Parallel Computing for Science & Engineering

01/30/2018

Instructors:
Lars Koesterke, TACC
Charlie Dey, TACC

Scientific Computing Terminology

Terms	Definition
NUMA	Non Uniform Memory Access. In SMP systems with multiple CPUs access time to different parts of memory may vary.
Affinity	Propensity to maintain a process or thread on a hardware execution unit.
SMP	Symmetric Multi-Process(ing/or). Single OS system with shared memory.
OpenMP	
– Directive	Comment statement (F90) or #pragma (C/C++) that specifies parallel operations and control.
– Construct	The lexical extent that a directive controls.
– Region	All code controlled by a directive– lexical extent + content of called routines.
Runtime	Code or a library within an executable that interacts with the operating system and can control code execution.

TACC

OpenMP-- Overview

- Standard is ~20 years old. Mature language
- The “language” is easily comprehended.
  You can start simple and expand.
- Light Weight from System Perspective
- Very portable –GNU and vendor compilers.
- Now extended to user accelerators (GPUs, etc.)
- Spend time finding parallelism can be the most difficult part. The parallelism may be hidden.
- Writing Parallel OpenMP code examples is relatively easy.
- Developing parallel algorithms and/or parallelizing serial code is much harder.
- Expert level requires awareness of scoping and synchronization.
- Expansion into other performance relevant areas like: thread pinning and memory pinning

OpenMP executable runs on an SMP*

- Shared Memory systems:
  - One Operating System
  - Instantiation of ONE process
  - Threads are forked (created) from within your program.
  - Multiple threads on multiple cores

*** SMP = Symmetric Multi-Processor:
The execution of the operating system has equal access to any of the "processors".**

What is OpenMP (Open Multi-Processing)

- **De facto standard for Scientific Parallel Programming on Symmetric Multi-Processor (SMP) Systems.**
- **It is an API (Application Program Interface) for designing and executing parallel Fortran, C, and C++ programs**
    - Based on threads, but
    - Higher-level than POSIX threads (Pthreads) (http://www.llnl.gov/computing/tutorials/pthreads/#Abstract)
- **Implemented by:**
    - **Pragmas/comments in code**
    - **Runtime Library (interface to OS and Program Environment)**
    - **Environment Variables**
- **Compiler option required to interpret/activate directives.**
- **http://www.openmp.org/ has tutorials and description.**
- **Directed by OpenMP ARB (Architecture Review Board)**

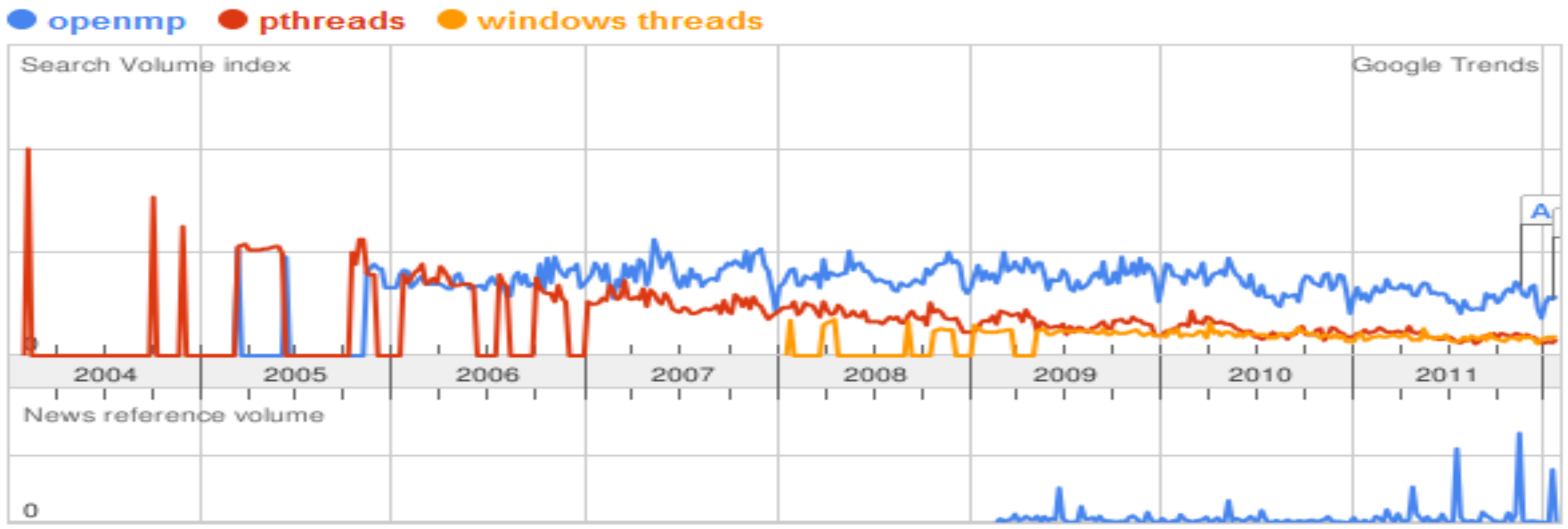
OpenMP History

- Primary OpenMP participants

  AMD, Cray, Fujitsu, HP, IBM, Intel, NEC, PGI, Oracle, MS, TI, CAPS, NVIDIA
  ANL, LLNL, cOMPunity, EPCC, LANL, NASA, ORNL, RWTH, TACC

OpenMP Fortran API, Version 1.0,	1997
OpenMP C API, Version 1.0,	1998
OpenMP 2.0 API for Fortran,	2000
OpenMP 2.0 API for C/C++,	2002
OpenMP 2.5 API for C/C++ & F90	2005
OpenMP 3.0 Tasks	May 2008
OpenMP 3.1	July 2011
OpenMP 4.0 **Affinity, Devices, Depend, SIMD**	July 2013
OpenMP 4.5	

OpenMP History



OpenMP 3.0: The World is still flat, no support for NUMA (yet)!
OpenMP is hardware agnostic, it has no notion of data locality.
The Affinity problem: How to maintain or improve the nearness of threads and their most frequently used data.
Or:
Where to run threads?
Where to place data?

http://terboven.wordpress.com/

Thread binding was added in OpenMP 4.0

Advantages/Disadvantages of OpenMP

- Pros
  - Shared Memory Parallelism is easier to learn.
  - Coarse-grained or fine-grained parallelism
  - Parallelization can be incremental
  - Widely available, portable
  - Converting serial code to OpenMP parallel can be easier than converting to MPI parallel.
  - SMP hardware is prevalent now.
    - Supercomputers **and** your desktop/laptop (and your phones)
    - GPUs (Graphics Cards), MICs (Many-cores CPUs)
- Cons
  - Scalability limited by memory architecture.
  - Available on SMP systems “only”.
  - Beware: “Upgrading” large serial code may be hard.
- Takeaway
  - Threads are essential to use today’s hardware effectively

OpenMP Parallel Directives

Supports parallelism by Directives in Fortran, C/C++,…

Unlike others that require base language changes and constructs

Unlike MPI which supports parallelism through communication lib.

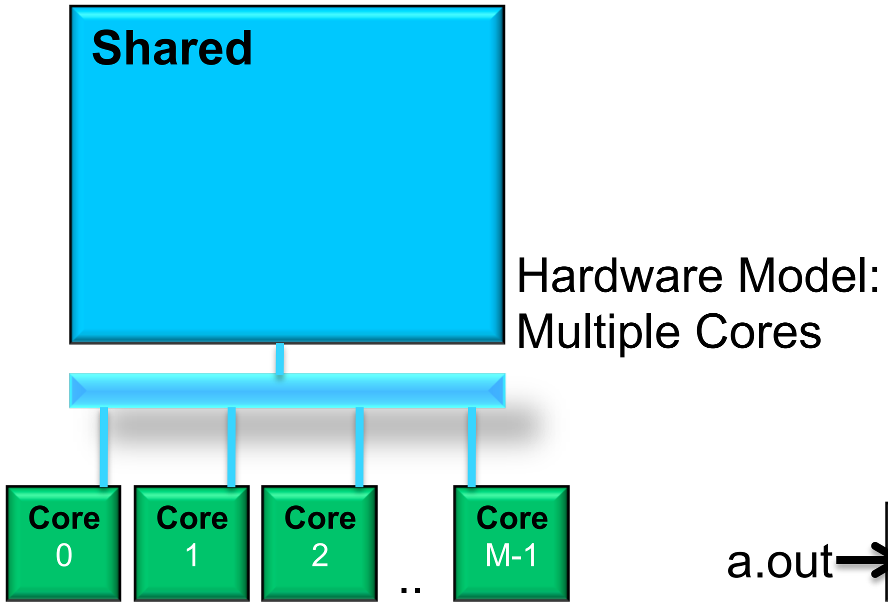
Processes on an SMP System

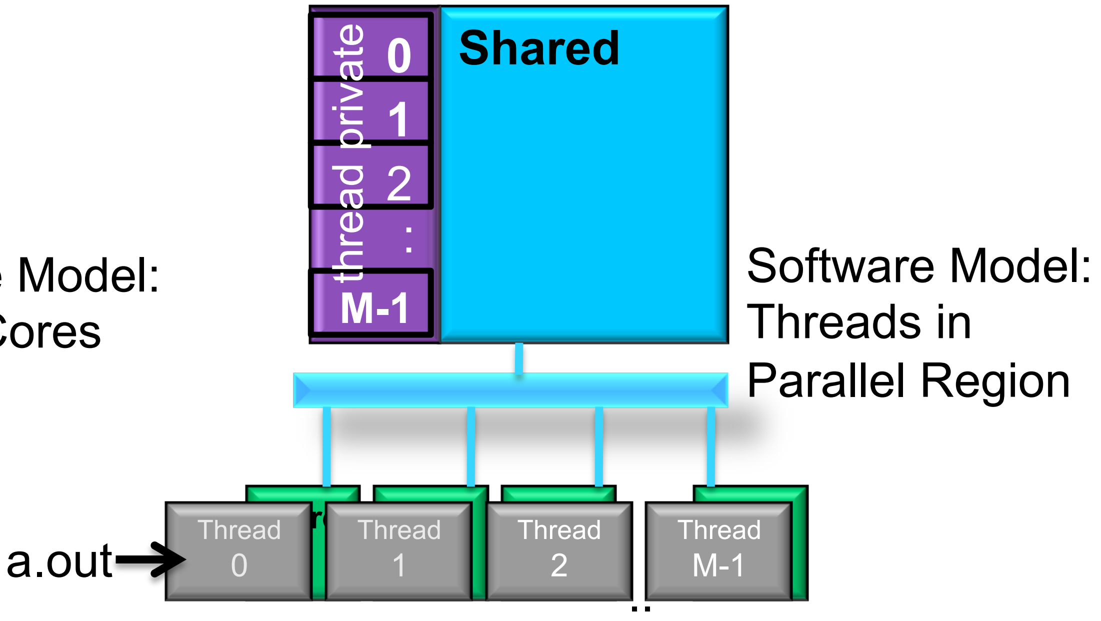
- The OS starts a process
  - One instance of your computer program, the “a.out”
- Many processes may be executed on a single core through “time sharing” (time slicing).
  - The OS allows each process to run for awhile.
- The OS may run multiple processes concurrently on different cores.
- Security considerations
  - Independent processes have no direct communication (exchange of data) and are not able to read another process’s memory.
- Speed considerations
  - Time sharing among processes has a large overhead.

OpenMP Threads

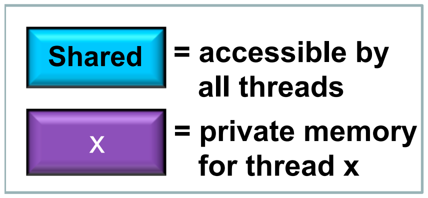
- Threads are instantiated (forked) in a program
- Threads run concurrently*
- All threads (forked from the same process) can read the memory allocated to the process.
- Each thread is given some private memory only seen by the thread.
- *When the # of threads forked exceeds the # of cores, time sharing (TS) will occur. Usually you would not do this. (But TS with user threads is less expensive than TS with processes). → max: one thread per 'core'
- Implementation of threads differs from one OS to another.

Programming with OpenMP on Shared Memory Systems





M threads are usually mapped to M cores.



Examples of Parallel Computing

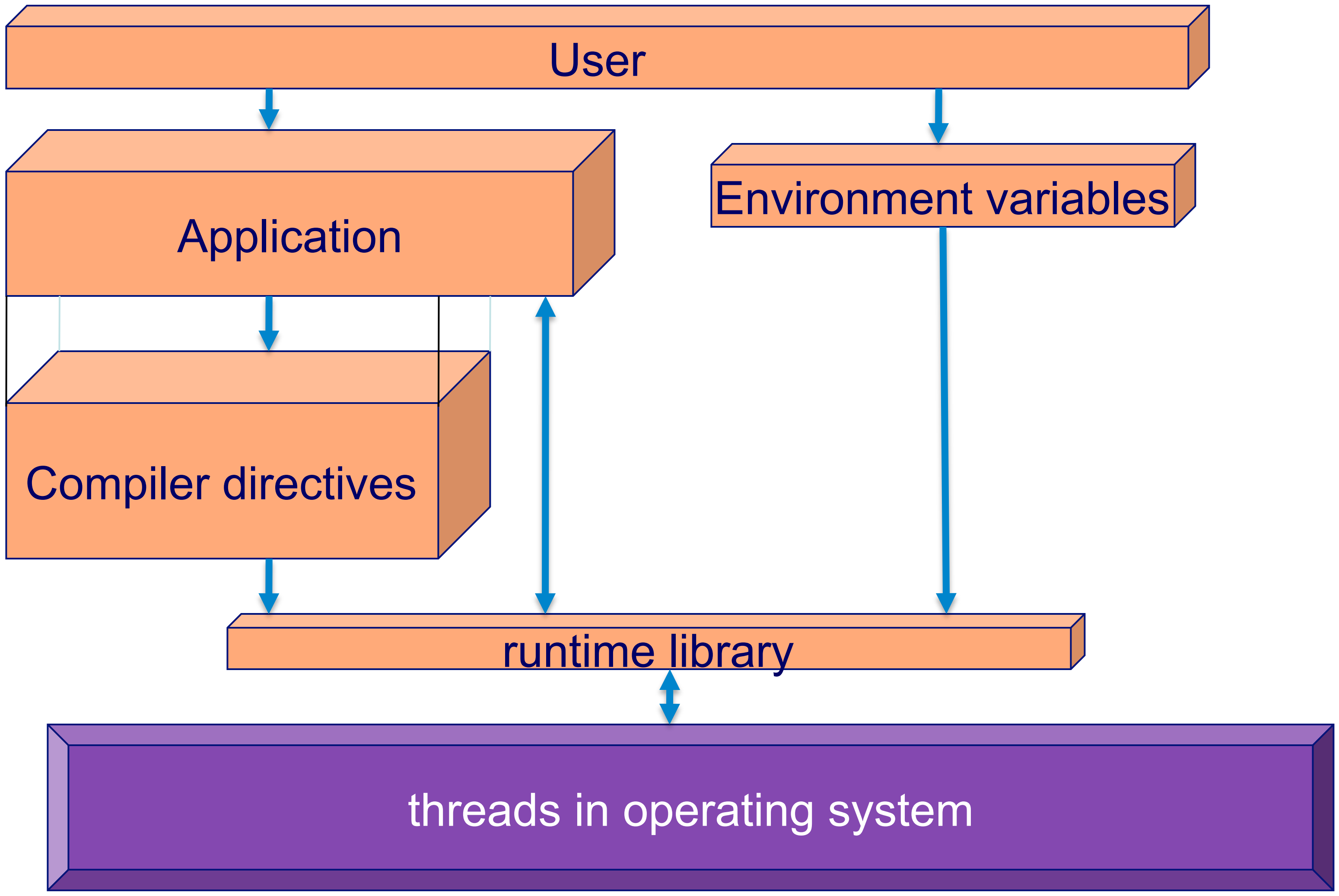
- Concurrent execution of computational work (tasks).
  - Tasks execute independently
  - Variable Updates must be mutually exclusive
  - Synchronization through barriers

```
// We use loops for
// repetitive tasks
for (i=0; i<N; i++){

  a[i] = b[i] + c[i];
}
```

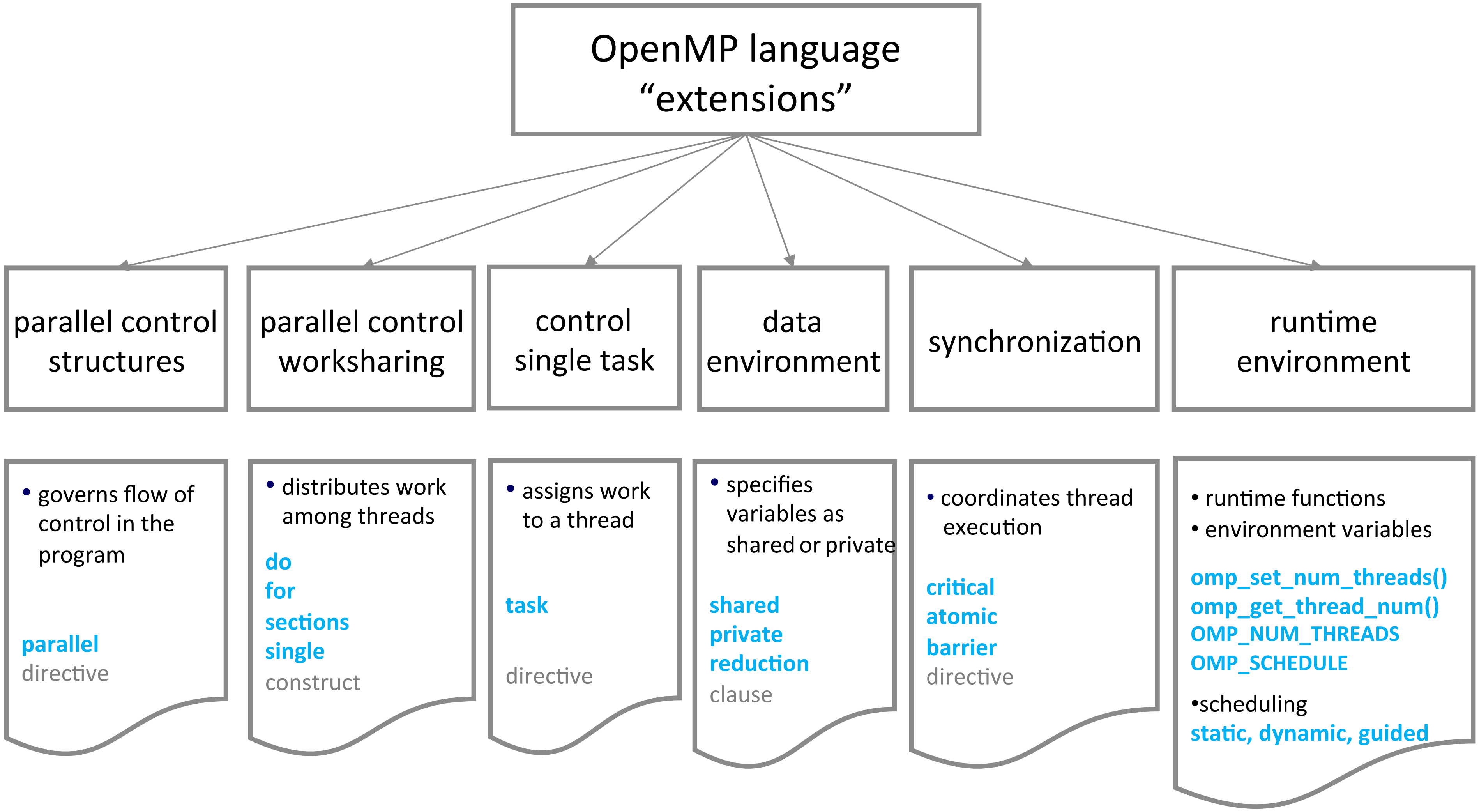
```
// We often update
// variable(s)
for (i=0; i<N; i++){

  prod = prod + b[i]*c[i];
}
```

Parallel Directives …

OpenMP Architecture



OpenMP Constructs



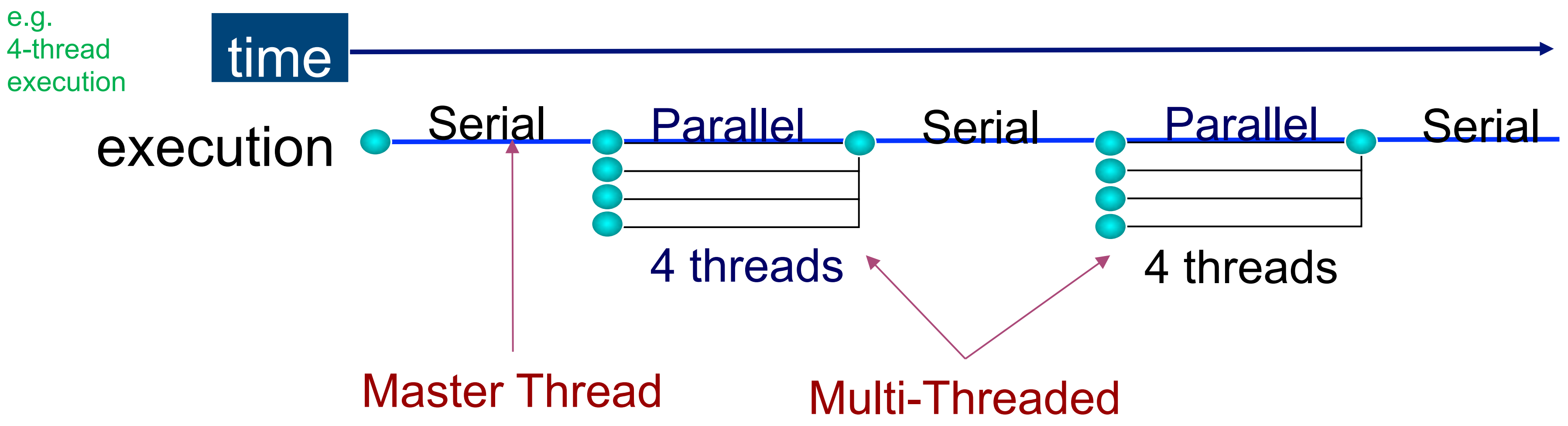
OpenMP language "extensions"

- **parallel control structures**
  - governs flow of control in the program
  - parallel
  - directive
- **parallel control worksharing**
  - distributes work among threads
  - do
  - for
  - sections
  - single
  - construct
- **control single task**
  - assigns work to a thread
  - task
  - directive
- **data environment**
  - specifies variables as shared or private
  - shared
  - private
  - reduction
  - clause
- **synchronization**
  - coordinates thread execution
  - critical
  - atomic
  - barrier
  - directive
- **runtime environment**
  - runtime functions
  - environment variables
  - omp_set_num_threads()
  - omp_get_thread_num()
  - OMP_NUM_THREADS
  - OMP_SCHEDULE
  - scheduling
  - static, dynamic, guided

Directive Sentinels: "!$omp" and "#pragma omp" not shown.

OpenMP Fork-Join Parallelism

- **Programs begin as a single process**: master thread
- Master thread executes in serial mode until the parallel region construct is encountered
- Master thread **creates (forks)** a team of parallel **threads** that simultaneously execute tasks **in a parallel region**
- After executing the statements in the parallel region, team threads synchronize and terminate (**join**) but master continues



OpenMP Syntax

- OpenMP Directives: **Sentinel**, **construct** and **clauses**

  #pragma omp ***construct*** **[*clause* [[,]*clause*]…]** **C**

  !$omp ***construct*** **[*clause* [[,]*clause*]…]** **F90**

- Example

  #pragma omp parallel num_threads(4) **C**

  !$omp parallel num_threads(4) **F90**

- Function prototypes and types are in the file:

  #include <omp.h> **C**

  use omp_lib **F90**

- Most OpenMP constructs apply to a “structured block”, that is, a block of one or more statements with one point of entry at the top and one point of exit at the bottom

OpenMP Directives

- **OpenMP directives begin with special comments/pragmas that open-aware compilers interpret. Directive sentinels are:**

F90	**!$OMP**
C/C++	**# pragma omp**

Syntax: ***sentinel*** **construct** ***clauses*** ***defaults used when no clauses present***

Fortran

```
!$OMP parallel
  ...
!$OMP end parallel
```

C/C++

```
# pragma omp parallel
  {...}
```

Fortran **Parallel regions are enclosed by** **enclosing directives.**
C/C++ **Parallel regions are enclosed by** **curly brackets.**

Parallel Region

```
    ...
1  !$omp parallel
2        code statements
3        call work(…)
4  !$omp end parallel
```

```
...
#pragma omp parallel
{ code statements
       work(…)
}
```

Line 1 — **Team of threads formed.**

Lines 2-3 — **This is the parallel region**
Each thread executes code block and subroutine call or function.
No branching (in or out) in a parallel region.

Line 4 — **All threads synchronize at end of parallel region (implied barrier).**

Parallel Region

```
   ...
1 !$omp parallel
2       code statements
3       call work(…)
4  !$omp end parallel
```

```
...
#pragma omp parallel
{ code statements
       work(…)
}
```

Line 1	**Team of threads formed.**
Lines 2-3	**This is the parallel region**
	Each thread executes code block and subroutine call or function.
	No branching (in or out) in a parallel region.
Line 4	**All threads synchronize at end of parallel region (implied barrier).**

In example above, user must explicitly create independent work (tasks) in the code block and routine (using thread id and total thread count).

Parallel Region & Thread Number

```
1   use omp_lib
        ...
2       nt  = 1
3   !$omp parallel
4       nt = omp_get_num_threads()
5       call work(nt)
6   !$omp end parallel

7   #include <omp.h>
       ...
8       int nt=1;
9   #pragma omp parallel
10  {
11      nt = omp_get_num_threads();
12      ierr=work(nt);
13  }
```

Every thread can inquire the total number of threads (nt in line 4).

Parallel Region & Thread Number

```
!$ use omp_lib
     ...
   nt   = 1
!$omp parallel private(id)
!$ nt = omp_get_num_threads()
   call work(nt)
!$omp end parallel

#include <omp.h>
  ...
   int nt=1;
#pragma omp parallel
{
   nt = omp_get_num_threads();
   ierr=work(nt);
}
```

```
#ifdef _OPENMP
...
#endif
```

For compiling without OpenMP, comment out runtime routines (!$) in F90; use ifdef's in C/C++.

Parallel Region

```
    ...
!$omp parallel

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

```
...
#pragma omp parallel
{
  for(i=0;i<n;i++){
    work(i);
    a[i] = b[i]+c[i];
  }
}
```

In above example the do/for loop iterations are **not** split among the threads. The example shows replicated work.

Note: see discussion of private variables later

Parallel Region with Worksharing Construct

```
   ...
!$omp parallel

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

```
...
#pragma omp parallel
{
#pragma omp for
  for(i=0;i<n;i++){
    work(i);
    a[i] = b[i]+c[i];
  }
}
```

In above example the do/for loop iterations are split among the threads via the do/for worksharing constructs.

Parallel Region & Work-Sharing

Use OpenMP directives to specify Parallel Region, Work-Sharing constructs, and Mutual Exclusion

```
parallel


end parallel
```

Use parallel … end parallel for F90
Use parallel {…} for C

Code block	Each Thread Executes
do / for	Work Sharing
sections	Work Sharing
single	One Thread (Work sharing)
master	One Thread
critical	One Thread at a time
atomic	One Thread at a time

```
parallel do/for
parallel sections
```

A single worksharing construct (e.g. a do/for) may be combined on a parallel directive line.

OpenMP Combined Directives

- Combined directives
  - parallel do/for and parallel sections
  - Same as parallel region containing only do/for or sections worksharing construct

```
!$omp parallel do
  do i = 1, 100
    a(i) = b(i)
  end do
```

```
#pragma omp parallel for
for(i=0;i<100;i++){
  a[i] = b[i];
}
```

trip count required
no exit
cycle ok

trip count required
no break
limited C++ throw.
continue ok

Work Sharing – do/for

Worksharing (WS) constructs: do/for, sections, and single

- **With Worksharing: Threads execution their "share" of statements in a PARALLEL region.**
- **Do/for Worksharing may require run-time work distribution and scheduling**

```
1 !$OMP PARALLEL DO
2    do i=1,n
3     a(i)=b(i)+c(i)
     enddo
5 !$OMP END PARALLEL DO
```

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

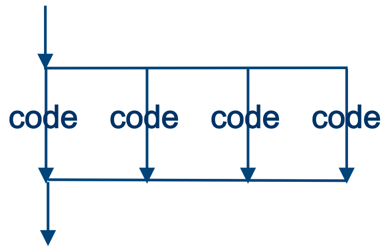
Line 1 Team of threads formed (parallel region).
Line 2-4 Loop iterations are split among threads.
Implied barrier at "enddo" and "}".
Line 5 (Optional) end of parallel loop.

- **Each loop iteration must be independent of other iterations.**

Replicated and Work Share Constructs

- **Replicated:** **Work blocks are executed by all threads.**
- **Work Sharing:** **Work is divided among threads.**

```
PARALLEL
   {code}
END PARALLEL
```



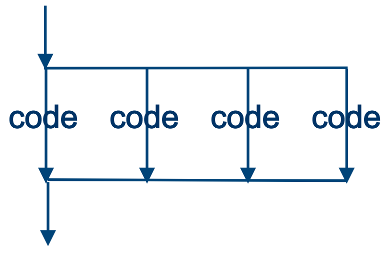
Replicated

Replicated and Work Share Constructs

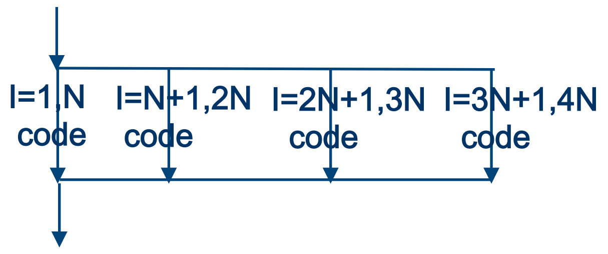
- **Replicated:** **Work blocks are executed by all threads.**
- **Work Sharing:** **Work is divided among threads.**

```
PARALLEL
   {code}
END PARALLEL
```

```
PARALLEL DO
  do I = 1,N*4
     {code}
  end do
END PARALLEL DO
```



Replicated



Work Sharing

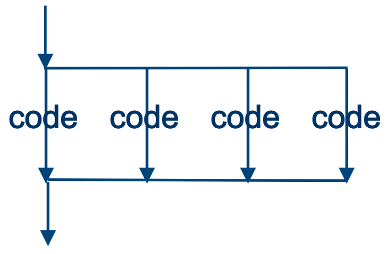
Replicated and Work Share Constructs

- **Replicated: Work blocks are executed by all threads.**
- **Work Sharing: Work is divided among threads.**

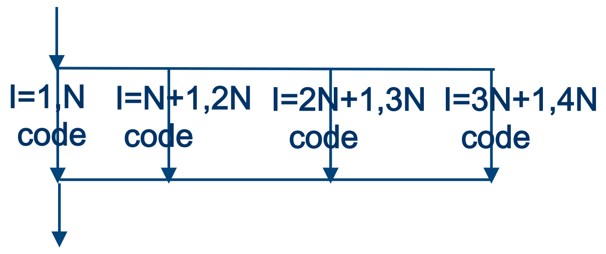
```
PARALLEL
  {code}
END PARALLEL
```

```
PARALLEL DO
  do I = 1,N*4
    {code}
  end do
END PARALLEL DO
```

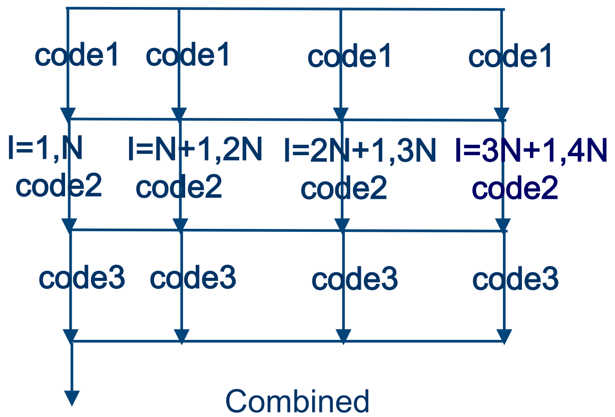
```
PARALLEL
    {code1}
DO
  do I = 1,N*4
    {code2}
  end do
    {code3}
END PARALLEL
```



Replicated



Work Sharing



Combined

OpenMP Worksharing Scheduling

Clause Syntax: parallel do/for schedule(*schedule-type* [,*chunk-size*])

Schedule Type

schedule (static[, chunk])

- Threads receive chunks of iterations in thread order, round-robin. (Divided “equally” if no chunk size.)
- Good if every iteration contains same amount of work
- May help keep parts of an array in a particular processor's cache– good between parallel do/for’s.

schedule (dynamic[, chunk])

- Thread receives chunks as it (the thread) becomes available for more work
- Default chunk size may be 1
- Good for load-balancing

OpenMP Worksharing Scheduling

schedule (guided[, chunk])

- Thread receives chunks as the thread becomes available for work
- Chunk size decreases exponentially, until it reaches the chunk size specified (default is 1)
- Balances load and reduces number of requests for more work

schedule (runtime)

- Schedule is determined at run-time by the OMP_SCHEDULE environment value.
- Useful for experimentation

OpenMP Worksharing Scheduling

For example, loop with 100 iterations and 4 threads

- schedule(static)

Thread	0	1	2	3
Iteration	1-25	26-50	51-75	76-100

- schedule(dynamic, 15) *(one possible outcome)*

Thread	0	1	3	2	1	3	2
Iteration	1-15	16-30	31-45	46-60	61-75	76-90	90-100

- schedule(guided, 8) *(one possible outcome)*

Thread	0	1	2	3	3	2	3	1
Iteration	1-25	26-44	45-58	59-69	70-77	78-85	86-93	93-100

TACC

Parallel – Worksharing - Schedule

- Combined directives
  - parallel do/for
  - Schedule clause added

```
!$omp parallel do schedule(static,8)
  do i = 1, 100
    a(i) = b(i)
  end do
```

```
#pragma omp parallel for schedule(static,8)
    for(i=0;i<100;i++){
      a[i] = b[i];
    }
```

How will the loop iteration be scheduled?
Assume that the number of threads is 4

OpenMP WorkSharing -- Sections

- SECTIONS
  - Blocks of code are split among threads - task parallel style
  - A thread might execute more than one block or no blocks
  - Implied barrier

```
!$OMP SECTIONS

!$OMP SECTION
      CALL TASK1()
!$OMP SECTION
      CALL TASK2()
!$OMP SECTION
      CALL TASK3()

!$OMP END SECTIONS
```

```
#pragma omp sections
{
#pragma omp section
      { TASK1( ); }
#pragma omp section
      { TASK2 ( ); }
#pragma omp section
      { TASK3 ( ); }

}
```

OpenMP Worksharing -- Single

- SINGLE (or MASTER)
  - Block of code is executed only once by a single thread (or the master thread)
  - Implied barrier (ONLY single)

```
!$OMP single

  glob_count = glob_count + 1
  print *, glob_count

!$OMP end single
```

```
#pragma single
{
  glob_count++;
  printf("%d\n", glob_count);

}
```

Three major components

1. **`Parallel`**: Starting forking/joning threads
2. **`do/for`**, **`section`**, **`single`**: worksharing

3. *Synchronization*
   - **`Private`**
   - **`Reduction`**
   - **`Single`** and **`master`**
   - **`Critical`**

Component #3 will keep us busy

Private variables

Let’s go back to our simple parallel loop

Parallel construct

Worksharing (WS) constructs: do/for

```
1 !$OMP PARALLEL DO
2    do i=1,n
3     a(i)=b(i)+c(i)
     enddo
5 !$OMP END PARALLEL DO
```

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

Line 1 Team of threads formed (parallel region).
Line 2-4 Loop iterations are split among threads.
Implied barrier at “enddo” and “}”.
Line 5 (Optional) end of parallel loop.

- Each loop iteration must be independent of other iterations.
- **How many variables are used in the code snippet?**

OpenMP Data Scoping

SHARED - Variable is shared (seen) by all processors.
PRIVATE - Each thread has a private instance of the variable.

Defaults:
Fortran: **do** indices are private, all other variables are shared.
C: only OMP workshare **for** <u>indices</u> have private indices.
Both languages: Everything else is shared

```
!$omp parallel do shared(a), &
                     private(t1,t2)
  do i = 1,1000
    t1 = f(i); t2 = g(i)
    a(i) = sqrt(t1**2 + t2**2)
  end do
```

```
#pragma parallel for shared(a), \
                         private(t1,t2)
for(i=0; i<1000; i++){
  t1 = f[i];  t2 = g[i];
  a[i] = sqrt( (t1*t1 + t2*t2);
}
```

All threads have access to the same storage areas for a, but each loop has its own private copy of the loop index, i, t1, and t2.

OpenMP Clauses -- Scoping

#pragma omp directive-name [*clause* [[,]*clause*]…]
!$omp ***directive-name*** **[*clause* [[,]*clause*]…]**

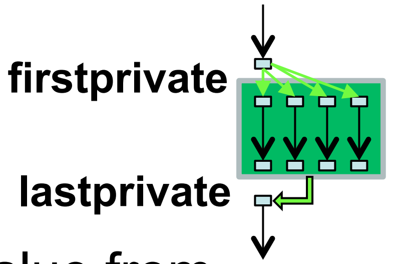
- Data scoping (See section 2.9.3.1-3of OpenMP 3.1 spec.)

  private(*variable list*)
  - Each thread has its own copy of the specified variable
  - Variables are undefined after work sharing region

  shared(*variable list*)
  - Threads share a single copy of the specified variable

  default(*type*)
  - A default of PRIVATE, SHARED or NONE can be specified
  - Note that loop counter(s) of work sharing constructs are always PRIVATE by default; everything else is SHARED by default

OpenMP Data Scoping



- Data scoping (continued)

  firstprivate(*variable list*)

  - Like PRIVATE, but copies are initialized using value from master thread's copy

  lastprivate(*variable list*)

  - Like PRIVATE, but final value is copied out to master thread's copy
  - For/DO: last iteration; SECTIONS: last section

  reduction(*op:variable*)

  - Each thread has its own copy of the specified variable
  - Can appear only in reduction operation
  - All copies are "reduced" back into the original master thread's variable

OpenMP Data Scoping

- Data scoping (continued)
  - do/for and parallel do/for constructs
    - index variable is automatically private
  - non-worksharing loops (nested loops)
    - Fortran: index variable is private (not so in C/C++)
- automatic storage variables
  - private, if with “duration” of scope inside the construct. (e.g. automatic variables in functions)

Variable Scoping, Fortran example

scope

```
program main
integer, parameter :: nmax=100
real*8   :: x(n,n)
integer :: n, j
...
n=nmax; y=0.0
!$omp parallel do
   do j=1,n
      call adder(x,n,j)
   end do
...
end program main
```

lexical extent

```
subroutine adder(a,m,icol)
integer,parameter :: nmax=100
real*8   :: a(m,m)
integer :: i, m, icol
save sum = 0.0     ! C: static
...
do i=1,m
   y(icol)=y(icol)+a(i,icol)
end do

sum=sum+y(icol)
...
end subroutine adder
```

dynamic extent

Worksharing Parallelism

- All about removing dependences <u>between threads</u>:
  - Remove dependences between threads.
  - Remove dependences between iterations (finer parallelism, often see this).
- Accomplished by:
  - Splitting dependences out of loop.
  - Exchange memory for dependences. (What??)
  - Exchanging loops
  - … Thinking outside of the equations (box)
- Making more work per thread.

Splitting dependencies out of loop

Loop fission

```
for(i=1; i<n; i++){
  b[i  ] = a[i  ]*r[i  ]
  a[i-1] = t[i-1]*s[i-1]
}
```

→

```
for(i=1; i<n; i++)
  b[i  ] = a[i  ]*r[i  ]

for(i=1; i<n; i++)
  a[i-1] = t[i-1]*s[i-1]
```

C/C++

```
do i=2,n
  b(i  ) = a(i  )*r(i  )
  a(i-1) = s(t-1)*t(t-1)
enddo
```

→

```
do i=2,n
  b(i  ) = a(i  )*r(i  )
enddo
do i=2,n
  a(i-1) = s(t-1)*t(t-1)
enddo
```

F90

TACC

Exchange memory for dependencies

Exchange Memory for dependence

```
for(i=1; i<n; i++)
   a[i-1] = f(a[i]);
```

→

```
for(i=1; i<n; i++)
  b[i-1] = f(a[i]);
for(i=0; i<n-1; i++)
  a[i]=b[i];
```

C/C++

```
do i=2,n
   a(i-1) = f(a(i))
enddo
```

→

```
do i=2,n
   b(i-1) = f(a(i))
enddo
do i=1,n-1
   a(i) = b(i)
enddo
```

F90

More Work per Thread

Loop Exchange (swap)

```
for(i=0; i<n-1; i++){
  for(j=0; j<n-1; j++){

    a[i][j]= a[i][j]+a[i+1][j];

}}
```

$\rightarrow$

```
for(j=0; j<n-1; j++){
  for(i=0; i<n-1; i++){

    a[i][j]= a[i][j]+a[i+1][j];

}}
```

C/C++

```
do i=1,n-1
  do j=1,n-1

    a(i,j)= a(i,j)+a(i+1,j)

enddo;enddo
```

$\rightarrow$

```
do j=1,n-1
  do i=1,n-1

    a(i,j)= a(i,j)+a(i+1,j)

enddo;enddo
```

F90

The Best Debug Tool

```
#pragma omp parallel private(tid)
{
  tid = omp_get_num_thread();
  printf("tid=%3.3d ...\n",tid);
{
```

C/C++

F90

```
!$omp parallel private(tid)

  tid = omp_get_num_thread()
  write(*,'("tid=",i3.3,"…”)') tid
!$omp end parallel
```

```
$ a.out |sort
tid=000 ...
tid=001 ...
...
tid=010 ...
```

Reduction

Let’s go back to our simple parallel loop

Parallel construct

Worksharing (WS) constructs: do/for

How can we parallelize this?

```
sum = 0
!$OMP PARALLEL
!$OMP DO
do i=1,n
  sum = sum + a(i)
enddo
!$OMP END PARALLEL
```

F90

```
sum = 0;
#pragma OMP PARALLEL
#pragma OMP for
for(i=0; i<n; i++){
  sum = sum + a[i];
}}
```

C/C++

F90

Reduction
Let’s go back to our simple parallel loop

Worksharing (WS)

```
integer :: a(n), sum


!$omp parallel


   !$omp do
     do i=1,n
        sum  =  sum      + a(i)
     enddo

!$omp end parallel
```

Reduction

F90

Worksharing (WS) – partial reductions

```
integer :: a(n), sum, psum(nt)


!$omp parallel private(id)

   id = omp_get_thread_num()
   !$omp do
     do i=1,n
       psum(id) = psum(id) + a(i)
     enddo

!$omp end parallel
```

Array for partial sums

Identify Thread

Each Thread's partial sum

Reduction

F90

Worksharing (WS) – partial reduction pre/post processing

```
integer :: a(n), sum=0, psum(nt)
do i=1,nt; psum(i)=0; end do

!$omp parallel private(id)

  id = omp_get_thread_num()
  !$omp do
    do i=1,n
      psum(id) = psum(id) + a(i)
    enddo

!$omp end parallel

do i=1,nt; sum=sum+psum(i); end do
```

Initialize Partial Sum

Reduce Partial Sums

C/C++

Reduction

Worksharing (WS) –

```
int a[n], sum;


#pragma omp parallel
{

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

C/C++

Reduction

Worksharing (WS) – partial reduction pre/post processing

```
int a[n], sum, psum[nt];


#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  #pragma omp for
    for(i=0; i<n; i++){
       psum[id] =  psum[id] + a[i];
    }
}
```

Array for partial sums

Identify Thread

Each Thread’s partial sum

C/C++

Reduction

Worksharing (WS) – partial reduction pre/post processing

```
int a[n], sum=0, psum[nt];
for(i=0; i<nt; i++) psum[i]=0;

#pragma omp parallel private(id)
{
  id = omp_get_thread_num();
  #pragma omp for
    for(i=0; i<n; i++){
       psum[id] =  psum[id] + a[i];
    }
}

for(i=0; i<nt; i++) sum += psum[id];
```

Initialize Partial Sum

Reduce Partial Sums

TACC

Reduction
But– OpenMP has the reduction clause to do this for you

Worksharing (WS) constructs:

Reduction operation

```
sum = 0
!$OMP PARALLEL reduction(+:sum)
!$OMP DO
do i=1,n
  sum = sum + a(i)
enddo
!$OMP END PARALLEL
```

F90

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

C/C++

Lets discuss:

How many ‘copies’ of sum exist?
Why do we have to specify the operator?
Why do we have to initialize ‘sum’?

TACC

Reduction

Let's go back to our simple parallel loop

Parallel construct

Worksharing (WS) constructs: do/for

Reduction operation

```
sum = 0
!$OMP PARALLEL reduction(+:sum)
!$OMP DO
do i=1,n
  sum = sum + a(i)
enddo
!$OMP END PARALLEL
```

F90

Lets discuss:

What happens if we initialize sum to 5.?

What happens if we change the reduction operation to 'multiply' (*)?

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

C/C++

F90

OpenMP Data Scoping

```
sum = 0
!$omp parallel do reduction(+:sum)
do i = 1, 1000
   sum = sum + a(i)
end do
! Each thread's copy of sum is added
! to original sum at end of loop

!$omp parallel do lastprivate(temp)
do i = 1, 1000
   temp = f(i)
end do
print *, 'f(1000) == ', temp
! temp is equal to f(1000) at end of loop
```

C/C++

OpenMP Data Scoping

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for(i=0;i<N;i++){
   sum = sum + a[i];
}
//Each thread's copy of sum is added
//to original sum at end of loop
printf("sum= %f\n",sum);

#pragma omp parallel for lastprivate(temp)
for(i=0;i<N;i++){
   temp = f[i];
}
printf("f(1000) == %f\n", temp);
//temp is equal to f(1000) at end of loop
```

TACC

Debugging

- To debug parallel code -- know semantics of the language
  - The semantics (directives) tells you the restraints on the parallel execution.
  - It is a prescription of behavior of a task relative to other tasks. Know the restraints and what is not constrained.
    - Reread code– think parallel
    - Review: expected behavior vs spec behavior vs implementation behavior – expected behavior does not take into account the unconstrained concurrency.

OpenMP Data Scoping

Sections: “Each structured block is executed once by one of the threads in the team in the context of its implicit task.” “The method of scheduling the structured blocks among the threads in the team is implementation defined.” pg 65-66.

```
cnt = 0
#pragma omp parallel
#pragma sections firstprivate(cnt)
{
   #pragma omp section
   { cnt += 1; printf(”%d\n",cnt);

   #pragma omp section
   { cnt += 1; printf(“%d\n”,cnt);
}
```

What happens in these 3 cases:

```
1. No clause
2. private(cnt)
3. firstprivate(cnt)
```

```
1 race condition
2 undefined or “initialized”
3 possible race condition
  (same thread executes both
  sections)
```

OpenMP Data Scoping

Sections: "Each structured block is executed once by one of the threads in the team in the context of its implicit task." "The method of scheduling the structured blocks among the threads in the team is implementation defined." pg 65-66.

```
cnt = 0
!$omp parallel
!$sections firstprivate(cnt)

   !$omp section
     cnt = cnt + 1; print*,cnt

   !$omp section
     cnt = cnt + 1; print*,cnt

!$omp end sections
!$omp end parallel
```

What happens in these 3 cases:

1. `No clause`
2. `private(cnt)`
3. `firstprivate(cnt)`

```
1 race condition
2 undefined or "initialized"
3 possible race condition
  (same thread executes both
  sections)
```

OpenMP Worksharing Directives

- NOWAIT clause
  - Threads encounter a barrier synchronization at end of worksharing constructs.
  - Specifies that threads completing assigned work can proceed.

C/C++

Include as clause in C/C++.

```
#pragma omp for      nowait
#pragma omp single   nowait
#pragma omp sections nowait
```

F90

Fortran include on end statement

```
!$omp end do       nowait
!$omp end single   nowait
!$omp end sections nowait
```

TACC

Try this on your own: Hello OpenMP

- Get on a compute node:

Login to stampede and execute idev to get a compute node for your work.

- Follow instructions in the README file:

Create a file (hello.f90 or hello.c) with contents on the next page.

- Useful commands:

```
top                                  # then hit the number 1 key to load on cores
cat /proc/cpuinfo                    # find the number of cores (processors)
/usr/bin/time -p  ./a.out                 # time the execution of a.out
```

OpenMP “Hello”

```
program hello
use omp_lib     !always do this
! integer :: omp_get_thread_num
print*, "hello, from master“

!$omp  parallel

print*, “id",omp_get_thread_num()

!$omp  end parallel
end program
```

```
#include <omp.h>
#include <stdio.h>
int main(){
printf("hello, from master\n");

#pragma omp parallel
  {
   printf("id%d\n",
              omp_get_thread_num());
   }
}
```

Compile with intel compiler

PGI compiler

```
pgf90  -mp hello.f90
pgcc   -mp hello.c
```

Intel compiler

```
ifort -qopenmp hello.f90
icc   -qopenmp hello.c
```

GNUcompiler

```
gfortran -fopenmp hello.f90
gcc        -fopenmp hello.c
```

OpenMP “Hello”

- Set Env. Var: export OMP_NUM_THREADS=4
- Execute: ./a.out
- The OpenMP routine omp_get_thread_num() reports unique thread #s between 0 and OMP_NUM_THREADS-1

OpenMP "Hello"

- Output after running on 4 threads :

```
hello, main
 thrd=1
 thrd=0
 thrd=3
 thrd=2
```

- Analysis of OpenMP output :
  - Threads are working completely independently
  - In real code threads usually have to cooperate to produce correct results, requiring synchronization