Lecture #11 - More OpenMP

AMath 483/583

Last Time

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

- **#pragma omp for** [schedule]
- #pragma omp barrier
- #pragma omp critical
- #pragma omp atomic
- reduction(op: list)

Last Time

- **Aside**: manual loop chunking — `chunk_size=1`

```
// a = array of length N
int nthreads = omp_get_num_threads();
#pragma omp parallel shared(a) \
  private(id)
{
  int id = omp_get_thread_num();
  for (int i=id; i<N; i += nthreads)
    a[i] = // computation
}
```

This Time

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

- `#pragma omp for` [schedule]
- `#pragma omp barrier`
- `#pragma omp critical`
- `#pragma omp atomic`
- `reduction(op: list)`

Synchronization

- Impose order constraints and protect access to shared data
- General Construct:

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


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

Synchronization

- Impose order constraints and protect access to shared data
- General Construct:

```
#pragma omp parallel
{
  int id = omp_get_thread_num();
  A[id] = // some big calcula
          // on current threa


  // use all of A in some fu
  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
- General Construct:

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

  #pragma omp barrier
  // use all of 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
```

Question from Last Time

- What if your code looked like this?

```
#pragma omp parallel
{
  // thread work Part A

  #pragma omp for
  for (…, …, …)
       { // for loop body }

  // thread work Part B
}
```

Question from Last Time

- What if your code looked like this?

```
#pragma omp parallel
{
   // thread work Part A

   #pragma omp for
   for (…, …, …)
        { // for loop body }

   // thread work Part B
}
```

No implicit barrier here.

Thread `j` will start doing it's share of the loop work even if thread `k` is still working on Part A.

Question from Last Time

- What if your code looked like this?

```
#pragma omp parallel
{
  // thread work Part A

  #pragma omp for
  for (…, …, …)
       { // for loop body }

  // thread work Part B
}
```

Implicit barrier here.

Unless `nowait` is used, threads stop here until all reach end of loop.

Then all threads begin Part B.

Question from Last Time

- What if your code looked like this?

```
#pragma omp parallel
{
  // thread work Part A

  #pragma omp for nowait
  for (…, …, …)
       { // for loop body }

  // thread work Part B
}
```

Now, no implicit barrier

Once thread `j` finished it's share of the loop work it will move on to Part B.

Even if thread `k` is still working on its share.

Another Question

- Why not create a new `omp parallel` block?

```
#pragma omp parallel
{
}

#pragma omp parallel
{
}
```

All threads end here (because deleted)

- **Answer**: creating / deleting threads requires overhead!

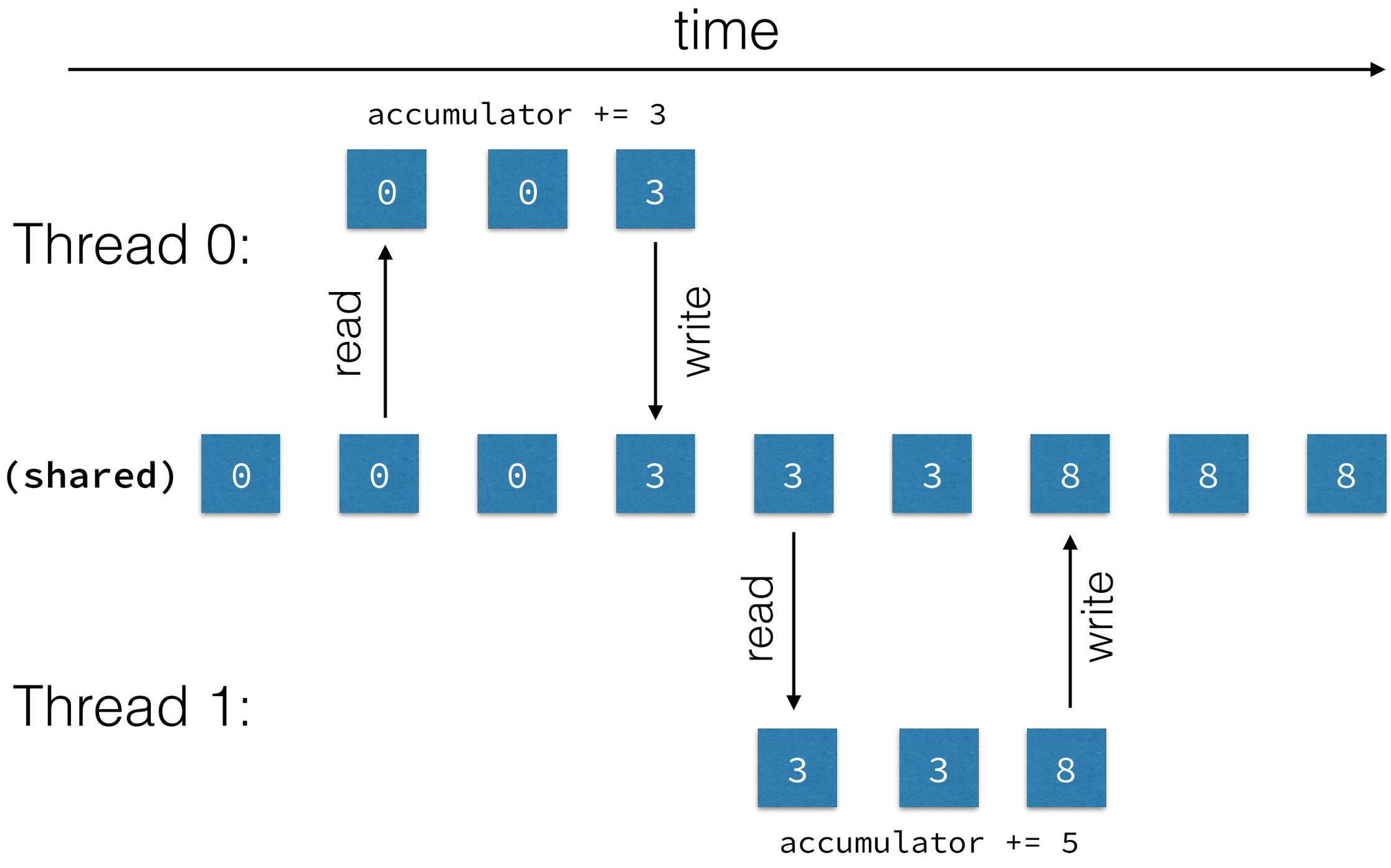
Synchronization - Critical

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

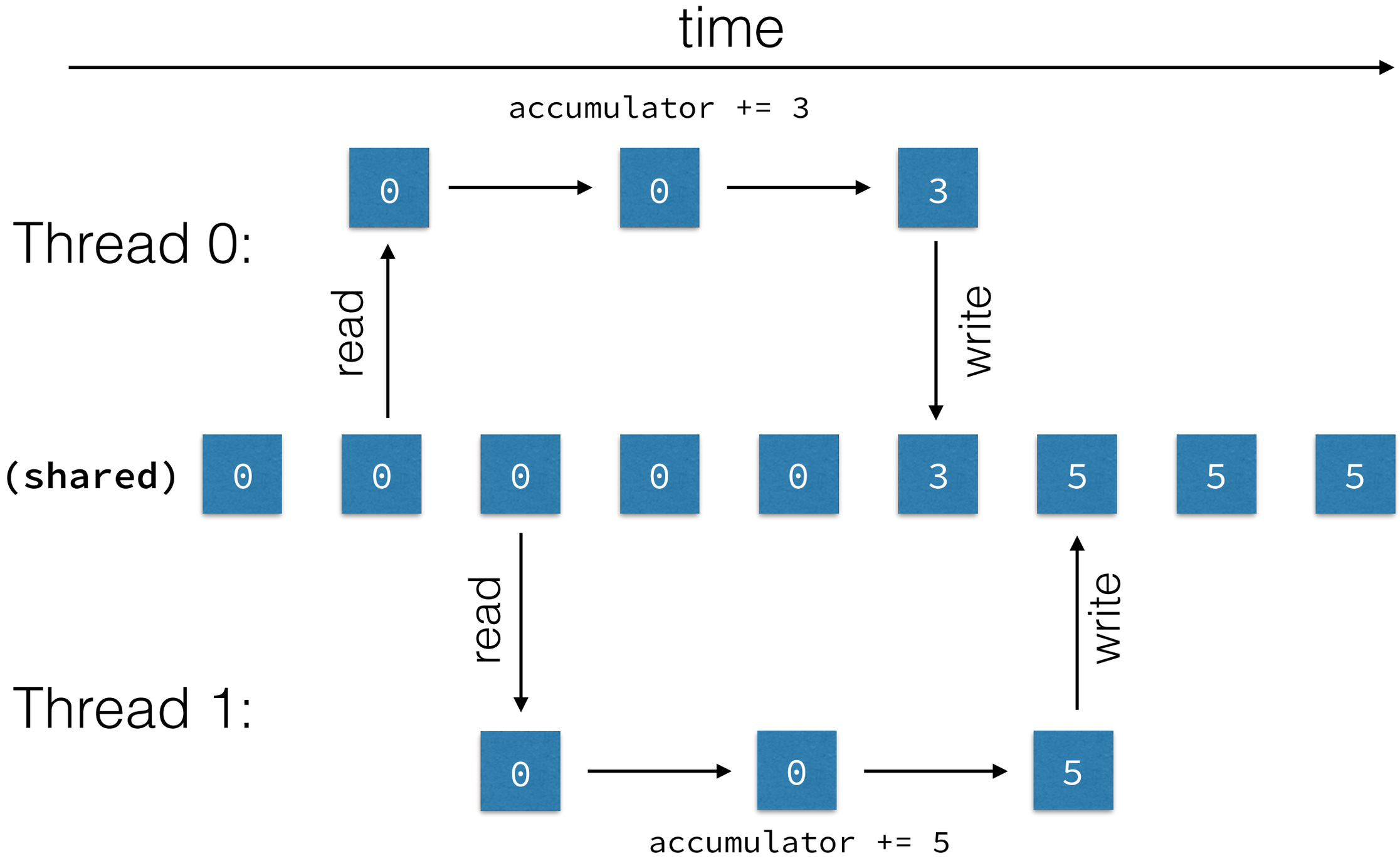
```
double accumulator = 0;
#pragma omp parallel
{
  double output;
  int thread_id = omp_get_thread_num();
  output = big_calculation(thread_id);

  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"

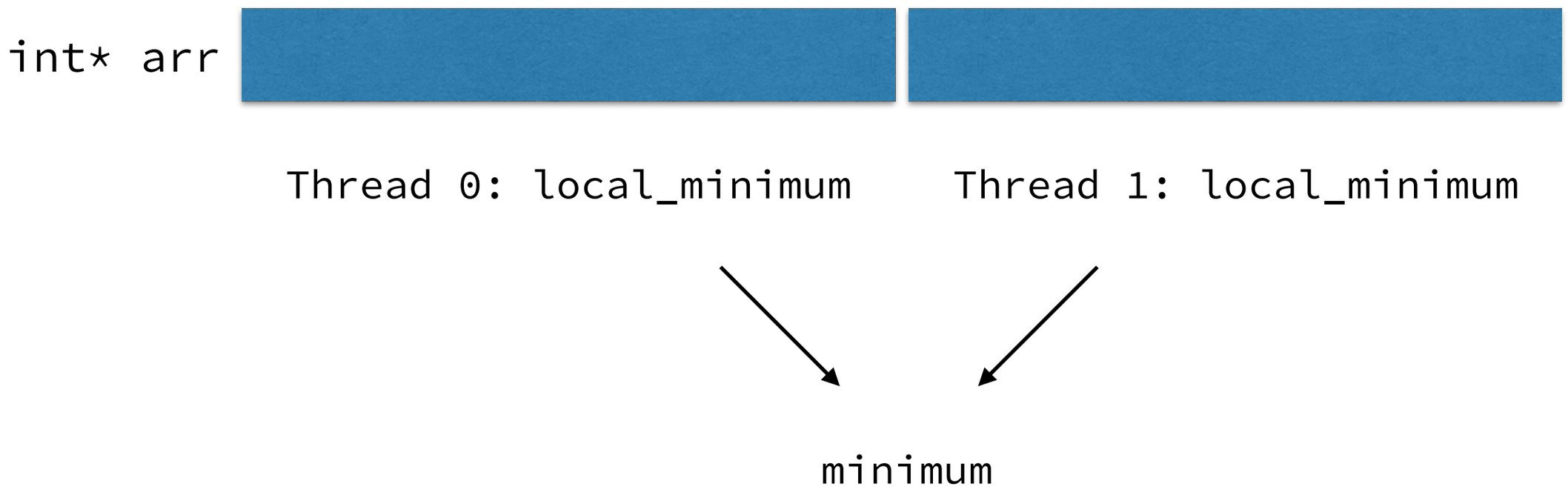
Example - Minimization

- Given an array (`int* a`) of ints find the minimum
- **Serial**: compare each element with the minimum and replace if necessary

```
// INT_MAX defined in limits.h
minimum = INT_MAX - 1;
for (int i=0; i<N; ++i)
  minimum = min(minimum, a[i]);
```

Example - Minimization

- **Parallelize**: each thread finds min in part of the array



Example - Minimization

- **Parallelize**: each thread finds min in part of the array

```
minimum = INT_MAX - 1;
#pragma omp parallel
{
  int local_minimum = minimum;

  // the parallel work: each thread gets a chunk of a
  #pragma omp for nowait
  for (int i=0; i<N; ++i)
    local_minimum = min(local_minimum, a[i]);

  // now make min across local minima (at each thread)
  minimum = min(local_minimum, minimum);
}
```

Example - Minimization

- **Parallelize**: each thread finds min in part of the array

```
minimum = INT_MAX - 1;
#pragma omp parallel
{
  int local_

  // the par                          a chunk of a
  #pragma om
  for (int i
    local_mi                         i]);

  // now make min across local minima (at each thread)
  minimum = min(local_minimum, minimum);
}
```

Potential Race Condition

Thread `j` - read `minimum`
Thread `k` - read `minimum`
Thread `j` - write `minimum`
Thread `k` - write `minimum`

Example - Minimization

- **Parallelize**: each thread finds min in part of the array

```
minimum = INT_MAX - 1;
#pragma omp parallel num_threads(32)
{
  int local_minim

  // the parallel                                    k of a
  #pragma omp for
  for (int i=0; i
    local_minimum

  #pragma omp critical
  minimum = min(local_minimum, minimum);
}
```

Serialized Portion

Thread j - read minimum
Thread k - read minimum
Thread j - write minimum
Thread k - write minimum

Minimization Alternative

- Each thread stores local min in array; find min outside parallel block

```
int nthreads = omp_get_num_threads();
int* local_minima = (int*) malloc(nthreads * sizeof(int));
#pragma omp parallel
{
  int id = omp_get_thread_num();
  int local_minima[id] = INT_MAX - 1;

  #pragma omp for nowait
  for (int i=0; i<N; ++i)
    local_minima[id] = min(local_minima[id], a[i]);
}

// now compute min from list of local minima
```

Minimization Alternative

- Each thread stores local min in array; find min outside parallel block

```
int nthreads = omp_get                    ();
int* local_minima                         ads * sizeof(int));
#pragma omp par
{
  int id = om
  int local_m

  #pragma omp fo
  for (int i=0; i
    local_minim      d] = min(local_minima[id], a[i]);
}

// now compute min from list of local minima
```

My Opinion

Not as clean as using critical block.

Demo

minimize.c

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;
}
```

- Less overhead than critical (see secondary references)

Reductions

- A common operation:

```
for (int i=0; i<N; ++i)
{
   expr = // compute…
   var = var OP expr;
}
```

Reductions

- A common operation:

```
for (int i=0; i<N; ++i)
{
   expr = // compute…
   var = var OP expr;
}
```

“Reduction Operation”

Carries potential for race conditions.

Example:

```
var = var + expr;
```

Reductions

- Could use `omp critical` or `omp atomic`
- `reduction(operator: list)` is a convenience function.
  - attach to `omp parallel`, `omp for`, or `omp sections`

```
#pragma omp parallel for  \
schedule(static,chunk)    \
reduction(+:result)
for (int i=0; i<N; ++i)
  result = result + (a[i] * b[i]);
```

Reductions

- Could use `omp critical` or `omp atomic`
- `reduction(operator: list)` is a convenience function.
  - attach to `omp parallel`, `omp for`, or `omp sections`

```
#pragma omp parallel for  \
schedule(static,chunk)    \
reduction(+:result)
for (int i=0; i<N; ++i)
  result = result + (a[i] * b[i]);
```

Private copy of `result` created for each thread.

Operations performed within each thread.

Final result is written to the global shared variable using reduction.

Reductions

- Could use `omp critical` or `omp atomic`
- `reduction(operato`ction.
  - attach to `omp par`ns

Reduction variables:

must be scalar
must be shared

warning:

```
#pragma omp parallel for    \
schedule(static,chunk)      \
reduction(+:result)
for (int i=0; i<N; ++i)
  result = result + (a[i] * b[i]);
```

Summary of OpenMP Concepts

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

- **`#pragma omp for`** `[schedule]`
- **`#pragma omp barrier`**
- **`#pragma omp critical`**
- **`#pragma omp atomic`**
- **`reduction(op: list)`**

Next Time

- Gradually parallelizing and improving a numerical integration calculation.