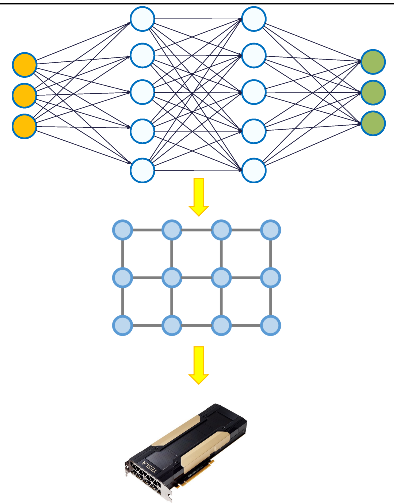
```
FILE *f = fopen("test.txt", "r");
int digit, count;
while ((count = fgetc(f)) != EOF) {
    ungetc(count, f); // what does this do?

    fscanf(f, "%d", &digit);
    printf("%d \n", digit);
}
fclose(f);
```

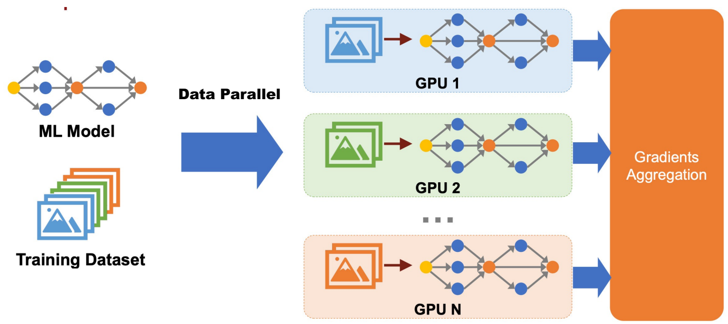

Wrap up so far

- Forster Method
  - Partitioning
  - Communication
  - Aggregation
  - Mapping
- Performance analysis
  - Speedup and Efficency
- Profiling
  - Gprof and other profiling tools
- Debugging C
  - Setting break points
  - Step over and step into
- Pthread
  - pthread_create, pthread_join
  - Critical Sections
  - Mutex
  - Semaphore
- MPI
  - Pointer-to-Point communication
    - MPI_Send and MPI_Receive
  - Collective communication
    - MPI_Reduce
    - MPI_Allreduce

Another Parallel Example

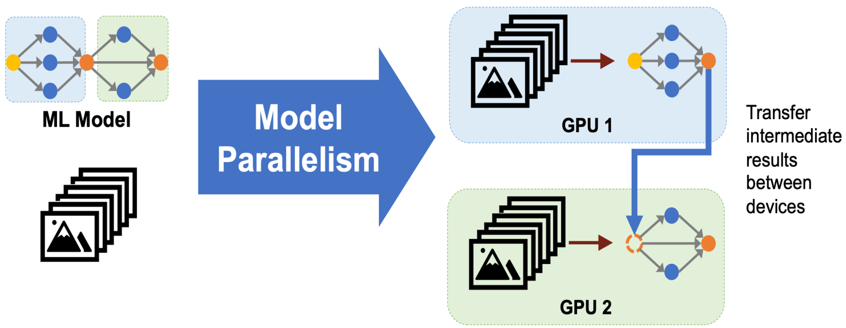


Another Parallel Example – cont.



- Each GPU saves a replica of the entire model
- Cannot train large models that exceed GPU device memory

Another Parallel Example – cont.



This week's topic

- 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

OpenMP

- An API for shared-memory parallel programming.
- MP = multiprocessing
- Designed for systems in which each thread or process can potentially have access to all available memory.

OpenMP vs. Pthreads

- Pthreads requires that the programmer explicitly specify the behavior of each thread. OpenMP allows the compiler and run-time system to determine some of the details of thread behavior.
- Any Pthreads program can be used with any C compiler, provided the system has a Pthreads library. OpenMP requires compiler support for some operations, and hence it's entirely possible that you may run across a C compiler that can't compile OpenMP programs into parallel programs.
- Pthreads is lower level. Cost: Specify every detail of the behavior of each thread. OpenMP can be simpler to code some parallel behaviors. Cost: Some low-level thread interactions can be more difficult to program.

Pragma

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that aren't part of the basic C specification.
- Compilers that don't support the pragmas ignore them.

#pragma

"Hello, World" Using OpenMP


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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Compiling and Running

gcc -g -Wall -fopenmp -o omp_hello omp_hello. c

./omp_hello 4

compiling

running with 4 threads

possible outcomes

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

Demo

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

OpenMP Syntax

- Most of the constructs in OpenMP are compiler directives
  - #pragma omp directive [clause list]
- A parallel directive: # pragma omp parallel
  - The number of threads that run the following structured block of code is determined by the run-time system.
- Function prototypes and types in the file
  - #include <omp.h>

Note

- There may be system-defined limitations on the number of threads that a program can start.
- The OpenMP standard doesn't guarantee that this will actually start thread_count threads.
- Most current systems can start hundreds or even thousands of threads.
- Unless we're trying to start a lot of threads, we will almost always get the desired number of threads.

In Case the Compiler doesn't Support OpenMP

```
# include <omp.h>
```

```
#ifdef _OPENMP
# include <omp.h>
#endif
```


In Case the Compiler doesn't Support OpenMP

```
# ifdef _OPENMP
  int my_rank = omp_get_thread_num ( );
  int thread_count = omp_get_num_threads ( );
# else
  int my_rank = 0;
  int thread_count = 1;
# endif
```


OpenMP Library Functions

- **Control the number of threads and Processors:**

```
#include <omp.h>
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel ();
```

- **Set and monitor thread creation:**

```
#include <omp.h>
void omp_set_dynamic(int dynamic_threads);
int omp_get_dynamic ();
void omp_set_nested (int nested);
int omp_get_nested ();
```

- **Mutex:**

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
int omp_test_lock(omp_lock_t *lock);
```

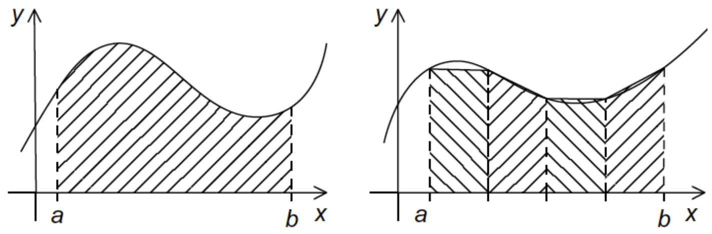
- **OpenMP also supports nested lock, which has similar semantics with simple lock.**

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

The Trapezoidal Rule

Xi'an Jiaotong-Liverpool University 西交利物浦大学



If each subinterval has the same length h and if we define $h=(b-a)/n$, $x_i=a+ih$, $i=0,1,2,\ldots,n$, then our approximation will be

$$h[f(x_0)/2+f(x_1)+f(x_2)+\cdots+f(x_{n-1})+f(x_n)/2]$$

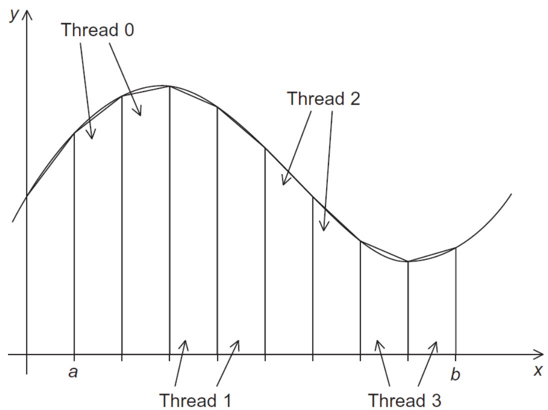
Serial Algorithm

Xi'an Jiaotong-Liverpool University 西交利物浦大学

```
/* Input: a, b, n */
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++) {
   x_i = a + i*h;
   approx += f(x_i);
}
approx = h*approx;
```


Assignment of Trapezoids to Threads

Xi'an Jiaotong-Liverpool University 西交利物浦大学



Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

- Unpredictable results when two (or more) threads attempt to simultaneously execute: global_result += my_result ;(critical section)
- Recall:

Race Condition, Critical Section

Mutual Exclusion

Critical Directive:

pragma omp critical

global_result += my_result ;

only one thread can execute the following structured block at a time

First OpenMP Trapezoidal Rule Program(1)

```
#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 */
```


First OpenMP Trapezoidal Rule Program(2)

```
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 */
```

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

Definition of Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.

Scope in OpenMP

- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.

Example: Hello, World

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

void Hello(void);  /* Thread function */

int main(int argc, char* argv[]) {
   /* Get number of threads from command line */
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Private Scope

Example: Trapezoidal Rule(1)

```
#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 */
```

Shared Scope

Example: Trapezoidal Rule(2)

Private Scope

```
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 */
```

Shared Scope

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

Analysis of Trapezoidal Rule Program(1)

In the OpenMP program, this more complex version is used to get global_result by adding each thread's local calculation.

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

Although we'd prefer this for a serial implementation.

```
double Trap(double a, double b, int n);
```

```
global_result = Trap(a, b, n);
```

Analysis of Trapezoidal Rule Program(2)

For the pointer version, we need to add each thread's local calculation to get global_result. If we use this, there's no critical section!

```
double Local_trap(double a, double b, int n);
```

If we fix it like this, we force the threads to execute sequentially.

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
#     pragma omp critical
      global_result += Local_trap(double a, double b, int n);
   }
```


Analysis of Trapezoidal Rule Program(3)

We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
      double my_result = 0.0;  /* private */
      my_result += Local_trap(double a, double b, int n);
#     pragma omp critical
      global_result += my_result;
   }
```


Syntax of Reduction Clause

```
reduction(<operator>: <variable list>)
```

+, *, -, &, |, ^, &&, ||

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

Reduction

A reduction clause can be added to a parallel directive.

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

- When a variable is included in a reduction clause, the variable itself is shared. However, a private variable is created for each thread in the team.
- In the parallel block each time a thread executes a statement involving the variable, it uses the private variable. When the parallel block ends, the values in the private variables are combined into the shared variable.

Summary

- Start working on your individual assessment 2