Schedule

- Beyond Threads
- Tasking
- SIMD
- Affinity
- C++
- OpenMP Offloading
  - GPUs in HPC
  - Introduction to the offloading paradigms
  - OpenMP target directive
  - OpenMP Mapping
  - OpenMP Variants, Metadata, interoperabilites

OpenMP beyond basics

CSC Summer Institute

October 9-11, 2023

Kent Milfeld (TACC)

Emanuele Vitali (CSC)

Slides available at:

General

Parallelize for orders of magnitude better performance.
CPU/GPU=> teams/grids =>threads+SIMD/SMT

Teach basics with concepts, not just syntax

- This is the debut for this course – the schedule may vary
- Content is about Concepts: Describes Features, Syntax, and use with examples (+labs).
- Some content comes from other tutorials/classes/documents (sources are referenced)

Outline of Intro

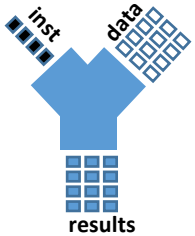
- History of OpenMP
- Summary of the Basics
- Hints, tricks and and Tips for OpenMP development
- Getting setup for Lumi/Frontera

OpenMP

OpenMP beyond threads

OpenMP is NOT just about threads

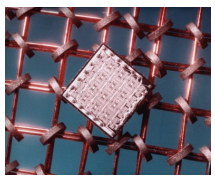
OpenMP deals with multiple levels of Parallelism

target	Across Devices	
parallel do/for, task	Across Cores	
simd	Across SIMD Lanes	

OpenMP beyond threads

OpenMP is NOT just about threads

OpenMP deals with Memory

Affinity	HW thrds, cores, sockets	
Allocations	API memory routines	 *
Code Transformation	loop transform directives	$for\ (i,j,k) \rightarrow for(l,m,n)$

* https://commons.wikimedia.org/wiki/File:Electronic_Memory.jpg
Steve Jurvetson from Menlo Park, USA, CC BY 2.0 <https://creativecommons.org/licenses/by/2.0>, via Wikimedia Commons

OpenMP -- board and membership

The OpenMP Architecture Review Board (ARB)
published first OpenMP Specification in 1997.
It is a non-profit organization with the following members:

https://www.openmp.org

OpenMP -- Information

Specifications, Examples and Reference Guide (Cheat Sheet)
and stackoverflow link are available at:

https://openmp.org -- click on Specifications

https://stackoverflow.com/questions/tagged/openmp

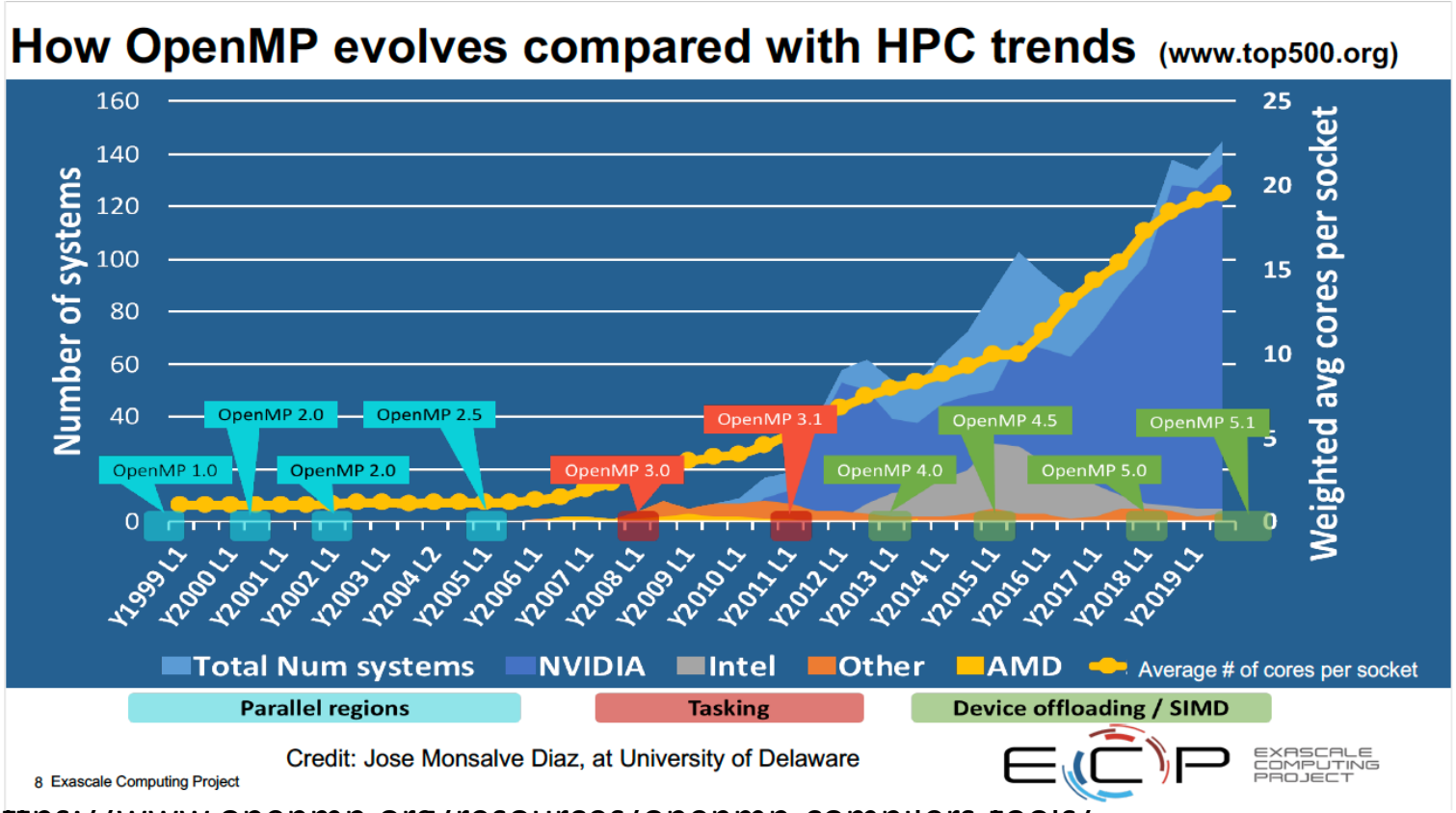
- Recommended forum for technical questions is Stack Overflow, using the tag OpenMP
- Alternatively post messages on:
  - Twitter: @OpenMP_ARB
  - LinkedIn: OpenMP Users
  - Facebook: OpenMP Group
  - Email: info@openmp.org

https://www.openmp.org/resources/openmp-compilers-tools/

OpenMP Releases

OpenMP Version	Year (Release)	_OPENMP (YYYYMM)	Examples (YY-MM)
5.2	2021	202111	2022-11
5.1	2020	202011	2021-08
5.0	2018	201811	2020-05
4.5	2015	201511	2016-11
4.0	2013	201307	2015-03
3.1	2011	201107	
3.0	2008	200805	

OpenMP -- Evolution



https://www.openmp.org/resources/openmp-compilers-tools/

Taken from: 2020 Exascale Computing Project Annual Meeting: https://www.ecpannualmeeting.com/archive/2020/overview.php
OpenMP 4.5 and 5.0 Tutorial (Offload) https://www.ecpannualmeeting.com/archive/2020/assets/overview/sessions/ff2020%20ECP-Tutorial-with-ECP-template.pdf

OpenMP -- early development in some areas

2.5->3.0 May 2008
- Environment Variables and setter:
- Tasking, thread limit, active levels, stack size, wait policy, set_num_threads, get_ancestor_thread_num,get levels

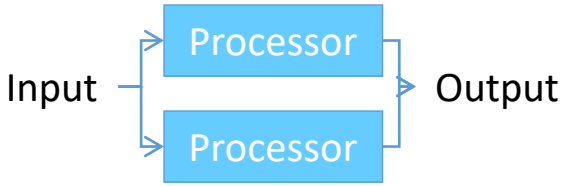
3.0->3.1 Jul 2011
- Tasking (final, mergeable, taskyield)
- Affinity (PROC_BIND)

3.1->4.0 Jul 2013
- target directives (data, update, declare, teams, distribute...,)
- SIMD directive
- User Defined Reduction
- Affinity (OMP_PLACES)
- Tasking (defined scheduling points, taskgroup, depend clause)
- OMP_DISPLAY_ENV

Summary of Basics

Why Parallel?

Power = CV^2F



Cap = 1.0c
Volts = 1.0v
Freq = 1.0f
Power = $1.0cv^2f$

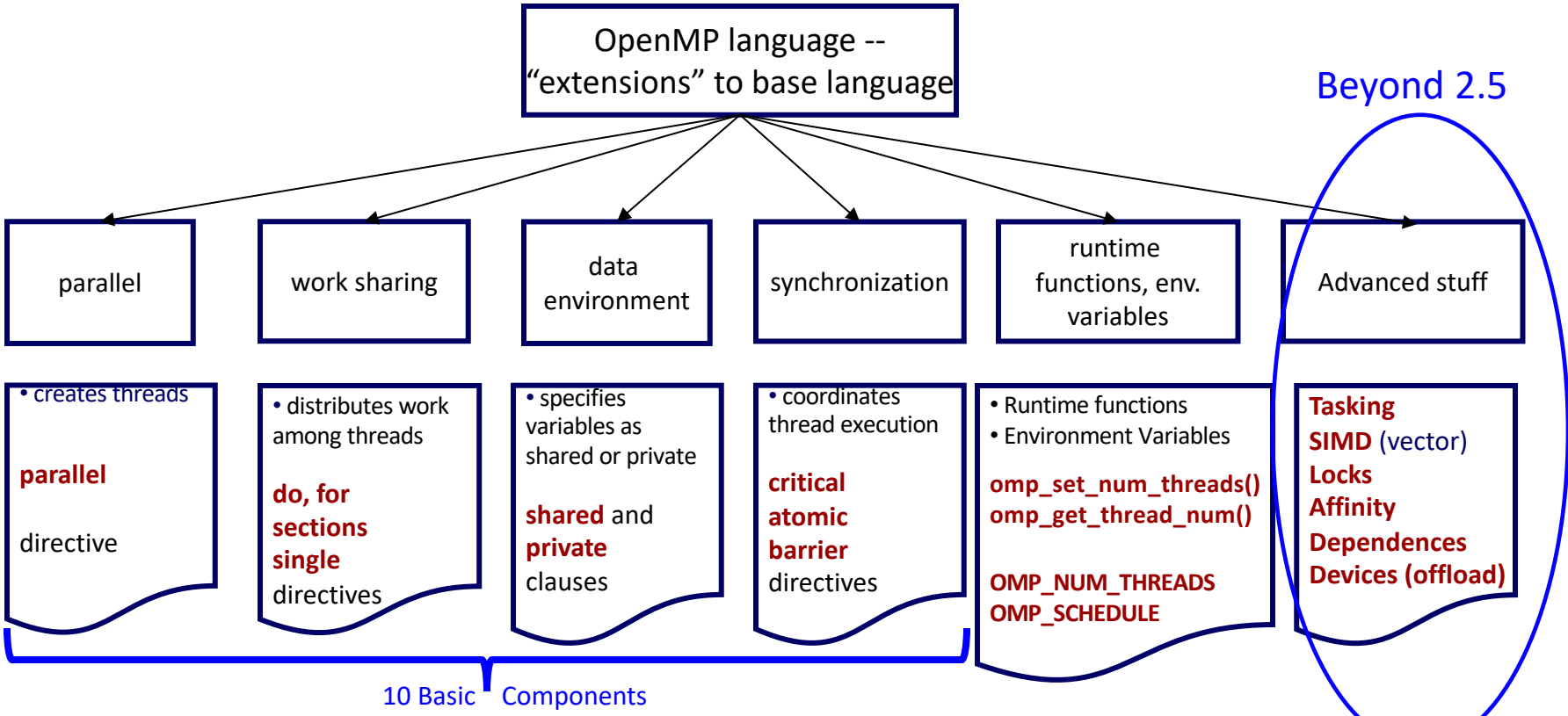
Do same work with 2 processors at 0.5 freq.

Voltage ~ Frequency
2x more wires ->
~2x Capacitance

Cap = 2.2c
Volts = 0.6v
Freq = 0.5f
Power = $0.4cv^2f$

https://www.youtube.com/watch?v=cMWGeJyrc9w&index=2&list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwn

OpenMP 2.5 Basics Constructs



OpenMP Syntax

- Compiler directive syntax:

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

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

- Example

Fortran

```
print*,"serial"

!$omp  parallel num_threads(4)
   …
!$omp end parallel


print*,"serial"
```

C/C++

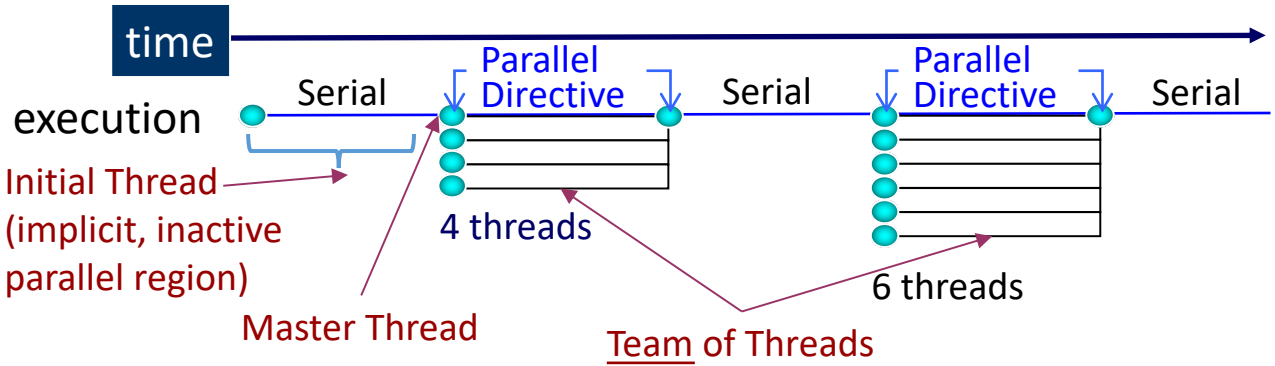
```
printf("serial\n");

#pragma omp parallel num_threads(4)
{
    …  // can be statements or just function call w.o. {}
}

printf("serial\n");
```

Early (2.5 legacy) Execution Model

- Programs begin as a single, *initial thread.*
- A thread encountering a parallel construct it becomes a *master thread*.
- After executing the statements in the parallel region, team threads synchronize and terminate (join) but *initial thread* continues



Note:
threads – concept of task comes in 3.0.

OpenMP Directive Scope

- OpenMP directives are comments/pragmas in source:

  F90 : **!$omp** free-format*

  C/C++ : **#pragma omp** sentinel

* Fortran Fixed Format:
!$OMP, **C$OMP** or ***$OMP**

- Parallel regions are marked by enclosing parallel directives

  F90 : !$omp parallel … !$omp **end** parallel

  C/C++ : #pragma omp on block **{ …}**, or single statement

- Work-sharing loops are marked by parallel do/for

Fortran

```
!$omp parallel
  ...
!$omp end parallel

!$omp parallel
  call foo(...)
!$omp end parallel
```

C/C++

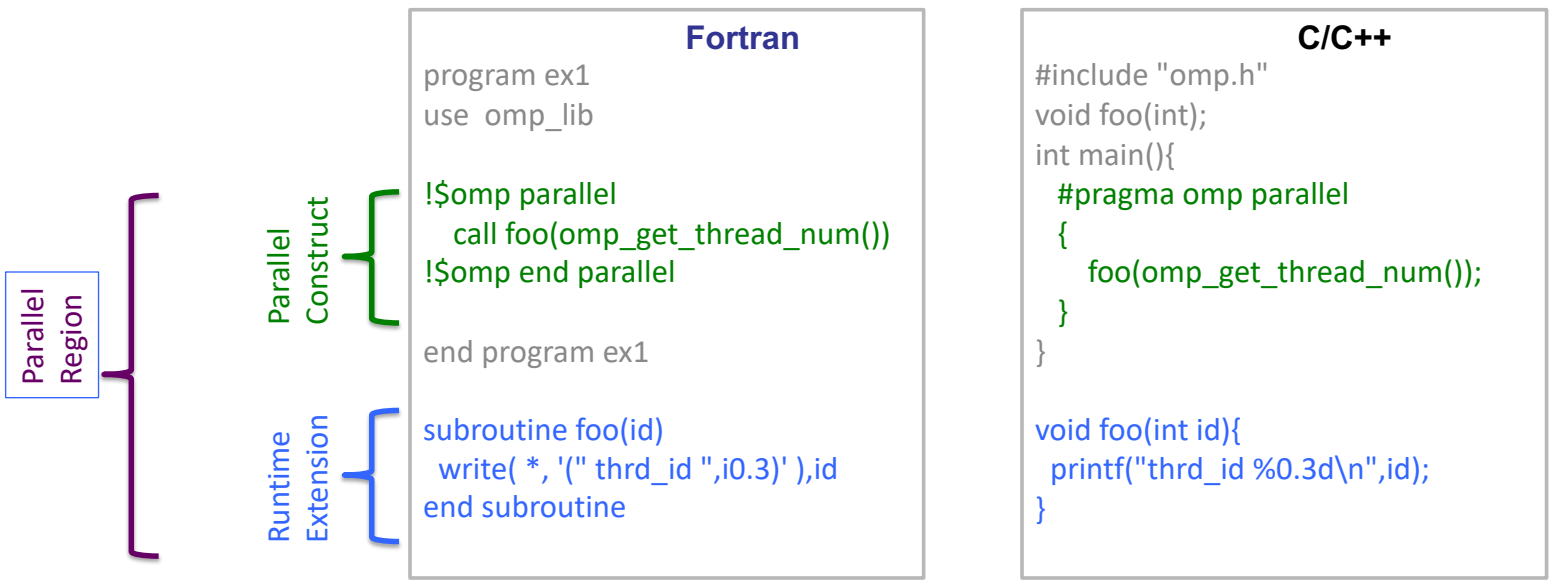
```
#pragma omp parallel
  {...
  }

#pragma omp parallel
  foo(...);
```

Any directive that uses omx or ompx in the sentinel is implementation defined

OpenMP Directive Scope

- construct – the lexical extent of executable directive
- region – all code encountered in a construct (construct + routines)



Parallel Region & work ID

F90

For example, to create a 4-thread parallel region.

```
$ export OMP_NUM_THREADS=4; a.out
```

```
real ::  A(1000); integer :: ID
!$omp parallel


   ID = omp_get_thread_num()
   call foo(ID, A)
!$omp end parallel
```

But we need to make id
Private to the thread– more later

Each thread executes the code within the structured block

Thread numbers range from 0 to Nthreads-1

Each thread calls foo(ID,A) with a different `ID {= 0 to 3}`

Parallel Region & work ID

C/C++

For example, to create a 4-thread Parallel region.

```
$ export OMP_NUM_THREADS=4; a.out
```

```
double A[1000];
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    foo(ID, A);
}
```

In C++ a local variables are private to the thread…

Each thread executes the code within the structured block

Thread numbers range from 0 to Nthreads-1

Each thread calls foo(ID,A) with a different `ID {= 0 to 3}`

Parallel Region & Worksharing

Use OpenMP directives to specify Parallel Region & Worksharing constructs

```
...omp parallel
{


} $!omp end parallel
```

C/C++ F90

Code block	Each Thread Executes
do / for	Worksharing
sections	Worksharing
single	Worksharing (one thread)

!$ and #pragma not shown here.

Work-sharing Directives assign threads to units of work.
There is an **implied barrier** at the end of a worksharing construct!

The *master* directive has no barrier, and is therefore not worksharing.

Parallel Region and Worksharing

C/C++

```
1  #pragma omp parallel
2  {
3    #pragma omp for
4    for (i=0; i<N; i++)
5    {
6      a[i] = b[i] + c[i];
7    }
8  }
```

Line 1 Team of threads formed (parallel region).

Line 3-7 Loop iterations are split among threads.
implied barrier at }

Each loop iteration must be independent of other iterations.

Parallel Region and Worksharing

```
1   !$OMP PARALLEL                    F90
2      !$OMP DO
3      do i=1,N
4         a(i) = b(i) + c(i)
5      enddo
6   !$OMP END PARALLEL
```

Line 1 Team of threads formed (parallel region).

Line 3-4 Loop iterations are split among threads by DO construct.

Line 5 !$OMP END DO is optional after the enddo.

Line 5 Implied barrier at enddo.

Each loop iteration must be independent of other iterations.

Parallel Region & Combined Constructs

Fortran

```
!$omp parallel
  !$omp do
    do ...
    end do
  !$omp end do
!$omp end parallel


!$omp parallel do
  do ...
  end do
                    !*
```

C/C++

```
# pragma omp parallel
{
  #pragma omp for
  for (...){...}
}



# pragma omp parallel for
  for(){...}
```

* Since Fortran DO has a block structure "!$omp end do" Is not required

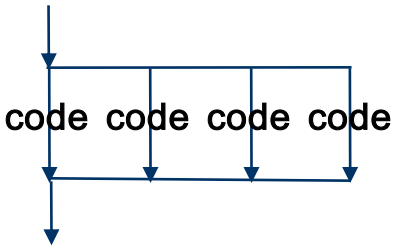
Parallel Region & Combined Constructs

Replicated : Work blocks are executed by all threads.

Work-Sharing : Work is divided among threads.

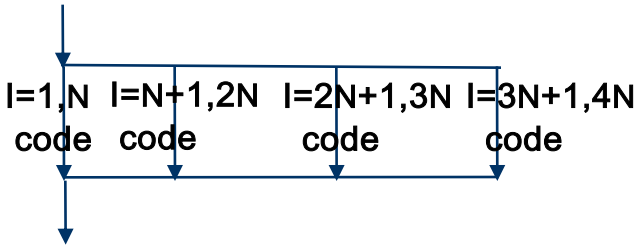
4 Threads

```
!$OMP PARALLEL
  {code}
!$OMP END PARALLEL
```



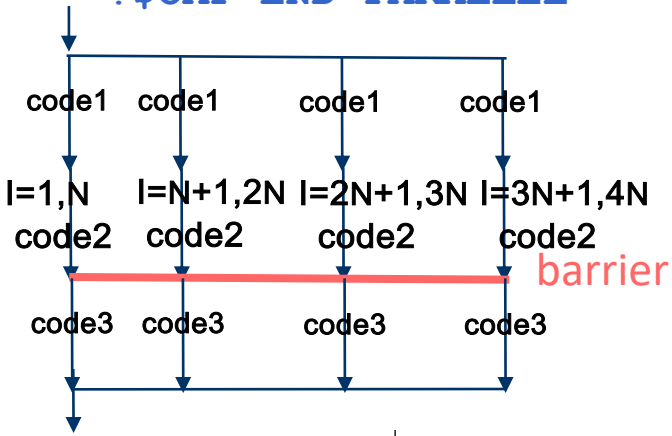
Replicated

```
!$OMP PARALLEL
!$OMP DO
  do i = 1,N*4
    {code}
  end do
!$OMP END PARALLEL DO
```



Work-Sharing

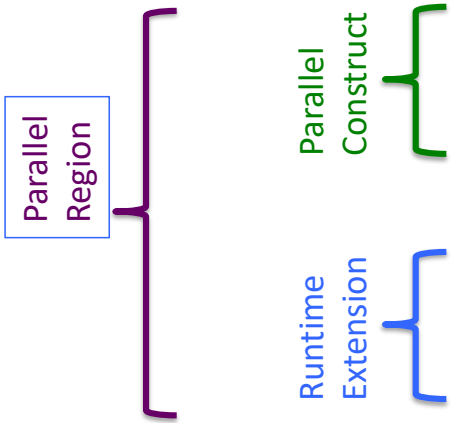
```
!$OMP PARALLEL
  {code1}
!$OMP DO
  do i = 1,N*4
    {code2}
  end do
  {code3}
!$OMP END PARALLEL
```



Combined

OpenMP Directive Scope

- construct – the lexical extent of executable directive
- region – all code encountered in a construct (construct + routines)



Fortran

```
program ex1
use  omp_lib

!$omp parallel
   call foo(omp_get_thread_num())
!$omp end parallel

end program ex1

subroutine foo(id)
  write( *, '(" thrd_id ",i0.3)' ),id
end subroutine
```

C/C++

```
#include "omp.h"
void foo(int);
int main(){
  #pragma omp parallel
  {
     foo(omp_get_thread_num());
  }
}

void foo(int id){
  printf("thrd_id %0.3d\n",id);
}
```

Schedule Clause for Worksharing

```
!$omp do schedule(...)
   do i=1,128
     A(i)=B(i)+C(i)
   enddo
```

```
#pragma omp schedule(.
   for(i=0;i<128;++i)
     A[i]=B[i]+C[i];
```

schedule(static)

Each CPU receives one set of contiguous iterations

schedule(static, C)

Iterations are divided round-robin fashion in chunks of size C

schedule(dynamic, C)

Iterations handed out in chunks of size C as CPUs become available

schedule(guided, C)

Each of the iterations are handed out in pieces of exponentially decreasing size, with C minimum number of iterations to dispatch each time

schedule (runtime)

Schedule and chunk size taken from the OMP_SCHEDULE environment variable

schedule(auto)

Decision is delegated to compiler and/or runtime

schedule(monotonic: ...)

Each thread executes the chunks that it is assigned in increasing logical iteration order (nonmonotonic – any order).

schedule(simd: ...) simd size is considered in chunk size—see SIMD.

Kind

Modifier

OpenMP Data Environments

Clauses control the data(-sharing) attributes of vars within a parallel region:

shared, private, reduction, firstprivate, lastprivate

Default variable scope (in parallel region):

1. Variables declared in main/program (C/F90) are shared by default
2. Global variables are shared by default
3. Automatic variables within routines called within a parallel region are private (reside on a stack private to each thread)
4. Loop index of worksharing loops are private.
5. Default scoping rule can be changed with `default` clause

Private and Shared Data

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

C/C++

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

F90

Reductions

- Operation that combines multiple elements to form a single result
- A variable that accumulates the result is called a reduction variable
- In parallel loops reduction operators and variables must be declared

```
float asum=0.0, aprod=1.0;

#pragma omp parallel for \
     reduction(+:asum)    \
     reduction(*:aprod)

  for (i=0; i<n; i++){
    asum  = asum  + a[i];
    aprod = aprod * a[i];
  }
```

C/C++

```
real asum=0.0, aprod=1.0

!$omp  parallel do        &
!$omp& reduction(+:asum)  &
!$omp& reduction(*:aprod)

   do i = 1,n
      asum  = asum  + a(i)
      aprod = aprod * a(i)
   enddo
```

F90

Each thread has a private **asum** and **aprod**, initialized to the operator's identity

After the loop execution, the master thread collects the private values of each thread and finishes the (global) reduction

Synchronization

- Synchronization is used to impose order constraints and to protect access to shared data
- High-Level Synchronization
  - critical
  - atomic Now has acquire and release semantics!
  - barrier
  - ordered
- Low-Level Synchronization
  - locks

Parallel Region & Combined Constructs

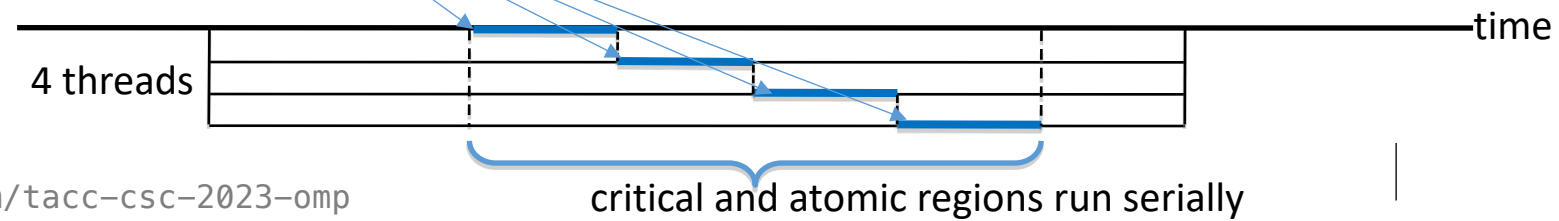
When only one thread at a time can execute a section of code (to avoid race conditions), create a **critical** region. In essence the code executes (and can even run worse than) serially.

Use an **atomic** directive for ("simple") atomic operations that possibly have hardware atomic support. The expression must have canonical (recognizable) form.

```
#pragma omp parallel shared(sum,x,y)
{...
   #pragma omp critical
   {
      update(x);
      update(y);
      sum=sum+1;
   }
}
```

```
!$omp parallel shared(sum,x,y)
 ...
   !$omp critical
     update(x);
     update(y);
     sum=sum+1;
   !$omp end critical

!$omp end parallel
```



nowait clause (non) synchronization

When a work-sharing region is executed, a barrier is implied - all threads must reach the barrier before any can proceed.

A **nowait** clause at the end of a worksharing region can remove an implied barrier.

```
#pragma omp parallel
{
  #pragma omp for nowait
      {
         for (i=0; i<n; i++)
           {work(i);}
      }
  #pragma omp for schedule(guided,k)
      {
         for (i=0; i<m; i++)
           {x[i]=y[i]+z[i];}
      }
}
```

C/C++

```
!$OMP PARALLEL

   !$OMP DO
      do i=1,n
         work(i)
      enddo
   !$OMP END DO NOWAIT
   !$OMP DO schedule(guided,k)
      do i=1,m
         x(i)=y(i)+z(i)
      enddo
   !$OMP END DO
!$OMP END PARALLEL
```

F90

OpenMP -- Summary of the basics

Concept	What to learn	Level	Optimize
Setup	How to compile OMP_NUM_THREADS	Basic	1 thread per 'core'
Parallel region	Forking/joining threads	Easy	Minimize number of fork/join
Work-sharing/replicated work	What do the threads do? 'omp do/for'	Work-sharing: easy Replicated: medium	Optimize scheduling Remove implicit barriers
Avoiding race conditions	----------	Will take effort!	----------
- Private variables	Why/how to shelter data	medium	
- reduction	Condensing a result from pieces	medium	
- Critical/atomic	All threads, but one thread at a time	harder	Do not serialize everything
- Single/master	One thread, and only one thread	harder	See 'min. fork/join'
Advanced	----------		
- Hybrid	MPI + OpenMP	medium	Interplay MPI/OpenMP
- Thread/memory pinning		medium	Utilize all cores
- SIMD	Vectorization with OpenMP	hard	Utilize vector lanes
- Tasking	Irregular problems	hard	

Hints, Tricks and Tips for OpenMP development

OpenMP Version (Compliance)

Timers

Warm up

OMP_SCHEDULE

Programming style

omp_get... _num_thread(), _thread_num(), etc.

The Compliance Table

Specification and Technical Report	Compliance Version	_OPENMP (YYYYMM)
TR: Nov 2022	for 6.0	
Spec: Nov 2021	5.2	202111
Spec: Nov 2020	5.1	202011
Spec: Nov 2018	5.0	201811
Spec: Nov 2015	4.5	201511
Spec: Jul 2013	4.0	201307
Spec: Jul 2011	3.1	201107

Releases now occur at SuperComputing (in Nov).

Some compilers specify _OPENMP with a TR report --if they have early-adapter implementations

There are many earlier TRs.

If you don't memorize the table, see openmp.org/specifications where the Spec and TR links are specified with versions and dates.

See Cray man page intro_openmp
(it has details about feature implementation)!

timers -- gettimeofday

Usually accurate to ns.
Call syntax may vary.
Fortran can easily call the C version

```
double gtod_secbase = 0.0E0;
double gtod_timer_() {
   struct timeval tv;
   double sec;

   gettimeofday(&tv, NULL);

// Always remove the LARGE sec value for improved accuracy
   if(gtod_secbase == 0.0E0)
      gtod_secbase = (double)tv.tv_sec;
   sec = (double)tv.tv_sec - gtod_secbase;

   return sec + 1.0E-06*(double)tv.tv_usec;
}
```

timers -- time stamp counter (tsc)

Highly specific to HW (x86 below, with 2 32-bit registers)
Every Clock Period (CP) counter is updated.
Accuracy is about 20 CP!
But, to get seconds, use the “clock rate”, most often just report CPs.

```
static __inline unsigned long long tsc(void){
  unsigned long a, d;
  unsigned long long d2;

  __asm__ __volatile__ ("rdtsc" : "=a" (a), "=d" (d));

//                          read time stamp counter
  d2 = d;
  return (unsigned long long) a | (d2 << 32);
};

unsigned long long   tsc_timer(void){ return  tsc(); }
unsigned long long  tsc_timer_(void){ return  tsc(); }
```

timers -- class timer

Collects time with a label – good for measuring multiple events.
Easy to use and incorporate.
Uses gettimeofday

```
#include “timer.hpp”
Timer timer;

timer.start(”for loop: 1”);
  for(...)...;
timer.stop()
          ...
timer.start(”function: 2”);
  fun();
timer.stop()

timer.print();
```

```
use mod_timer
type(cls_timer) :: timer

call timer%start(”do loop:  1")
   do i=1,N ...
call timer%stop
            ...
call timer%start(”function: 2")
call foo2()
call timer%stop

call timer%print
```

OpenMP Version (Compliance)

Timers

Warm up

OMP_SCHEDULE

Programming style

omp_get... _num_thread(), _thread_num(), etc.

warm up

- It may take considerable time to get threads from the OS for the first parallel region
- When benchmarking, always request the threads first in a do-nothing parallel region, for doing timings.

```
#include <omp.h>
...
#pragma omp parallel
if(omp_get_num_threads()<
   omp_get_thread_num()) exit(1);

timer.start("parallel loop: 1");
  #pragma omp parallel for
    for(...)...;
timer.stop()

timer.print();
```

```
use omp_lib
...
!$omp parallel
  if( omp_get_num_threads()< &
      omp_get_thread_num() ) stop

  call timer%start("do loop:  1")
     !$omp parallel do
     do i=1,N ...
  call timer%stop

  call timer%print
```

OMP_SCHEDULE env. var.

- When evaluating performance, you can quickly change the schedule kind by setting the schedule clause to runtime and setting the OMP_SCHEDULE environment variable to the kind.

```
$ env OMP_SCHEDULE='static'      ./a.out
$ env OMP_SCHEDULE='dynamic,10' ./a.out


 #pragma omp parallel for \
             schedule(runtime)
   for(int i=0; i<N; i++) {...}
```

```
$ env OMP_SCHEDULE='static'      ./a.out
$ env OMP_SCHEDULE='dynamic,10' ./a.out


  !$omp  parallel do &
  !omp&  schedule(runtime)
    do i=N;  ...; end do
```

OpenMP Version (Compliance)

Timers

Warm up

OMP_SCHEDULE

Programming style

omp_get... _num_thread(), _thread_num(), etc.

Program styles

- Consistent style can help organize across program files/projects
- Remember, it will be read again by a person and the compiler
  -- Use a style that allows a reader to easily evaluate what you have done (and yourself 6 months later)
- Personal experience (KFM) for Fortran coders, for any non-trivial procedure use *implicit none*. (Also, in subroutines *use omp_lib, only : omp_get...* lets readers know what is used.)
- Sometimes, going outside the style box helps in reading and debugging code.

Which "style" is easier to read, understand the logic, and modify?

```
subroutine func2(A)
   implicit none
   double precision   :: A(100,100)
   integer :: i1,i2,j1,j2

   do i1 =  1, 100,  4
      do j1 =  1, 100, 16
         do i2 = i1,  i1 + 3
            do j2 = j1, min(j1+15,100)
               A(j2,i2) = A(j2,i2) + 1
   end do; end do; end do; end do

end subroutine
```

```
subroutine func2(A)
   implicit none
   double precision   :: A(100,100)
   integer            :: i1,i2,j1,j2

   do i1 =  1, 100,  4
   do j1 =  1, 100, 16
   do i2 = i1,  i1 + 3
   do j2 = j1, min(j1+15,100)
      A(j2,i2) = A(j2,i2) + 1
   end do; end do; end do; end do

end subroutine
```

OpenMP API routines

There are many API routines that can help you

* Threads and Teams:	Count, number(ID), max -- getters and setters
* Target:	Device count, number, Is Initial Device
* Parallel:	In parallel region
* Memory	allocation routines (host and device)
* Affinity	Information – display and setters

Most are self descriptive:

OpenMP APIs

Most are self descriptive:

```
omp_set_num_threads
omp_get_num_threads
omp_get_max_threads
omp_get_thread_num
omp_in_parallel

omp_set_schedule
omp_get_schedule
omp_get_thread_limit
omp_get_team_size

omp_get_proc_bind
omp_get_num_places
omp_get_place_num_procs
...
omp_set_affinity_format
omp_get_affinity_format
omp_display_affinity
omp_capture_affinity

omp_get_num_teams
omp_get_team_num
omp_set_num_teams
omp_get_max_teams
omp_set_teams_thread_limit
omp_get_teams_thread_limit
omp_get_num_procs
omp_set_default_device
omp_get_default_device
omp_get_num_devices
omp_get_device_num
omp_is_initial_device
omp_get_initial_device
```

```
omp_target_alloc
omp_target_free
omp_target_is_present
omp_target_is_accessible
omp_target_memcpy
omp_target_memcpy_rect
omp_target_memcpy_async
omp_target_memcpy_rect_async
omp_target_associate_ptr
omp_target_disassociate_ptr
omp_get_mapped_ptr

omp_init_lock
omp_init_lock_with_hint
omp_init_nest_lock_with_hint
omp_destroy_lock
omp_set_lock
omp_unset_lock
omp_test_lock

omp_get_wtime
omp_get_wtick

omp_init_allocator
omp_destroy_allocator
omp_set_default_allocator
omp_get_default_allocator
omp_alloc
omp_free
omp_calloc
omp_realloc
............NOT all included!
```

Program Information Displays

- DISPLAY Environment Variable (EV) info

```
export OMP_DISPLAY_ENV=TRUE | FALSE
```

Shows ICVs associated with the EVs
after runtime evaluates EVs (or their defaults).
Use `VERBOSE` to show values of runtime
variables that may be modified by vendor-specific.

*ICV = internal control variables associated with a data environment

- 5.1

API routine:

```
omp_display_env(verbose) //C|C++
call omp_display_env(verbose)   !! F90
```

Program Information

```
OPENMP DISPLAY ENVIRONMENT BEGIN
_OPENMP='202011'
[host] OMP_NUM_THREADS: value is not defined
[host] OMP_MAX_ACTIVE_LEVELS='1'
[host] OMP_SCHEDULE='static'
[host] OMP_STACKSIZE='4M'
[host] OMP_DISPLAY_AFFINITY='FALSE'
[host] OMP_AFFINITY_FORMAT='(null)'
[host] OMP_PLACES: value is not defined
[host] OMP_PROC_BIND: value is not defined
[host] OMP_TARGET_OFFLOAD=DEFAULT
[host] OMP_DEFAULT_DEVICE='0'
[host] OMP_ALLOCATOR='omp_default_mem_alloc'
[host] OMP_MAX_TASK_PRIORITY='0'
OPENMP DISPLAY ENVIRONMENT END
```

_OPENMP: YYYYMM

Year & Month of Spec.
201511 – 4.5
201811 – 5.0
202011 – 5.1
202111 – 5.2

Some vendors are Technical Report compliant
202008 – TR9
202107 – TR10
202211 – TR11

Not all ENV VARs shown.
Order has been changed.

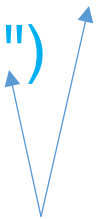
Program Information Displays

- Display Affinity Info

  ENV. VAR: export OMP_DISPLAY_AFFINITY=TRUE | FALSE

  API routine: omp_display_affinity(NULL) //C|C++

  call omp_display_affinity("") !! F90



Can put format string here.

Wall Clock Timer

- Every developer should have an easy why to evaluate performance for code optimization and changes
- Total time:

  wrapper: `/usr/bin/time -p` ./a.out

  wrapped: `date +%s`
  ./a.out
  `date +%s` `#in secs.`

```
$ /usr/bin/time -p a.out
real 5.00
user 4.50
sys  0.01
```

```
$ date; ./a.out; date
1693573722
1693573727
```

Bash function nanosecond timer:

```
nstimer(){ t0=`date +%s.%N`; $*; t1=`date +%s.%N`; bc <<<"$t1 - $t0" ; }
```

```
$ nstimer echo Hello World
Hello World
.002101725
```

thread number

- API routine: omp_get_thread_num()

- In normal programming used for task-parallel work assignment in a parallel region.

- Identifying threads is useful for (debugging):
  - ✓ Making sure multiple threads are executing tasks
  - ✓ Critical in resolving affinity problems
  - ✓ Resolving race conditions by slowing down one thread in a race.

primary thread

An OpenMP thread that has thread number 0. A primary thread may be an initial thread or the thread that encounters a parallel construct, creates a team, generates a set of implicit tasks, and then executes one of those tasks as thread number 0.

Example of Class Timer for C/C++/F90 codes

- See timer.hpp and timer.f90

```
#include "timer.hpp";
int main(){
  Timer timer;

  timer.start("CPU:foo1");
  foo1();
  timer.stop();

  timer.start("CPU:foo2");
  foo2();
  timer.stop();

  timer.print();
}
```

```
#include “timer.f90”
program main
  use  mod_timer
  type(cls_timer) :: timer
  call timer%start("CPU:foo1")
  call foo1()
  call timer%stop

  call timer%start("CPU:foo2")
  call foo2()
  call timer%stop

  call timer%print
end
```

Output from Class Timer for C/C++/F90 codes

- OUTPUT:

```
Action    ::    time/s     Time resolution = 1e-06
------
CPU:foo1 ::    1.9e-05
CPU:foo2 ::    1.0e-06
```

Useful Unix Utils

- |& tee *file* for displaying stdout/err and saving to *file*

- *Contributions welcome:*_____

Remember how to get to these references

- OpenMP References

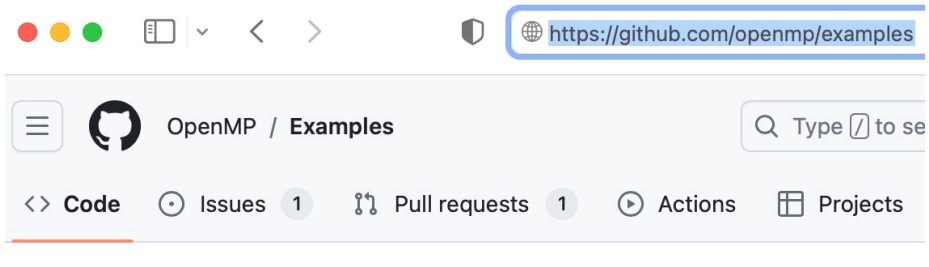
  https://www.openmp.org Look under Specifications

  OpenMP API 5.2 Specification

  OpenMP API 5.2.1 Examples

  OpenMP API 5.2 Reference Guide

- https://github.com/openmp/examples get example codes here



LABS adv_intro

For Exercises, git clone

```
https://github.com/csc-training/advancedOpenMP.git
```

```
$ cd OpenMP_cpu
$ cd adv_intro

$ ls
1_env_apis                  README_dev_access.md
2_timers                    README.md
3_private_if_clauses        job_lumi_C
```