KOÇ UNIVERSITY

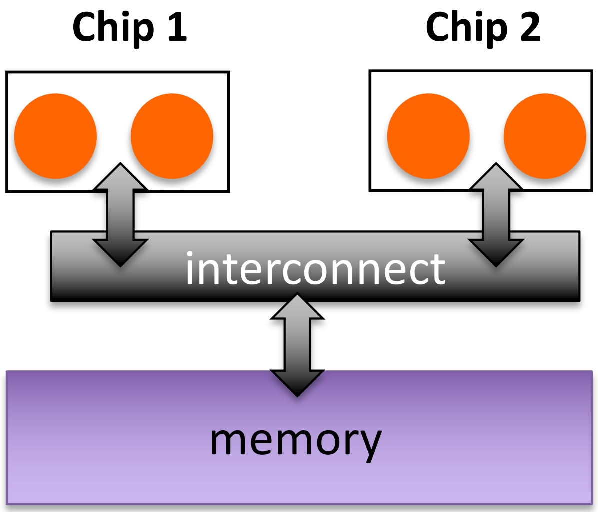
Shared-Memory Programming

Didem Unat

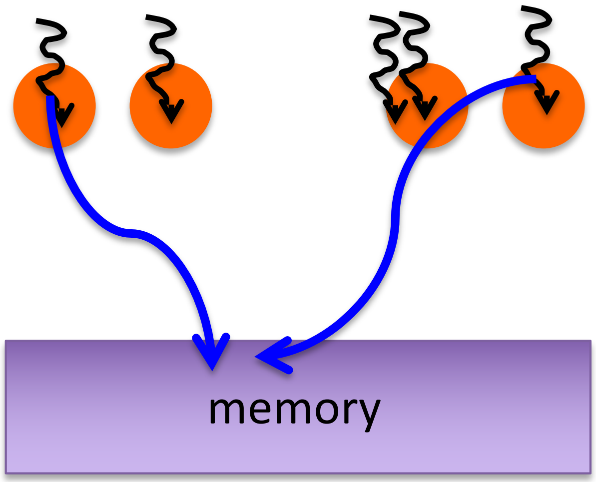
COMP 429/529 Parallel Programming

Shared-Memory Programming Model

- More correct name: Shared-address space programming
  - Threads communicate through shared memory as opposed to messages
  - Threads coordinate through synchronization (also through shared memory).



Recall shared memory system
(can be either UMA, NUMA)



Shared Memory Programming with Threads

- Several thread libraries out there
  - Pthreads, OpenMP, TBB, Cilk, Qthreads, C++11

- Pthreads is the POSIX (Portable Operating System Interface for Unix) Thread Library
  - Very low level of multi-threaded programming
  - Most widely used for systems-oriented code

- OpenMP is a standard
  - High level support for parallel programming on shared memory

OpenMP

- Chapter 5 from the textbook
- Tutorial: https://computing.llnl.gov/tutorials/openMP/
- Standardization Committee: http://www.openmp.org

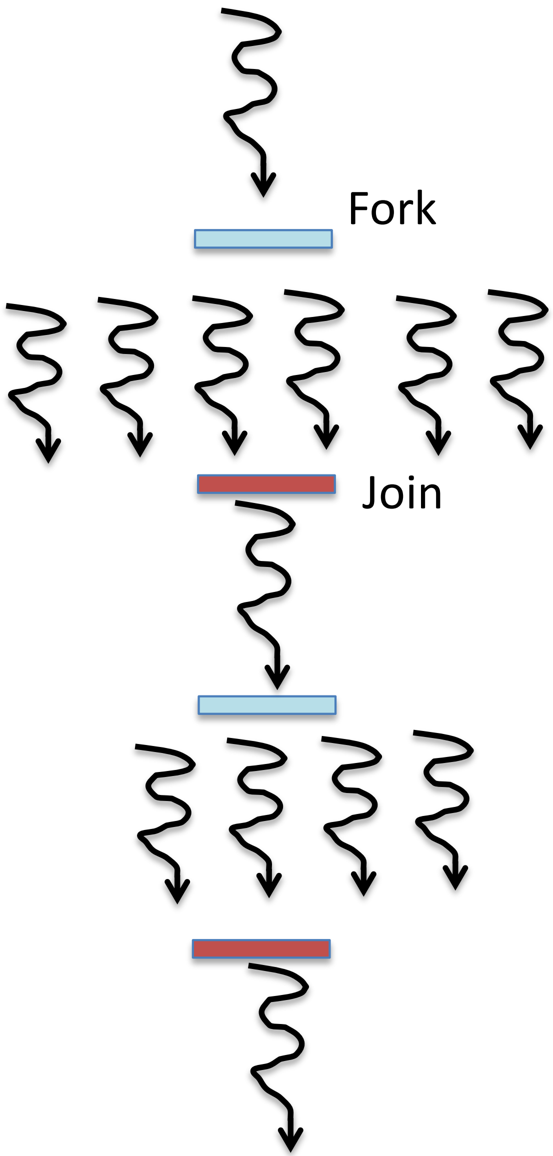
- Model for shared-memory parallel programming
  - Prevailing approach in scientific computing community
  - A simplified alternative to Pthreads

- OpenMP
  - MP= multiprocessing
  - Open= open specification, developed by community

- Extensions to existing programming languages (Fortran, C and C++)
  - Consists of compiler directives,
  - Runtime routines and environment variables

Fork-Join Execution Model

- Fork-join model of parallel execution
  - Begin execution as a single process (**master thread**)
- Start of a parallel construct:
  - Master thread creates team of threads (**worker threads**)
- Completion of a parallel construct:
  - Threads in the team synchronize -- (**implicit barrier**)
- Only master thread continues execution after join
  - A spawned thread executes asynchronously until it completes its task
  - Threads may or may not execute on different processors/cores



OpenMP uses #pragmas

- Programmer annotates the code with pragmas (directives)
  - Pragmas are special preprocessor instructions.

- Compilers that don't support the pragmas **ignore them.**

- The interpretation of OpenMP pragmas
  - They modify the statement immediately following the pragma
  - This could be a compound statement such as a loop

#pragma omp …

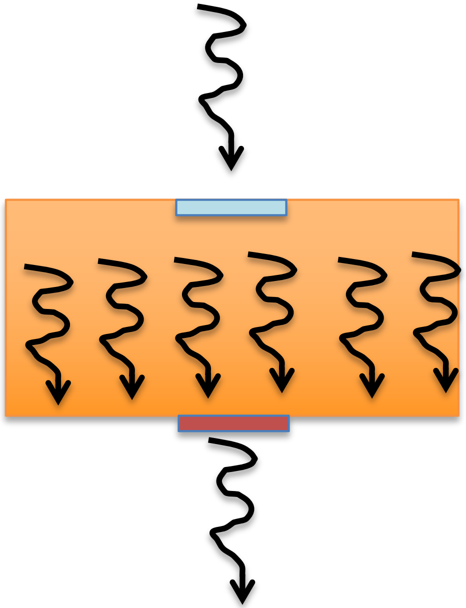
OpenMP Constructs

- Today, we will cover
  - `Parallel Region`
  - `Parallel For loop`
  - `Critical Directive`
  - `Reduction`

OpenMP Parallel Region

- A parallel region is a block of code executed by all threads in the team
- Each thread executes the **same code redundantly** **(SPMD)**

```
#pragma omp parallel [ clause [ clause ] ... ]
{
    //structured block
}
```



Hello World- Serial

```
#include <stdio.h>
#include <stdlib.h>


int main (int argc, char* argv[]){

    int thread_count= 4;



    int myID =0;

    printf("Hello from thread %d of %d\n", myID, thread_count);


    return 0;
}
```

Hello World in OpenMP

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

int main (int argc, char* argv[]){

    int thread_count= 4;



    int myID =0;


    printf("Hello from thread %d of %d\n", myID, thread_count);


    return 0;
}
```

Include OpenMP Library

Hello World in OpenMP

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

int main (int argc, char* argv[]){

    int thread_count= 4;

    int myID = omp_get_thread_num();

    printf("Hello from thread %d of %d\n", myID, thread_count);

    return 0;
}
```

Get thread ID

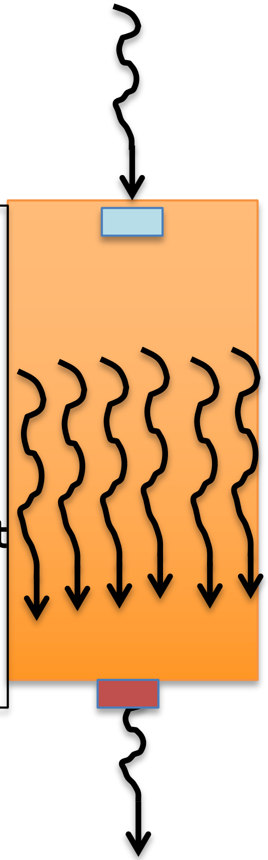
Thread IDs start from 0 and go to number of threads -1

Hello World in OpenMP

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

int main (int argc, char* argv[]){

    int thread_count= 4;

    #pragma omp parallel
    {
    int myID = omp_get_thread_num();


    printf("Hello from thread %d of %d\n", myID, thread_count

    }
    return 0;
}
```

a parallel region spawns threads

Threads join. Implicit barrier!



Hello World in OpenMP

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

int main (int argc, char* argv[]){

    int thread_count= 4;

    #pragma omp parallel num_threads(thread_count)
    {
    int myID = omp_get_thread_num();


    printf("Hello from thread %d of %d\n", myID, thread_count);

    }
    return 0;
}
```

Optionally specify number of threads to create

Hello World in OpenMP

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

int main (int argc, char* argv[]){

    int thread_count= 4;

    #pragma omp parallel num_threads(thread_
    {
    int myID = omp_get_thread_num();

    int num_threads = omp_get_num_threads();
    printf("Hello from thread %d of %d\n", myID, num_threads);

    }
    return 0;
}
```

How many threads actually created?

The OpenMP standard doesn't guarantee that *num_threads* clause will actually start *thread_count* many threads.

OpenMP Runtime Library

- In addition to compiler directives/pragmas, OpenMP has a runtime

```
int omp_get_num_threads(void);
```

- Returns the number of threads currently in the team executing the parallel region from which it is called

```
int omp_get_thread_num(void);
```

- Returns the thread number, within the team, that lies between 0 and omp_get_num_threads()-1, inclusive. The master thread of the team is thread 0

Lab 1: Hello World

- ssh username@login.kuacc.ku.edu.tr
- Copy the lab from here to your home directory
  - /kuacc/users/dunat/COMP429/OpenMP/Labs/Lab1-hello.c

Lab 1: Hello World

- Request time in an interactive queue
  - srun -N 1 -n 4 -p short --time=00:30:00 --pty bash
- Compile and run this code as it is
  - What do you get?
  - TODO 1
    - myID = omp_get_thread_num();
  - TODO 2
    - num_threads = omp_get_num_threads();
  - Compile and run, what do you observe?

Lab 1: Hello World

- Don’t forget to compiler with –fopenmp
- How many threads did it use? Why?
- TODO 3
  - int thread_count = 6;
- TODO 4
  - num_threads (thread_count)
- TODO 5
  - Move int myID and num_threads outside of parallel region
  - What do you observe?

Hello World in OpenMP

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

int main (int argc, char* argv[]){

    int thread_count= 4;

    int num_threads = omp_get_num_threads();
    int myID = omp_get_thread_num();


    #pragma omp parallel num_threads(thread_count)
    {

    printf(“Hello from thread %d of %d\n”, myID, num_threads);

    }

    return 0;
}
```

Calling this outside of a parallel region will get us 1.

Calling this outside of a parallel region will get us 0.

Compiling

```
gcc -fopenmp -o omp_hello omp_hello.c
```

```
./omp_hello
```

running

compiling

possible outcomes

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4

Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4
Hello from thread 3 of 4

Hello from thread 3 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 0 of 4

Hello World in OpenMP

- Things to think about
  - Code should be correct without the pragmas and library calls
  - There is an implicit thread identifier
  - Thread creation and termination (fork and join) are implicit, managed by compiler and runtime
  - Barrier at the end of a parallel region is implicit

OpenMP Clauses

- Text that modifies a directive/pragma/construct.
  - For example, the **num_threads** clause can be added to a parallel directive.
    - It allows the programmer to specify the number of threads that should execute the following block.

```
# pragma omp  parallel   num_threads ( thread_count )
```

$\underbrace{\texttt{parallel}}_{\text{directive}}$ $\underbrace{\texttt{num\_threads}}_{\text{clause}}$

- There are many more clauses – will learn some of them soon

Parallel Loop Construct

- Compiler calculates loop bounds for each thread directly from *serial* source (computation decomposition)
- Compiler also manages data partitioning
- Implicit barrier at the end of the loop
- This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

<table>
<tr><th>Serial Program</th><th>Parallel Program</th><th>Parallel Program</th></tr>
<tr>
<td><pre>double res[N];

for(int i=0; i < N; i++)
  do_huge_comp(res[i]);</pre></td>
<td><pre>double res[N];

#pragma omp parallel
{
  #pragma omp for
  for(int i=0; i < N; i++)
      do_huge_comp(res[i]);
}</pre></td>
<td><pre>double res[N];

#pragma omp parallel for
for(int i=0; i < N; i++)
      do_huge_comp(res[i]);</pre></td>
</tr>
</table>

Limitations and Semantics of Parallel for

- Requirements ensure iteration count is predictable
- Not all loops can be parallelizable by the compiler
  - Loop index: signed integer
  - Program correctness cannot depend on which thread executes a particular iteration
  - Loop control parameters must be the same for all threads
  - Termination Test: <,<=,>,=> with loop **invariant** integer
  - Incr/Decr by loop invariant int; change each iteration
  - Count up for <,<=; count down for >,>=
  - Illegal to branch out of a loop

```
#pragma omp parallel for
for(int i=lbound; i < ubound; i++)
      do_huge_comp(res[i]);
```

Data Sharing

- Shared memory parallel programs often employ two types of data
  - Shared data, visible to all threads
  - Private data, visible to a single thread (often stack-allocated)

- Pthreads:
  - Global-scoped variables are shared
  - Stack-allocated variables are private

- OpenMP:
  - **shared** variables are shared
  - **private** variables are private
  - Default is **shared**
  - Loop index is **private**

Hello World in OpenMP

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

int main (int argc, char* argv[]){

    int thread_count= 4 ;

    #pragma omp parallel num_threads(thread_count)
    {
    int myID = omp_get_thread_num();


    prinf("Hello from thread %d of %d\n", myID, thread_count);

    }
    return 0;
}
```

Thread_count is shared by all threads!

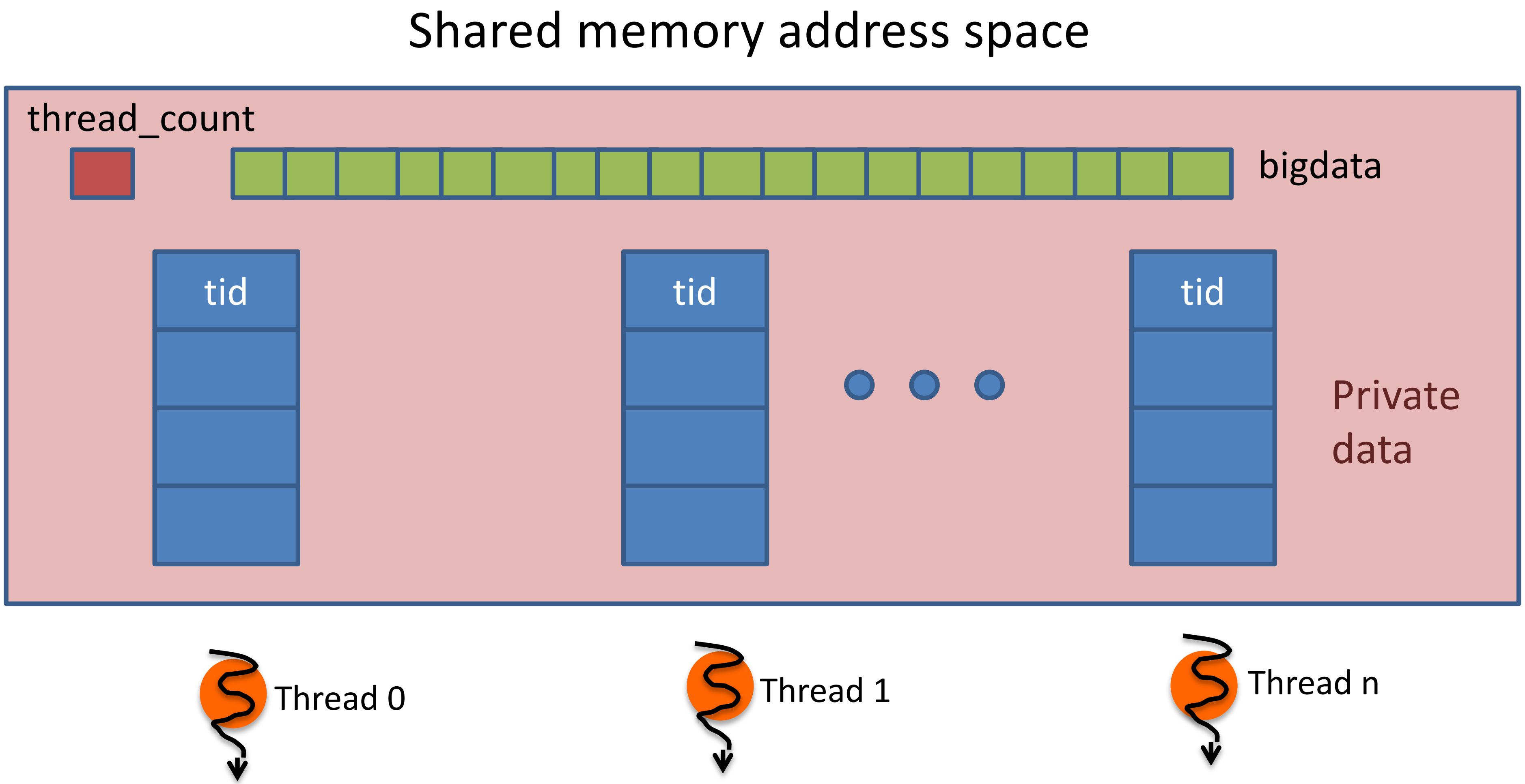
myID is private to each thread.

Data Sharing

- OpenMP:
  - **shared** variables are shared
  - **private** variables are private
  - Variables created inside a parallel region are **private**
  - Default is **shared**
  - Loop index is **private**

```
int bigdata[1024];
int tid, thread_count;
#pragma omp parallel shared ( bigdata ) private ( tid )
  {
    //any local variable declared in this scope is private
    /* Calc. here */
  }
```

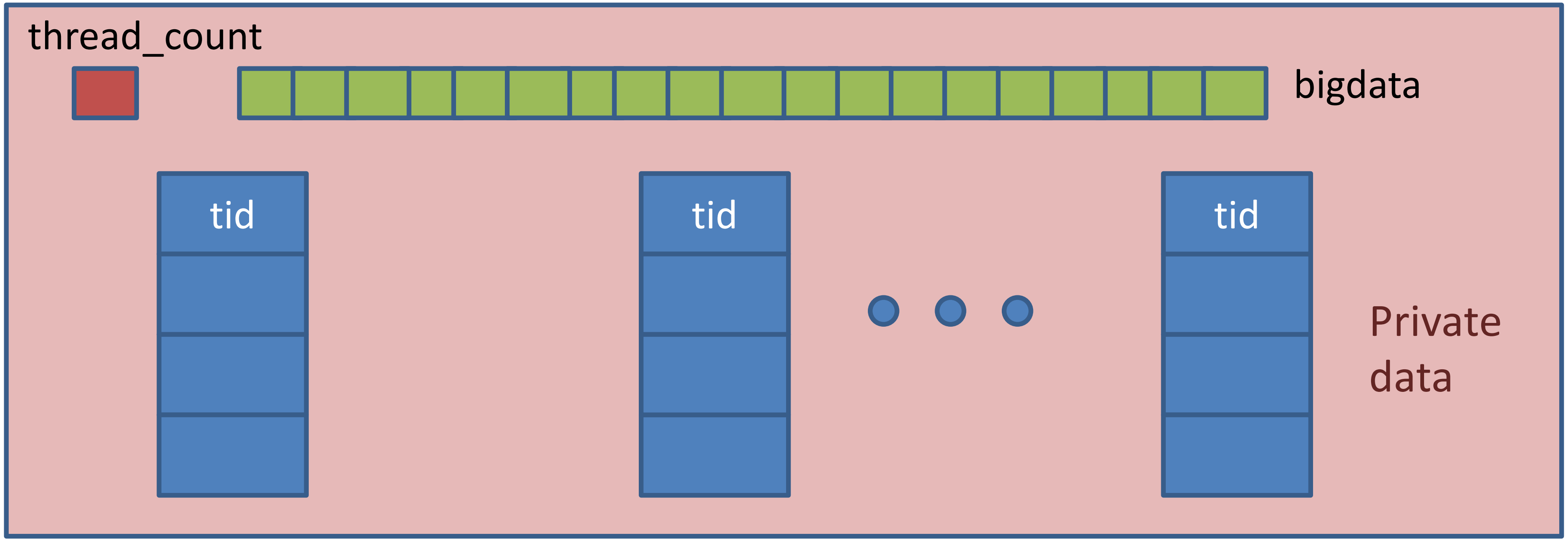
Memory View



- Threads have a private memory space (stack) and shared memory space (heap and global data)

Memory View

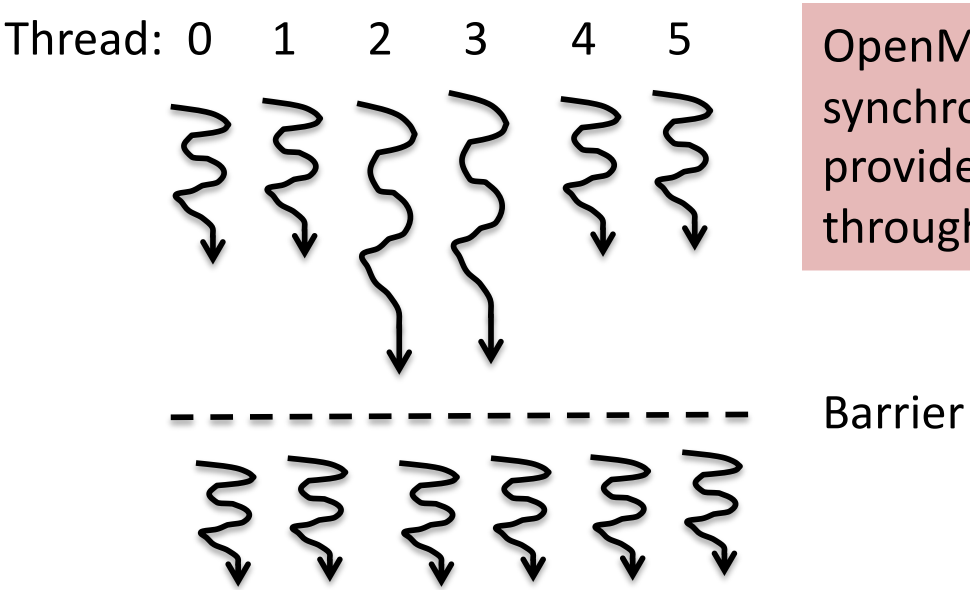
Shared memory address space



- What should you declare as private/shared?
- What is the advantage/disadvantage of declaring everything as shared?
- What is the advantage/disadvantage of declaring everything as private?

Barriers

- Synchronizing threads to make sure that they all are at the same point in a program is called a **barrier.**
- No thread can cross the barrier until all the threads have reached it.



OpenMP doesn't just synchronizes threads but provides **memory consistency** through barriers

Even though threads 2 and 3 reached barrier, they will wait for others to arrive.
Then all threads cross the barrier point together.

OpenMP Critical Directive

- Enclosed code executed by all threads, but
  - **restricted to only one thread at a time**

```
#pragma omp critical [ ( name ) ] new-line
   structured-block
```

- A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name.
- All unnamed critical directives map to the same unspecified name.
- Acts like a mutex/lock

The critical Construct

```
int dequeue(float *a);
void work(int i);

void critical_example(float *x, float *y) {
    int ix_next, iy_next;

    #pragma omp parallel shared(x, y)
    {
        #pragma omp critical (xaxis)
            ix_next = dequeue(x);
        work(ix_next);

        #pragma omp critical (yaxis)
            iy_next = dequeue(y);
        work(iy_next);
     }
}
```

Naming the critical sections

The critical Construct

```
int dequeue(float *a);
void work(int i);

void critical_example(float *x, float *y) {
    int ix_next, iy_next;

    #pragma omp parallel shared(x, y)
    {
        #pragma omp critical (xaxis)
            ix_next = dequeue(x);
        work(ix_next);

        #pragma omp critical (yaxis)
            iy_next = dequeue(y);
        work(iy_next);
     }
}
```

What is wrong with this program?

The critical Construct

```
int dequeue(float *a);
void work(int i);

void critical_example(float *x, float *y) {
    int ix_next, iy_next;

    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
            ix_next = dequeue(x);
        work(ix_next);

        #pragma omp critical (yaxis)
            iy_next = dequeue(y);
        work(iy_next);
     }
}
```

Recall our parallel sum example

- Recall our parallel sum example from previous lecture

```
int items_per_task = n/t;
mutex m;
int my_sum=0, my_x, sum=0;
int start = thread_id * items_per_task;

for (i=start; i<start + items_per_task; i++)  {
     my_x = Compute_next_value(…);
     my_sum += my_x;
}
mutex_lock(m);
sum+= my_sum;
mutex_unlock(m);
```

OpenMP version of sum example

- Replaced mutex with a critical section

```
~~int items_per_task = n/t;~~
~~mutex m;~~
int my_sum=0, my_x, sum=0;
~~int start = thread_id * items_per_task;~~
#pragma omp parallel for
for (i=0; i< N; i++)  {
     my_x = Compute_next_value(…);
     my_sum += my_x;
}
#pragma omp critical
sum+= my_sum;
```

Is this correct?

No, my_x and my_sum should be private, otherwise race condition occurs!

OpenMP version of sum example

- Specify thread-private variables

```
int items_per_task = n/t;
mutex m;
int my_sum=0, my_x, sum=0;
int start = thread_id * items_per_task;
#pragma omp parallel for private(my_x, my_sum)
for (i=0; i< N; i++)  {
     my_x = Compute_next_value(…);
     my_sum += my_x;
}
#pragma omp critical
sum+= my_sum;
```

Is this correct?

No, critical section should be in a parallel region, otherwise only master thread executes that section

OpenMP version of sum example

- Critical section should be in a parallel region

```
int items_per_task = n/t;
mutex m;
int my_sum=0, my_x, sum=0;
int start = thread_id * items_per_task;
#pragma omp parallel private(my_x, my_sum) shared(sum)
{
    my_sum = 0;
    #pragma omp for
    for (i=0; i< N; i++)  {
            my_x = Compute_next_value(…);
            my_sum += my_x;
    }
    #pragma omp critical
    sum+= my_sum;
}
```

OpenMP Reduce

- OpenMP has reduce operation

Operator Variable

```
sum = 0;
#pragma omp parallel for private(my_x) reduction(+:sum)
for (i=0; i < 100; i++) {
    my_x = Compute_next_value(…);
    sum += my_x;
}
```

- Reduce ops and init() values (C and C++):

+	0	bitwise &	0	logical &	1
-	0	bitwise |	0	logical |	0
*	1	bitwise ^	0		

OpenMP Reduction

- OpenMP runtime/compiler has an ``efficient" implementation for parallel reduction

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

Acknowledgments

- These slides are inspired and partly adapted from
  - Mary Hall (Univ. of Utah)
  - The course book (Pacheco)