COMP 605: Introduction to Parallel Computing Topic: Shared Memory Programming with OpenMP

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

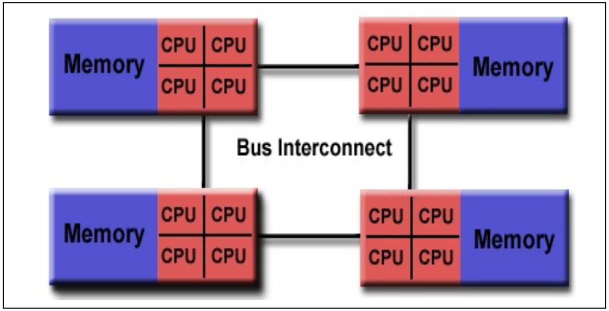
Presented: 04/11/17
Last Update: 04/10/17

What is OpenMP?

- OpenMP = Open Multir-Processing
- an API that supports multi-platform shared memory multiprocessing programming.
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPUs, all of which have access to main memory
- Applications built using hybrid model of parallel programming:
    - Runs on a computer cluster using both OpenMP and Message Passing Interface (MPI)
    - OR through the use of OpenMP extensions for non-shared memory systems.
- See:
    - `http://openmp.org/`
    - `http://en.wikipedia.org/wiki/OpenMP`

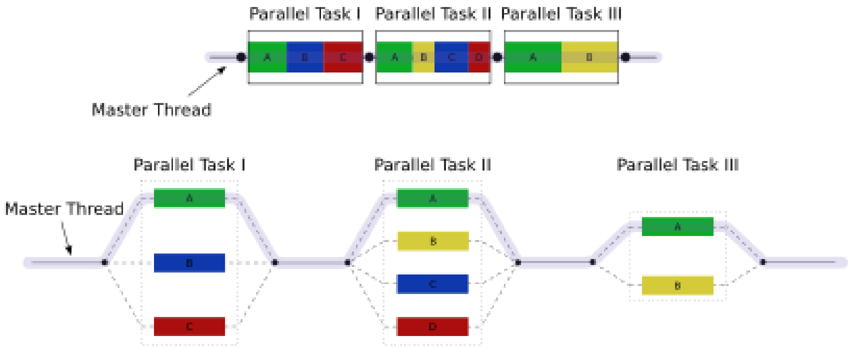
What is OpenMP?

- OpenMP grew out of the need to standardize different vendor specific directives related to parallelism.
- Pthreads not scaleable to large systems and does not support incremental parallelism very well.
- Correlates with evolution of hybrid architectures: shared memory and multi PE architectures being developed in early '90s.
- Structured around parallel loops and was meant to handle dense numerical applications.



Source: `https://computing.llnl.gov/tutorials/openMP`

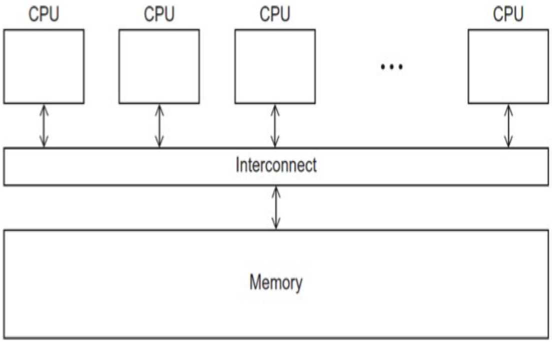
OpenMP is an implementation of *multithreading*



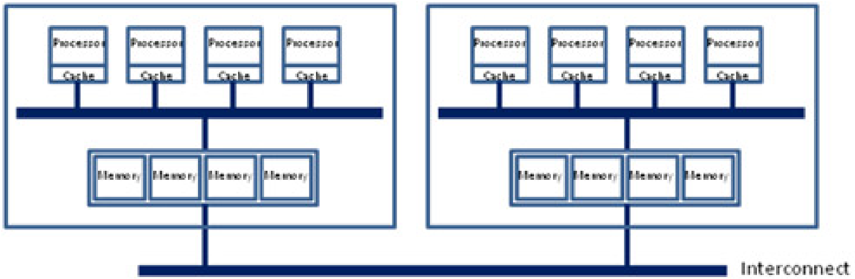
Source: `http://en.wikipedia.org/wiki/OpenMP`

- Method of parallelizing where a master thread forks a specified number of slave threads
- Tasks are divided among them.
- Threads run concurrently.

Shared memory architecture 1



Non Uniform Memory Access (NUMA)



- Hierarchical Scheme: processors are grouped by physical location
- located on separate multi-core (PE) CPU packages or nodes.
- Processors (PEs) within a node share access to memory modules via UMA shared memory architecture.
- PE's may also access memory from the remote node using a shared interconnect

Source: `https://software.intel.com/en-us/articles/optimizing-applications-for-numa`

OpenMP Features & Advantages

- Portable, threaded, shared-memory programming specification with light syntax
- Exact behavior depends on OpenMP implementation!
- Requires compiler support (C or Fortran)
- Allows programmer to define and separate serial and parallel regions
- Does not "detect" parallel dependencies or guarantee speedup
- Can use OpenMP to parallelize many serial for loops with only small changes to the source code.
- Task parallelism.
- Explicit thread synchronization.
- Standard problems in shared-memory programming

OpenMP Challenges

- Currently only runs efficiently in shared-memory multiprocessor platforms
- Scalability is limited by memory architecture.
- Cache memories
- Dealing with serial libraries
- Thread safety
- Unreliable error handling.
- Mostly used for loop parallelization
- Requires a compiler that supports OpenMP
- Lacks fine-grained mechanisms to control thread-processor mapping.
- Synchronization between subsets of threads is not allowed.
- Can be difficult to debug, due to implicit communication between threads via shared variables.

OpenMP: General Code Structure

```
 #include <omp.h>
main () {
   int var1, var2, var3;
   Serial code
   . . .
  /* Beginning of parallel section.
   Fork a team of threads. Specify variable scoping*/
   #pragma omp parallel private(var1, var2) shared(var3)
   {
      /* Parallel section executed by all threads */
     . . .
     /* All threads join master thread and disband*/
    }
    Resume serial code
    . . .
}
```

OpenMP: A Very Simple Hello World

```
/*
 *  File:  omp_hello_env.c
 *  Compile:  gcc -g -Wall -fopenmp -o omp_hello_env omp_hello_env.c
 *
 * In this example, the number of threads is set
 * using the value for the environment variable
 *     OMP_NUM_THREADS
 * It can be set via the command line:
 *   OMP_NUM_THREADS=8  ./omp_hello_env
*/
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

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

  #pragma omp parallel default(shared) private(thd_rank, nthds)
  {
    nthds = omp_get_num_threads();
    thd_rank = omp_get_thread_num();
    printf("Hello from thread %d out of %d\n",
           thd_rank, nthds);
  }
return 0;
}
```

How are the number of threads set????

OpenMP: Using/Setting OMP_NUM_THREADS

- You can run using just the name of the executable and default number of threads via the command line:

```
[mthomas] ./omp_hello
Hello from thread 3 out of 4
Hello from thread 0 out of 4
Hello from thread 1 out of 4
Hello from thread 2 out of 4
```

- You can change the number of threads via the command line (or set the env var in your shell):

```
[mthomas] OMP_NUM_THREADS=8 ./omp_hello
Hello from thread 3 out of 8
Hello from thread 6 out of 8
Hello from thread 7 out of 8
Hello from thread 4 out of 8
Hello from thread 5 out of 8
Hello from thread 0 out of 8
Hello from thread 1 out of 8
Hello from thread 2 out of 8
```

OpenMP: Using/Setting OMP_NUM_THREADS

You can pass the number of threads as a command line argument:

```
/* File:     omp_hello.c
 * Purpose:  A parallel hello, world program that uses OpenMP
 * Compile:  gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
 * Run:      ./omp_hello <number of threads>
  */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

/*--------------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

/*-------------------------------------------------------------------
 * Function:    Hello
 * Purpose:     Thread function that prints message
 */
void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

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

}  /* Hello */
```

OpenMP: Using/Setting OMP_NUM_THREADS

```
[mthomas]%
[mthomas@tuckoo]$ mpicc -g -Wall -fopenmp -o omp_hello omp_hello.c


[mthomas@tuckoo ch5]$ ./omp_hello 10
Hello from thread 6 of 10
Hello from thread 4 of 10
Hello from thread 5 of 10
Hello from thread 0 of 10
Hello from thread 1 of 10
Hello from thread 7 of 10
Hello from thread 2 of 10
Hello from thread 3 of 10
Hello from thread 9 of 10
Hello from thread 8 of 10
```

OpenMP: You can set OMP_NUM_THREADS in a batch script

```
[mthomas] cat batch.omp_hello_thdarg
#!/bin/sh
# run using:
#    qsub -v T=16 batch.omp_hello
#    qsub -v T=16 batch.omp_hello_thdarg
#
#PBS -V
#PBS -l nodes=1:core16
#PBS -N omp_hello_thdarg
#PBS -j oe
#PBS -r n
#PBS -q batch
cd $PBS_O_WORKDIR
echo PBS: current home directory is $PBS_O_HOME

####################################
# set number of threads using env var
OMP_NUM_THREADS=${T}
export OMP_NUM_THREADS=${T}
./omp_hello
####################################


####################################
#or pass to program:
#./omp_hello_thdarg $T
####################################
```

What to do if compiler does not support OpenMP

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

```
#ifdef _OPEN_MP
#include <omp.h>
#endif
 ...
int rank;
int thd_cnt;
 ...
#ifdef _OPEN_MP
rank=omp_get_thread_num();
thd_cnt=omp_get_num_threads();
#else
rank=0;
thd_cnt=1;
#endif
 ...
```

OpenMP Directive: #pragma

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

- #pragma is first OpenMP directive.
- Scope of a directive is one block of statements {. . . }
- OpenMP determines # threads to create, synchronize, destroy
- Start threads running thread function Hello.
- *num_threads(thread_count)* is an OpenMP clause
- Similar (but less work) to the Pthread command:

```
pthread_create(&thread_handles[i], NULL, Thread_work, (void*) i);
```

- Special preprocessor instructions.
- Typically added to a system to allow behaviors that arent part of the basic C specification.
- Portable: compilers that don't support the pragmas ignore them.

OpenMP: Parallel Region Construct

- Defines a block of code to be executed by the threads:

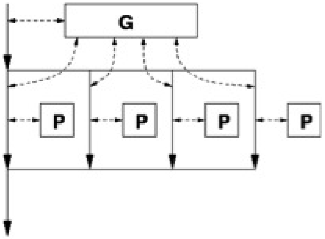
```
#  pragma omp parallel num_threads(thread_count)
        {
               ...
           } (implied barrier)
```

- Example clauses:
  - if (expression): only in parallel if expression evaluates to true
  - private(list): everything private and local (no relation with variables outside the block).
  - shared(list): data accessed by all threads
  - default (none — shared)
  - reduction (operator: list)
  - firstprivate(list), lastprivate(list)

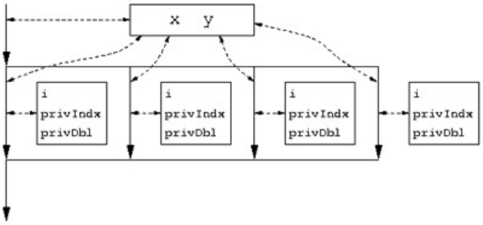
OpenMP: Data Model

- Private and shared variables
- Global data space: accessed by all parallel threads.
- Private space: only be accessed by the thread.
- Parallel for loop index private by default.

```
#pragma omp parallel for private(
     privIndx, privDbl )
  for ( i = 0; i < arraySize; i++){
     for(privdx=0; privdx <16;privdx++){
        privDbl=( (double)privdx)/16;
      y[i]=sin(exp(cos( -exp(sin(x[i])))))
                   + cos( privDbl );
   }    }
```



P = private data space
G = global data space



execution context for "arrayUpdate_II"

OpenMP pragma directives

#pragmal omp	**#Desc**
atomic	Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
atomic	Identifies a specific memory location that must be updated atomically and not be exposed to multiple, simultaneous writing threads.
parallel	Defines crit. block to be run by multiple threads in parallel. With specific exceptions, all other OpenMP directives work within parallelized regions defined by this directive.
for	Work-sharing construct identifying an iterative for-loop whose iterations should be run in parallel.
parallel for	Shortcut combination of omp parallel and omp for pragma directives, used to define a parallel region containing a single for directive.
ordered	Work-sharing construct identifying a structured block of code that must be executed in sequential order.
section(s)	Work-sharing construct identifying a non-iterative section of code containing one or more subsections of code that should be run in parallel.
parallel sections	Shortcut combination of omp parallel and omp sections pragma directives, used to define a parallel region containing a single sections directive.
single	Work-sharing construct identifying section of code to be run by a single avail. thread.
master	Synchronization construct identifying a section of code that must be run only by the master thread.
critical	Synchronization construct identifying a statement block that must be executed by a single thread at a time.
barrier	Synchronizes all the threads in a parallel region.
flush	Synchronization construct identifying a point at which the compiler ensures that all threads in a parallel region have the same view of specified objects in memory.
threadprivate	Defines the scope of selected file-scope data variables as being private to a thread, but file-scope visible within that thread.

Some OpenMP Comments & Observations

- In OpenMP terminology, the collection of threads executing the parallel block the original thread and the new threads is called a *team*
- the original thread is called the *master*
- additional threads are called *slaves*
- the master starts p-1 new threads
- implicit barrier: formed after the hello thread – all threads must return to this point in the code
- all threads share STDIO

Binding OpenMP Thread to a Processor

- There may be system-defined limitations on the number of threads that a program can start.
- OpenMP standard does not guarantee that the directive will actually start the number of requested threads.
- Modern systems can start hundreds or thousands of threads
- OpenMP typically will scale to the number of cores on a node
- OpenMPI contains features that allow you to force binding to get one thread per core.
- Two mechanisms:
    - Set the environment variable *OMP_PROC_BIND* to logical *true* or *false* via the command line, in your .bashrc file, or in the batch script (or the command line)
    - use an API to find out the binding setting: *omp_proc_bind_t omp_get_proc_bind(void)*
      Note: this requires a special library version, which is not on tuckoo

Binding OpenMP Thread to a Processor

```
/*  File:  omp_info.c
 *  Written by: Mary Thomas, April, 2016
 *  Compile:  gcc -g -Wall -fopenmp -o omp_info omp_info.c
 */
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <omp.h>

void Usage(char* prog_name);

int main(int argc, char* argv[]) {
   int omp_req_thds, omp_nprocs, omp_core, omp_numthds, omp_tid;

   if (argc != 2) {
      fprintf(stderr, "  usage: %s <omp_req_thds> \n", argv[0]);
      exit(0);
   }
   omp_req_thds = strtol(argv[1], NULL, 10);

   /* get the total number of processors available to the device */
   omp_nprocs  = omp_get_num_procs();
   /*  set the number of threads to override any ENV vars */
   omp_set_num_threads(omp_req_thds);
   printf("omp_nprocs=%d,  omp_req_thds=%d\n", omp_nprocs,omp_req_thds);

   # pragma omp parallel num_threads(omp_req_thds)  private(omp_core,omp_numthds, omp_tid)       {
      omp_core        = sched_getcpu();
      omp_numthds = omp_get_num_threads(); /* get number of OpenMP threads */
      omp_tid           = omp_get_thread_num(); /* get OpenMP thread ID */
      printf("OMP region: omp_tid=%d, omp_core=%d, omp_numthds=%d \n",
               omp_tid, omp_core, omp_numthds);
   }
}
```

OpenMP Thread to Processor Bindings: Batch Script

```
[mthomas@tuckoo] cat batch.omp_info
#!/bin/sh
# run using:
#    qsub -v T=16,B=false batch.omp_info
#
#PBS -V
#PBS -l nodes=1:core16
#PBS -N omp_info
#PBS -j oe
#PBS -r n
#PBS -q batch
cd $PBS_O_WORKDIR
echo PBS: current home directory is $PBS_O_HOME

# set binding of threads to processors
# logical true or false
# set the ENV var in this script, or on the command line
OMP_PROC_BIND=$B
export OMP_PROC_BIND

# use this if you are not setting number of
# threads in program
#set number of cores using command line arg
OMP_NUM_THREADS=${T}
export OMP_NUM_THREADS=${T}

# either of these lines will work:
OMP_PROC_BIND=$B ./omp_info $T
####./omp_info $T

```

OpenMP Thread to Processor Bindings: Output

Setting the environment variable OMP_PROC_BIND to logical *true* will force the system to bind 1 thread to 1 processor. The default is *false* If the number of threads is larger than the number of processors, the system will begin to assign multiple threads.

```
[mthomas] qsub -v B=true,T=16 batch.omp_info
6738.tuckoo.sdsu.edu
[mthomas] cat omp_info.o6738 | sort
omp_nprocs=16,  omp_req_thds=16
OMP region: omp_tid=0, omp_core=0, omp_numthds=16
OMP region: omp_tid=1, omp_core=1, omp_numthds=16
OMP region: omp_tid=2, omp_core=2, omp_numthds=16
OMP region: omp_tid=3, omp_core=3, omp_numthds=16
OMP region: omp_tid=4, omp_core=4, omp_numthds=16
OMP region: omp_tid=5, omp_core=5, omp_numthds=16
OMP region: omp_tid=6, omp_core=6, omp_numthds=16
OMP region: omp_tid=7, omp_core=7, omp_numthds=16
OMP region: omp_tid=8, omp_core=8, omp_numthds=16
OMP region: omp_tid=9, omp_core=9, omp_numthds=16
OMP region: omp_tid=10, omp_core=10, omp_numthds=16
OMP region: omp_tid=11, omp_core=11, omp_numthds=16
OMP region: omp_tid=12, omp_core=12, omp_numthds=16
OMP region: omp_tid=13, omp_core=13, omp_numthds=16
OMP region: omp_tid=14, omp_core=14, omp_numthds=16
OMP region: omp_tid=15, omp_core=15, omp_numthds=16
```

```
[mthomas] qsub -v B=false,T=16 batch.omp_info
6736.tuckoo.sdsu.edu
[mthomas] cat omp_info.o6736 | sort
omp_nprocs=16,  omp_req_thds=16
OMP region: omp_tid=0, omp_core=12, omp_numthds=16
OMP region: omp_tid=1, omp_core=1, omp_numthds=16  ***
OMP region: omp_tid=2, omp_core=2, omp_numthds=16
OMP region: omp_tid=3, omp_core=13, omp_numthds=16
OMP region: omp_tid=4, omp_core=3, omp_numthds=16
OMP region: omp_tid=5, omp_core=4, omp_numthds=16
OMP region: omp_tid=6, omp_core=14, omp_numthds=16
OMP region: omp_tid=7, omp_core=5, omp_numthds=16
OMP region: omp_tid=8, omp_core=6, omp_numthds=16
OMP region: omp_tid=9, omp_core=15, omp_numthds=16
OMP region: omp_tid=10, omp_core=7, omp_numthds=16
OMP region: omp_tid=11, omp_core=0, omp_numthds=16  ***
OMP region: omp_tid=12, omp_core=8, omp_numthds=16
OMP region: omp_tid=13, omp_core=0, omp_numthds=16  ***
OMP region: omp_tid=14, omp_core=9, omp_numthds=16
OMP region: omp_tid=15, omp_core=1, omp_numthds=16  ***
```

OpenMP Thread Bindings: Requesting More Threads than Cores

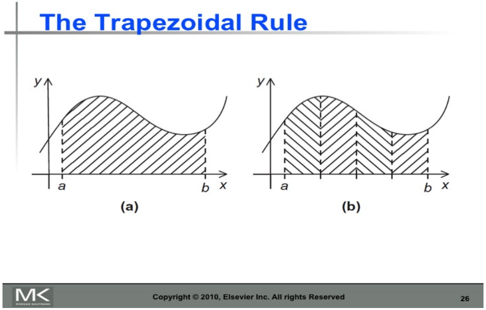
```
[mthomas] qsub -v B=false,T=32 batch.omp_info
6740.tuckoo.sdsu.edu
[mthomas] cat omp_info.o6740 | sort
omp_nprocs=16,  omp_req_thds=32
OMP region: omp_tid=0, omp_core=11, omp_numthds=32
OMP region: omp_tid=1, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=2, omp_core=2, omp_numthds=32
OMP region: omp_tid=3, omp_core=12, omp_numthds=32
OMP region: omp_tid=4, omp_core=0, omp_numthds=32
OMP region: omp_tid=5, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=6, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=7, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=8, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=9, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=10, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=11, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=12, omp_core=0, omp_numthds=32    ****
OMP region: omp_tid=13, omp_core=3, omp_numthds=32
OMP region: omp_tid=14, omp_core=2, omp_numthds=32
OMP region: omp_tid=15, omp_core=2, omp_numthds=32
OMP region: omp_tid=16, omp_core=2, omp_numthds=32
OMP region: omp_tid=17, omp_core=3, omp_numthds=32
OMP region: omp_tid=18, omp_core=3, omp_numthds=32
OMP region: omp_tid=19, omp_core=3, omp_numthds=32
OMP region: omp_tid=20, omp_core=3, omp_numthds=32
OMP region: omp_tid=21, omp_core=3, omp_numthds=32
OMP region: omp_tid=22, omp_core=3, omp_numthds=32
OMP region: omp_tid=23, omp_core=3, omp_numthds=32
OMP region: omp_tid=24, omp_core=3, omp_numthds=32
OMP region: omp_tid=25, omp_core=3, omp_numthds=32
OMP region: omp_tid=26, omp_core=3, omp_numthds=32
OMP region: omp_tid=27, omp_core=3, omp_numthds=32
OMP region: omp_tid=28, omp_core=3, omp_numthds=32
OMP region: omp_tid=29, omp_core=3, omp_numthds=32
OMP region: omp_tid=30, omp_core=3, omp_numthds=32
OMP region: omp_tid=31, omp_core=1, omp_numthds=32
```

OpenMP Thread Bindings: Requesting More Threads than Cores

```
mthomas] qsub -v B=true,T=32 batch.omp_info
6739.tuckoo.sdsu.edu
[[mthomas] cat omp_info.o6739 | sort
omp_nprocs=16,  omp_req_thds=32
OMP region: omp_tid=0, omp_core=0, omp_numthds=32
OMP region: omp_tid=1, omp_core=1, omp_numthds=32
OMP region: omp_tid=2, omp_core=2, omp_numthds=32
OMP region: omp_tid=3, omp_core=3, omp_numthds=32
OMP region: omp_tid=4, omp_core=4, omp_numthds=32
OMP region: omp_tid=5, omp_core=5, omp_numthds=32
OMP region: omp_tid=6, omp_core=6, omp_numthds=32
OMP region: omp_tid=7, omp_core=7, omp_numthds=32
OMP region: omp_tid=8, omp_core=8, omp_numthds=32
OMP region: omp_tid=9, omp_core=9, omp_numthds=32
OMP region: omp_tid=10, omp_core=10, omp_numthds=32
OMP region: omp_tid=11, omp_core=11, omp_numthds=32
OMP region: omp_tid=12, omp_core=12, omp_numthds=32
OMP region: omp_tid=13, omp_core=13, omp_numthds=32
OMP region: omp_tid=14, omp_core=14, omp_numthds=32
OMP region: omp_tid=15, omp_core=15, omp_numthds=32
OMP region: omp_tid=16, omp_core=0, omp_numthds=32
OMP region: omp_tid=17, omp_core=1, omp_numthds=32
OMP region: omp_tid=18, omp_core=2, omp_numthds=32
OMP region: omp_tid=19, omp_core=3, omp_numthds=32
OMP region: omp_tid=20, omp_core=4, omp_numthds=32
OMP region: omp_tid=21, omp_core=5, omp_numthds=32
OMP region: omp_tid=22, omp_core=6, omp_numthds=32
OMP region: omp_tid=23, omp_core=7, omp_numthds=32
OMP region: omp_tid=24, omp_core=8, omp_numthds=32
OMP region: omp_tid=25, omp_core=9, omp_numthds=32
OMP region: omp_tid=26, omp_core=10, omp_numthds=32
OMP region: omp_tid=27, omp_core=11, omp_numthds=32
OMP region: omp_tid=28, omp_core=12, omp_numthds=32
OMP region: omp_tid=29, omp_core=13, omp_numthds=32
OMP region: omp_tid=30, omp_core=14, omp_numthds=32
OMP region: omp_tid=31, omp_core=15, omp_numthds=32
```

The Trapezoid Rule for Numerical Integration

Solve the Integral: $\int_a^b F(x)dx$


The Trapezoidal Rule

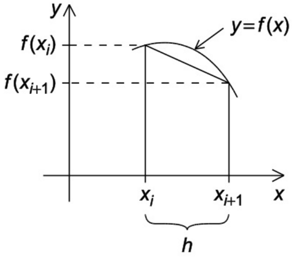
Where $F(x)$ can be any function of x: $f(x^2)$, $f(x^3)$
See Pacheco IPP (2011), Ch3.

Trapezoid Equations

Integral: $\int_a^b f(x)dx$

Area of 1 trapezoid: $= \frac{h}{2} \lfloor f(x_i) + f(x_{i+1}) \rfloor$

Base: $h = \frac{b-a}{n}$



Endpoints: $x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h, \ldots, \quad x_{n-1} = a + (n-1)h, \quad x_c = b$

Sum of Areas: $Area = h \left[\frac{f(x_0)}{2} + f(x_{i+1}) + f(x_{i+1}) + \ldots + f(x_{n-1}) \frac{f(x_n)}{2} \right]$

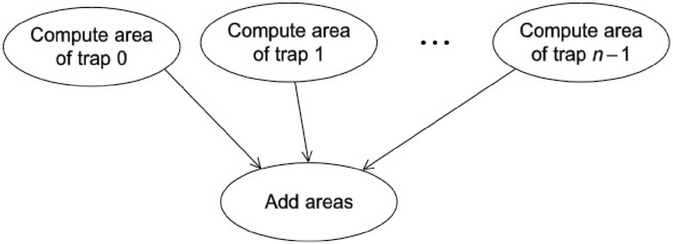
Trapezoid Problem: Serial Algorithm

```
/*  Input: a ,b, n  */
h = (b-a)/n ;
approx - (F(a) + F(b))/2.0
for (i=0; i<= n-1; i++) {
   x_i = a + i*H;
   approx += f(x_i);
}
approx = h* approx
```

Parallelizing the Trapezoidal Rule

PCAM Approach

- Partition problem solution into tasks.
- Identify communication channels between tasks.
- Aggregate tasks into composite tasks.
- Map composite tasks to cores.



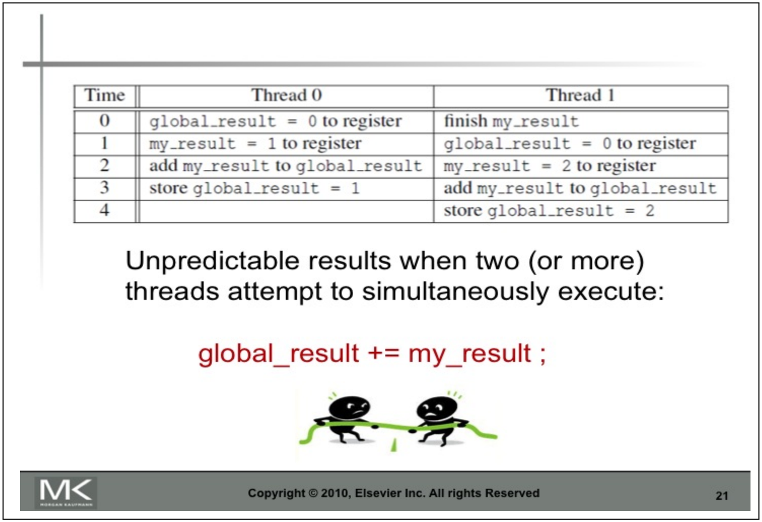
Two types of tasks:

Compute area of 1 trapezoid

Compute area sums

First OpenMP Version of the Trap Alg.

1. We identified two types of tasks:
    a. computation of the areas of individual trapezoids, and
    b. adding the areas of trapezoids.
2. There is no communication among the tasks in the first collection, but each task in the first collection communicates with task 1b.
3. We assumed that there would be many more trapezoids than cores. So we aggregated tasks by assigning a contiguous block of trapezoids to each thread (and a single thread to each core).

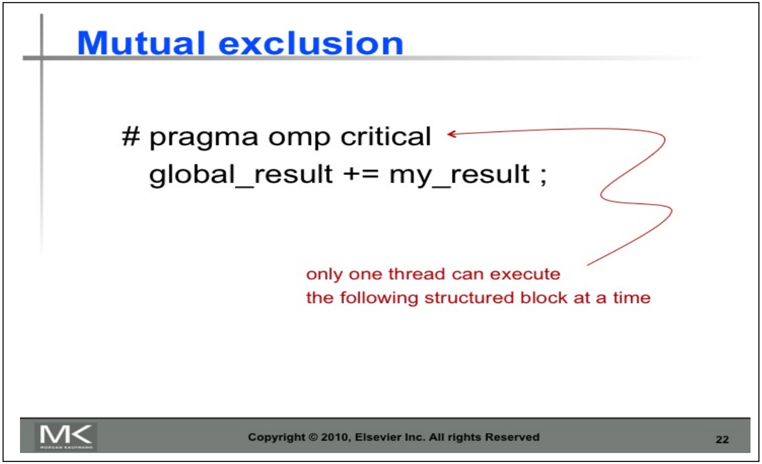


Time	Thread 0	Thread 1
0	global_result = 0 to register	finish my_result
1	my_result = 1 to register	global_result = 0 to register
2	add my_result to global_result	my_result = 2 to register
3	store global_result = 1	add my_result to global_result
4		store global_result = 2

Unpredictable results when two (or more) threads attempt to simultaneously execute:

global_result += my_result ;

Results in a race condition

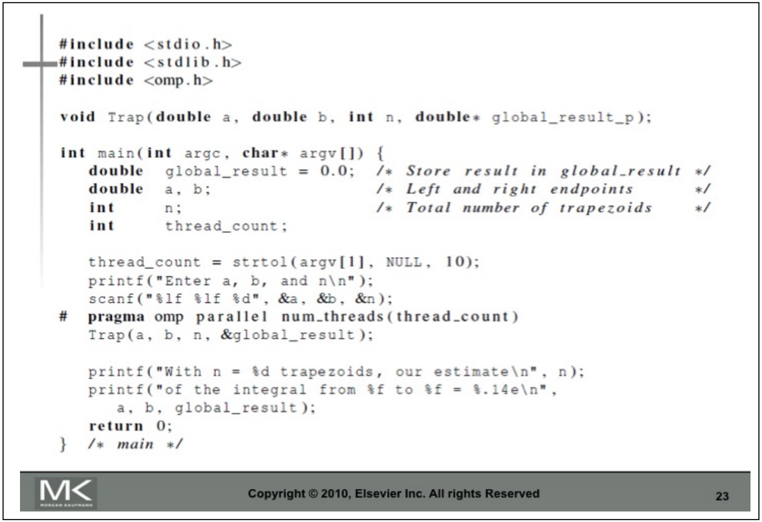


Mutual exclusion

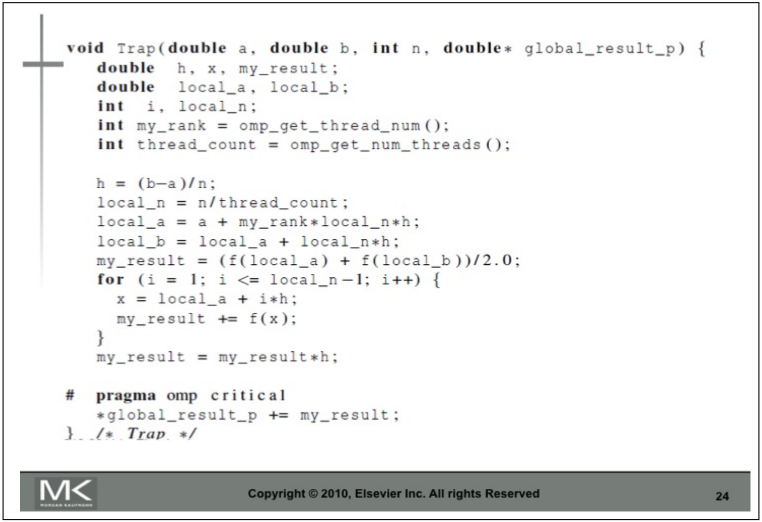
```
# pragma omp critical
  global_result += my_result ;
```

only one thread can execute
the following structured block at a time

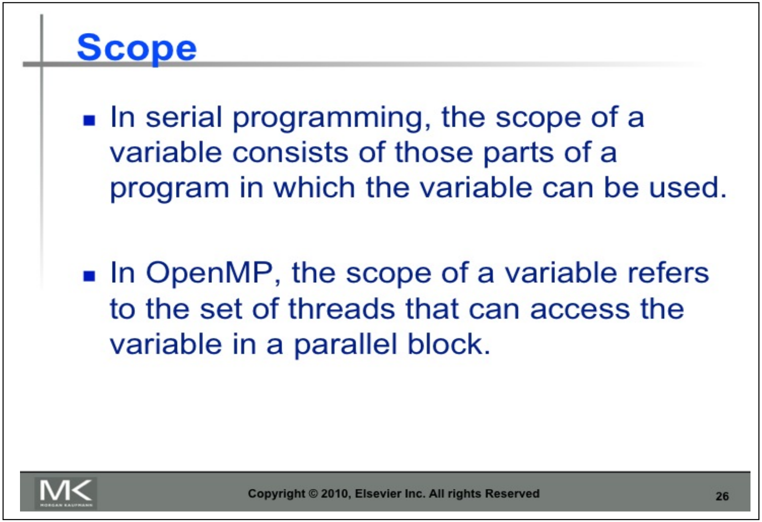
critical directive tells compiler that system needs to provide mutually exclusive access control for the block of code.



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

void Trap(double a, double b, int n, double* global_result_p);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

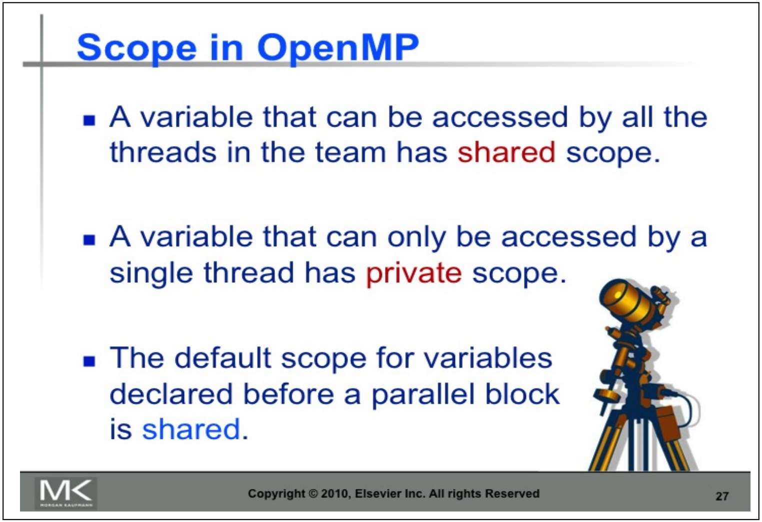


```
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```



Scope

- In serial programming, the scope of a variable consists of those parts of a program in which the variable can be used.
- In OpenMP, the scope of a variable refers to the set of threads that can access the variable in a parallel block.



Scope in OpenMP

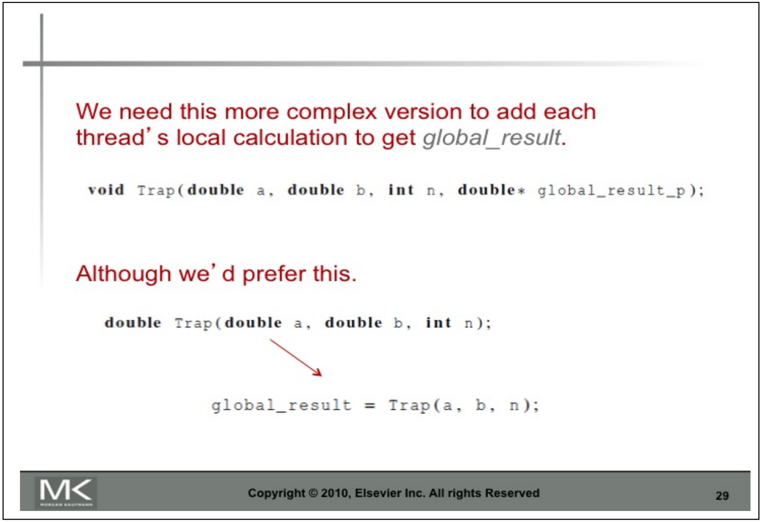
- A variable that can be accessed by all the threads in the team has shared scope.
- A variable that can only be accessed by a single thread has private scope.
- The default scope for variables declared before a parallel block is shared.

- for C, variables defined in *main* have global; variables defined in a *function* have function scope.
- for OpenMP: the scope of a variable is associated with the set of threads that can access the variable in a parallel block.
- **shared scope**:
  - the default scope for variables defined outside a parallel block
  - e.g. *global_results* was declared in *main*, so it is shared by all threads
- **private scope**:
  - a variable that can only be accessed by a single thread
  - The default scope for variables declared inside a parallel block is private (e.g. all vars in defined in Trap).

```
   int main(int argc, char* argv[]) {
/* Store result in global_result */
   double  global_result = 0.0;
    /* Left and right endpoints */
   double  a, b;
   int     n;    /* Total number of trapezoids*/
   int     thread_count;

   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
   if (n % thread_count != 0) Usage(argv[0]);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

```
 * Function:    Trap
 * Purpose:     Use trapezoidal rule to
 *         estimate definite integral
 * Input args:
 *    a: left endpoint
 *    b: right endpoint
 *    n: number of trapezoids
 *    global_result_p: pointer to global trap sum
 * Output arg:
 * integral: estimate of integral from a to b of f(x)
 */
  void Trap(double a, double b, int n,
                double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

#  pragma omp critical
   *global_result_p += my_result;
}  /* Trap */
```

OpenMP: Reduction Clause

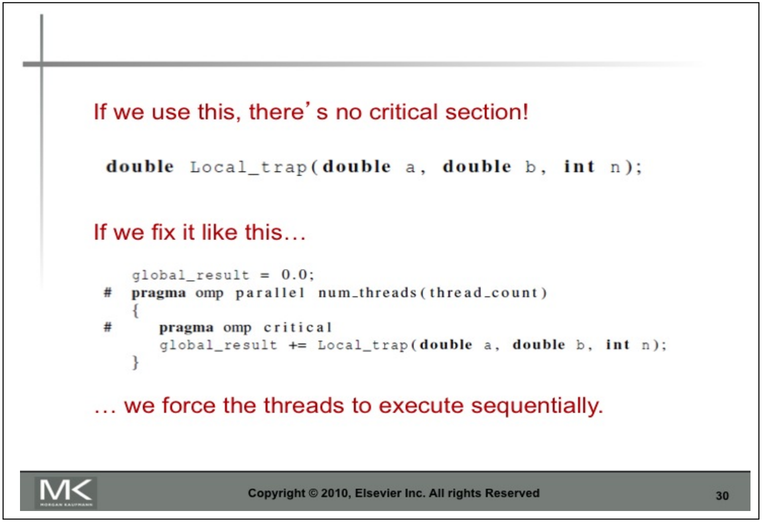


We need this more complex version to add each thread's local calculation to get *global_result.*

```
void Trap(double a, double b, int n, double* global_result_p);
```

Although we'd prefer this.

```
double Trap(double a, double b, int n);

            global_result = Trap(a, b, n);
```



If we use this, there's no critical section!

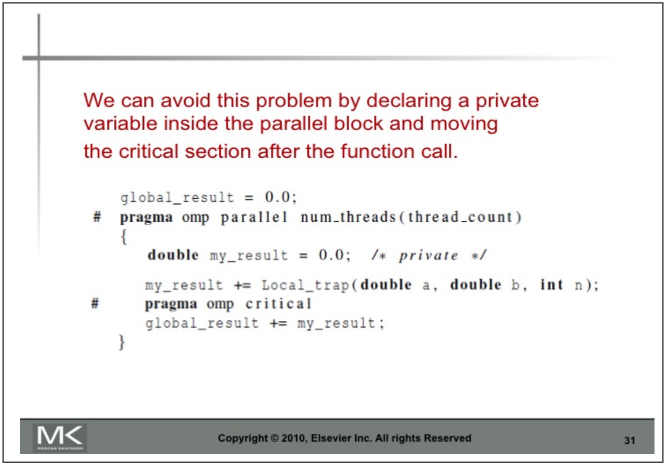
```
double Local_trap(double a, double b, int n);
```

If we fix it like this…

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
#      pragma omp critical
       global_result += Local_trap(double a, double b, int n);
    }
```

… we force the threads to execute sequentially.

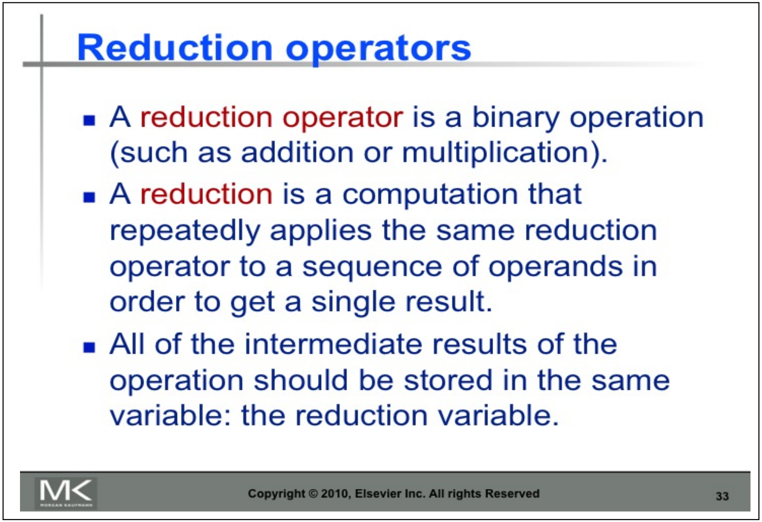
Local_Trap does not have reference to the global variable global_result



We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

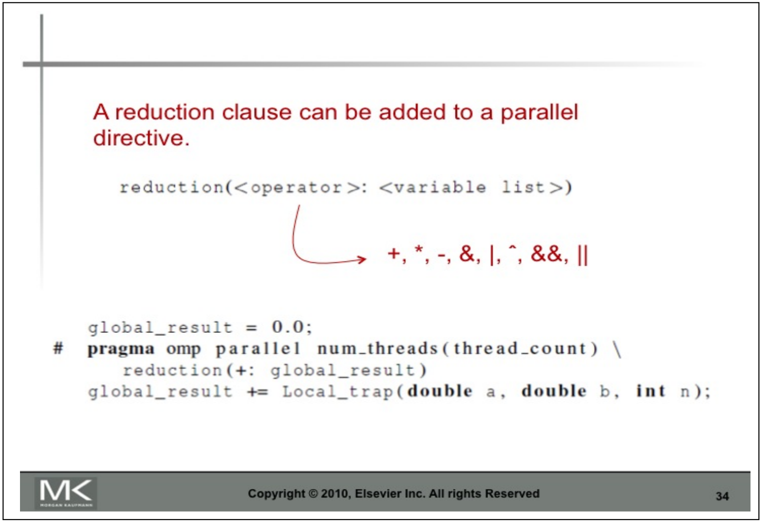
```
      global_result = 0.0;
#     pragma omp parallel num_threads(thread_count)
      {
         double my_result = 0.0;  /* private */
         my_result += Local_trap(double a, double b, int n);
#        pragma omp critical
         global_result += my_result;
      }
```

Notes: the call to Local_Trap is inside the parallel block, but outside critical section; my_result is private to each thread



Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.



A reduction clause can be added to a parallel directive.

```
reduction(<operator>: <variable list>)
```

$\longrightarrow$ +, *, -, &, |, ^, &&, ||

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(double a, double b, int n);
```

A few comments

- OpenMP (1) creates private thread variable, (2) stores result for thread, and (3) creates critical section block.
- subtraction ops are not guaranteed (not associative or commutative):

$$\begin{aligned} &result = 0; \\ &for\,(i = 1; \quad i \leq 4; \quad i++) \\ &result \; -= \; i; \end{aligned}$$

- floating point arithmetic is not associative, so results are not guaranteed:

$$a + (b + c) \text{ may not equal } (a + b) + c$$