Parallel Programming with OpenMP

Module Outline

- Refresher on OpenMP
  - Model of parallelism
  - Components
  - Scope of variables
  - Synchronization
- OpenMP 3.0's Tasking

Refresher on OpenMP

- Most popular shared memory programming standard
  - – Backed by industry consortium
  - – Open, not proprietary
  - – Supported by most compilers, including GNU (starting from gcc version 4.2)
  - – Still evolving (version 3.1 as of July 2011)
- Consists of directives, run-time system, and libraries

Refresher on OpenMP

- Initially, designed for expressing loop-level parallelism (I.e. parallelism between loop iterations)

Sequential Program

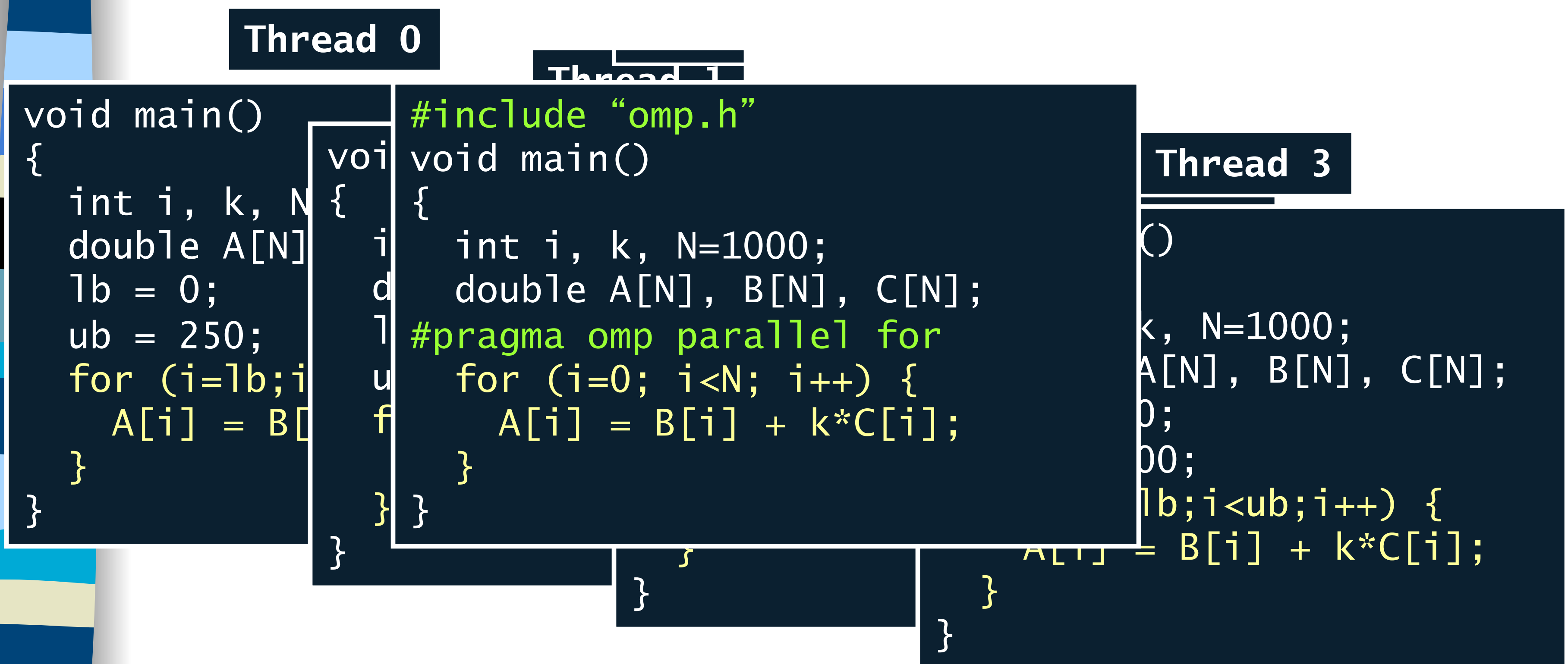
```
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i]
  }
}
```

Parallel Program

```
#include "omp.h"
void main()
{
  int i, k, N=1000;
  double A[N], B[N], C[N];
#pragma omp parallel for
  for (i=0; i<N; i++) {
    A[i] = B[i] + k*C[i];
  }
}
```

During Execution

- Single Program Multiple Data (SPMD)
  - The same code applied to different data



OpenMP Fork-and-Join model

```c
printf("program begin\n");
N = 1000;
```

Serial

```c
#pragma omp parallel for
for (i=0; i<N; i++)
    A[i] = B[i] + C[i];
```

Parallel

```c
M = 500;
```

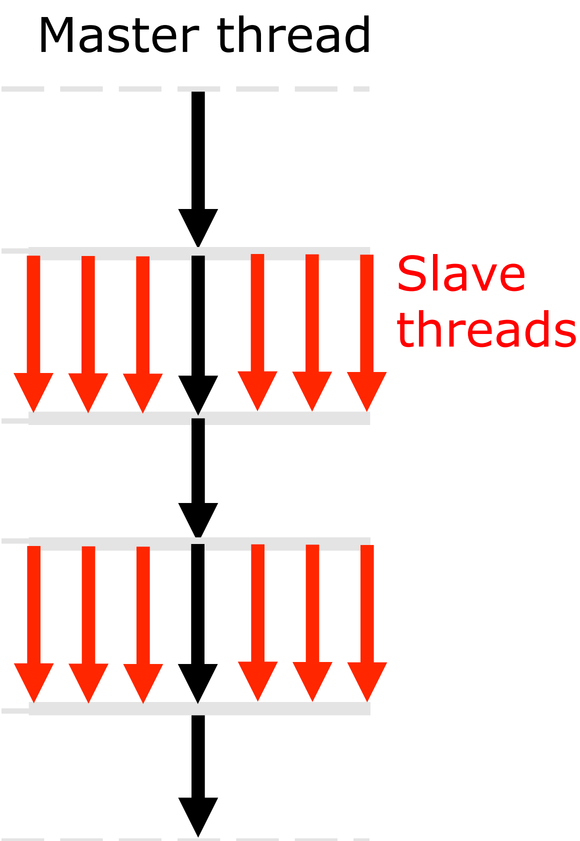
Serial

```c
#pragma omp parallel for
for (j=0; j<M; j++)
    p[j] = q[j] - r[j];
```

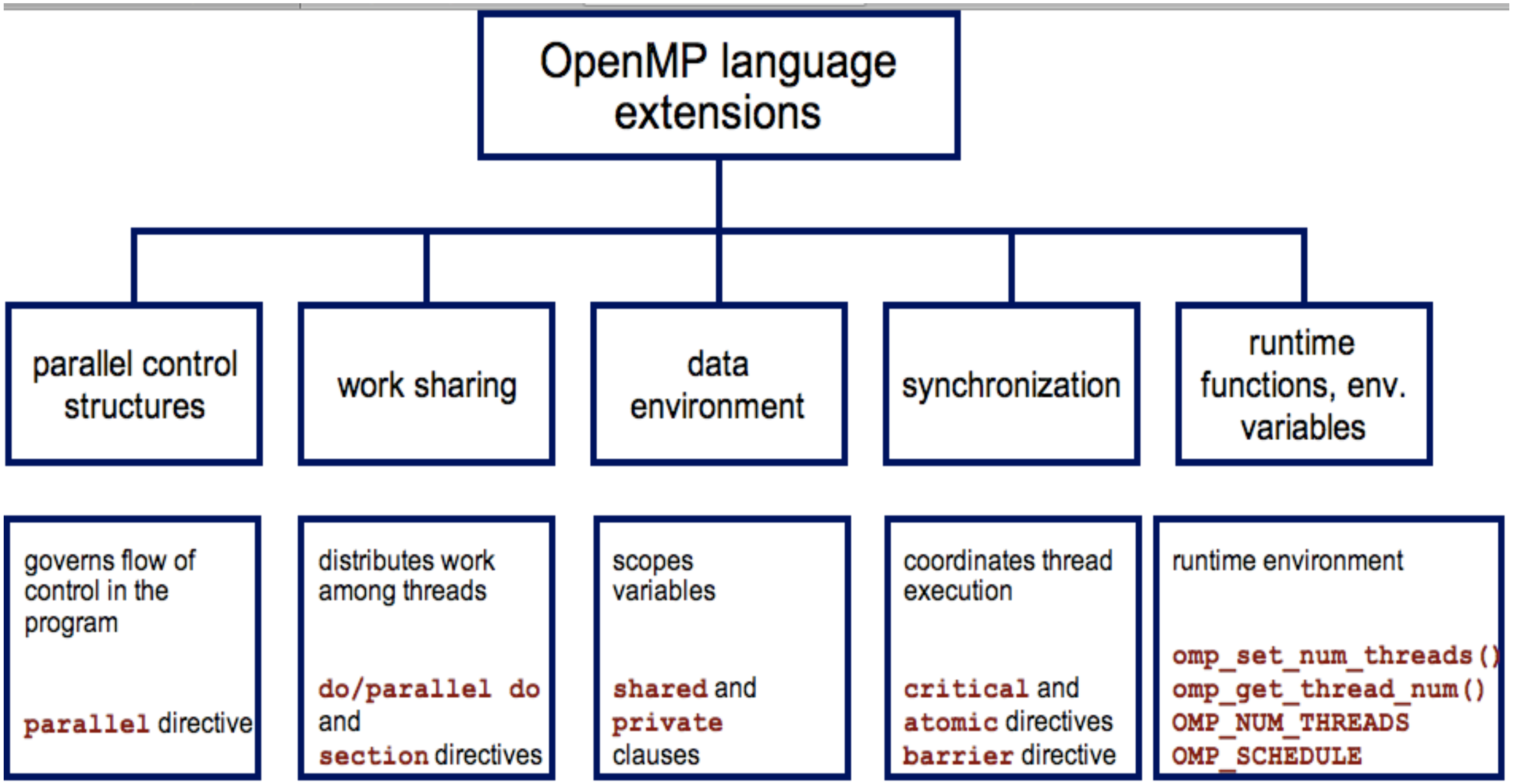
Parallel

```c
printf("program done\n");
```

Serial



OpenMP's Components



Directives format

Refer to http://www.openmp.org and the OpenMP 2.0 specifications in the course web site for more details

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

For example,

```
#pragma omp for [clause[[,] clause] ... ] new-line
for-loop
```

The clause is one of

- **private**(*variable-list*)
- **firstprivate**(*variable-list*)
- **lastprivate**(*variable-list*)
- **reduction**(*operator*: *variable-list*)
- **ordered**
- **schedule**(*kind*[, *chunk_size*])
- **nowait**

Parallel for loop

- Parallel for loop:

Except here.
Iterations divided
over multiple threads

Threads spawned here

Code executed by
all threads

Threads joined here

```
#include <omp.h>
//…
#pragma omp parallel
{
  …
   #pragma omp for private(i)
   for(i=0; i<n; i++)
         A[I]= A[I]*A[I]- 3.0;
}
```

- OpenMP allows us to express both the parallel region and work sharing of loop iterations by:

Threads spawned here
and iterations divided
over multiple threads

```
#pragma omp parallel for private(i)
   for(i=0; i<n; i++)
         A[I]= A[I]*A[I]- 3.0;
```

Parallel Section

- Parallel sections:

Threads spawned here

Each section executed by one thread

```
#pragma omp parallel shared(A,B) private(i)
{
 #pragma omp sections
 {
    #pragma omp section
    for(i=0; i<n; i++)
         A[i]= A[i]*A[i]- 4.0;
    #pragma omp section
    for(i=0; i<n; i++)
         B[i]= B[i]*B[i] + 9.0;
 } // end omp sections
} // end omp parallel
```

Type of variables

- Most important: shared, private, reduction, firstprivate, lastprivate
- Semi-private data for parallel loops:
  - ***reduction***: variable that is the target of a reduction operation performed by the loop, e.g., sum
  - ***firstprivate***: initialize the private copy from the value of the shared variable prior to parallel section
  - ***lastprivate***: upon loop exit, master thread holds the value seen by the thread assigned the last loop iteration (for parallel loops only)

Synchronization Constructs in OpenMP

- To enclose code in critical section, use #pragma omp critical

```
#include <omp.h>
//…
#pragma omp parallel
{
 …
 #pragma omp parallel for private(i) {
   for(i=0; i<n; i++) {
     #pragma omp critical {
       sum = sum + A[I];
     }
   }
 }
}
```

- To enclose code that is to be executed by just one thread (the master thread), use #pragma omp master

Barriers

- Barriers are implicit after each parallel section
- When barriers are not needed for correctness, use **`nowait`** clause
- schedule clause will be discussed later

```
#include <omp.h>
//…
#pragma omp parallel
{
 …
 #pragma omp parallel for nowait private(i)
   for(i=0; i<n; i++)
         A[I]= A[I]*A[I]- 3.0;
}
```

Other Synchronization Constructs

- #pragma omp atomic ensures critical section, but:
  - has lower overheads (implemented directly with machine instructions)
  - restricted to simple ops: adding/subtracting
    - e.g. #pragma_omp_atomic
    - x = x+1;
- Named locks
  - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock(), omp_destroy_lock()
  - Use when you use multiple locks (fine-grain lock programming)

Declaring Variables

- Principle 1: declare variables as shared as much as possible
  - Read-only variables
  - A variable (or an element of a matrix) is only written and read by one thread
- Principle 2: variables that may be overwritten by another thread should be declared as private
  - Declaring a variable private creates replicas
    - e.g. private(X)
      - X (the original copy)
      - X_private[0] (replica for thread 0)
      - X_private[1] (replica for thread 1)
      - …

Declaring Variables

- Principle 3: If a variable will aggregate values from all elements of an array or matrix, we can perform reduction
  - e.g. reduction(sum)

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; i++) {
  sum = sum + a[i];
} // end omp parallel
```

  - is equivalent to:

```
sum = mysum = 0;
#pragma omp parallel firstprivate(mysum) {
  #pragma omp for
    for (i=0; i<N; i++)
      mysum = mysum + a[i];
  #pragma omp critical {
    sum = sum + mysum
  }
} // end omp parallel
```

Restrictions on Reduction

- reduction(op: var1, var2, …)
  - op can only be one of
    - arithmetic: +, *
    - logical: &, |, &&, ||, ^
    - mathematical: min, max (only in Fortran)

Declaring Variables

- Principle 4: some variables are treated as private by default
  - Variables declared inside the parallel region
  - Stack and local variables of a function
  - Function arguments are not private! (unless declared inside the parallel region)

Specifying Number of Threads

- Method 1: using environment variable, e.g.
  - setenv OMP_NUM_THREADS 2
  - a.out
  - In this case, OpenMP library spawns 2 threads to execute all parallel sections
- Method 2: hardwire it in the code

```
#include <omp.h>
…
omp_set_num_threads(4);
#pragma omp parallel  // 4 threads will run
{
  // do work here
}

omp_set_num_threads(omp_num_procs());
#pragma omp parallel  // as many threads as CPUs
{
  // do work here
}
```

Lab Assignment: Reduction

- Step 1: Go to Lab/Reduction
- Step 2: Parallelize the loop that reduces to sum
  - put #pragma omp parallel for reduction(+:sum)
- Step 3: Parallelize the loop that reduces to amax
  - declare "double my_amax;"
  - put #pragma omp parallel for private(my_amax) {…}
  - Replace "amax = abs_max(amax,…)" with "my_amax = abs_max(my_amax, …)". This causes each thread to accumulate its own amax value. We still need to reduce this at the end.
  - Add #pragma omp critical {if (fabs(my_amax) > fabs(amax)) amax = my_amax;} before exiting the parallel section
- Step 4: Compile with gcc -fopenmp flag.
- Step 5: Experiment with different number of threads
  - setenv OMP_NUM_THREADS x

OpenMP 3.0

- OpenMP version 3.0 added tasking ability (2008)
  - useful for irregular parallelism that is not loop-based and not easy to specify using sections
- **#pragma omp task [***clause***[[,]***clause***] ...]** *structured-block*
- clause is one of
  - **if (***expression***)**
  - **untied**
  - **shared (***list***)**
  - **private (***list***)**
  - **firstprivate (***list***)**
  - **default(shared | none)**

Example of Tasking

- Suppose we are traversing a linked list and spawns a thread to process each node
  - we can let the master thread spawns a thread to process each node it visits

```
#pragma omp parallel {
  #pragma omp single {
    p = head;
    while (p != NULL) {
      #pragma omp task firstprivate(p) {
         process (p);
      }
      p = p->next; ;
    } // end while
  }
}
```

  - must be careful to ensure task size > thread creation and management overheads