A Journey Through the World of OpenMP

Mission-Critical Computing
NSF CENTER FOR SPACE, HIGH-PERFORMANCE, AND RESILIENT COMPUTING (SHREC)

PCA Lecture

Dr. Alan George
Mickle Chair Professor of ECE
University of Pittsburgh

David Langerman
Evan Gretok
Research Students
University of Pittsburgh

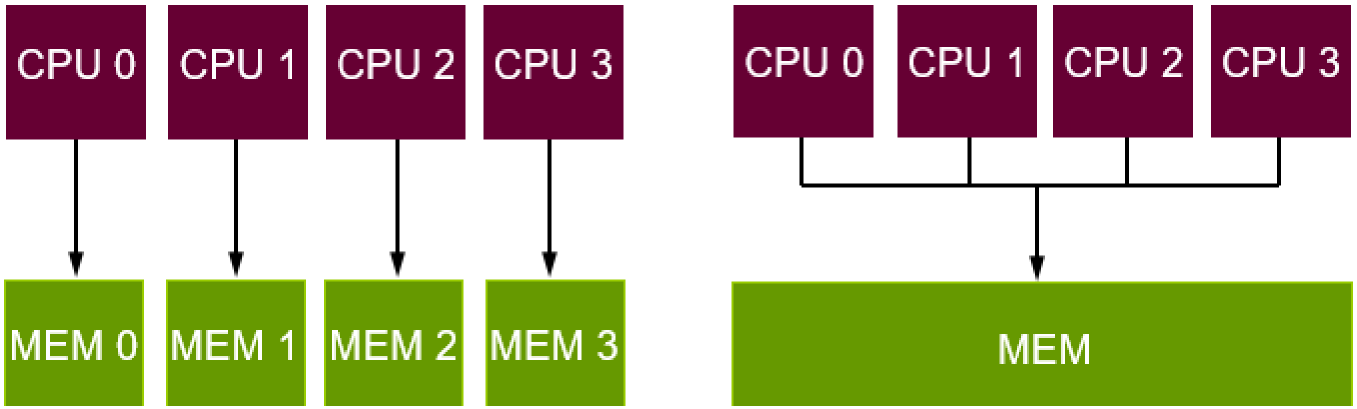
University of Pittsburgh
BYU BRIGHAM YOUNG UNIVERSITY
Virginia Tech
UF UNIVERSITY of FLORIDA

Overview

- Introduction to OpenMP
- Division of Work
- Serial Considerations
- Granularity and Scheduling
- Extra Fun

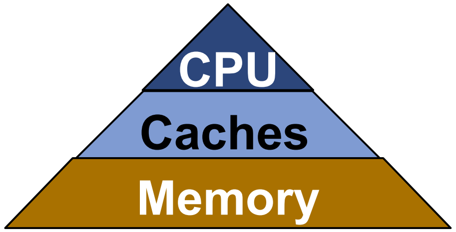
Shared Memory

- **Distributed Memory Programming**
  - Start multiple processes on multiple networked systems
  - Processes carry out work and communicate through message-passing
  - Processes coordinate through message-passing or synchronization
- **Shared Memory Programming**
  - Start a single process on one system and fork threads
  - Threads each carry out work and communicate through shared memory
  - Threads coordinate through synchronization

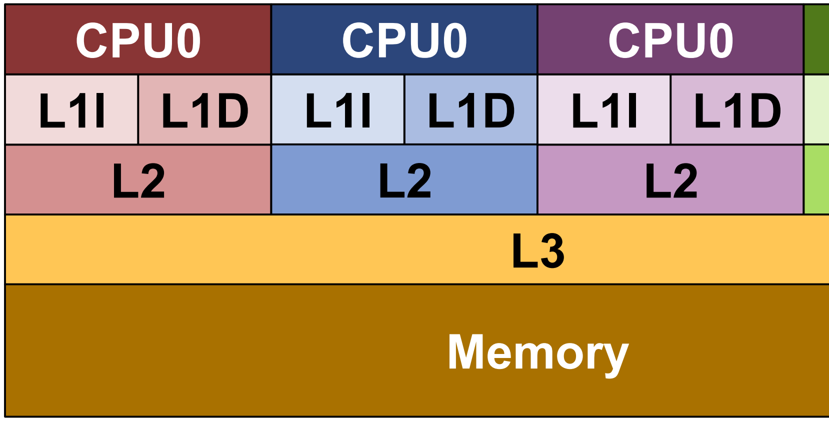


Distributed Memory **Shared Memory**

Memory Hierarchy



- **Cache Pyramid**
  - Simple illustration of concept
  - Higher is smaller and faster
- **Reality**
  - Multicore processors share uniquely
    - Shared L3 cache
    - Shared main memory space
    - Hyperthreading may share L2



- **(A)symmetric Multiprocessing**
  - Multiple cores in one or both states
    - Symmetric runs same operating system/programs on multiple cores
    - Asymmetric may run entirely different operating systems on each core
- **A little too local for Marco Polo…**

OpenMassivePastry

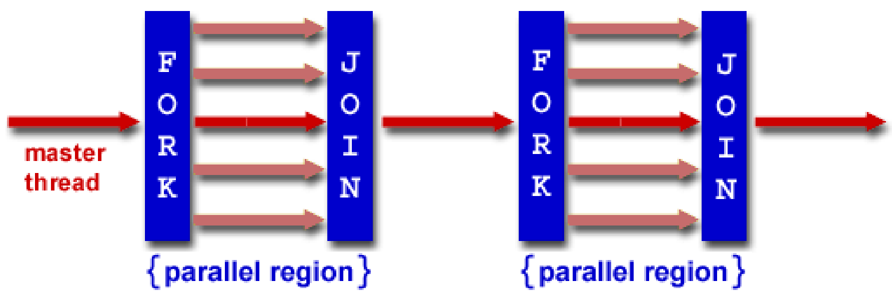
OpenMassivePasture

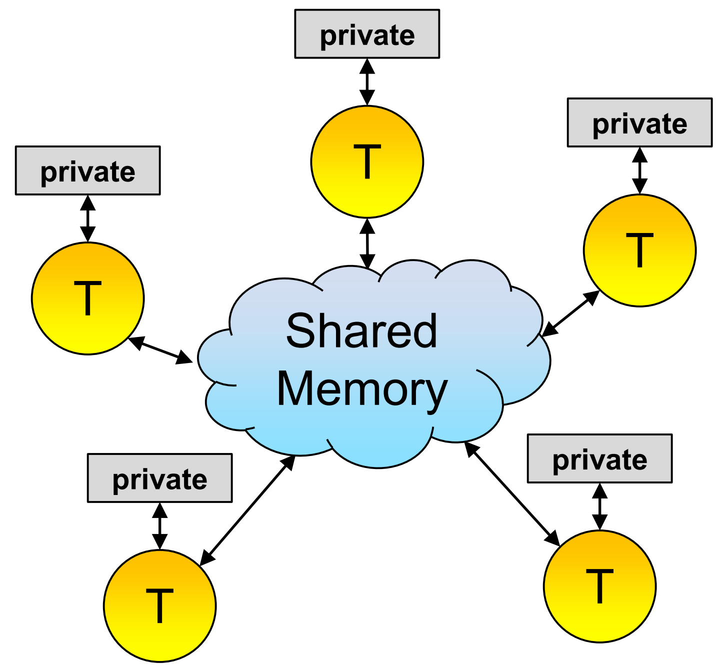
OpenMP Introduction

- **What does OpenMP stand for?**
  - **Open** specifications for **Multi-Processing** via collaborative work between interested parties from the hardware and software industry, government, and academia
- **OpenMP is an Application Programming Interface (API)**
  - Used to explicitly direct ***multi-threaded, shared-memory parallelism***
  - Components: compiler directives, library routines, environment variables
- **OpenMP is directive-based**
  - Directives invoke parallel computations on shared-memory multiprocessors
- **OpenMP API specified for C/C++ and Fortran**
- **Portable: supported by HP, IBM, Intel, SGI, SUN, and others**
- **De facto standard for writing shared memory programs**

OpenMP Basics

- **OpenMP uses fork-join model of parallel execution**
  - OpenMP program begins with single **master thread**
  - Master thread executes sequentially until a **parallel region** is encountered
  - Creates **team of parallel threads** (**FORK**) when parallel region is encountered
  - When parallel region ends the threads synchronize and terminate (**JOIN**)
  - This leaves only the master thread





- **Thread-based programming model**
  - All threads have access to **shared memory**
  - Each thread has access to its own **private memory**
- **Synchronization mostly implicit**
  - More on synchronization later

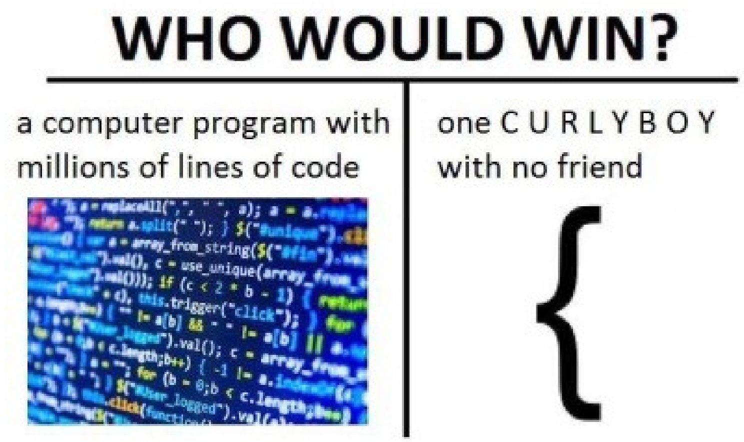
OpenMP Compilation/Execution

- **Compile the Program**
  - Terminal command: “gcc -fopenmp <filename>.c –o <executable_name>”
  - Compiles <filename>.c and outputs <executable_name>
  - Be sure to add the –fopenmp compiler flag
    - Everything needed for OpenMP is already in gcc, just waiting to be turned on

- **Execute the Program**
  - Terminal command: “./<executable_name> <arguments>”
  - Can pass in number of cores among other arguments

Basic OpenMP Pragmas

- **Pragma OMP Parallel**
  - OpenMP operates through "#pragma omp" compiler directives
  - "pragma omp parallel" is used to specify the start of a parallel region
  - Each thread will execute given code inside of given scope



Code Example

```c
#include <omp.h>

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

  #pragma omp parallel
  {
    printf( "Hello from a random thread!\n" );
  }

  return 0;
}
```

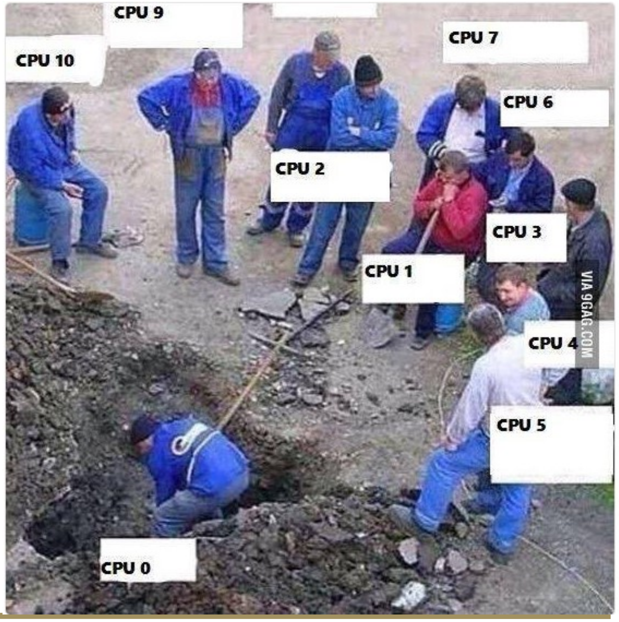
Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./example
Hello from a random thread!
Hello from a random thread!
Hello from a random thread!
Hello from a random thread!
$
```

Pragma OMP Parallel Parameters

- **private(…)**
  - List of variables to keep private, accessible to one thread
- **shared(…)**
  - List of variables to make shared, accessible to each and every thread
- **default(none | shared)**
  - Makes variables private or shared by default if not specified
- **num_threads(#)**
  - Best way to set number to cores to be utilized

- **All variables called out must be previously declared!**



OpenMP Functions

- **int omp_get_num_threads( )**
  - Returns number of threads being used in parallel regions
- **int omp_get_max_threads( )**
  - Returns maximum number of threads this system can process in parallel
- **int omp_get_thread_num( )**
  - Returns the thread number of the thread
  - Thread numbers of indexed starting with 0 up to (omp_get_num_threads()-1)
- **void omp_set_num_threads(int num_threads)**
  - Sets the number of threads (cores) to use in subsequent parallel regions
  - Can cause unexpected results; best to use "num_threads( )" in pragma
- **And more!**

OpenMP Hello World Program

Code Example

```c
#include <omp.h>

int main( int argc, char **argv ) {
  int num_threads = 0;  // Number of Threads
  int thread_id   = 0;  // ID Number of Running Thread

  // Fork the threads, giving each one a private copy of:
  #pragma omp parallel private( num_threads, thread_id )
  {
    // Get the thread number
    thread_id     = omp_get_thread_num( );
    printf( "Hello World from Thread %d\n", thread_id );

    // Have master print the total number of threads used.
    if( thread_id == 0 ) {
      num_threads = omp_get_num_threads( );
      printf( "Number of Threads = %d\n", num_threads );
    }
  }  // All of the threads rejoin the master thread.
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp...
(builds)
$ ./example
Hello World from Thread 0
Hello World from Thread 1
Hello World from Thread 2
Hello World from Thread 3
Number of Threads = 4
$
```

Note that order is never guaranteed.

Shared/Private/Default Example

Code Example

```
#include <omp.h>

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

  int id = 0; // Variables declared before pragma.
  int i  = 0;
  int m  = 0;
  int x  = 2;

  #pragma omp parallel private( id, i ) shared ( m ) \
                       default( shared )
  {
    id = omp_get_thread_num( );
    if( id == 0 ) {  // Only id = 0 master makes changes.
      i = 3;    // Change i, private, just this thread.
      m = 17;   // Change m, shared, all threads.
      x++;      // Change x, default shared, all.
    }
    printf( "Thread %d: %d %d %d\n", id, i, m, x );
  }

  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./example
Thread 0: 3 17 3
Thread 1: 0 17 3
Thread 2: 0 17 3
Thread 3: 0 17 3
$
```

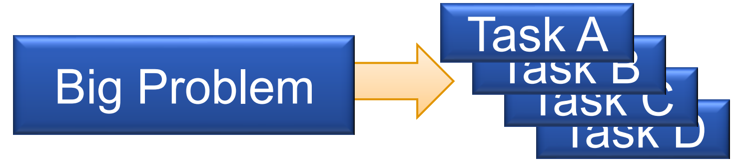
Note that order of threads is never guaranteed.

Strange things can happen.
O_O

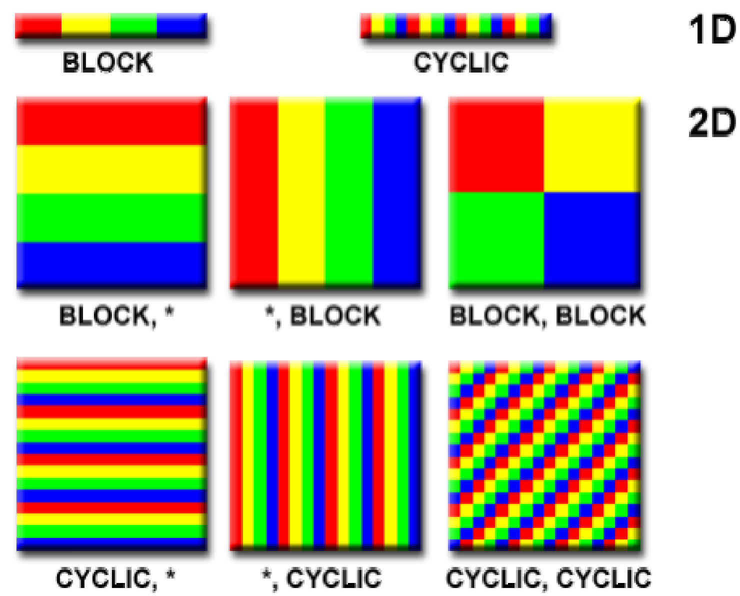
Division of Work

Smile and nod if you are paying attention.

Division of Work



- **Using OpenMP to divide work**
  - Creation of parallel region with additional directive of task to divide
  - Simplest is “pragma omp parallel for”
  - Will discuss “single” and “sections” later
- **Type of division depends on application, scheduling, and desires**
  - Vector addition may be best served by cyclic
  - Image processing often lends itself to even block division over an axis
- **Number of divisions depends on cores**
- **Partitioning methods**
  - In domain decomposition each task performs the same function and gets a portion of the data (our focus, for now)
  - In functional decomposition, each task performs a different function (cool, later)



Parallel For

- **Pragma OMP Parallel For**
  - Append “pragma omp for” before for loop inside “pragma omp parallel”
  - A work-sharing construct, divides iterations of a for loop between cores
  - Must ensure there are no “loop carry” dependencies between iterations
  - Implicit barrier at conclusion of for loop - “no thread gets left behind”

Code Example

```c
#include <omp.h>
int main( int argc, char **argv ) {
  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations
  #pragma omp parallel shared( n ) private( i )
  {
    #pragma omp for
    for( i = 0; i < n; i++ ) {
      printf( “Thread %d of %d – Iteration %d\n”,
              omp_get_thread_num( ),
              omp_get_max_threads( ), i         );
    }  // Note scopes – no “{}” with for.
  }
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./basicFor
Thread 0 of 4 – Iteration 0
Thread 0 of 4 – Iteration 1
Thread 0 of 4 – Iteration 2
Thread 1 of 4 – Iteration 3
Thread 1 of 4 – Iteration 4
Thread 1 of 4 – Iteration 5
Thread 2 of 4 – Iteration 6
Thread 2 of 4 – Iteration 7
Thread 2 of 4 – Iteration 8
Thread 3 of 4 – Iteration 9
Thread 3 of 4 – Iteration 10
Thread 3 of 4 – Iteration 11
$
```

Parallel For – One Line

Code Example

```
#include <omp.h>

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

  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations

  #pragma omp parallel for shared( n ) private( i )

    for( i = 0; i < n; i++ ) {
      printf( “Thread %d of %d – Iteration %d\n”,
              omp_get_thread_num( ),
              omp_get_max_threads( ), i           );
    }  // Note scopes – no “{}” at all!

  return 0;

}
```

Output (4 Cores)

```
$ gcc –fopenmp ...
(builds)
$ ./oneLineFor
Thread 0 of 4 – Iteration 0
Thread 0 of 4 – Iteration 1
Thread 0 of 4 – Iteration 2
Thread 1 of 4 – Iteration 3
Thread 1 of 4 – Iteration 4
Thread 1 of 4 – Iteration 5
Thread 2 of 4 – Iteration 6
Thread 2 of 4 – Iteration 7
Thread 2 of 4 – Iteration 8
Thread 3 of 4 – Iteration 9
Thread 3 of 4 – Iteration 10
Thread 3 of 4 – Iteration 11
$
```

Wall Clock Time

- **double omp_get_wtime( )**
  - Returns double-precision floating-point elapsed wall-clock time in seconds
  - Called before and after parallel code
  - Simple arithmetic to determine overall elapsed time

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {

   double start = 0.0;   // Start Time
   double end   = 0.0;   // End Time
   double total = 0.0;   // Total Execution Time

   start        = omp_get_wtime( );   // Get Start Measure
   #pragma omp parallel               // Perform Task
   {
     printf( "Hello from a random thread!\n" );
   }
   end          = omp_get_wtime( );   // Get End Measure
   total        = end - start;        // Calculate Total
   printf( "Execution Time: %lf", total );
   return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./ompTime
Hello from a random thread!
Hello from a random thread!
Hello from a random thread!
Hello from a random thread!
Execution Time: 0.00023
$
```

Varying Cores

- **Varying number of cores per run**
  - Pass in number of cores as an argument when invoking the program
  - Parse that input argument within the code and convert the string to integer
  - Use the num_threads(x) parameter discussed before to set for the region

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int    cores = 0;    // Number of Cores
  double start = 0.0;  // Start Time
  double end   = 0.0;  // End Time
  double total = 0.0;  // Total Execution Time
  cores        = atoi( argv[1] );
  start        = omp_get_wtime( );  // Get Start Measure
  #pragma omp parallel num_threads( cores )
  {
    printf( “Hello from a random thread!\n” );
  }
  end          = omp_get_wtime( );  // Get End Measure
  total        = end - start;       // Calculate Total
  printf( “Execution Time: %lf”, total );
  return 0;
}
```

Output (4 Cores)

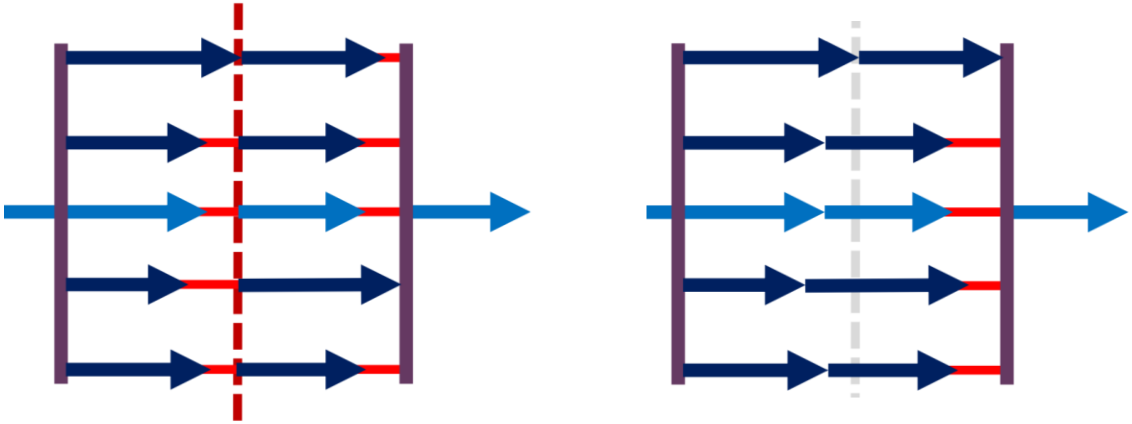
```
$ gcc -fopenmp ...
(builds)
$ ./howMany 4
Hello from a random thread!
Hello from a random thread!
Hello from a random thread!
Hello from a random thread!
Execution Time: 0.000460
$ ./howMany 2
Hello from a random thread!
Hello from a random thread!
Execution Time: 0.000312
$ ./howMany 206

  ???
```

What Time Really Means

- **Serial Baseline**
  - Execution time of serial application implemented without OpenMP
- **Execution Time**
  - Per-core execution time measured with OpenMP for each number of cores
- **Speedup = $Time_{Serial}$ / $Time_{Parallel}$**
  - Unitless ratio of serial execution time over parallel execution time
  - Goal is to realize best possible speedup doing useful work
- **Parallel Efficiency = Speedup / Cores**
  - Percentage measure of how efficiently additional cores are being used
- **Useful Work**
  - Operations and functions that would have been conducted in serial program
- **Parallel Overhead**
  - Additional operations for OpenMP, sharing, timing - minimize

Barriers

- **Concept**
  - All threads wait at barrier until all others reach it
  - Helps ensure operations are complete before moving on
- **Implicit Barriers**
  - Using a parallel for pragma implies a barrier at the end
  - This is used to ensure the whole loop is complete before moving on
  - Can append a "nowait" to remove this implied barrier
  - May improve speed, but must watch out for dependencies



Parallel For

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations
  #pragma omp parallel shared( n ) private( i )
  {
    #pragma omp for
    for( i = 0; i < n; i++ ) {
      printf( "Thread %d of %d – Iteration %d\n",
              omp_get_thread_num( ),
              omp_get_max_threads( ), i          );
    }

    #pragma omp for
    for( i = 0; i < n; i++ ) {
      printf( "Thread %d of %d – Iteration %d\n",
              omp_get_thread_num( ),
              omp_get_max_threads( ), i          );
    }

  }
  return 0;
}
```

Output (4 Cores)

```
$ gcc –fopenmp ...
(builds)
$ ./noWait
Thread 0 of 4 – Iteration 0
Thread 0 of 4 – Iteration 1
Thread 0 of 4 – Iteration 2
Thread 1 of 4 – Iteration 3
Thread 1 of 4 – Iteration 4
Thread 1 of 4 – Iteration 5
Thread 2 of 4 – Iteration 6
Thread 2 of 4 – Iteration 7
Thread 2 of 4 – Iteration 8
Thread 3 of 4 – Iteration 9
Thread 3 of 4 – Iteration 10
Thread 3 of 4 – Iteration 11
Thread 0 of 4 – Iteration 0
Thread 0 of 4 – Iteration 1
Thread 0 of 4 – Iteration 2
Thread 1 of 4 – Iteration 3
Thread 1 of 4 – Iteration 4
Thread 1 of 4 – Iteration 5
Thread 2 of 4 – Iteration 6
Thread 2 of 4 – Iteration 7
...
$
```

Parallel For Nowait

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations
  #pragma omp parallel shared( n ) private( i )
  {
    #pragma omp for nowait
    for( i = 0; i < n; i++ ) {
      printf( "Thread %d of %d - Iteration %d\n",
              omp_get_thread_num( ),
              omp_get_max_threads( ), i           );
    }

    #pragma omp for nowait
    for( i = 0; i < n; i++ ) {
      printf( "Thread %d of %d - Iteration %d\n",
              omp_get_thread_num( ),
              omp_get_max_threads( ), i           );
    }

  }
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./noWait
(Similar to previous, but orders
merge and times overlap.  Please
explore!)
$

Significantly faster speeds!
```

Parallel For Nowait with Barrier

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations
  #pragma omp parallel shared( n ) private( i )
  {
    #pragma omp for nowait
    for( i = 0; i < n; i++ ) {
      printf( "Thread %d of %d - Iteration %d\n",
              omp_get_thread_num( ),
              omp_get_max_threads( ), i           );
    }

    #pragma omp barrier

    #pragma omp for nowait
    for( i = 0; i < n; i++ ) {
      printf( "Thread %d of %d - Iteration %d\n",
              omp_get_thread_num( ),
              omp_get_max_threads( ), i           );
    }

  }
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./noWait
(Barrier overrides noWait.
While order is not guaranteed,
there will be no overlap.)
$

Roughly the same performance as
standard parallel for with
implicit barrier.
```

Ordered

- **Guarantees proper execution order…but at what cost?**
  - Iterations executed serially, one thread at a time…no parallelism
  - Only applicable to parallel for and each iteration can have only one
  - Requires "ordered" clause on initial pragma and inside at specific function

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations
  #pragma omp parallel for ordered \
              shared( n ) private( i )
  for( i = 0; i < n; i++ ) {
    #pragma omp ordered
    printf( "Thread %d of %d – Iteration %d\n",
              omp_get_thread_num( ),
              omp_get_max_threads( ), i         );
    }
  }
  return 0;
}
```

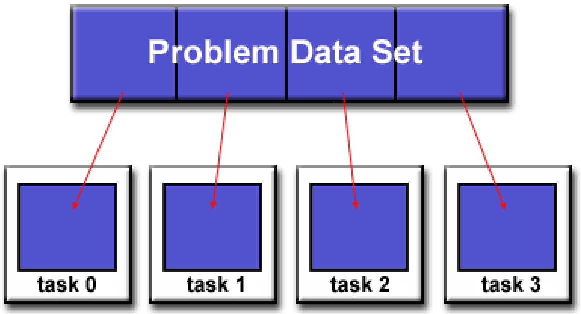
Output (4 Cores)

```
$ gcc ...
(builds)
$ ./ordered
Thread 0 of 4 – Iteration 0
Thread 0 of 4 – Iteration 1
Thread 0 of 4 – Iteration 2
Thread 1 of 4 – Iteration 3
Thread 1 of 4 – Iteration 4
Thread 1 of 4 – Iteration 5
Thread 2 of 4 – Iteration 6
Thread 2 of 4 – Iteration 7
Thread 2 of 4 – Iteration 8
Thread 3 of 4 – Iteration 9
Thread 3 of 4 – Iteration 10
Thread 3 of 4 – Iteration 11…
$
```

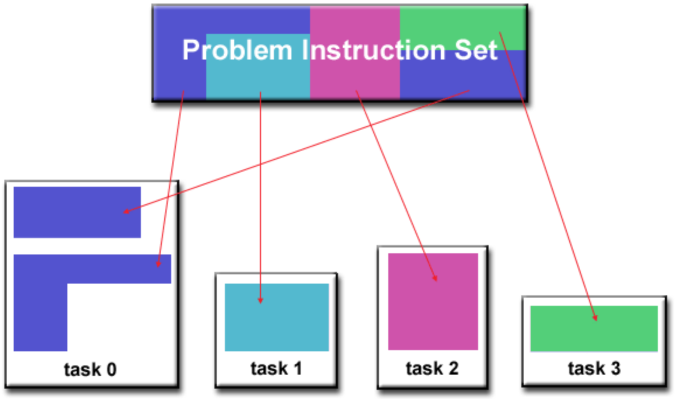
Serial Considerations

Raise your left hand if you are still paying attention.

Serial Considerations



- **Why back to serial?**
  - Overcome things that do not parallelize well
  - Divide functions between cores
  - Cases where file operations or output must be serial
- **One-for-all or all-for-one?**
  - Single core can perform some functions while other cores work together
  - Each core can work independently on its own function
  - Could pipeline cores so that data moves between them in series



Single

- **Pragma OMP Single**
  - Runs given code on a single core
  - Can be coupled with parallel for other tasks running on other cores
  - Like for, work-sharing construct with an implicit barrier at end

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf( "Thread %d of %d - Single\n",
              omp_get_thread_num( ),
              omp_get_max_threads( )        );
    }
  }
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./single
Thread 0 of 4 - Single
$
```

Sections

- **Pragma OMP Sections**
  - Creates section to run on single thread
  - Can be used to break problems into separate functions that run in parallel
  - Like for, work-sharing construct with an implicit barrier at end

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int i = 0;   // Loop Iterator
  int n = 12;  // Number of Iterations
  #pragma omp parallel
  {
    #pragma omp sections
    {
      #pragma omp section
      functionA( );

      #pragma omp section
      functionB( );
    }
  }
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./sections
(Words from Function A)
(Words from Function B)
$
```

Master

- **Only the master thread does this**
  - Can also accomplish with if(omp_get_thread_num( ) == 0)
  - Reduces amount of code, more tidy
  - No implicit barrier for the other threads

Code Example

```c
#include <omp.h>
int main( int argc, char **argv )  {
  int num_threads = 0;  // Number of Threads
  int thread_id   = 0;  // ID Number of Running Thread
  #pragma omp parallel private( num_threads, thread_id )
  {
    thread_id = omp_get_thread_num( );
    printf( "Hello World from Thread = %d\n", thread_id );
    #pragma omp master
    {
      num_threads = omp_get_num_threads( );
      printf( "Number of Threads = %d\n", num_threads );
    }
  }
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./example
Hello World from Thread = 0
Hello World from Thread = 1
Hello World from Thread = 2
Hello World from Thread = 3
Number of Threads = 4
$
```

Note that order is never guaranteed.

Critical

- **Concept**
  - Ensures only one core can access shared variable at a time
  - Helps avoid race conditions where value is incorrectly altered
- **Race Condition**
  - Thread one gets data and changes it
  - Thread two gets data
  - Thread one stores data
  - Thread two performs operation on old data and overwrites thread one’s store

Thread A		Thread B		
Instruction	Register	Instruction	Register	Count
		LOAD Count	10	10
LOAD Count	10			10
ADD #1	11			10
STORE Count	11			11
		SUB #1	9	11
		STORE Count	9	9

Critical

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int *a        = malloc( 25 * sizeof( int ) );
  int  i        = 0;   // Loop Iterator
  int  n        = 25;  // Number of Iterations
  int  localSum = 0;   // Private Local Sum
  int  totalSum = 0;   // Shared Total Sum
  int  thread   = 0;   // Thread Number
  // Fill Array with Values 1 to 25
  for( i = 0; i < n; i++ ) {
    a[i] = i + 1;
  }
  #pragma omp parallel                 \
              shared( n, a, totalSum ) \
              private( thread, localSum )
  {
    thread   = omp_get_thread_num( );
    localSum = 0;
    #pragma omp for
    for( i = 0; i < n; i++ ) {
      localSum += a[i];
    }
    #pragma omp critical( totalSum )
    {
      totalSum += localSum;
      printf( "Thread %d has local sum %d and adds
               to total sum %d.\n",
               thread, localSum, totalSum );
    }
  }
```

Output (4 Cores)

```
$ gcc ...
(builds)
$ ./noWait
Thread 0 has local sum 99 and
adds to total sum 99.
Thread 1 has local sum 135 and
adds to total sum 234.
Thread 2 has local sum 28 and
adds to total sum 262.
Thread 3 has local sum 63 and
adds to total sum 325.
Total sum at end is 325.
$
```

Problem?

Atomic

- **Specifies a specific memory location to update atomically**
  - Does not need to worry about overhead to lock for multiple threads
  - Operation in hardware guarantees no interference
  - Consider machine code that allows immediate incrementation, assignment
- **A number of operations allowed**
  - Arithmetic increment (++, +=), decrement (--, -=)
  - Arithmetic assignment (+=, -=, *=, /=)
  - Bit shifting assignment (<<=, >>=)
  - Bitwise logical assignment (&=, |=, ^=)
- **Watch out**
  - Specific syntax
  - Small set of operations
  - Single statement

Expr1++	++expr1
Expr1--	--Expr1
Expr1 += expr2	Expr1 -= expr2
Expr1 *= expr2	Expr1 /= expr2
Expr1 <<= expr2	Expr1 >>= expr2
Expr1 &= expr2	Expr1 ^=expr2
Expr1 |= expr2	

Atomic Example

Code Example

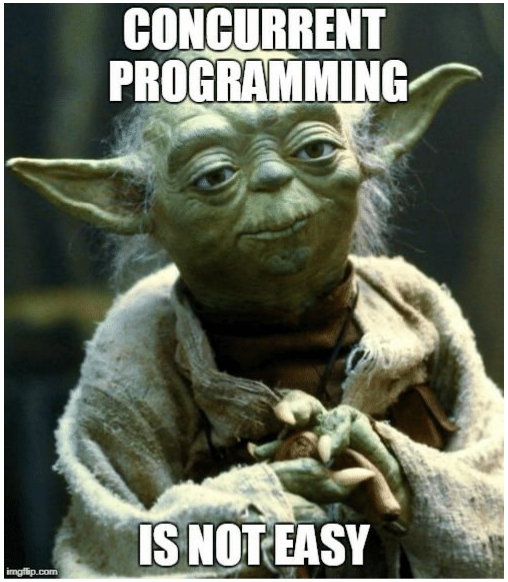
```
#include <omp.h>
int main( int argc, char **argv ) {
  int *a        = malloc( 25 * sizeof( int ) );
  int  i        = 0;   // Loop Iterator
  int  n        = 25;  // Number of Iterations
  int  localSum = 0;   // Private Local Sum
  int  totalSum = 0;   // Shared Total Sum
  // Fill Array with Values 1 to 25
  for( i = 0; i < n; i++ ) {
    a[i] = i + 1;
  }
  #pragma omp parallel                 \
              shared( n, a, totalSum ) \
              private( localSum )
  {
    localSum = 0;
    #pragma omp for
    for( i = 0; i < n; i++ ) {
      localSum += a[i];
    }
    #pragma omp atomic
    totalSum += localSum;
  }
  printf( "Total sum at end is %d.\n", totalSum )
  return 0;
}
```

Output (4 Cores)

```
$ gcc ...
(builds)
$ ./atomic
Total sum at end is 325.
Time:  0.000244141
$
```

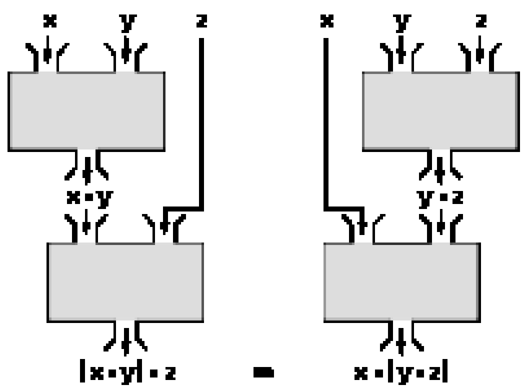
Critical vs. Atomic

- **Critical**
  - Surround any block of code, only one thread at a time can enter
  - Significant overhead entering and exiting, must block other threads
  - Can have names to group and categorize access restrictions
    - Unnamed criticals are considered the same across the entire program
    - May name critical sections to let each separate names to execute in parallel
    - Lets you do some pretty interesting things
      - Criticals with different names performing similar tasks
      - Criticals with the same name performing very different tasks
      - Must be sure you are protecting things appropriately and approaching options correctly
- **Atomic**
  - Significantly less overhead
  - Takes advantage of hardware if platform supports it
  - Does not block atomic operations on different variables
  - Small set of operations supported by atomic



Reduction

- **Handles combining values from parallel threads**
  - Private copy for each thread is operated on by each in parallel region
  - When complete, reduction applies operation to each of these private copies
  - Result ends up in specified global shared variable
- **A number of operations allowed**
  - Arithmetic addition (+), multiplication (*), subtraction (-)
  - Bitwise logic AND (&), OR (|), XOR (^)
  - Logical AND (&&), OR (||)
  - Initial conditions allow for first add from 0, mult from 1
- **Watch out**
  - Must be valid operation for this variable
  - Not shared or private
  - Associativity not guaranteed



Operator	Initial value
+	0
*	1
-	0

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Reduction Example

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int *a   = malloc( 25 * sizeof( int ) );
  int  i   = 0;   // Loop Iterator
  int  n   = 25;  // Number of Iterations
  int  sum = 0;   // Only One Sum
  // Fill Array with Values 1 to 25
  for( i = 0; i < n; i++ ) {
    a[i]   = i + 1;
  }
  #pragma omp parallel for                    \
              shared( n, a ) private( thread ) \
              reduction( + : sum )
  {
    for( i = 0; i < n; i++ ) {
      sum += a[i];
    }
  }
  printf( "Total sum at end is %d.\n", totalSum )
  return 0;
}
```

Output (4 Cores)

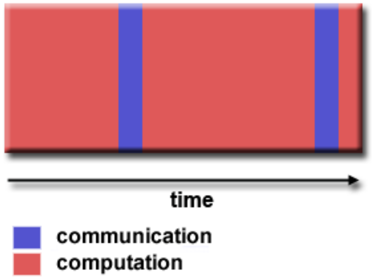
```
$ gcc -fopenmp ...
(builds)
$ ./reduction
Total sum at end is 325.
$
```

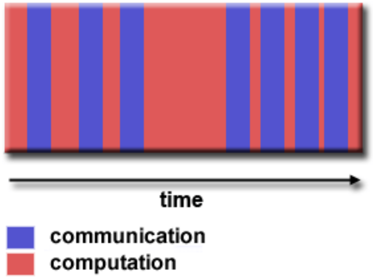
Granularity and Scheduling

Wave both hands if you are paying attention.

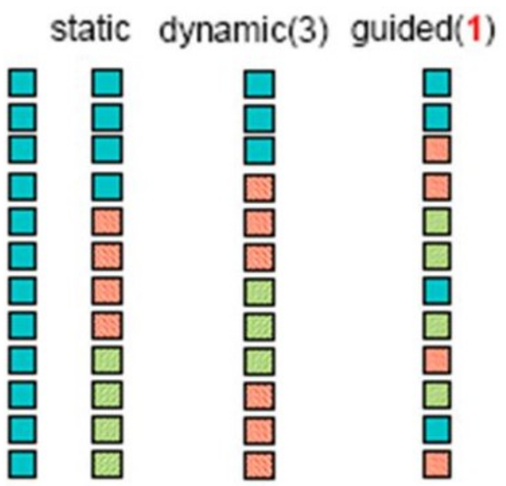
Granularity

- **What is granularity?**
  - Size of divided groups of data
  - Measure of computation to communication/synchronization
- **Why is it important?**
  - Want to maximize computation
  - Comm/sync time is not completing work
  - Applies differently to different applications/architectures
- **What kinds are there?**
  - Coarse granularity – a boulder
    - Large groups of data after division
    - Large amounts of work are done between communication
    - For OpenMP, not communication, more like “distribution” or “assignment”
  - Fine granularity – sand
    - Work split into very small pieces, maybe a single iteration
    - Only a small amount of work done before we must stop to get more





Scheduling



- **What is scheduling?**
  - Manages how iterations of a for loop are divided
  - Allows optimization for different program needs
  - Must balance granularity, overhead, performance
- **How do I use this?**
  - Append “schedule(type)” to pragma omp parallel for
- **What kinds are there?**
  - Static – iterations divided and assigned evenly at compile time
  - Static, Chunk – static, with size of divided chunks parameterizable
  - Dynamic – when core completes one, takes another from pile
  - Dynamic, Chunk – dynamic, but takes more than one to reduce overhead
  - Guided – dynamic, starts with big chunks, then they get smaller as it runs

Scheduling Example

Code Example

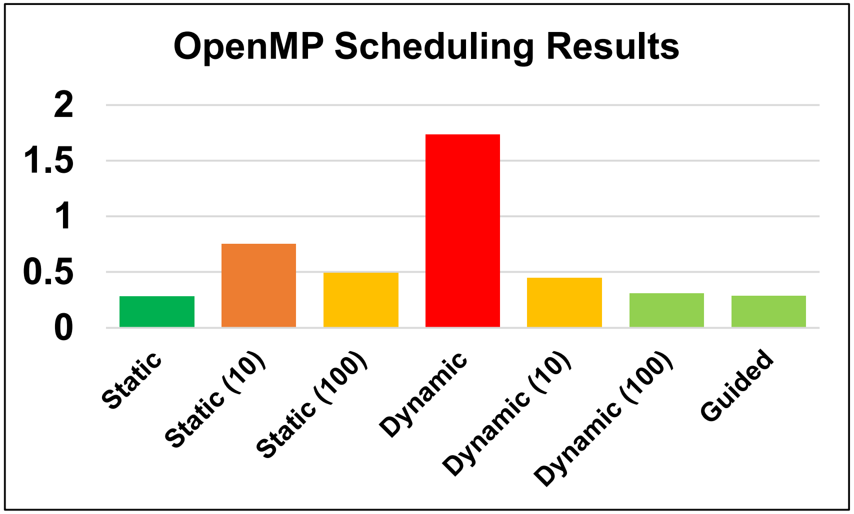
```c
#include <omp.h>
#define SIZE 100000000
int main( int argc, char **argv ) {
  srand( time( NULL ) );   // Seed Random
  int i              = 0;  // Loop Iterator
  // Initialize Vector Memory Spaces
  double *mainVector = malloc( SIZE * sizeof(double) );
  double *divVector  = malloc( SIZE * sizeof(double) );
  double *solVector  = malloc( SIZE * sizeof(double) );
  // Fill Vectors
  for( i = 0; i < SIZE; i++ ) {
    mainVector[i]    = ( rand( ) % 10000000 ) * 0.01;
    divVector[i]     = ( rand( ) % 1000 )     * 0.01;
    solVector[i]     = 0;
  }
  // Perform Processing
  double start       = omp_get_wtime( );
  #pragma omp parallel for num_threads( 4 )     \
     shared( mainVector, divVector, solVector ) \
     private( i ) schedule( static )
  for( i = 0; i < SIZE; i++ ) {
    solVector[i]     = mainVector[i] / divVector[i];
  }
  double end         = omp_get_wtime( );
  double solTime     = end - start;
  printf( "Complete - %lf Seconds\n", solTime );
  return 0;
}
```

Output (4 Cores)

```
$ gcc -fopenmp ...
(builds)
$ ./vectorOps
Complete - 0.282029 Seconds
$

Can tweak schedule to dynamic.

$ gcc -fopenmp ...
(builds)
$ ./vectorOps
Complete - 1.735515 Seconds
$

Wowza.  That's a lot of
overhead.  Can we alter the
chunk size?  Maybe try 1000?

$ gcc -fopenmp ...
(builds)
$ ./vectorOps
Complete - 0.286736 Seconds
$
```

Scheduling Results

- **Timings for Various Scheduling Methods**
  - Static 0.282029
  - Static, 10 0.753599
  - Static, 100 0.491518
  - Dynamic 1.735515
  - Dynamic, 10 0.445480
  - Dynamic, 100 0.307367
  - Guided 0.284483



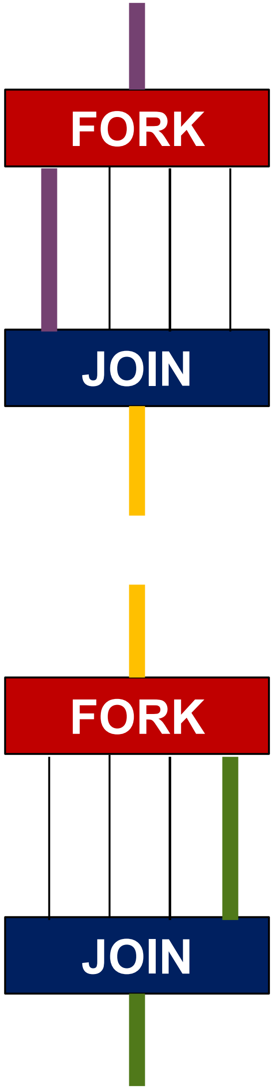
- **Visualizing the Balance**
  - Increasing chunk size of static division can induce overhead
  - Dynamic scheduling often induces significant overhead for small chunks
  - For some applications, division may cause load imbalance dynamic solves
  - Guided provides a suitable balance for many applications between the two

Extra Fun

You're doing so great! Almost there! WOOOO!

First/LastPrivate

- **Alters private behavior relating to first/last iteration**
- **Private**
  - Not initialized by default, just existing binary garbage
  - Order not guaranteed, may remain garbage
- **FirstPrivate**
  - Handled at start of parallel region
  - Automatically initializes values of variables in threads
  - Value of variable in master thread is carried into others
- **LastPrivate**
  - Handled at the end of a parallel region
  - Sets value of variable in master thread
  - Can be from for loop iterations or sections
  - Value taken from last iteration or section to execute
  - Sequential order implied, comes from highest thread index



FirstPrivate Example

Code Example

```
#include <omp.h>
int main( int argc, char **argv ) {
  int thread = 0;  // Thread Number
  int num    = 3;  // Number
  printf( "Master Thread\n  Num Value = %d\n", num );
  printf( "Parallel Region\n" );
  #pragma omp parallel private( thread ) \
                       firstprivate( num )
  {
    #pragma omp master
    {
      printf( "  Num Value = %d\n", num );
    }
    thread = omp_get_thread_num( );
    num    = thread * thread + num;
    printf( "  Thread %d - Value %d\n", thread, num );
  }
  printf( "Master Thread\n  Num Value = %d\n", num );
  return 0;
}
```

Output (4 Cores)

```
$ gcc ...
(builds)
$ ./firstprivate
Master Thread
  Num Value = 3
Parallel Region
  Num Value = 3
  Thread 0 – Value 3
  Thread 1 – Value 4
  Thread 2 – Value 7
  Thread 3 – Value 12
Master Thread
  Num Value = 3
$
```

LastPrivate Example

Code Example

```c
#include <omp.h>
int main( int argc, char **argv ) {
  int thread = 0;  // Thread Number
  int num    = 3;  // Number
  int i      = 0;  // Loop Iterator
  printf( "Master Thread\n  Num Value = %d\n", num );
  printf( "Parallel Region\n" );
  #pragma omp parallel private( thread )
  {
    #pragma omp master
    {
      printf( "  Num Value = %d\n", num );
    }
    #pragma omp for lastprivate( num )
    for( i = 0; i < 4; i++ ) {
      thread = omp_get_thread_num( );
      num    = thread * thread + num;
      printf( "  Thread %d - Value %d\n", thread, num );
    }
  }
  printf( "Master Thread\n  Num Value = %d\n", num );
  return 0;
}
```

Output (4 Cores)

```
$ gcc ...
(builds)
$ ./private
Master Thread
  Num Value = 3
Parallel Region
  Num Value = 3  (before for)
  Thread 0 - Value 0
  Thread 3 - Value 9
  Thread 2 - Value 4
  Thread 1 - Value 1
Master Thread
  Num Value = 9
$
```

Note that it still returns value from highest index even if order changes.

ThreadPrivate

- **Let thread variables persist between parallel regions**
  - Same rules as private apply
    - Must specify variable after declaration
    - Each thread gets own copy, not visible to others
  - Undefined on first entry unless copyin is used
- **Watch out**
  - Must explicitly turn off dynamic threads
  - Number of threads must remain constant
- **CopyIn**
  - Allows assignment of value to all threads in team
  - Master thread is used as copy source for value
- **CopyPrivate**
  - Used with single directive
  - Copies value from single thread to all private instances in other threads

ThreadPrivate Example

Code Example

```c
#include <omp.h>
int num = 0;
int main( int argc, char **argv ) {
  // Disable Dynamic Threads
  omp_set_dynamic( 0 );
  // Set Num as ThreadPrivate
  #pragma omp threadprivate( num )
  int thread = 0;  // Thread Number
  num        = 6;  // Change Master Num Value
  printf( "First Parallel Region\n" );
  #pragma omp parallel private( thread ) copyin( num )
  {
    thread = omp_get_thread_num( );
    num    = thread * thread + num;
    printf( "  Thread %d - Value %d\n", thread, num );
  }

  printf( "Master Thread\n" );

  printf( "Second Parallel Region\n" );
  #pragma omp parallel
  {
    thread = omp_get_thread_num( );
    printf( "  Thread %d - Value Remains %d\n",
            thread, num                          );
  }
  return 0;
}
```

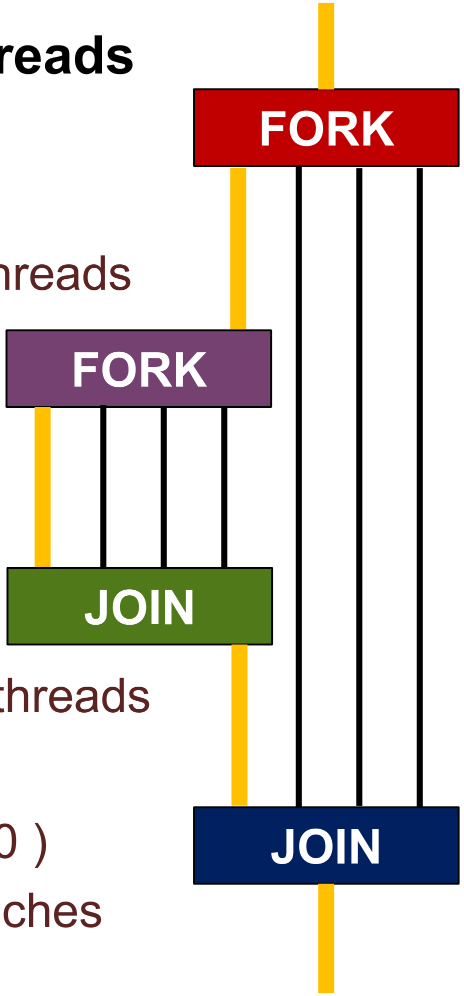
Output (4 Cores)

```
$ gcc ...
(builds)
$ ./threadprivate
First Parallel Region
  Thread 2 – Value 10
  Thread 1 – Value 7
  Thread 3 – Value 15
  Thread 0 – Value 6
Master Thread
Second Parallel Region
  Thread 0 – Value Remains 6
  Thread 1 – Value Remains 7
  Thread 2 – Value Remains 10
  Thread 3 – Value Remains 15
$
```

Note that values persist with thread number! No pesky dynamic reassignment!

Nested Parallelism

- **Thread within team forks again, creating more threads**
  - Original thread becomes master of created threads
- **Advantages**
  - Can quickly split functions more deeply to utilize more threads
  - May layer and combine functional and data parallelism
- **Disadvantages**
  - Requires more overhead – are we overdoing it?
  - Often less efficient – load balancing easier at low level
  - Additional complexity managing more thread teams
  - System may be "oversubscribed" by creating too many threads
- **Usage**
  - Must use omp_set_nested(1) and omp_set_dynamic(0)
  - Create parallel regions as before, assigning thread branches



Nested Parallelism On Example

Code Example

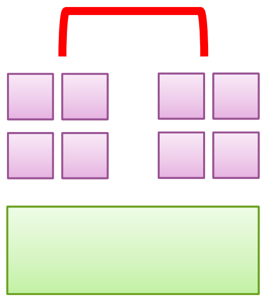
```
#include <omp.h>

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

  omp_set_nested( 1 );    // Enable Nested Parallelism
  omp_set_dynamic( 0 );   // Disable Dynamic Threads

  // Outer Level Parallel Region - 2 Threads
  #pragma omp parallel num_threads( 2 )
  {

    printf( "Outer Level - See this twice.\n" );

    // Inner Level Parallel Region - 2 Threads Each
    #pragma omp parallel num_threads( 2 )
    {

      printf( "Inner Level - See this four times!\n" );

    }

  }

  return 0;
}
```

Output (4 Cores)

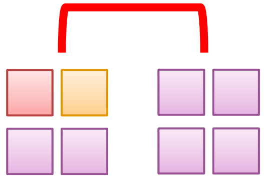
```
$ gcc ...
(builds)
$ ./nested
Outer Level – See this twice.
Outer Level – See this twice.
Inner Level – See this four
times!
Inner Level – See this four
times!
Inner Level – See this four
times!
Inner Level – See this four
times!
$
```

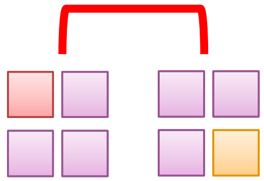
Thread Affinity (Binding)

- **Controls assignment of thread to specific core(s)**
  - Performed via environment variable assignment
  - Affects communication, memory use, performance
  - Helps group execution for NUMA architectures
    - Communication over bus may reduce performance
    - Lack of bandwidth for all cores in one socket may also impact performance
    - Just like everything else, a delicate balance



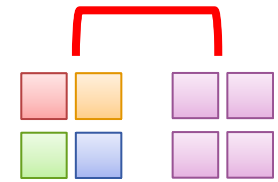
- **OMP_PROC_BIND = …**
  - False – no binding enforced
  - True – locks threads to one core throughout execution
    - Simple affinity for locked-to-core performance, similar to threadprivate
  - Master – collocates threads with the master thread
    - Convenient for recursive thread creation
  - Close – place threads close to the master
    - Minimize communication cost between each thread
  - Spread – spread threads out as much as possible
    - Maximize bandwidth available to each thread

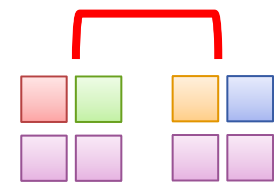


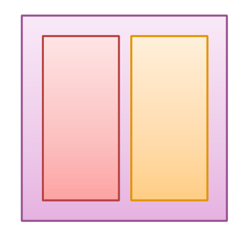


Thread Affinity (Binding)

- **OMP_PLACES = …**
  - Cores
    - Default setting, assigns to cores sequentially
    - May cause collisions that reduce performance
  - Sockets
    - Alternates assigning threads to cores of each socket
    - Helpful for bandwidth
  - Threads
    - Ties OpenMP execution to specific hardware thread
    - For processors with hardware multithreading
  - Manually – {location:number:stride}
    - OMP_PLACES="{0:8:1},{8:8:1}“
      - Two sockets, eight consecutive cores per socket
    - OMP_PLACES="{0},{1},{2},...,{15}“
      - Equivalent to setting individual cores
    - OMP_PLACES="{0:4:8}:4:1“
      - Four sockets; place 0, 8, 16, 24 repeated four times; core 0 of some socket, core 1 of another, etc.
  - Via GCC Compiler - GOMP_CPU_AFFINITY=0,1,8,9







Thank You! Any Questions?

You did it! I will be quiet now!