The Abdus Salam International Centre for Theoretical Physics

United Nations Educational, Scientific and Cultural Organization

IAEA International Atomic Energy Agency

1967-19

Advanced School in High Performance and GRID Computing

3 - 14 November 2008

Introduction to OpenMP (first part)

BROWN Shawn T.

Carnegie Mellon University
Pittsburgh Supercomputing Center
300 S. Craig Street
U.S.A.

Introduction to OpenMP

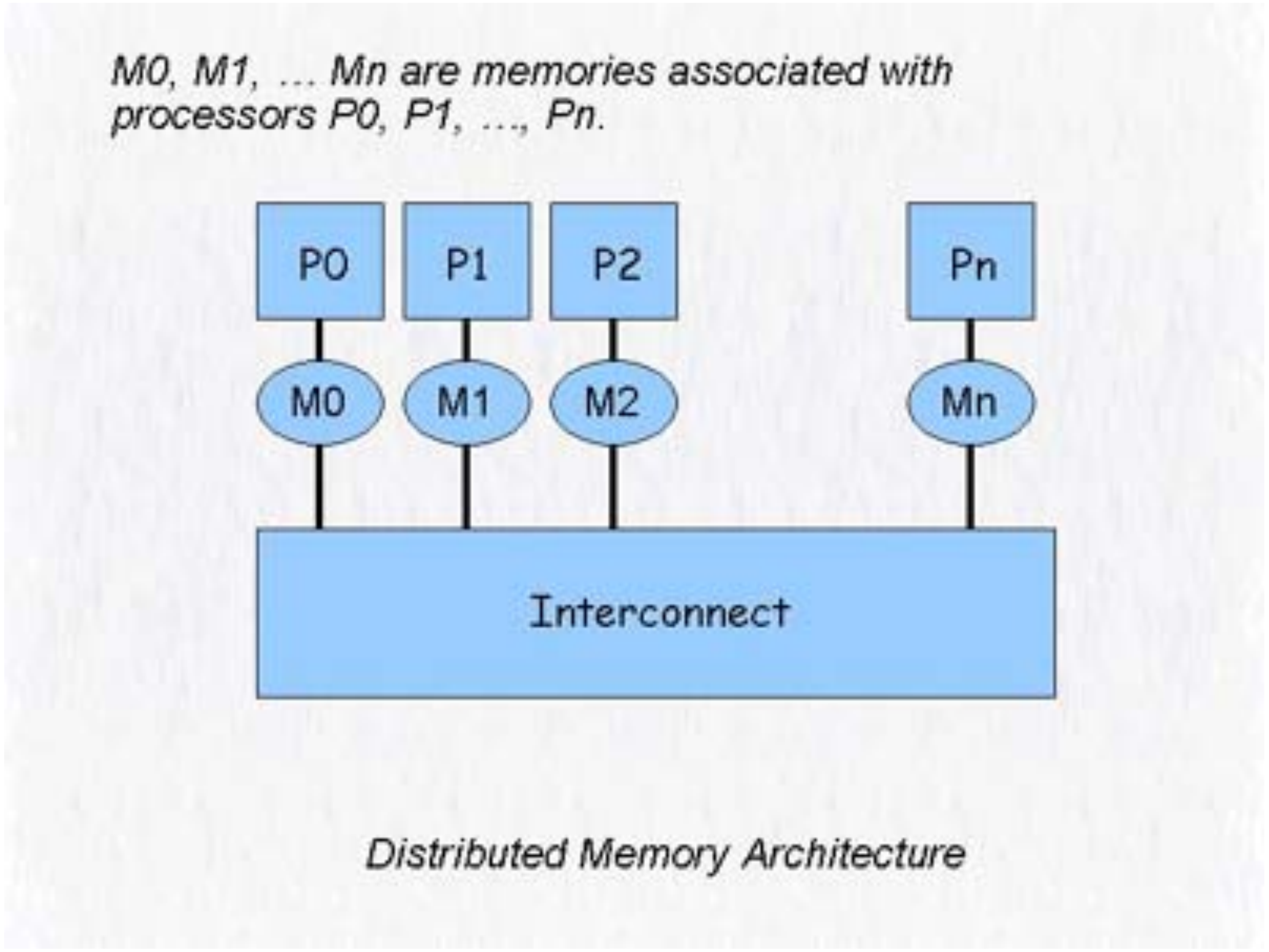
Shawn T. Brown

Scientific Specialist
Pittsburgh Supercomputing Center

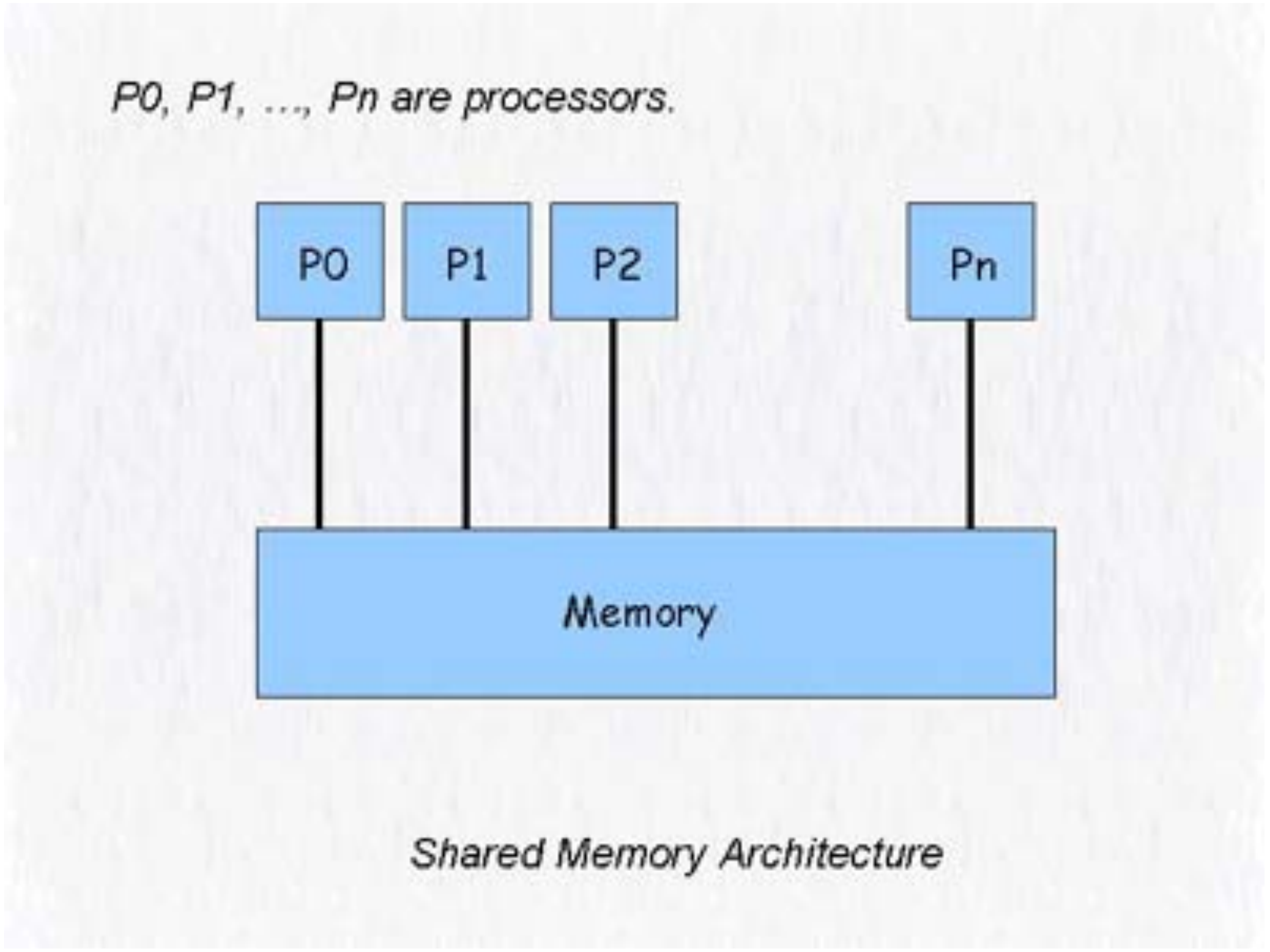
Jeff Gardner (U. of Washington)
Phil Blood (PSC)

© 2008 Pittsburgh Supercomputing Center

Different types of parallel platforms: Distributed Memory


M0, M1, … Mn are memories associated with processors P0, P1, …, Pn.
P0
P1
P2
Pn
M0
M1
M2
Mn
Interconnect
Distributed Memory Architecture

Different types of parallel platforms: Shared Memory



© 2008 Pittsburgh Supercomputing Center

Different types of parallel platforms: Shared Memory

- SMP: Symmetric Multiprocessing
  - Identical processing units working from the same main memory
  - SMP machines are becoming more common in the everyday workplace
    - Dual-socket motherboards are very common, and quad-sockets are not uncommon
    - 2 and 4 core CPUs are now commonplace
    - Intel Larabee: 12-48 cores in 2009-2010
- ASMP: Asymmetric Multiprocessing
  - Not all processing units are identical
  - Cell processor of PS3

© 2008 Pittsburgh Supercomputing Center

Parallel Programming Models

- Shared Memory
  - Multiple processors sharing the same memory space
- Message Passing
  - Users make calls that explicitly share information between execution entities
- Remote Memory Access
  - Processors can directly access memory on another processor
- These models are then used to build more sophisticated models
  - Loop Driven
  - Function Driven Parallel (Task-Level)

Shared Memory Programming

- SysV memory manipulation
  - One can actually create, manipulate, shared memory spaces.
- Pthreads (Posix Threads)
  - Lower level Unix library to build multi-threaded programs
- OpenMP (www.openmp.org)
  - Protocol designed to provide automatic parallelization through compiler pragmas.
  - Mainly loop driven parallelism
  - Best suited to desktop and small SMP computers
- Caution: Race Conditions
  - When two threads are changing the same memory location at the same time.

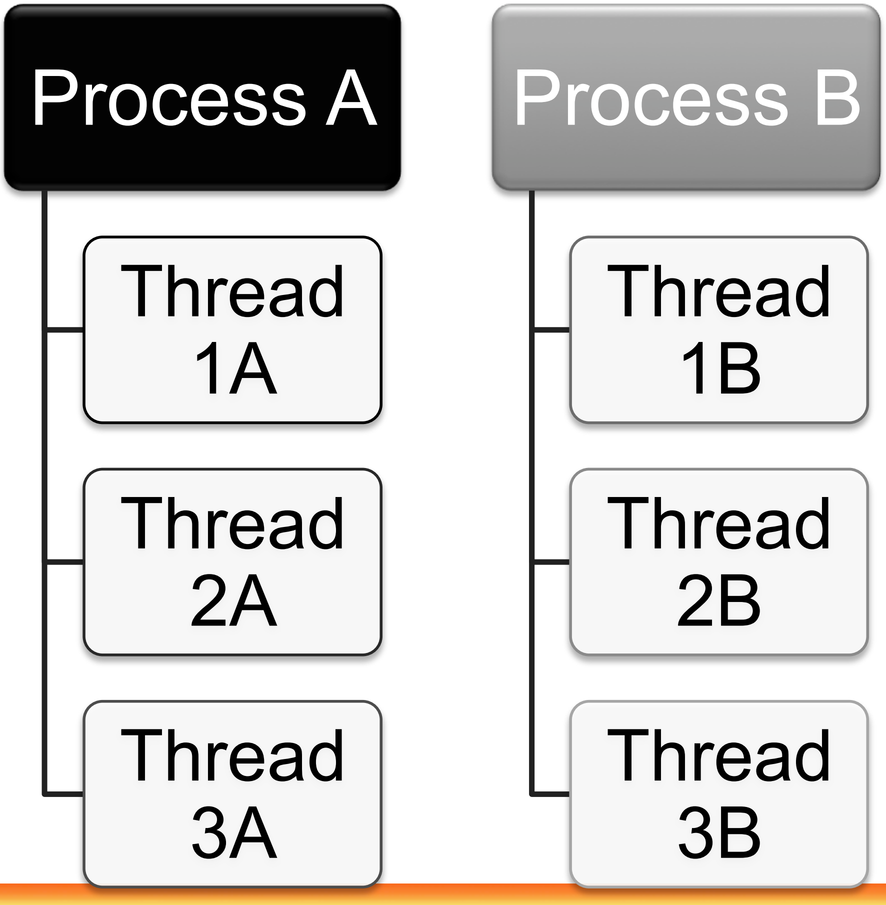
© 2008 Pittsburgh Supercomputing Center

Introduction

- OpenMP is designed for shared memory systems.
- OpenMP is not a programming language
  - it is a specification, usually implemented through compiler directive pragmas
- OpenMP is easy to use
  - achieve parallelism through compiler directives
  - or the occasional function call
- OpenMP is a "quick and dirty" way of parallelizing a program.
- OpenMP is usually used on existing serial programs to achieve moderate parallelism with relatively little effort

Computational Threads

• Tasks on an operating system are layed out on the proecessor as independent **Processes** that do not share memory space
• Within a process there can be several shared execution units known as **Threads**.



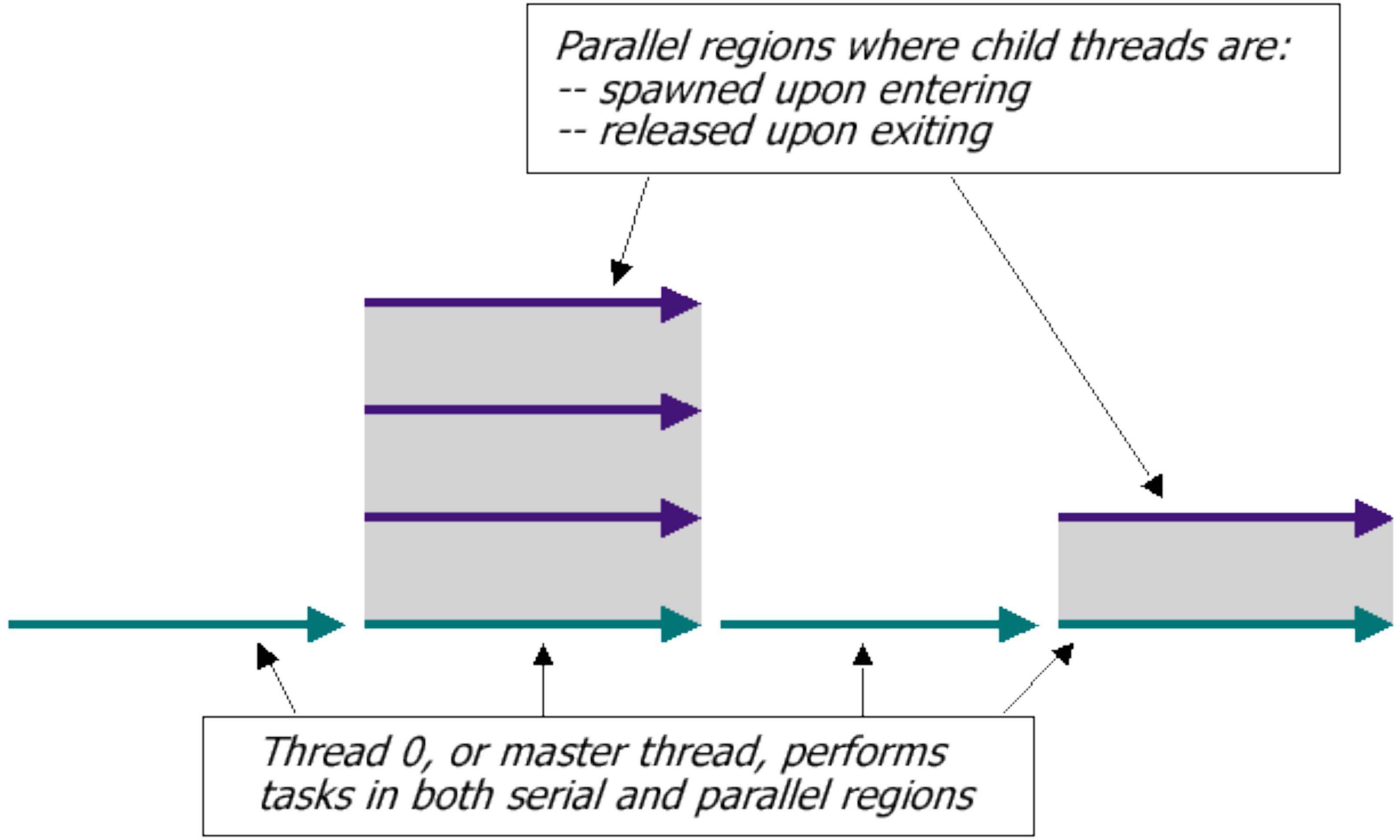
© 2008 Pittsburgh Supercomputing Center

OpenMP Execution Model

- In MPI, all processes are active all the time
  - Created at the initialization
- In OpenMP, execution begins only on the master thread.
  - Child threads are spawned and released as needed.
  - Threads are spawned when program enters a parallel region.
  - Threads are released when program exits a parallel region

© 2008 Pittsburgh Supercomputing Center

OpenMP Execution Model


Parallel regions where child threads are:
-- spawned upon entering
-- released upon exiting
Thread 0, or master thread, performs tasks in both serial and parallel regions

© 2008 Pittsburgh Supercomputing Center

PSC PITTSBURGH SUPERCOMPUTING CENTER

Parallel Region Example: For loop

Fortran:

```
!$omp parallel do
do i = 1, n
 a(i) = b(i) + c(i)
enddo
```

C/C++:

```
#pragma omp parallel for
for(i=1; i<=n; i++)
    a[i] = b[i] + c[i];
```

This comment or pragma tells openmp compiler to spawn threads *and* distribute work among those threads

These actions are combined here but they can be specified separately between the threads

© 2008 Pittsburgh Supercomputing Center

Pros of OpenMP

- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement
- Programming model is “serial-like” and thus conceptually simpler than message passing
- Compiler directives are generally simple and easy to use
- Legacy serial code does not need to be rewritten

© 2008 Pittsburgh Supercomputing Center

Cons of OpenMP

- Codes can only be run in shared memory environments!
  - In general, shared memory machines beyond ~8 CPUs are much more expensive than distributed memory ones, so finding a shared memory system to run on may be difficult
- Compiler must support OpenMP
  - whereas MPI can be installed anywhere
  - However, gcc 4.2 now supports OpenMP

© 2008 Pittsburgh Supercomputing Center

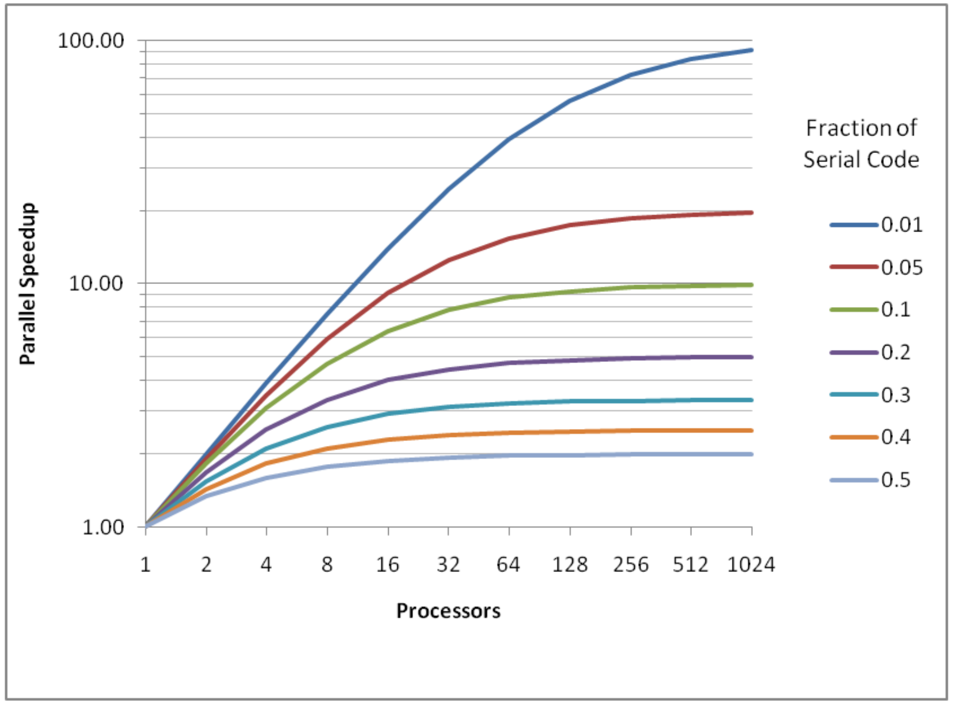
Cons of OpenMP

- In general, only moderate speedups can be achieved.
  - Because OpenMP codes tend to have serial-only portions, Amdahl's Law prohibits substantial speedups

- Amdahl's Law:

  F = Fraction of serial execution time that cannot be parallelized

  N = Number of processors



Execution time = $\dfrac{1}{F + (1 - F)/N}$

If you have big loops that dominate execution time, these are ideal targets for OpenMP

Goals of this lecture

- Exposure to OpenMP
  - Understand where OpenMP may be useful to you now
  - Or perhaps 4 years from now when you need to parallelize a serial program, you will say, “Hey! I can use OpenMP.”
- Avoidance of common pitfalls
  - How to make your OpenMP actually get the same answer that it did in serial
  - A few tips on dramatically increasing the performance of OpenMP applications

Compiling and Running OpenMP

True64:	**-mp**
SGI IRIX:	**-mp**
IBM AIX:	**-qsmp=omp**
Portland Group:	**-mp**
Intel:	**-openmp**
gcc (4.2)	**-fopenmp**

© 2008 Pittsburgh Supercomputing Center

Compiling and Running OpenMP

- OMP_NUM_THREADS environment variable sets the number of processors the OpenMP program will have at its disposal.
- Example script

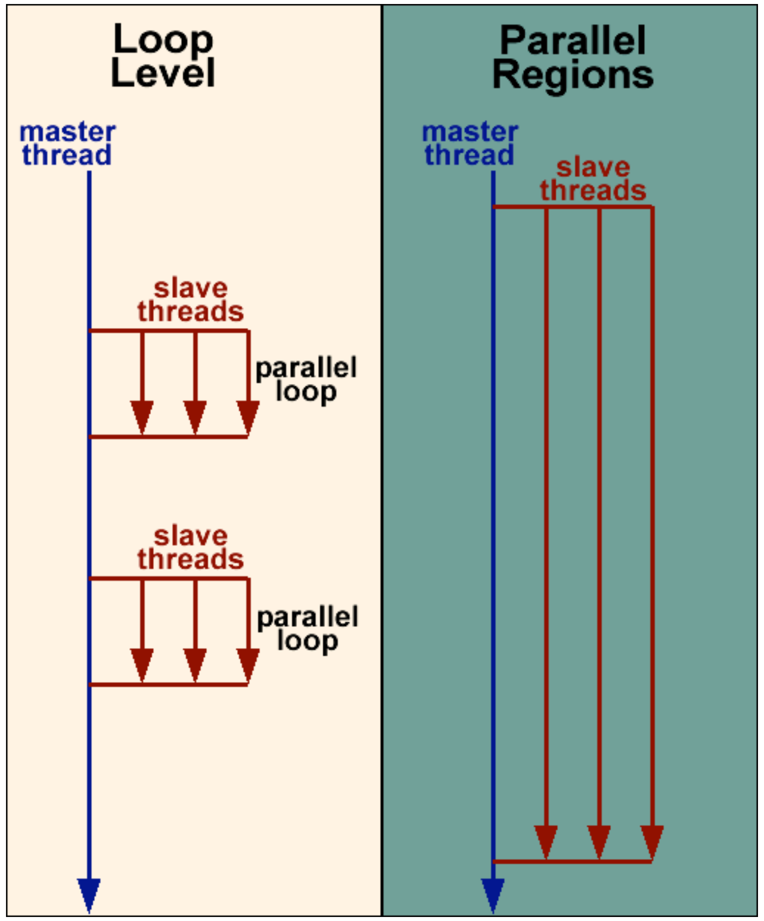
```
#!/bin/tcsh
setenv OMP_NUM_THREADS 4
mycode < my.in > my.out
```

© 2008 Pittsburgh Supercomputing Center

PSC
PITTSBURGH SUPERCOMPUTING CENTER

OpenMP Basics: 2 Approaches to Parallelism

Divide loop iterations among threads: We will focus mainly on loop level parallelism in this lecture



Divide various sections of code between threads

© 2008 Pittsburgh Supercomputing Center

Sections: Functional parallelism

```
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
                block1
        #pragma omp section
                block2
    }
}
```

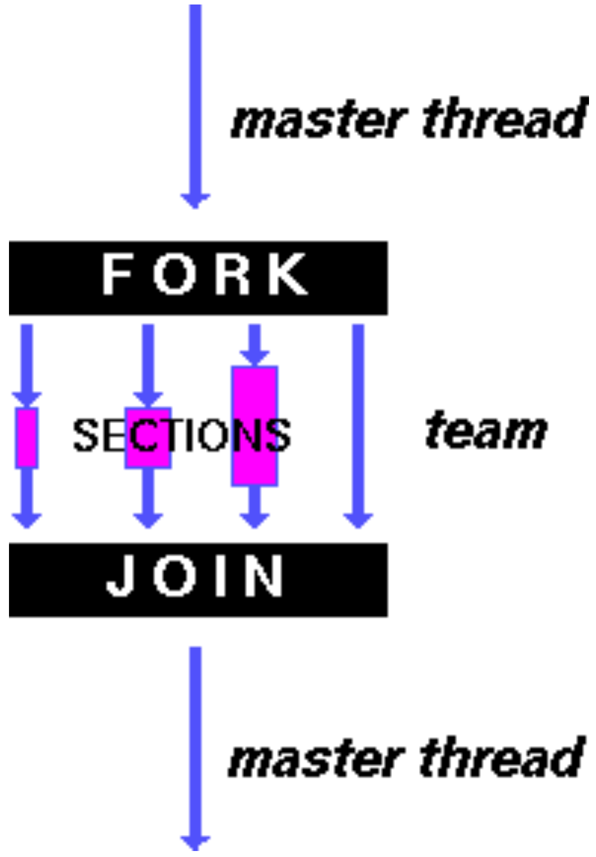


Image from: https://computing.llnl.gov/tutorials/openMP

© 2008 Pittsburgh Supercomputing Center

Parallel DO/for: Loop level parallelism

Fortran:

```
!$omp parallel do
do i = 1, n
 a(i) = b(i) + c(i)
enddo
```

C/C++:

```
#pragma omp parallel for
for(i=1; i<=n; i++)
    a[i] = b[i] + c[i];
```

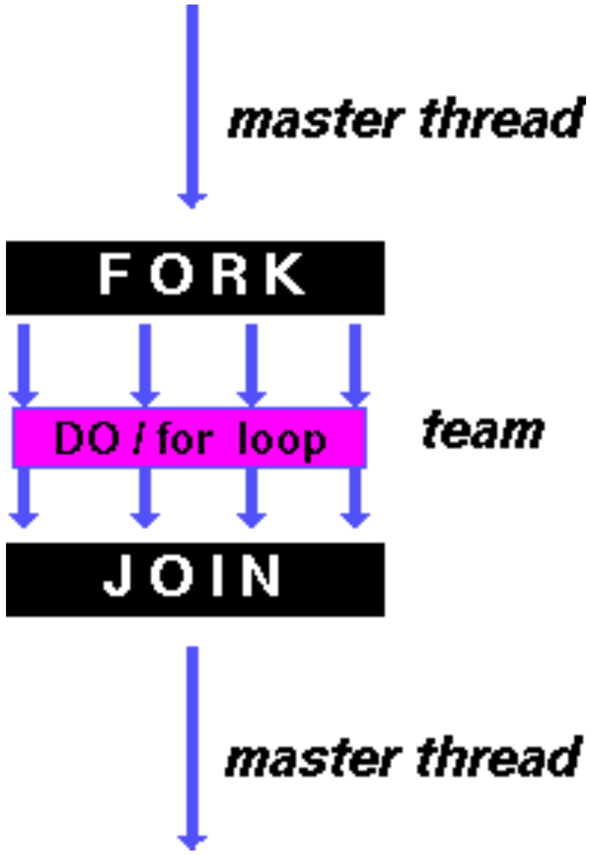


Image from: https://computing.llnl.gov/tutorials/openMP

© 2008 Pittsburgh Supercomputing Center

Pitfall #1: Data dependencies

- Consider the following code:

```
a[0] = 1;
for(i=1; i<5; i++)
  a[i] = i + a[i-1];
```

- There are dependencies between loop iterations.
- Sections of loops split between threads will not necessarily execute in order
- Out of order loop execution will result in undefined behavior

© 2008 Pittsburgh Supercomputing Center

Pitfall #1: Data dependencies

3 simple rules for data dependencies

1. All assignments are performed on arrays.
2. Each element of an array is assigned to by at most one iteration.
3. No loop iteration reads array elements modified by any other iteration.

© 2008 Pittsburgh Supercomputing Center

Avoiding dependencies by using Private Variables (Pitfall #1.5)

- Consider the following loop:

```
#pragma omp parallel for
{
   for(i=0; i<n; i++){
     temp = 2.0*a[i];
     a[i] = temp;
     b[i] = c[i]/temp;
   }
}
```

- By default, all threads share a common address space. Therefore, all threads will be modifying *temp* simultaneously

© 2008 Pittsburgh Supercomputing Center

Avoiding dependencies by using Private Variables (Pitfall #1.5)

- The solution is to make *temp* a thread-private variable by using the “private” clause:

```
#pragma omp parallel for private(temp)
{
  for(i=0; i<n; i++){
      temp = 2.0*a[i];
      a[i] = temp;
      b[i] = c[i]/temp;
  }
}
```

© 2008 Pittsburgh Supercomputing Center

Avoiding dependencies by using Private Variables (Pitfall #1.5)

- Default OpenMP behavior is for variables to be shared. However, sometimes you may wish to make the default private and explicitly declare your shared variables (but only in Fortran!):

```
!$omp parallel do default(private) shared(n,a,b,c)
  do i=1,n
   temp = 2.0*a(i)
   a(i) = temp
   b(i) = c(i)/temp;
  enddo
!$omp end parallel do
```

Private variables

- Note that the loop iteration variable (e.g. *i* in previous example) is *private* by default

- Caution: The value of any variable specified as *private* is *undefined* both upon entering and leaving the construct in which it is specified

- Use *firstprivate* and *lastprivate* clauses to retain values of variables declared as *private*

© 2008 Pittsburgh Supercomputing Center

Use of function calls within parallel loops

- In general, the compiler will not parallelize a loop that involves a function call unless is can guarantee that there are no dependencies between iterations.
  - sin(x) is OK, for example, if x is private.

- A good strategy is to inline function calls within loops. If the compiler can inline the function, it can usually verify lack of dependencies.

- System calls do not parallelize!!!

© 2008 Pittsburgh Supercomputing Center

Pitfall #2: Updating shared variables simultaneously

Consider the following serial code:

```
the_max = 0;
for (i=0;i<n; i++)
   the_max = max(myfunc(a[i]), the_max);
```

- This loop can be executed in any order, however the_max is modified every loop iteration.
- Use "critical" clause to specify code segments that can only be executed by one thread at a time:

```
#pragma omp parallel for private(temp)
{
   for(i=0; i<n; i++){
     temp = myfunc(a[i]);
     #pragma omp critical
     the_max = max(temp, the_max);
   }
}
```

© 2008 Pittsburgh Supercomputing Center

Reduction operations

- Now consider a global sum:

```
for(i=0; i<n; i++)
   sum = sum + a[i];
```

- This can be done by defining “critical” sections
  - Very very slow and unscalable.

- OpenMP provides a reduction clause (much faster):

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

© 2008 Pittsburgh Supercomputing Center

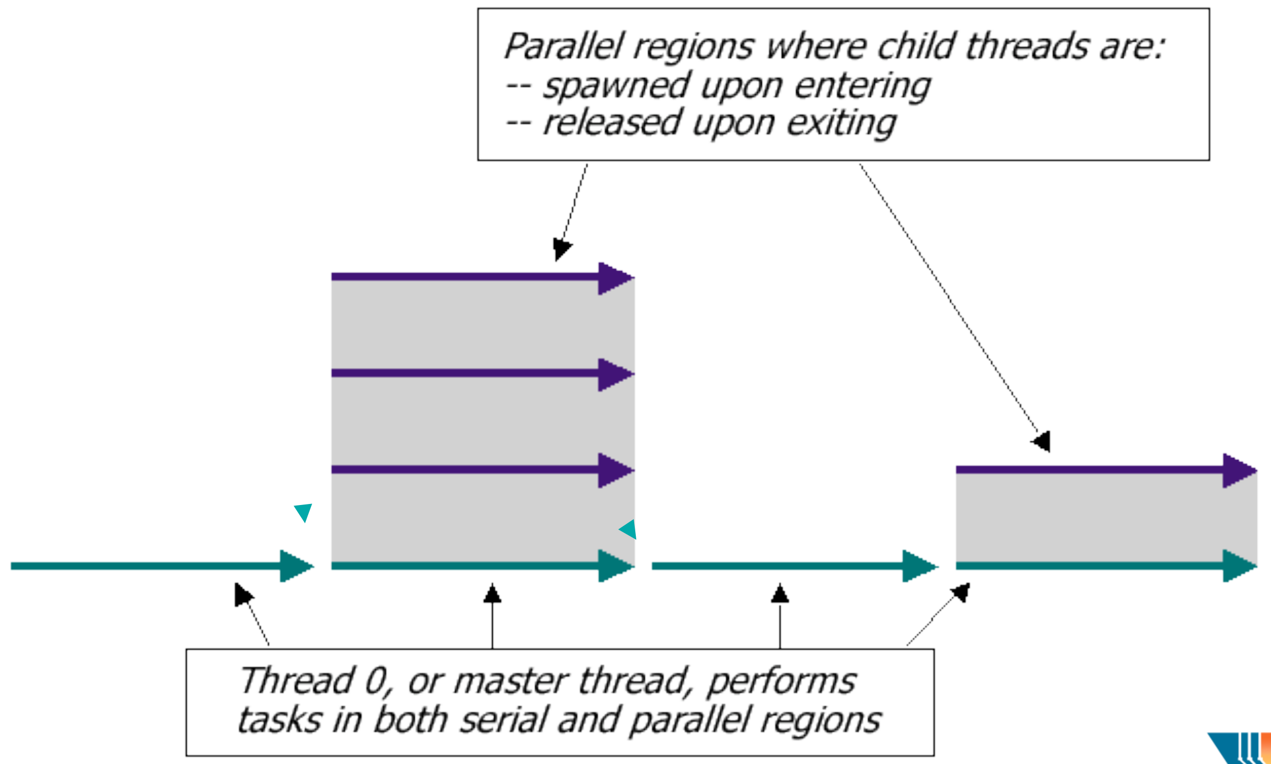
Reduction operations

- C/C++ reduction-able operators (and initial values):
  - + (0)
  - - (0)
  - * (1)
  - & (~0)
  - | (0)
  - ^ (0)
  - && (1)
  - || (0)

© 2008 Pittsburgh Supercomputing Center

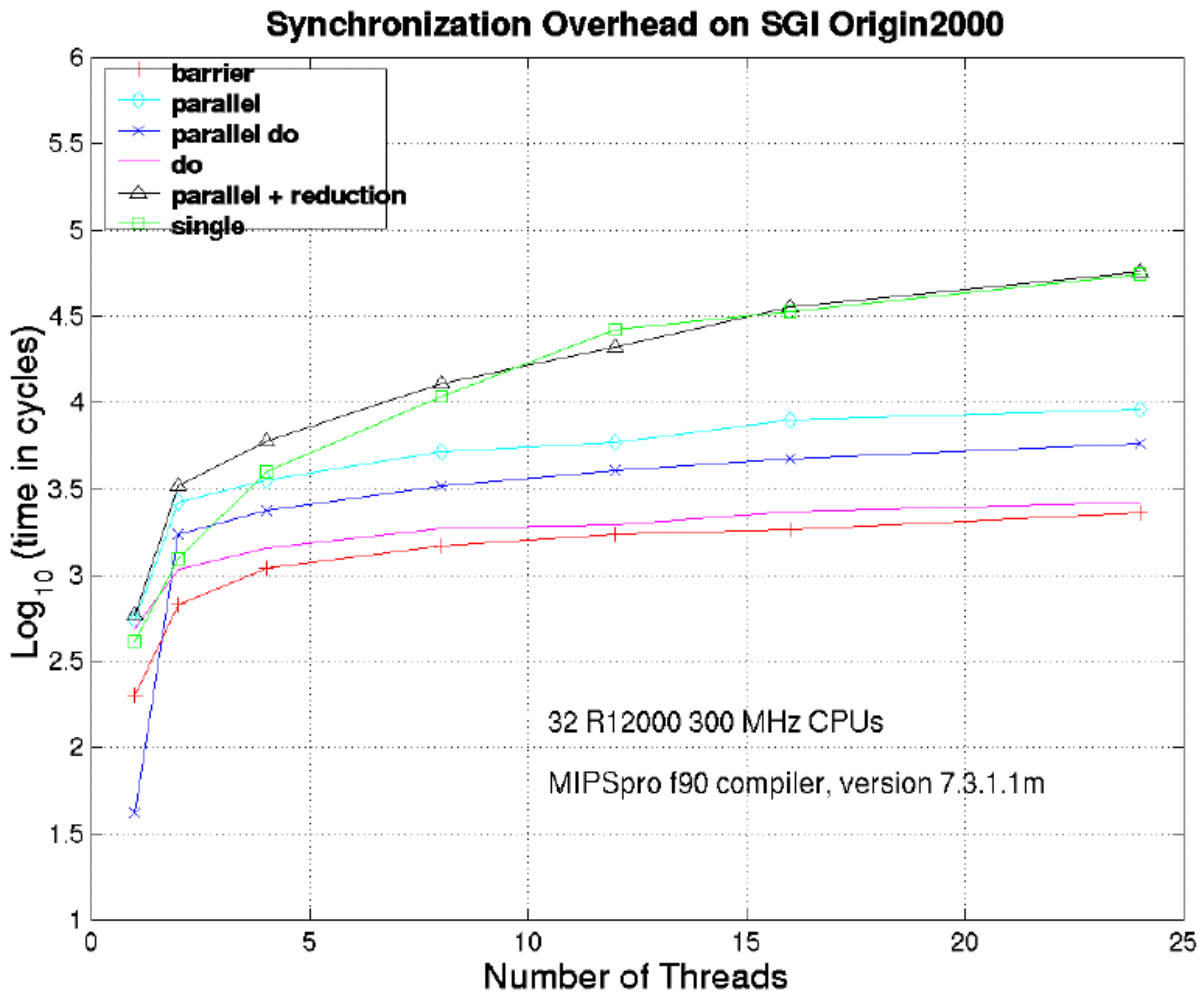
Pitfall #3: Parallel overhead

- Spawning and releasing threads results in *significant* overhead.


Parallel regions where child threads are:
-- spawned upon entering
-- released upon exiting
Thread 0, or master thread, performs tasks in both serial and parallel regions

© 2008 Pittsburgh Supercomputing Center

Pitfall #3: Parallel overhead



© 2008 Pittsburgh Supercomputing Center

Pitfall #3: Parallel Overhead

- Spawning and releasing threads results in *significant* overhead.
- Therefore, you want to make your parallel regions as large as possible
  - Parallelize over the largest loop that you can (even though it will involve more work to declare all of the private variables and eliminate dependencies)
  - Coarse granularity is your friend!

© 2008 Pittsburgh Supercomputing Center

Separating “Parallel” and “For” directives to reduce overhead

- In the following example, threads are spawned only once, **not** once per loop:

```c
#pragma omp parallel {
   #pragma omp for
   for(i=0; i<maxi; i++)
    a[i] = b[i];
   #pragma omp for
   for(j=0; j<maxj; j++)
    c[j] = d[j];
}
```

```fortran
!$omp parallel
!$omp do
do i=1,maxi
      a(i) = b(i)
enddo
!$omp end do !(optional)

!$omp do
do i=1,maxj
      c(j) = d(j)
enddo
!$omp end do !(optional)
!$omp end parallel !
(required)
```

Use “nowait” to avoid barriers

- At the end of every loop is an implied barrier.
- Use “nowait” to remove the barrier at the end of the first loop:

```
#pragma omp parallel {
   #pragma omp for nowait
   for(i=0; i<maxi; i++)
     a[i] = b[i];
   #pragma omp for
   for(j=0; j<maxj; j++)
     c[j] = d[j];
}
```

$\leftarrow$ Barrier removed by “nowait” clause

© 2008 Pittsburgh Supercomputing Center

Use “nowait” to avoid barriers

In Fortran, “nowait” goes at end of loop:

```
!$omp parallel
!$omp do
do i=1,maxi
   a(i) = b(i)
enddo
!$omp end do nowait

!$omp do
do i=1,maxj
   c(j) = d(j)
enddo
!$omp end do
!$omp end parallel
```

← Barrier removed by “nowait” clause

© 2008 Pittsburgh Supercomputing Center

PSC
PITTSBURGH SUPERCOMPUTING CENTER

Other useful directives to avoid releasing and spawning threads

- #pragma omp master

  !$omp master ... !$omp end master
  - Denotes codes within a parallel region to only be executed by the master
- #pragma omp single
  - Denotes code that will be performed only one thread
  - Useful for overlapping serial segments with parallel computation.
- #pragma omp barrier
  - Sets a global barrier within a parallel region

© 2008 Pittsburgh Supercomputing Center

Thread stack

- Each thread has its own memory region called the thread stack
- This can grow to be quite large, so default size may not be enough
- This can be increased (e.g. to 16 MB):

csh:

```
limit stacksize 16000; setenv KMP_STACKSIZE 16000000
```

bash:

```
ulimit -s 16000; export KMP_STACKSIZE=16000000
```

© 2008 Pittsburgh Supercomputing Center

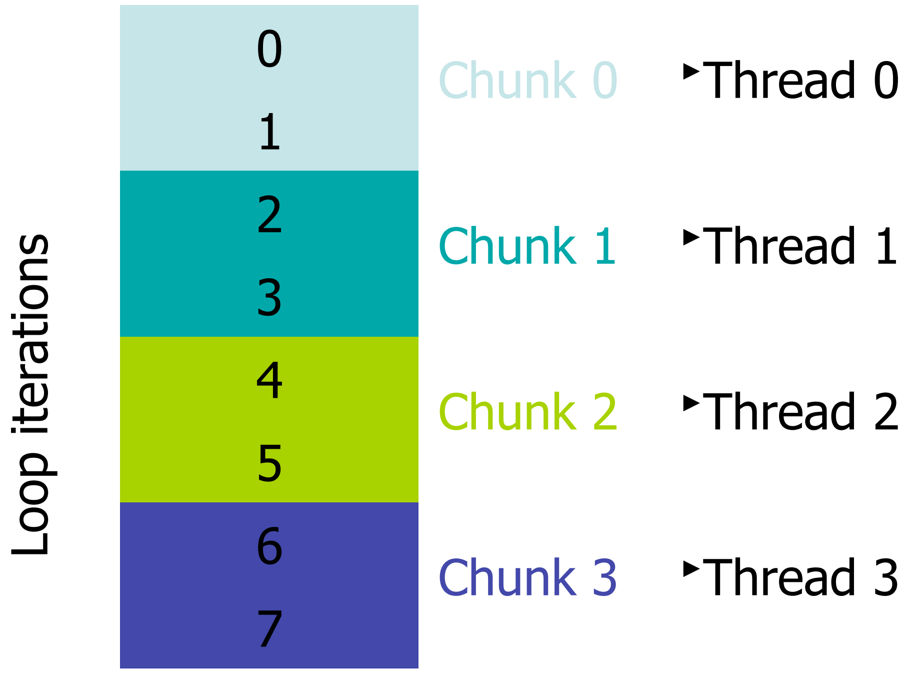
Useful OpenMP Functions

- `void omp_set_num_threads(int num_threads)`
  - Sets the number of OpenMP threads (overrides OMP_NUM_THREADS)
- `int omp_get_thread_num()`
  - Returns the number of the current thread
- `int omp_get_num_threads()`
  - Returns the total number of threads currently participating in a parallel region
  - Returns “1” if executed in a serial region
- For portability, surround these functions with `#ifdef _OPENMP`
- `#include <omp.h>`

© 2008 Pittsburgh Supercomputing Center

Optimization: Scheduling

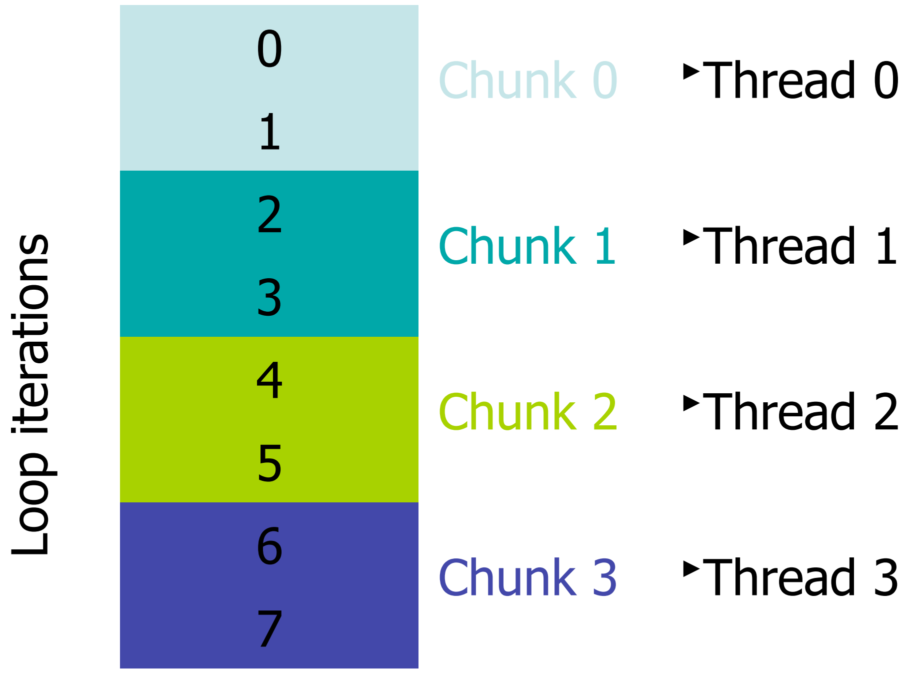
- OpenMP partitions workload into "chunks" for distribution among threads
- Default strategy is static:



© 2008 Pittsburgh Supercomputing Center

Optimization: Scheduling

- This strategy has the least amount of overhead
- However, if not all iterations take the same amount of time, this simple strategy will lead to load imbalance.



Optimization: Scheduling

- OpenMP offers a variety of scheduling strategies:
  - `schedule(static,[`*`chunksize`*`])`
    - Divides workload into equal-sized chunks
    - Default *chunksize* is Nwork/Nthreads
      - Setting *chunksize* to less than this will result in chunks being assigned in an interleaved manner
    - Lowest overhead
    - Least optimal workload distribution

Optimization: Scheduling

- `schedule(dynamic,[`*`chunksize`*`])`
  - Dynamically assigned chunks to threads
  - Default *chunksize* is 1
  - Highest overhead
  - Optimal workload distribution
- `schedule(guided,[`*`chunksize`*`])`
  - Starts with big chunks proportional to (number of unassigned iterations)/(number of threads), then makes them progressively smaller until chunksize is reached
  - Attempts to seek a balance between overhead and workload optimization

© 2008 Pittsburgh Supercomputing Center

Optimization: Scheduling

- `schedule(runtime)`
  - Scheduling can be selected at runtime using OMP_SCHEDULE
  - e.g. `setenv OMP_SCHEDULE "guided, 100"`
- In practice, often use:
  - Default scheduling (static, large chunks)
  - Guided with default chunksize
- Experiment with your code to determine optimal strategy

© 2008 Pittsburgh Supercomputing Center

What we have learned

- How to compile and run OpenMP progs
- Private vs. shared variables
- Critical sections and reductions for updating scalar shared variables
- Techniques for minimizing thread spawning/ exiting overhead
- Different scheduling strategies

© 2008 Pittsburgh Supercomputing Center

Summary

- OpenMP is often the easiest way to achieve moderate parallelism on shared memory machines
- In practice, to achieve decent scaling, will probably need to invest some amount of effort in tuning your application.
- More information available at:
  - https://computing.llnl.gov/tutorials/openMP/
  - http://www.openmp.org
  - *Using OpenMP*, MIT Press, 2008

© 2008 Pittsburgh Supercomputing Center