Lecture #10 - OpenMP

AMath 483/583

Announcements

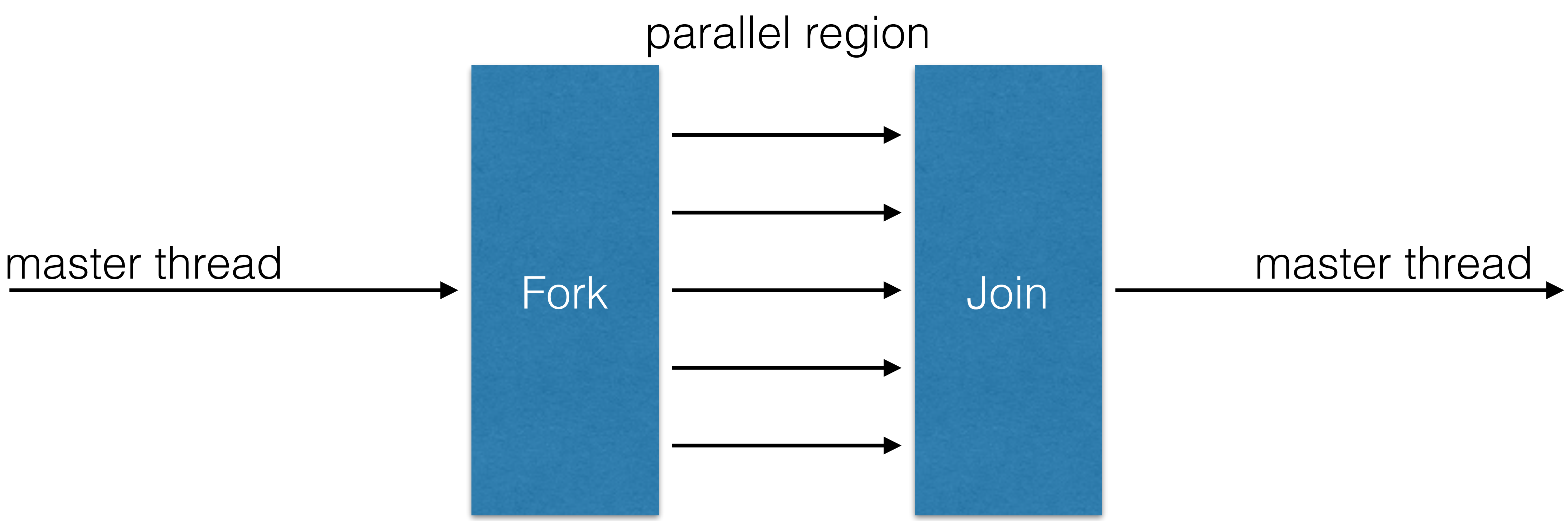
- Remember: Homework #2 due next Friday, not Monday
- References and code will be available later today

Note

- SMC currently will only give you two cores
- Four later
- Today’s demo is on my four-core machine
  - warning default compiler on OS X is “clang”
  - “gcc-5”

OpenMP - Basic Idea

- “Fork - Join” model
  - begin with single thread (master thread)
  - FORK: master thread creates team of parallel threads
  - JOIN: when threads complete action they synchronize and terminate



OpenMP Compiler Directives

```
#pragma omp [directive] [clause ...]
                        if (scalar_expression)
                        private (list)
                        shared (list)
                        default (shared || none)
                        firstprivate (list)
                        reduction (operator: list)
                        copyin (list)
                        num_threads (integer-expression)
```

OpenMP Compiler Directives

- Examples

```
#pragma omp parallel [clause]
{
  // block of parallel code
}

#pragma omp parallel for [clause]
for (…)
{
    // for loop body
}

#pragma omp barrier
// wait for all threads to arrive here before proceeding
```

Hello World

- Each thread prints “hello, world”

```
#include “omp.h”
int main()
{
  #pragma omp parallel
  {
    int id = omp_get_thread_num();
    printf(“Hello(%d),”, id);
    printf(“world(%d)”, id);
  }
}
```

OpenMP header file

Parallel region with default number of threads

Runtime library function to get current thread #

End of parallel region

Hello World

- Each thread prints "hello, world"

```
#inc
int
{
   #p
   {
      int id = omp_get_thread_num();
      printf("Hello(%d),", id);
      printf("world(%d)", id);
   }
}
```

Compile Using

```
$ gcc -fopenmp hello_openmp.c -o hello
```

rary function to
ent thread #

End of parallel region

Setting Thread Numbers

- Statically (at compile-time)

```
#pragma omp parallel thread_num(4)
{
  // parallel code
}
```

- Dynamically (at run-time)

```
omp_set_num_threads(4);  // a run-time function
#pragma omp parallel
{
  // parallel code
}
```

Demo

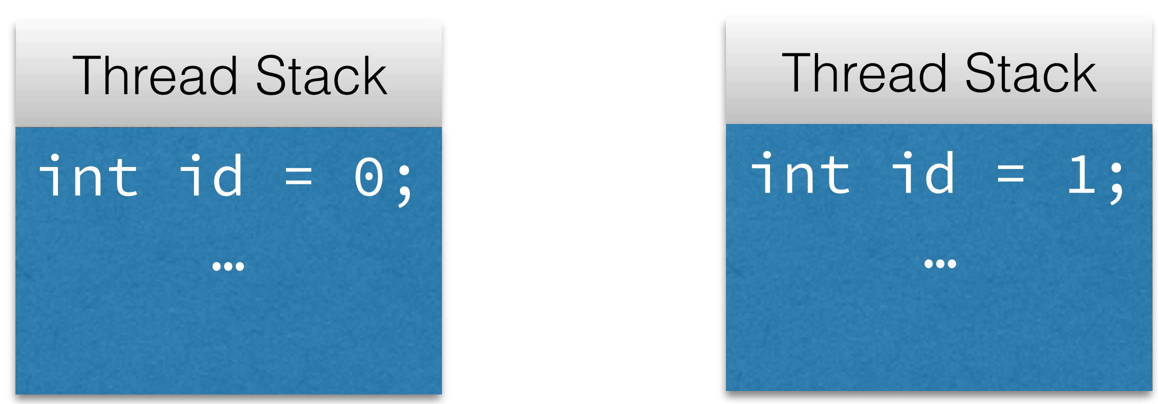
Setting run-time threads via command line:

`dynamic_threads.c`

Private Variables

- Variables declared *inside* a parallel region are private to each thread

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  ...
}
```

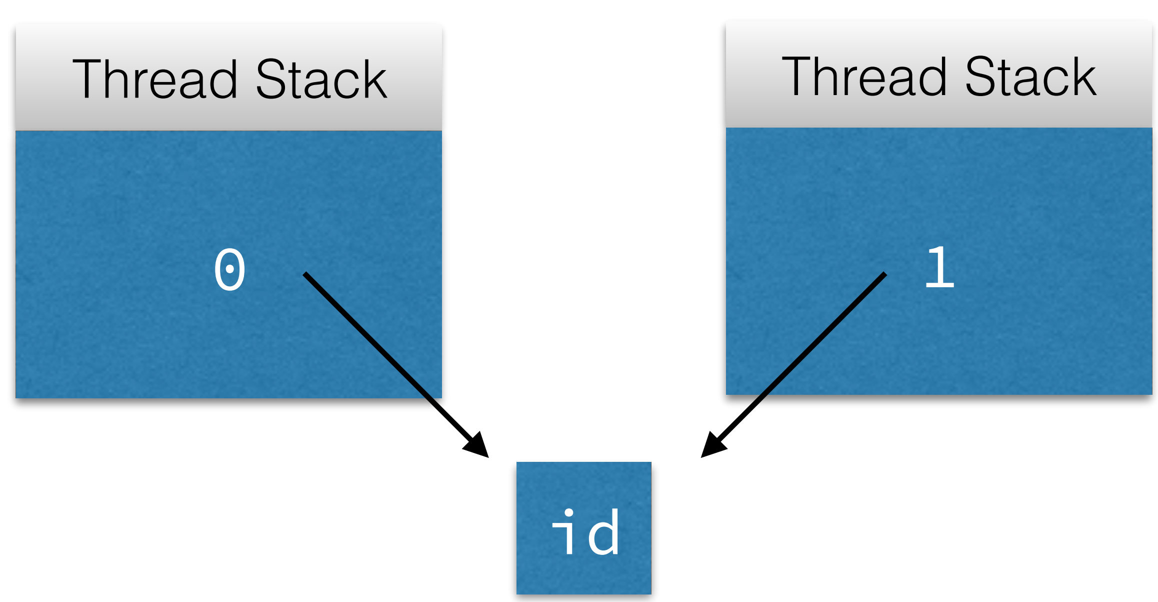


Private Variables

- Variables declared *<u>outside</u>* a parallel region are shared / accessible by each thread

```
int id;
pragma omp parallel
{
  id = omp_get_thread_num();
  ...
}
```

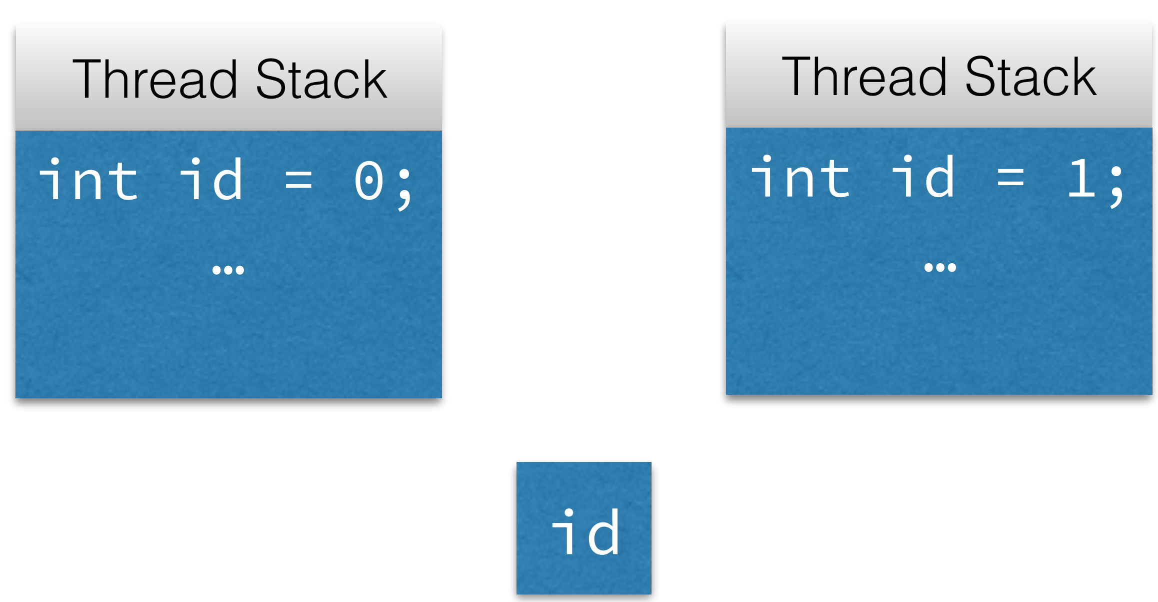
Danger! Multiple threads are writing to same location.



Private Variables

- **Solution**: variables can be declared "`private`" —> each thread will have a local copy

```
int id;
pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  ...
}
```



Demo

Private variables

Loops

- Share iterations of a loop across threads
    - useful in “data parallel” situations

```
int i;
for (i=0; i<10; ++i)
  // loop body
```

Loops

- **Default**: each thread gets one iteration
  get next available iteration when complete

```
int i;
#pragma omp parallel private(i) num_threads(4)
{
  #pragma omp for
  for (i=0; i<10; ++i)
  {
    // loop body
  }
}
```

Loops

- **Default**: each thread gets one iteration
  get next available iteration when complete

```
int i;
#pragma omp parallel private(i) num_threads(4)
{
  #pragma omp for
  for (i=0; i<10; ++i)
  {
    // loop body
  }
}
```

Declare parallel region

Loops

- **Default**: each thread gets one iteration
  get next available iteration when complete

```
int i;
#pragma omp parallel private(i) num_threads(4)
{
  #pragma omp for
  for (i=0; i<10; ++i)
  {
    // loop body
  }
}
```

Tell threads to split up work in `for` loop

Loops

- **Default**: each thread gets one iteration
  get next available iteration when complete

```
int i;
#pragma omp parallel private(i) num_threads(4)
{
  for (i=0; i<10; ++i)
  {
    // loop body
  }
}
```

WARNING

If "`#pragma omp for`" is <u>omitted</u> then all threads do same for loop

Loops

- **Example**: `vec_add`

```
void vec_add(double* out, double* v, double* w, double* N)
{
  #pragma omp parallel shared(out,v,w) num_threads(4)
  {
    int i;  // implicitly private
    #pragma omp for
    for (i=0; i<N; ++i)
        out[i] = v[i] + w[i]
  }
}
```

Although implicit, good idea to explicitly declare “shared” for readability.

Demo

Parallel `vec_add` and timing

Shortcut

- Combine parallel directives and for directives

```
#pragma omp parallel for [num_threads(N)]
for (. . .)
  { . . . }
```

- Convenience function.

Reminder: Private Variables

- Consider the following:

```
double x, dx = 1.0 / (N + 1.0);


for (int i=0; i<N; ++i)
{
  x = i*dx;
  y[i] = sin(x) * cos(x);
}
```

Reminder: Private Variables

- **Incorrect**:

```
double x, dx = 1.0 / (N + 1.0);

#pragma omp parallel for
for (int i=0; i<N; ++i)
{
  x = i*dx;
  y[i] = sin(x) * cos(x);
}
```

Reminder: Private Variables

- **Incorrect**:

```
double x, dx = 1.0 / (N + 1.0);

#pragma omp parallel for
for (int i=0; i<N; ++i)
{
  x = i*dx;
  y[i] = sin(x) * cos(x);
}
```

By default, x is a shared variable.

Thread 0: set x
Thread 1: set different x
Thread 0: evaluate y[i]

Reminder: Private Variables

- **Incorrect**:

```
double x, dx = 1.0 / (N + 1.0);

#pragma omp parallel for private(x)
for (int i=0; i<N; ++i)
{
  x = i*dx;
  y[i] = sin(x) * cos(x);
}
```

Now each thread computes own copy of x

Loop Chunking

- `vec_add` — work done by each thread is small

Thread 0	`out[0] = v[0] + w[0]; out[5] = v[5] + w[5]; ...`
Thread 1	`out[2] = v[2] + w[2]; out[4] = v[4] + w[4]; ...`
...	

- Thread scheduler works hard at finding next iteration for each thread

Loop Chunking

- Instead, give each thread a “chunk” of iterations

Thread 0	`do iterations i=0 through i=(M-1)`
Thread 1	`do iterations i=M through i=(2*M-1)`
...	

- In some instances this is more efficient
- *Static* vs. *dynamic* chunking - assign at start or as you go?

Loop Chunking

- **schedule:** declare chunking behavior

```
void vec_add(double* out, double* v, double* w, double* N)
{

  #pragma omp parallel shared(out,v,w) num_threads(4)
  {
    int i;
    #pragma omp for
    for (i=0; i<N; ++i)
        out[i] = v[i] + w[i]
  }
}
```

Loop Chunking

- **schedule:** declare chunking behavior

```
void vec_add(double* out, double* v, double* w, double* N)
{

  #pragma omp parallel shared(out,v,w) num_threads(4)
  {
    int i;
    #pragma omp for schedule(static,128)
    for (i=0; i<N; ++i)
        out[i] = v[i] + w[i]
  }
}
```

Set chunk size to 128 and use static scheduling

(each thread gets 128 iterations of the loop at a time)

Loop Chunking

- **schedule:** declare c

Chunk size can be stored in
(shared) variable.
(Each thread needs to know.)

```
void vec_add(double* out,                          uble* N)
{
  int chunk_size = 128;
  #pragma omp parallel shared(out,v,w,chunk_size) num_threads(4)
  {
    int i;
    #pragma omp for schedule(static,chunk_size)
    for (i=0; i<N; ++i)
        out[i] = v[i] + w[i]
  }
}
```

Chunking Strategies

- `schedule(static [,chunk])` — deal out blocks of iterations of size "chunk" to each thread
- `schedule(dynamic [,chunk])` — each thread grabs "chunk" iterations off of queue until all iterations have been handled
- `schedule(guided [,chunk])` — threads dynamically grab blocks of iterations. Block size starts large and shrinks down to "chunk" size
- `schedule(auto)` — runtime can "learn" from previous executions of same loop

Demo

Chunking and performance tuning

Synchronization

- Impose order constraints and protect access to shared data
- Case study:

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  A[id] = // some big calculation


  // use A in some function to compute Bi
  B[id] = func(A, id);
}
```

Synchronization

- Impose order constraints and protect access to shared data

- Case study:

```
#pragma omp parallel
{
  int id = omp_get_thread_n
  A[id] = // some big calcu


  // use A in some functio
  B[id] = func(A, id);
}
```

Problem!

A is not necessarily fully formed at this point!

(Need to wait for all threads)

Synchronization - Barrier

- Impose order constraints and protect access to shared data
- Case study:

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  A[id] = // some big calculation

  #pragma omp barrier
  // use A in some function to compute Bi
  B[id] = func(A, id);
}
```

Threads wait here until all threads arrive

Synchronization - Barrier

- Some OpenMP directives have natural barriers

```
#pragma omp for
for (. . .)
  { . . . }
// all threads synchronize at end of loop
// before proceeding

#pragma omp for nowait
for (. . .)
  { . . . }
// thread i will not wait for thread j to
// finish at last iterations of loop
```

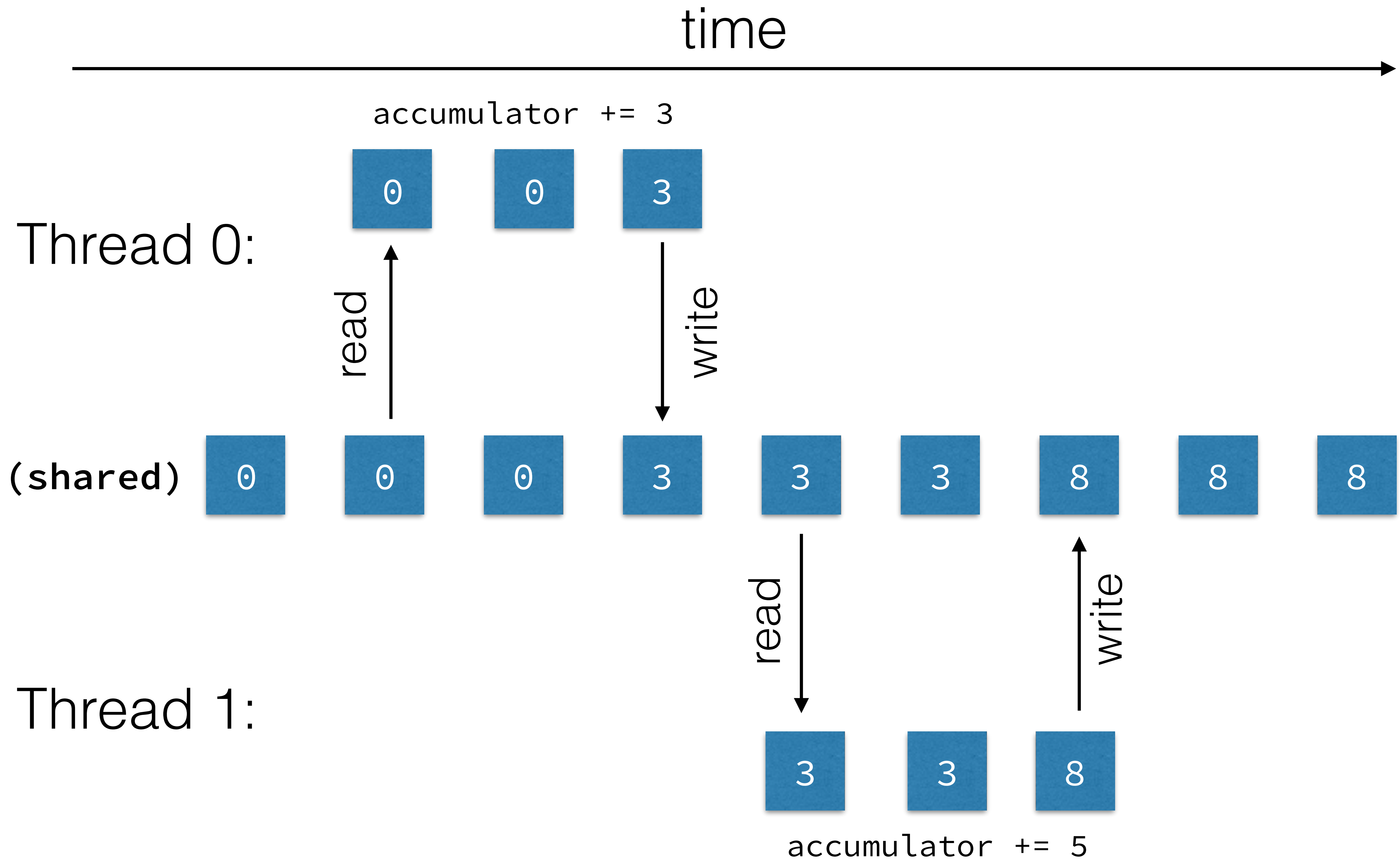
Synchronization - Critical

- Mutual exclusion (mutex) — only one thread at a time can enter critical region

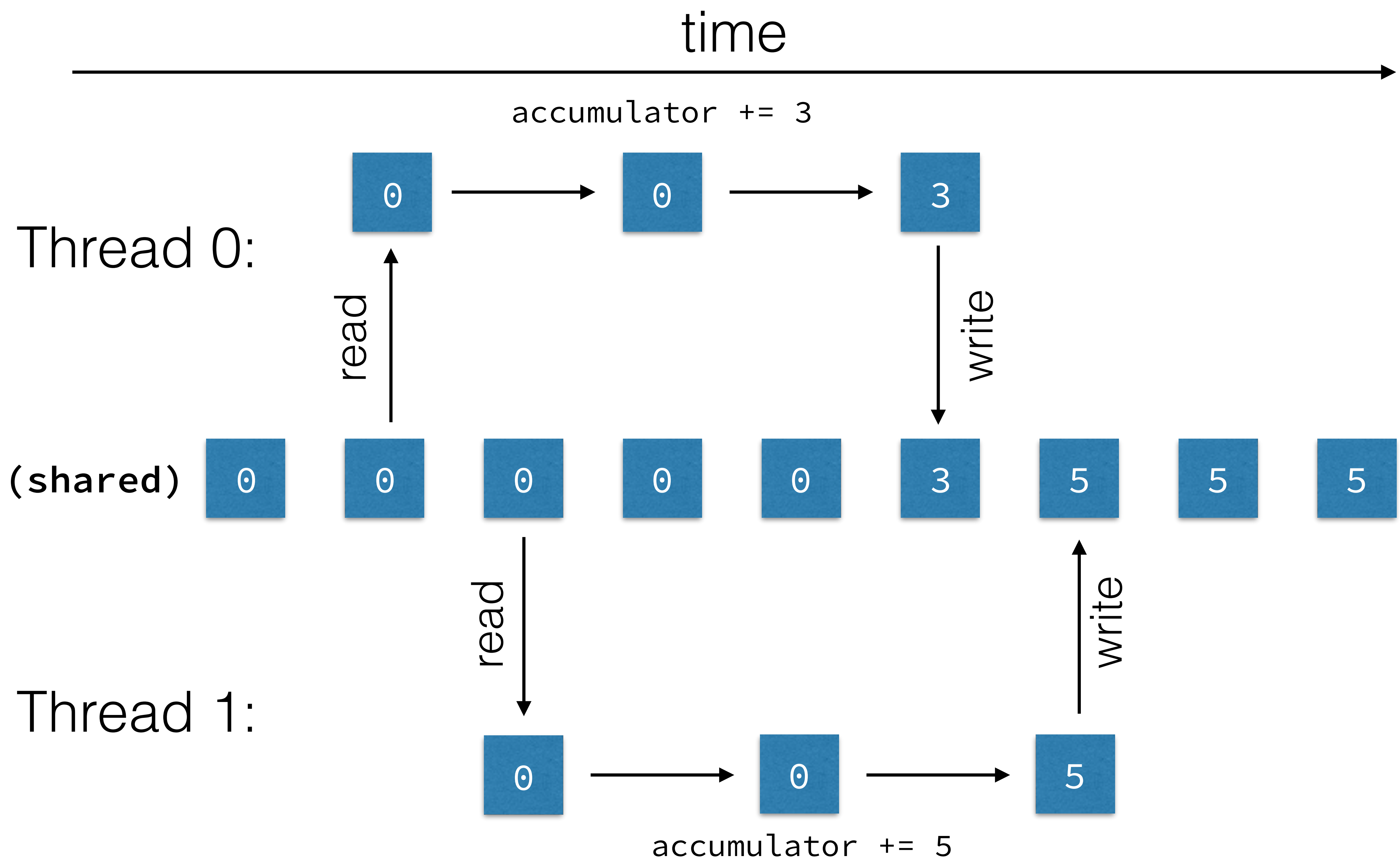
```
double accumulator = 0;
#pragma omp parallel
{
  double output;
  int thread_id = omp_get_thread_num();
  output = big_calculation(thread_id);


  accumulator += output;
}
```

“Race condition” — multiple threads with desynchronized read / writes

Synchronization - Critical



Synchronization - Critical



Synchronization - Critical

- Mutual exclusion (mutex) — only one thread at a time can enter critical region

```
double accumulator = 0;
#pragma omp parallel
{
  double output;
  int thread_id = omp_get_thread_num();
  output = big_calculation(thread_id);

  #pragma omp critical
  accumulator += output;
}
```

"Only one thread can execute following line at one time"

Synchronization - Atomic

- Mutex — but only for updates of memory locations
  - statement inside atomic must be of form:

    ```
    shared_mem_loc BINOP= expression
    ```

    e.g.
    ```
    accumulator += output;
    accumulator *= output;
    ```

  - in-place operations also allowed:

    ```
    ++accumulator;
    accumulator--;
    ```

Synchronization - Atomic

- “Atomic” is used in other languages for similar constructs

```
double accumulator = 0;
#pragma omp parallel
{
  double output;
  int thread_id = omp_get_thread_num();
  output = big_calculation(thread_id);

  #pragma omp atomic
  accumulator += output;
}
```

Summary of OpenMP Concepts - Part 1

- **#pragma omp parallel** [shared(…)]
  [private(…)]
  [num_threads(int)]

- **#pragma omp for** [schedule]

- **#pragma omp barrier**

- **#pragma omp critical**

- **#pragma omp atomic**

Next Time

- Gradually parallelizing and improving a numerical integration calculation.