CME213/ME339
Lecture 14

Eric Darve
Erich Elsen
Austin Gibbons

Department of Mechanical Engineering
Institute for Computational and Mathematical Engineering
Stanford University

Spring 2012

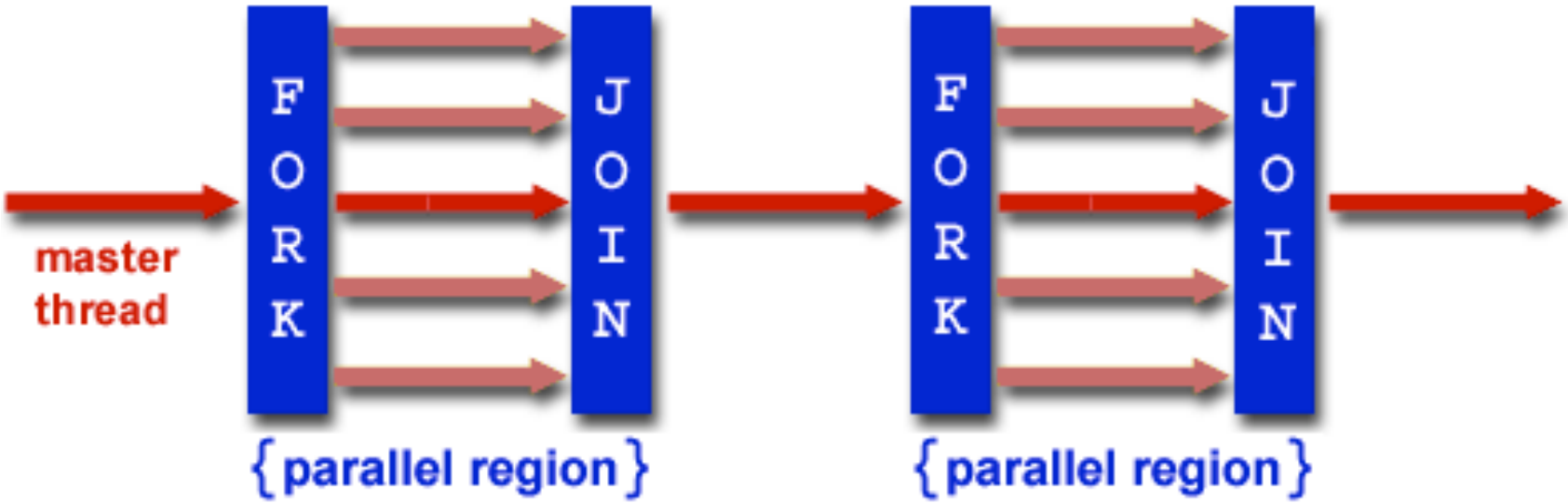
Pthread and OpenMP

- Pthread is a relatively low-level primitive.
- Anything can be done in Pthread and efficiently too. But at the same time, even the simplest calculation is tedious to set up.
- Just think that the only input to your function should be `void *`.
- OpenMP is a directive-based language: helps rid the programmer of the mechanics of manipulating threads.
- OpenMP works in FORTRAN, C, C++. We will focus on C++.

Overview of OpenMP

- OpenMP is based on the fork/join programming model.
- An executing OpenMP program starts as a single thread.
- At points in the program where parallel execution is desired, the program forks additional threads to form a team of threads.
- The threads execute in parallel across a region of code called a parallel region.
- At the end of the parallel region, the threads wait until the full team arrives, and then they join back together.
- At that point, the original or master thread continues until the next parallel region.

Fork/join


master
thread
FORK
JOIN
{parallel region}
FORK
JOIN
{parallel region}

Design concepts behind OpenMP

- OpenMP was designed around two concepts: sequential equivalence and incremental parallelism.
- Sequential equivalence: the code will execute correctly if one or multiple threads are present.
- Incremental parallelism: it is possible to evolve a sequential program into a parallel program in small chunks.
- *These are only design guidelines. They are not always satisfied in practice.*

A simple OpenMP program

```
1  #include <stdio.h>
2  #include "omp.h"
3  int main() {
4  #pragma omp parallel
5  {
6      printf("It wasn't until I could get out of Stanford");
7      printf(" that I could sit down and think about my life.\n");
8  }
9  }
```

- This will print out `It wasn't...` once for each thread.
- A structured block defines the beginning and end of the parallel section: block of one or more statements with one point of entry at the top and one point of exit at the bottom.

Shared and private variables

- The basic assumption is as follows: variables allocated before the parallel section are shared, while variables defined inside the parallel section are private to each thread.

```
1  int main() {
2  int i = 5; // Private variable
3  #pragma omp parallel
4  {
5      int c; // a variable local or private to each thread
6      c = omp_get_thread_num();
7      printf("c = %d, i = %d\n",c,i);
8  }}
```

```
1  c = 0, i = 5
2  c = 2, i = 5
3  c = 1, i = 5
```

Defining the number of threads to launch

- In the previous examples, the runtime system is allowed to select the number of threads.
- It is possible to change the operating system's default number of threads to use with OpenMP applications by setting the `OMP_NUM_THREADS` environment variable:
  `setenv OMP_NUM_THREADS 3`
- It can also be set inside the program:

```
1  #pragma omp parallel num_threads(3)
```

Worksharing

- In scientific computing most of the work happens inside of loops.
- OpenMP provides a natural mechanism to split the work among the different threads.
- It's really easy!

```
1  int main() {
2      int i;
3      double answer, res;
4      answer = 0.0;
5      for (i=0;i<N;i++) {
6          res = big_comp(i);
7          combine(answer,res);
8      }
9      printf("The answer to my complicated problem is %f\n",
10             answer);
11 }
```

Bad code

- Assume in the previous example that most of the work happens in `res = big_comp(i)`.
- In the previous example, we are stuck because `combine(answer,res)` makes the calculation completely sequential.
- We cannot proceed until the previous iteration is complete.
- This calculation cannot be parallelized.

Good code

```
1  int main() {
2      int i;
3      double res[N];
4      for (i=0;i<N;i++)
5          res[i] = big_comp(i);
6
7      double answer = 0.0;
8      for (i=0;i<N;i++)
9          combine(answer,res[i]);
10
11     printf("The answer to my complicated problem is %f\n",
12             answer);
13 }
```

This can be parallelized efficiently if `combine` is quick compared to `big_comp`.

Parallel code

As easy as:

```
1  int main() {
2      int i;
3      double answer, res[N];
4      answer = 0.0;
5      #pragma omp parallel
6      {
7          # pragma omp for
8          for (i=0;i<N;i++) {
9              res[i] = big_comp(i);
10         }
11     }
12     for (i=0;i<N;i++)
13         combine(answer,res[i]);
14
15     printf("The answer to my complicated problem is %f\n",
16            answer);
17 }
```

Parallel for loop

- The dummy loop index is always private.
- There is a shortcut notation:

```
1   #pragma omp parallel
2   {
3       # pragma omp for
4       for (i=0;i<N;i++) {
5           res[i] = big_comp(i);
6       }
7   }
8
9   #pragma omp parallel for
10      for (i=0;i<N;i++) {
11          res[i] = big_comp(i);
12      }
13  }
```

Data environment clause

- One of the tricky concept in OpenMP is understanding whether a variable should be shared or private.
- Clauses are used to make changes to the default choice.
- `default(shared)` or `default(none)`: make all variables defined prior to the parallel region either shared or private (each thread then has a local copy).
- `private` or `shared` can be used to specify a list of private or shared variables.
- `firstprivate`: private variable initialized to the corresponding value before the parallel directive.
- `reduction`: creates a private variable initialized to 0 — in general the identity element. At the end of the parallel construct, all the values are combined to define a single value.

Trapezoid rule of integration WRONG

```
1  int main() {
2      int i;
3      int num_steps = 1000000;
4      double x, pi, step, sum = 0.0;
5      step = 1.0/(double) num_steps;
6
7      #pragma omp parallel for
8      for (i=0; i < num_steps; ++i) {
9          x = (i+0.5)*step;
10         sum += 4.0/(1.0+x*x);
11     }
12     pi = step*num;
13     printf("Pi estimated using the trapezoid method %lf\n",pi);
14 }
```

Bug!

1. `x` is shared. Result of calculation is undefined.

Bug!

1. `x` is shared. Result of calculation is undefined.
2. `sum` is shared. This is a more subtle point but += also leads to an error.

Shared variables

- In most cases, we can use the `reduction` clause to clear up the problem.
- Problem solved! However, what is really going on here?
- The OpenMP API provides a relaxed-consistency, shared-memory model.
- Each thread is allowed to have its own *temporary* view of the memory. This corresponds to the fact that the hardware relies on many optimizations and can store a variable in many different memories: machine registers, cache, or other local storage, between the thread and the memory.
- This temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable.

- Consequence: it is not safe in general to have reads and writes to a shared variable. A synchronization point is required for correcteness. [More on this later.]
- In our previous example, we can fix this by using a `reduction` clause.

Trapezoid rule of integration RIGHT

```
1  int main() {
2      int i;
3      int num_steps = 1000000;
4      double x, pi, step, sum = 0.0;
5      step = 1.0/(double) num_steps;
6
7      #pragma omp parallel for private(x) reduction(+:sum)
8      for (i=0; i < num_steps; ++i) {
9          x = (i+0.5)*step;
10         sum += 4.0/(1.0+x*x);
11     }
12     pi = step*num;
13     printf("Pi estimated using the trapezoid method %lf\n",pi);
14 }
```

Tasks

- The other canonical way to run parallel code is to define independent tasks that will be executed by different threads.
- OpenMP provides two mechanisms: `sections` and `task`.
- Option 1: sections.

Sections

- The `sections` construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.
- Assume we have three tasks that can be executed concurrently, `taskA()`, `taskB()`, `taskC()`.

```
1  #pragma omp parallel
2  {
3      #pragma omp sections
4      {
5          #pragma omp section
6          {
7              taskA();
8          }
9          #pragma omp section
10         {
11             taskB();
12         }
13         #pragma omp section
14         {
15             taskC();
16         }
17     }
18 }
```

Task

- 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.

Tree traversal

```
1  struct node {
2    struct node *left, *right;
3  };
4  void traverse( struct node *p ) {
5    if (p->left)
6  #pragma omp task   // p is firstprivate by default
7        traverse(p->left);
8    if (p->right)
9  #pragma omp task   // p is firstprivate by default
10       traverse(p->right);
11   process(p);
12 }
13 int main() {
14     node *root = ...;
15 #pragma omp parallel
16     traverse(root);
17     // Probably not correct because all threads
18     // traverse the tree.
19 }
```

Tree traversal

```
1  ...
2  int main() {
3      node *root = ...;
4  #pragma omp parallel
5  #pragma omp single
6      traverse(root);
7      // OK!
8  }
```

Only one thread executes the code and forks other threads as it encounters `task`.

Post-ordering

- As specified, the order in which the traversal/`process` is occurring is basically unspecified.
- We need to add synchronization directives to get a specific ordering.
- Example: post-ordering; "traverse first" then process.

Tree traversal

```
1  void postorder_traverse( struct node *p ) {
2      if (p->left)
3          #pragma omp task   // p is firstprivate by default
4              postorder_traverse(p->left);
5      if (p->right)
6          #pragma omp task   // p is firstprivate by default
7              postorder_traverse(p->right);
8      #pragma omp taskwait
9      process(p);
10 }
```

Beware of scoping rules

`task` uses certain rules to guess the likely scope of variables (shared/private, it's called auto-scoping).

This is too complicated to cover in details here. Variables are in many cases `firstprivate`. In doubt specify explicitly the scope.

```
1  int fib(int n) {
2      int i, j;
3      if (n<2)
4          return n;
5      else {
6      #pragma omp task shared(i)
7          i = fib(n-1);
8      #pragma omp task shared(j)
9          j = fib(n-2);
10     #pragma omp taskwait
11
12         return i+j;
13 }}
```

Difference between task and sections

- OpenMP 3.0 tasking is more flexible and efficient compared to sections.
- The task directive can take an `if` clause to cause the task to be executed immediately or be deferred (that is, another thread will execute the task in parallel).
- Tasking has much better performance and scalability for nested parallel and recursive algorithms, compared to parallel sections.
- You can control the total number of threads (`OMP_NUM_THREADS`) with tasking, whereas with sections, with each newly created region you get `OMP_NUM_THREADS` new threads, which can oversubscribe the host.

Synchronization: flush

- A tricky concept.
- As explained previously, threads keep a "local" copy or view of variables. This is important for optimization.
- There are cases however where threads need to write to and read from shared variables.
- In that case a flush is required to make sure the code executes correctly.
- `flush` makes a threads temporary view of memory consistent with memory:
  1. if a thread modifies a shared variable, a flush (after) will result in writing to memory
  2. if a thread reads a variable, a flush (before) will make sure that the value is up to date

Execution of a flush region affects the memory and the temporary view of memory of only the thread that executes the region.

It does not affect the temporary view of other threads.

Other threads must themselves execute a flush operation in order to be guaranteed to observe the effects of the encountering threads flush operation.

```
1 #pragma omp flush [(list)]
```

Critical section

- Only one thread may enter that section.
- This can be very useful before a `printf` or `cout` otherwise the printouts from different threads can become interleaved.
- An optional name can be given: in that case, threads are restricted from entering any critical section with the same name.

```
1  #pragma omp critical [(name)]
```

Example

```
1  int main() {
2      int i;
3      double answer, res;
4      answer = 0.0;
5      #pragma omp parallel for private (res)
6      for (i=0;i<N;i++) {
7          res = big_comp(i);
8          # pragma omp critical
9          combine(answer,res);
10     }
11     printf("The answer to my complicated problem is %f\n",
12            answer);
13 }
```

Barriers

Explicit synchronization is possible using a barrier.

```
1  #pragma omp barrier
```

A barrier implies a flush as well.

Locks

For most programmers, these "high-level" constructs are sufficient.

In some cases a finer level of control over synchronization is required or the overhead of using some of the previous constructs may be too large.

Low-level synchronization functions based on locks are also provided by the language.

Schedule clause

This gives some measure of control over the scheduling of loop iterations onto threads.

```
1  #pragma omp for schedule(static [,chunk])
```

If `chunk` is specified, the loop is broken into blocks of size `chunk`. Blocks are assigned to threads in a round-robin fashion.

If `chunk` is not specified, the loop is broken into blocks of approximately equal sizes and each thread is assigned a single block.

Schedule: dynamic

```
1  #pragma omp for schedule(dynamic [,chunk])
```

If `chunk` is specified, the loop is broken into blocks of size `chunk`. Blocks are assigned to threads dynamically, that is "on demand."

This is useful when blocks take a varying amount of computational time. The overhead of this construct is greater.

If `chunk` is not specified, the block size is set to 1.

Variants

```
1 #pragma omp for schedule(guided [,chunk])
```

This is a variation on dynamic where large block sizes are chosen first and then progressively reduced. This allows reducing the cost of scheduling.

```
1 #pragma omp for schedule(runtime)
```

The schedule is specified at run time by `OMP_SCHEDULE`.

Matrix-matrix product

```
1  int chunk = 10;
2  #pragma omp parallel num_threads(8) private(tid,i,j,k)
3  {
4
5    tid = omp_get_thread_num();
6
7    /*** Do matrix multiply sharing iterations on outer loop ***/
8    /*** Display who does which iterations ***/
9    #pragma omp critical
10   printf("Thread %d starting matrix multiply...\n",tid);
11   #pragma omp for schedule (static, chunk)
12   for (i=0; i<NRA; i++) {
13     #pragma omp critical
14     printf("Thread=%d did row=%d\n",tid,i);
15     for (j=0; j<NCB; j++)
16         for (k=0; k<NCA; k++)
17             c[i][j] += a[i][k] * b[k][j];
18     }
19 }
```

Runtime library

- `omp_set_num_threads()`: requests that the OS provide that number of threads in subsequent parallel regions.
- `omp_get_num_threads()`: number of threads in the current team of threads.
- `omp_get_thread_num()`: ID of thread
- `omp_get_num_procs()`: number of processors available to execute the threaded program.