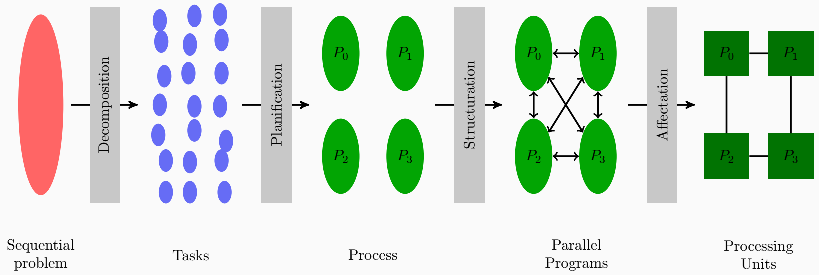
Toward HPC

Chapter 4 – OpenMP

M1 – MSIAM
March 13, 2018

So far . . .

Foster design



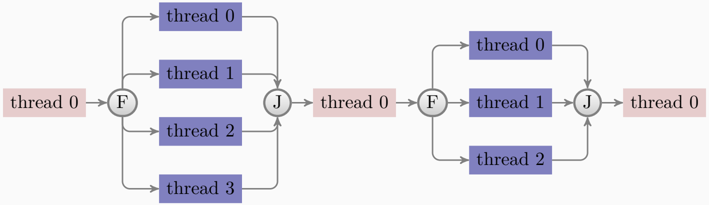
About OpenMP

What is OpenMP

- OpenMP provides high-level thread programming
- Multiple cooperating threads are allowed to run simultaneously
- Threads are created and destroyed dynamically in a fork-join pattern
  - An OpenMP program consists of a number of parallel regions
  - Between two parallel regions there is only one master thread
  - In the beginning of a parallel region, a team of new threads is spawned
- The newly spawned threads work simultaneously with the master thread
- At the end of a parallel region, the new threads are destroyed

Fork-Join Execution Model

- Parallelism is achieved by generating multiple threads that run in parallel
  - A fork Ⓕ is when a single thread is made into multiple, concurrently executing threads
  - A join Ⓙ is when the concurrently executing threads synchronize back into a single thread
- OpenMP programs essentially consist of a series of forks and joins.



Getting started, things to remember

- Remember the header file

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

- Insert compiler directives in C++ syntax as

```
#pragma omp...
```

- Compile with for example *c++ -fopenmp code.cpp*
- Execute
  - Remember to assign the environment variable OMP_NUM_THREADS
  - It specifies the total number of threads inside a parallel region, if not otherwise overwritten

General code structure

```
#include <omp.h>
main ()
{
 int var1, var2, var3;
 /* serial code */
 /* ... */
 /* start of a parallel region */
#pragma omp parallel private(var1, var2) shared(var3)
 {
  /* ... */
 }
 /* more serial code */
 /* ... */
 /* another parallel region */
#pragma omp parallel
 {
  /* ... */
 }
}
```

Parallel Region

Parallel region

- A parallel region is a block of code that is executed by a team of threads
- The following compiler directive creates a parallel region

```
#pragma omp parallel { ... }
```

- Clauses can be added at the end of the directive
- Most often used clauses
  - **default**(shared) or **default**(none)
  - **public**(list of variables)
  - **private**(list of variables)

Hello world, not again, please!

```
#include <omp.h>
#include <cstdio>
int main (int argc, char *argv[])
{
 int th_id, nthreads;
#pragma omp parallel private(th_id) shared(nthreads)
 {
  th_id = omp_get_thread_num();
  printf("Hello World from thread %d\n", th_id);
#pragma omp barrier
  if ( th_id == 0 ) {
   nthreads = omp_get_num_threads();
   printf("There are %d threads\n",nthreads);
  }
 }
 return 0;
}
```

Important OpenMP library routines

- **int** omp_get_num_threads(), returns the number of threads inside a parallel region
- **int** omp_get_thread_num(), returns the a thread for each thread inside a parallel region
- **void** omp_set_num_threads(**int**), sets the number of threads to be used
- **void** omp_set_nested(**int**), turns nested parallelism on/off

Single execution

#pragma omp single { ... }

The code is executed by one thread only, no guarantee which thread

Can introduce an implicit barrier at the end

#pragma omp master { ... }

Code executed by the master thread, guaranteed and no implicit barrier at the end.

Coordination and synchronization

#pragma omp barrier

Synchronization, must be encountered by all threads in a team (or none)

#pragma omp ordered { a block of codes }

is another form of synchronization (in sequential order). The form

#pragma omp critical { a block of codes }

and

#pragma omp atomic { single assignment statement }

is more efficient than

#pragma omp critical { a block of codes }

Data scope

OpenMP data scope attribute clauses:

- shared
- **private**
- firstprivate
- lastprivate
- reduction

What are the purposes of these attributes

- define how and which variables are transferred to a parallel region (and back)
- define which variables are visible to all threads in a parallel region, and which variables are privately allocated to each thread

Some remarks

- When entering a parallel region, the **private** clause ensures each thread having its own new variable instances. The new variables are assumed to be uninitialized.
- A shared variable exists in only one memory location and all threads can read and write to that address. It is the programmer's responsibility to ensure that multiple threads properly access a shared variable.
- The firstprivate clause combines the behavior of the private clause with automatic initialization.
- The lastprivate clause combines the behavior of the private clause with a copy back (from the last loop iteration or section) to the original variable outside the parallel region.

Worksharing constructs

Parallel for loop

- Inside a parallel region, the following compiler directive can be used to parallelize a for-loop:

```
#pragma omp for
```

- Clauses can be added, such as
  - schedule(**static**, chunk size)
  - schedule(dynamic, chunk size)
  - schedule(guided, chunk size) (non-deterministic allocation)
  - schedule(runtime)
  - **private**(list of variables)
  - reduction(**operator**:variable)
  - nowait

Load balancing

Schedule	When to Use
static	Even and predictable workload per iteration; scheduling may be done at compilation time, least work at runtime.
dynamic	Highly variable and unpredictable workload per iteration; most work at runtime
guided	Special case of dynamic scheduling; compromise between load balancing and scheduling overhead at runtime

Example code

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
main ()
{
 int i, chunk;
 float a[N], b[N], c[N];
 for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
 chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
 {
#pragma omp for schedule(dynamic,chunk)
  for (i=0; i < N; i++)
   c[i] = a[i] + b[i];
 } /* end of parallel region */
}
```

More on Parallel for loop

- The number of loop iterations can not be non-deterministic; **break**, **return**, exit, **goto** not allowed inside the for-loop
- The loop index is private to each thread
- A reduction variable is special
  - During the for-loop there is a local private copy in each thread
  - At the end of the for-loop, all the local copies are combined together by the reduction operation
- Unless the nowait clause is used, an implicit barrier synchronization will be added at the end by the compiler

```
// #pragma omp parallel and #pragma omp for
```

can be combined into

```
#pragma omp parallel for
```

What can happen with this loop?

What happens with code like this

```
#pragma omp parallel for
for (i=0; i<n; i++) {
 sum += a[i]*a[i];
}
```

All threads can access the sum variable, but the addition is not atomic! It is important to avoid race between threads. So-called reductions in OpenMP are thus important for performance and for obtaining correct results. OpenMP lets us indicate that a variable is used for a reduction with a particular operator. The above code becomes

```
sum = 0.0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<n; i++) {
 sum += a[i]*a[i];
}
```

Inner product

$$\sum_{i=0}^{n-1} a_i b_i$$

```
int i;
double sum = 0.;

/* allocating and initializing arrays */
/* ... */
#pragma omp parallel for default(shared) private(i) reduction(+:sum)
for (i=0; i<N; i++){
 sum += a[i]*b[i];
}
```

Different threads do different tasks

Different threads do different tasks independently, each section is executed by one thread.

```
#pragma omp parallel
{
#pragma omp sections
 {
#pragma omp section
  funcA ();
#pragma omp section
  funcB ();
#pragma omp section
  funcC ();
 }
}
```

Parallelizing nested for-loops

Serial code

```
for (i=0; i<100; i++){
 for (j=0; j<100; j++){
  a[i][j] = b[i][j] + c[i][j]
 }
}
```

- Why not parallelize the inner loop?
- Why must j be private?

Parallelizing nested for-loops

Parallelization

```
#pragma omp parallel for private(j)
for (i=0; i<100; i++){
 for (j=0; j<100; j++){
  a[i][j] = b[i][j] + c[i][j]
 }
}
```

- Why not parallelize the inner loop?
- Why must j be private?

Nested parallelism

When a thread in a parallel region encounters another parallel construct, it may create a new team of threads and become the master of the new team.

```
#pragma omp parallel num_threads(4)
{
 /* .... */
#pragma omp parallel num_threads(2)
 {
  //
 }
}
```

Tasking construct

```
struct node {
 struct node *left, *right;
};
void traverse( struct node *p ) {
 if (p->left)
#pragma omp task
   traverse(p->left);
 if (p->right)
#pragma omp task
   traverse(p->right);
 process(p);
}
int main() {
 node *root = ...;
#pragma omp parallel
#pragma omp single
 traverse(root);
}
```

Parallel tasks

When a thread encounters a task construct, a task is generated for the associated structured block.

The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task.

task should be called from within a parallel region for the different specified tasks to be executed in parallel.

The tasks will be executed in no specified order because there are no synchronization directives.

```
void postorder_traverse( struct node *p ) {
 if (p->left)
#pragma omp task
   postorder_traverse(p->left);
 if (p->right)
#pragma omp task
   postorder_traverse(p->right);
#pragma omp taskwait
 process(p);
}
```

What do you think this code does? How does the execution differ from the previous case?

Parallel tasks

OpenMP defines the concept of child task. A child task of a piece of code (region) is a task generated by a directive

```
#pragma omp task
```

found in that piece of code.

For example, in the previous code postorder_traverse(p->left) and postorder_traverse(p->right) are child tasks of the enclosing region.

taskwait specifies a wait on the completion of the child tasks of the current task (precisely the region the current task is executing).

Note that taskwait requires to wait for completion of the child tasks, but not completion of all descendant tasks (e.g., child tasks of child tasks).

Tree traversal with section

```
void traverse( struct node *p ) {
#pragma omp parallel sections
 {
#pragma omp section
   if (p->left)
     traverse(p->left);
#pragma omp section
   if (p->right)
     traverse(p->right);
 }
 process(p);
}
```

What do you think of this code does ?

Explanation

The problem with the previous code is that each thread entering one of the sections will call traverse, which leads to the creation of a new parallel region because of

```
#pragma omp parallel sections
```

The result is that this makes it more difficult in general to control the number of threads being generated by this implementation.

Common mistakes

Race condition

```
int nthreads;
#pragma omp parallel shared(nthreads)
{
  nthreads = omp_get_num_threads();
}
```

Common mistakes

Deadlock

```
#pragma omp parallel
{
 ...
#pragma omp critical
 {
  ...
#pragma omp barrier
 }
}
```

Common mistakes

Livelock

```
#pragma omp parallel
{
 flag[id] = true;
 while (flag[!id]){
  flag[id] = false;
  /*delay */;
  flag[id] = true;
 }
}
#pragma omp critical
flag[id] = false;
```

Not all computations are simple

Not all computations are simple loops where the data can be evenly divided among threads without any dependencies between threads

An example is finding the location and value of the largest element in an array

```
for (i=0; i<n; i++) {
 if (x[i] > maxval) {
  maxval = x[i];
  maxloc = i;
 }
}
```

Not all computations are simple, competing threads

All threads are potentially accessing and changing the same values, maxloc and maxval.

OpenMP provides several ways to coordinate access to shared values

```
#pragma omp atomic
```

Only one thread at a time can execute the following statement (not block). We can use the critical option

```
#pragma omp critical
```

Only one thread at a time can execute the following block atomic may be faster than critical but depends on hardware

How to find the max value using OpenMP

Write down the simplest algorithm and look carefully for race conditions. How would you handle them? The first step would be to parallelize as

```
#pragma omp parallel for
for (i=0; i<n; i++) {
 if (x[i] > maxval) {
  maxval = x[i];
  maxloc = i;
 }
}
```

Then deal with the race conditions

Write down the simplest algorithm and look carefully for race conditions. How would you handle them? The first step would be to parallelize as

```
#pragma omp parallel for
for (i=0; i<n; i++) {
#pragma omp critical
 if (x[i] > maxval) {
  maxval = x[i];
  maxloc = i;
 }
}
```

What can slow down OpenMP performance?

Performance poor because we insisted on keeping track of the maxval and location during the execution of the loop.

We do not care about the value during the execution of the loop, just the value at the end.

This is a common source of performance issues, namely the description of the method used to compute a value imposes additional, unnecessary requirements or properties

Idea: Have each thread find the maxloc in its own data, then combine and use temporary arrays indexed by thread number to hold the values found by each thread

Find the max location for each thread

```
int maxloc[MAX_THREADS], mloc;
double maxval[MAX_THREADS], mval;
#pragma omp parallel shared(maxval,maxloc)
{
 int id = omp_get_thread_num();
 maxval[id] = -1.0e30;
#pragma omp for
 for (int i=0; i<n; i++) {
  if (x[i] > maxval[id]) {
   maxloc[id] = i;
   maxval[id] = x[i];
  }
 }
}
```

Combine the values from each thread

```
#pragma omp flush (maxloc,maxval)
#pragma omp master
{
 int nt = omp_get_num_threads();
 mloc = maxloc[0];
 mval = maxval[0];
 for (int i=1; i<nt; i++) {
  if (maxval[i] > mval) {
   mval = maxval[i];
   mloc = maxloc[i];
  }
 }
}
```

Portable Sequential Equivalence

Portable Sequential Equivalence (PSE):

- when a program is sequentially equivalent if its results are the same with one thread or many threads
- For a program to be portable (runs the same on different platforms/compilers) it must execute identically when the OpenMP constructs are used or ignored

Strong SE: bitwise identical results

Weak SE: equivalent mathematically, not bitwise identical

Portable Sequential Equivalence

- Strong SE:
  - Locate all cases where a shared variable can be written by multiple threads
  - The access to the variable must be protected
  - If multiple threads combine results into a single value, enforce sequential order
- Weak SE:
  - Floating point arithmetic is not associative and not commutative
  - In most cases no particular grouping is mathematically preferred so why choose the sequential order?

Next. . .

Specific patterns

Some examples

Max subarray

```
Data: array
Result: maxSum, left, right, top, bottom
(maxSum, left, right, top, bottom) = ($-\infty$, 0, 0, 0, 0)
for $i \leftarrow 0$ to $n$ do
    for $j \leftarrow i$ to $n$ do
        temp(0:n-1) = 0
        for $k \leftarrow 0$ to $m$ do
            temp(k) += array(j,k)
        end
        sum = kadane(temp, start, finish)
        if $sum > maxSum$ then
            (maxSum, left, right, top, bottom) = (sum, i, j, start, finish)
        end
    end
end
```

Matrix-matrix multiplication

```
# include <cstdlib>
# include <cmath>
# include <ctime>
# include <iostream>
# include <omp.h>

using namespace std;

// Main function
int main ( )
{
  // brute force coding of arrays
  double a[500][500];
  double angle;
  double b[500][500];
  double c[500][500];
  int i;
  int j;
  int k;
```

Matrix-matrix multiplication

```
int n = 500;
double pi = acos(-1.0);
double s;
int thread_num;
double wtime;

cout << "\n";
cout << "  C++/OpenMP version\n";
cout << "  Compute matrix product C = A * B.\n";

thread_num = omp_get_max_threads ( );

//
//  Loop 1: Evaluate A.
//
s = 1.0 / sqrt ( ( double ) ( n ) );

wtime = omp_get_wtime ( );
```

Matrix-matrix multiplication

```
# pragma omp parallel shared (defualt) private ( angle, i, j, k )
{
# pragma omp for
 for ( i = 0; i < n; i++ )
 {
  for ( j = 0; j < n; j++ )
  {
   angle = 2.0 * pi * i * j / ( double ) n;
   a[i][j] = s * ( sin ( angle ) + cos ( angle ) );
  }
 }
 //
 //  Loop 2: Copy A into B.
 //
# pragma omp for
 for ( i = 0; i < n; i++ )
 {
  for ( j = 0; j < n; j++ )
  {
   b[i][j] = a[i][j];
  }
 }
```

Matrix-matrix multiplication

```
//  Loop 3: Compute C = A * B.
# pragma omp for
  for ( i = 0; i < n; i++ )
  {
   for ( j = 0; j < n; j++ )
   {
    c[i][j] = 0.0;
    for ( k = 0; k < n; k++ )
    {
     c[i][j] = c[i][j] + a[i][k] * b[k][j];
    }
   }
  }
 }
 wtime = omp_get_wtime ( ) - wtime;
 cout << "  Elapsed seconds = " << wtime << "\n";
 cout << "  C(100,100)  = " << c[99][99] << "\n";
 //
 //  Terminate.
 //
 cout << "\n";
 cout << "  Normal end of execution.\n";
 return 0;
}
```