OpenMP

Dr. Nitya Hariharan (Intel)

Preliminaries: Part 1

- Disclosure
  - The material in the lecture slides are adapted from lectures/tutorials by different sources
    - Tim Mattson (Intel)
    - Michael Klemm (Intel)
    - Edward Smyth (NAG)
    - Blaise Barney, LLNL
    - Hernandez et al., OpenMP 5.0/5.1 tutorial at ECP 2020 annual meeting
    - OpenMP 5.0.1 specification and examples https://www.openmp.org/resources/
    - Examples from “Using OpenMP” by Chapman et al.

Preliminaries: Part 2

- The lectures are split into different parts going from basics to advanced topics.
- Each section will have some exercises for you to try out.
- As far as possible, attempt the exercises on your own and try out different solutions. And please ask queries during the discussion hour.
- Grey boxes indicate some questions you can try and answer, solutions are given for some of them.
- You will find links to relevant talks or discussion boards in the notes, do have a look

Outline

- Introduction to OpenMP
- Thread creation
- PARALLEL and work-sharing constructs
- Data scoping

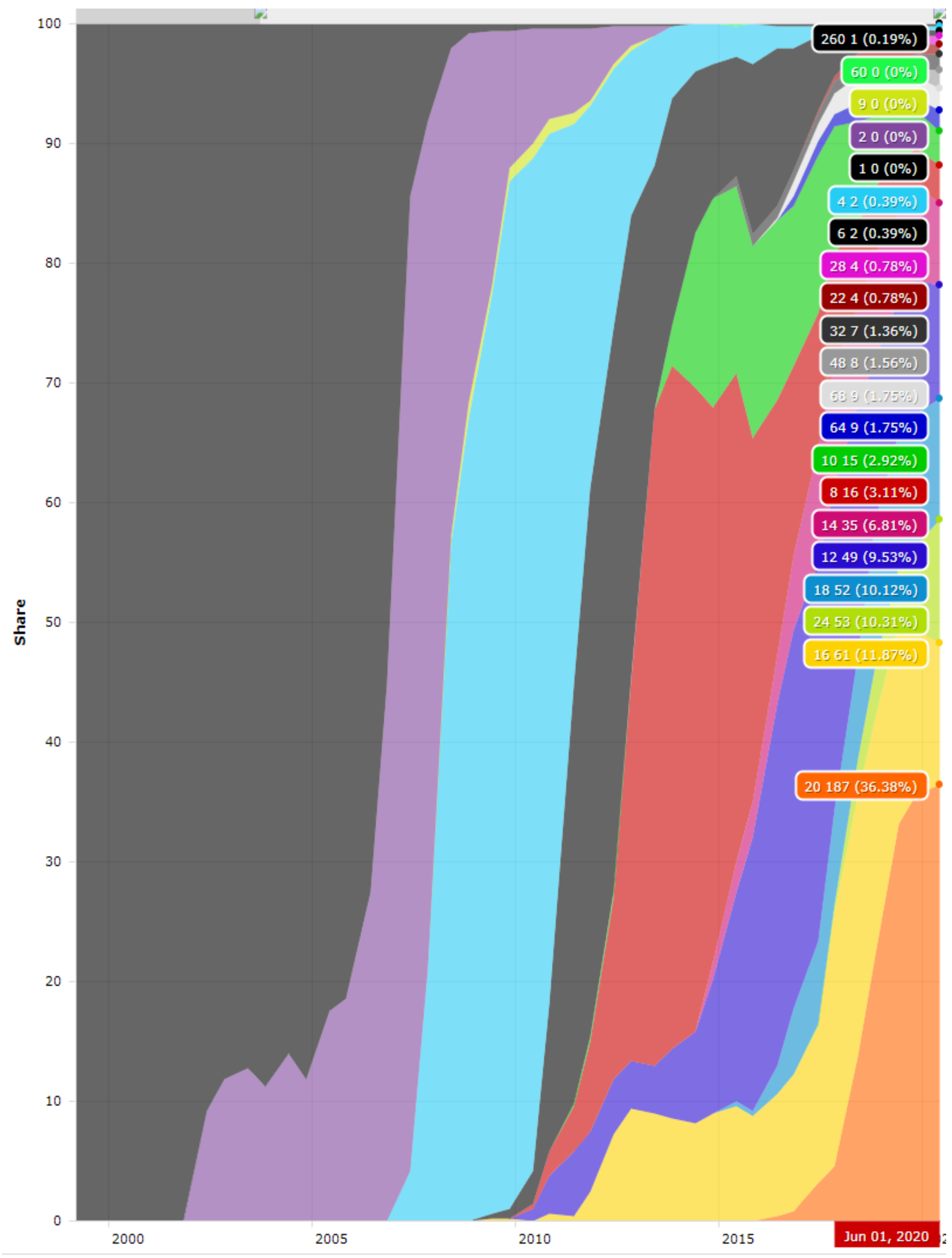
What is OpenMP*

OpenMP - Open Multi-Processing

- An API for developing multi-threaded (MT) applications
- Consists of a set of compiler directives and library routines for parallel application programmers
- Simplifies writing MT programs in Fortran, C and C++
- Augments vectorization and standardizes programming of various platforms
  - Embedded systems, accelerator devices (GPU), multi-core systems (CPU)
- Name and specification maintained by OpenMP Architecture Review Board

OpenMP ARB

"The OpenMP ARB's mission is to standardize directive-based multi-language high-level parallelism that is performant, productive and portable."



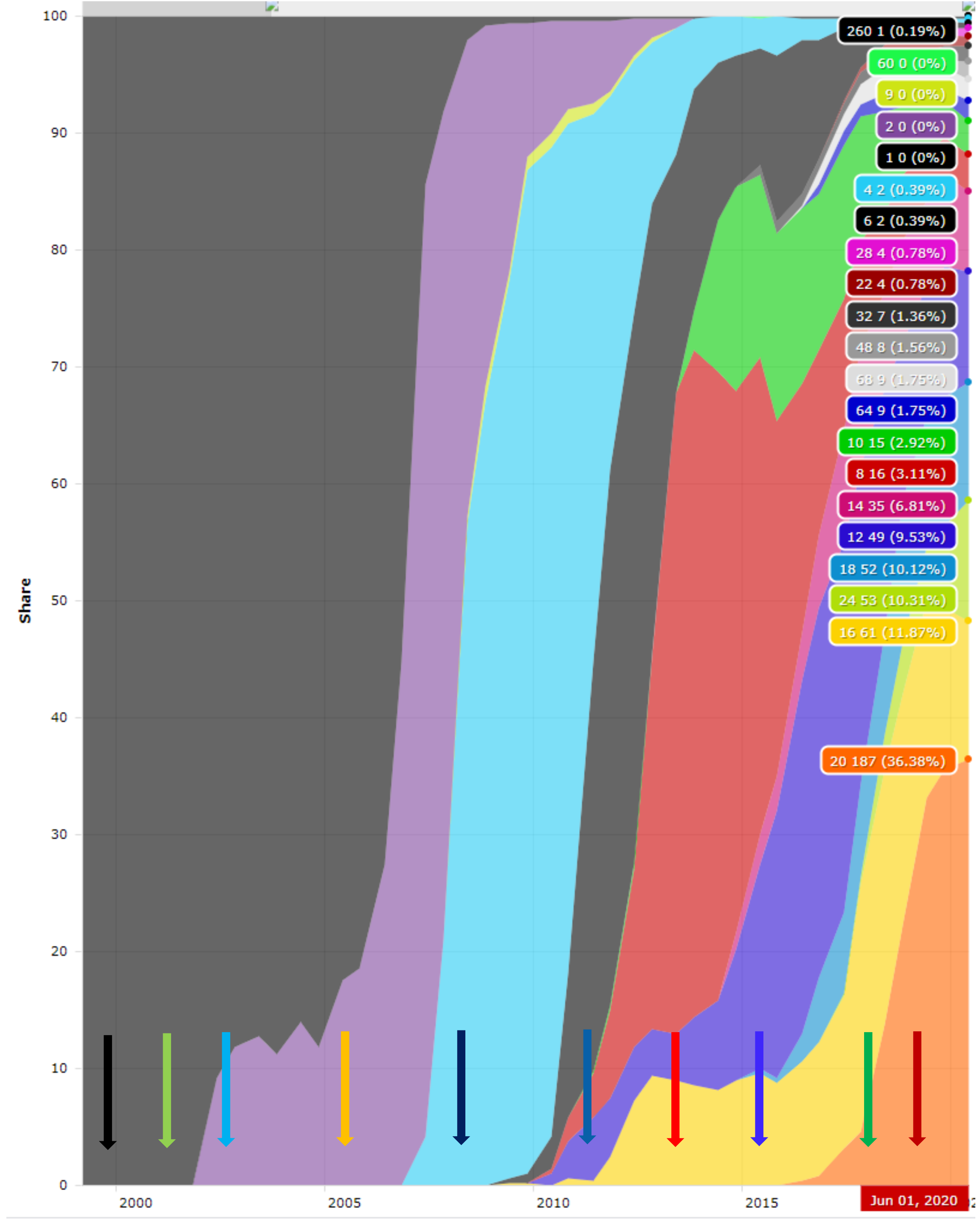
Cores per socket – systems share*

2007 onwards - multi-core processors dominated the landscape.

OpenMP benefited through increased greater parallelism.

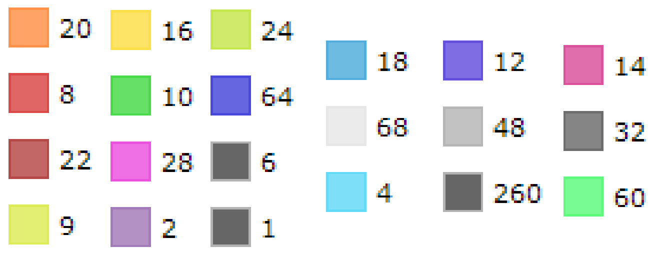
*https://www.top500.org/statistics/overtime/

Cores per socket – systems share*



OpenMP releases have followed the trend of increasing core counts

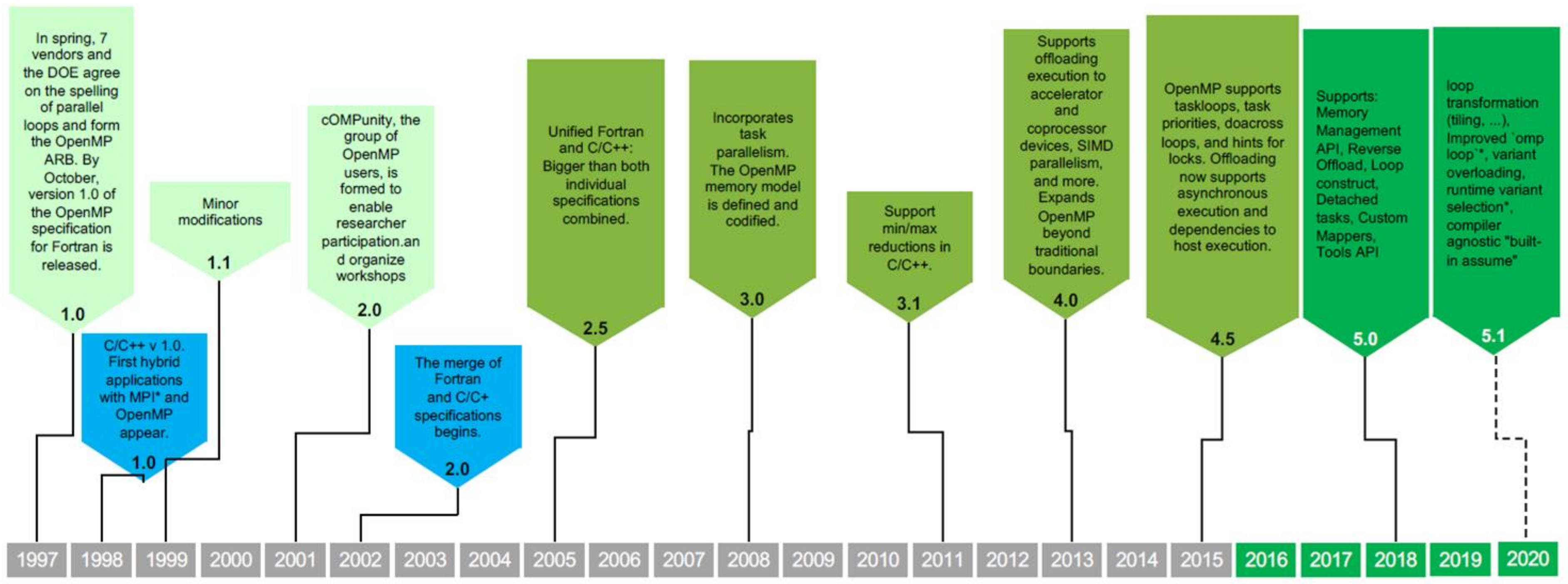
1999	- OpenMP 1.0
2000 - 01	- OpenMP 2.0
2002 - 03	- OpenMP 2.0
2005 - 06	- OpenMP 2.5
2008 - 09	- OpenMP 3.0
2011	- OpenMP 3.1
2013	- OpenMP 4.0
2015	- OpenMP 4.5
2018	- OpenMP 5.0
Late 2019	- OpenMP 5.1



*https://www.top500.org/statistics/overtime/

History of OpenMP

Increasing complexity



1.0: In spring, 7 vendors and the DOE agree on the spelling of parallel loops and form the OpenMP ARB. By October, version 1.0 of the OpenMP specification for Fortran is released.

1.0: C/C++ v 1.0. First hybrid applications with MPI* and OpenMP appear.

1.1: Minor modifications

2.0: cOMPunity, the group of OpenMP users, is formed to enable researcher participation.and organize workshops

2.0: The merge of Fortran and C/C+ specifications begins.

2.5: Unified Fortran and C/C++: Bigger than both individual specifications combined.

3.0: Incorporates task parallelism. The OpenMP memory model is defined and codified.

3.1: Support min/max reductions in C/C++.

4.0: Supports offloading execution to accelerator and coprocessor devices, SIMD parallelism, and more. Expands OpenMP beyond traditional boundaries.

4.5: OpenMP supports taskloops, task priorities, doacross loops, and hints for locks. Offloading now supports asynchronous execution and dependencies to host execution.

5.0: Supports: Memory Management API, Reverse Offload, Loop construct, Detached tasks, Custom Mappers, Tools API

5.1: loop transformation (tiling, ...), Improved `omp loop`*, variant overloading, runtime variant selection*, compiler agnostic "built-in assume"

Merged C/C++ and Fortran spec

Offload support

*image source – OpenMP 5.0 tutorial, Hernandez et. al., Exascale Computing Project

OpenMP solution stack

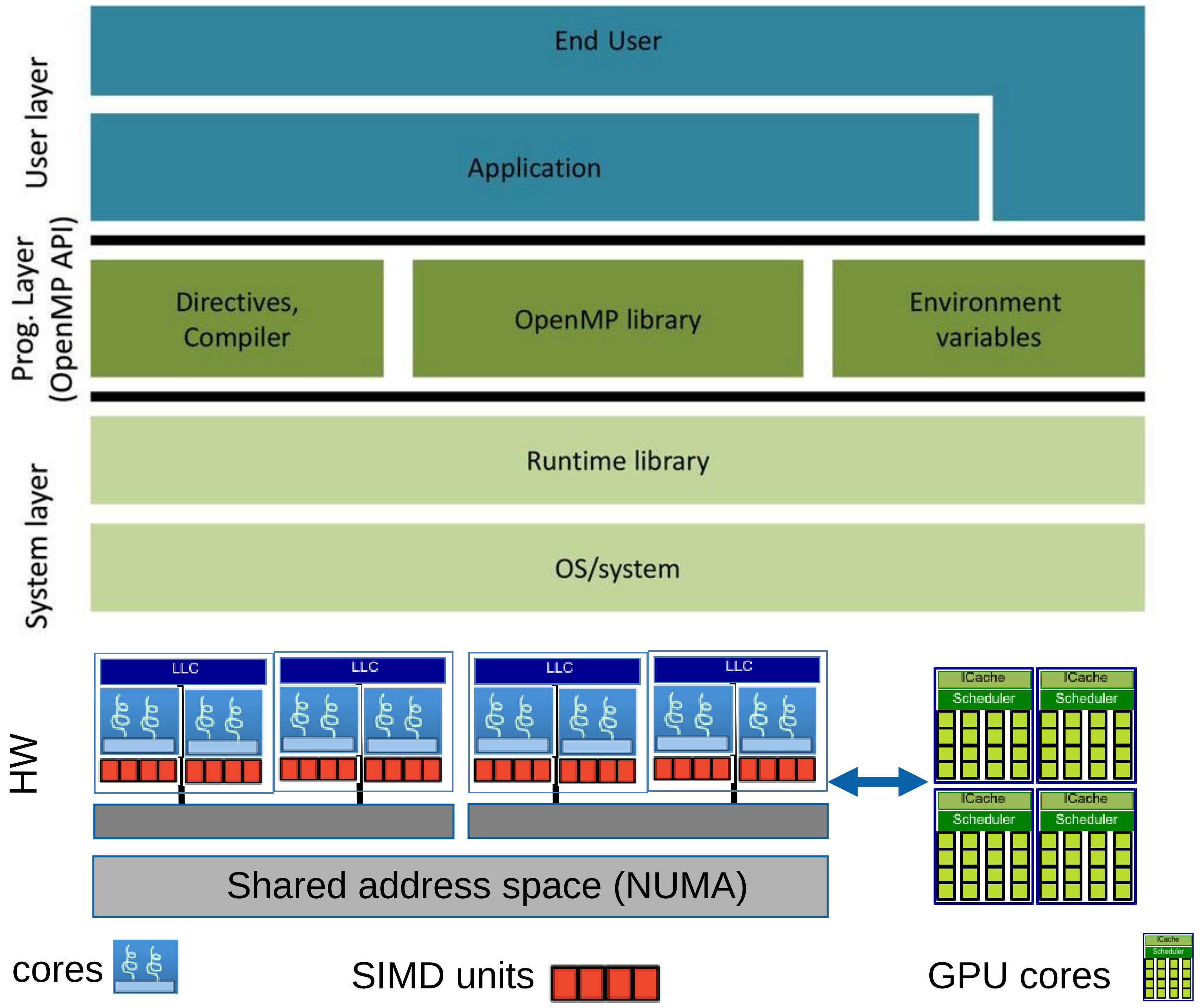


*image credit – Programming with OpenMP, Yonghong Yan

The OpenMP Common Core: Most OpenMP programs only use these 21 items

pragma, env variable, function, or clause	Concepts
#pragma omp parallel	Parallel region, teams of threads, structured block, interleaved execution across threads.
#pragma omp barrier/critical	Synchronization and race conditions, interleaved execution.
#pragma omp for/parallel for	Worksharing, parallel loops, loop carried dependencies.
#pragma omp single	Workshare with a single thread.
#pragma omp task/ taskwait	Tasks including the data environment for tasks.
setenv OMP_NUM_THREADS N	Setting the internal control variable (ICV) for the default number of threads with an environment variable
void omp_set_num_threads() int omp_get_thread_num() int omp_get_num_threads()	Default number of threads and ICV. SPMD pattern: Create threads in a parallel region and split up the work.
double omp_get_wtime()	Speedup and Amdahl's law, false sharing and other perf issues.
reduction(op:list)	Reductions of values across a team of threads.
schedule (static [,chunk]) schedule(dynamic [,chunk])	Loop schedules, loop overheads, and load balance.
shared(list), private(list), firstprivate(list)	Data environment.
default(none)	Force explicit definition of each variable’s storage attribute
nowait	Disabling implied barriers on workshare constructs, the high cost of barriers, and the flush concept (but not the flush directive).

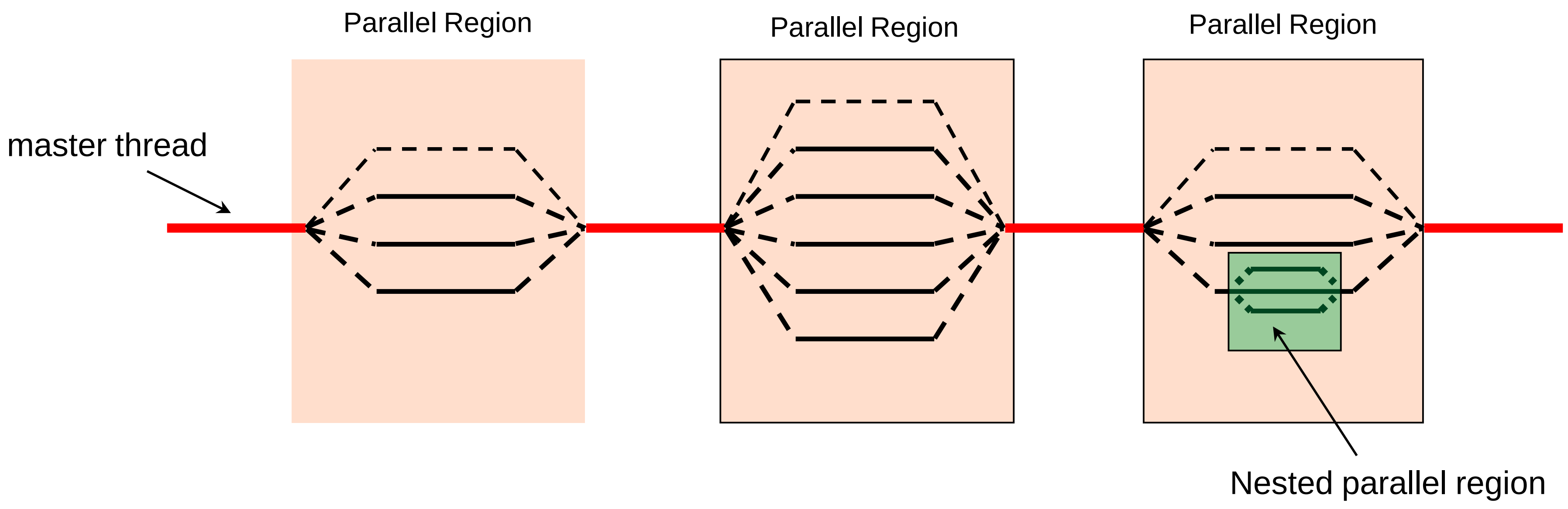
Outline

- Introduction to OpenMP
- Thread creation
- PARALLEL and work-sharing constructs
- Data scoping

OpenMP Programming Model

Fork-Join Parallelism:

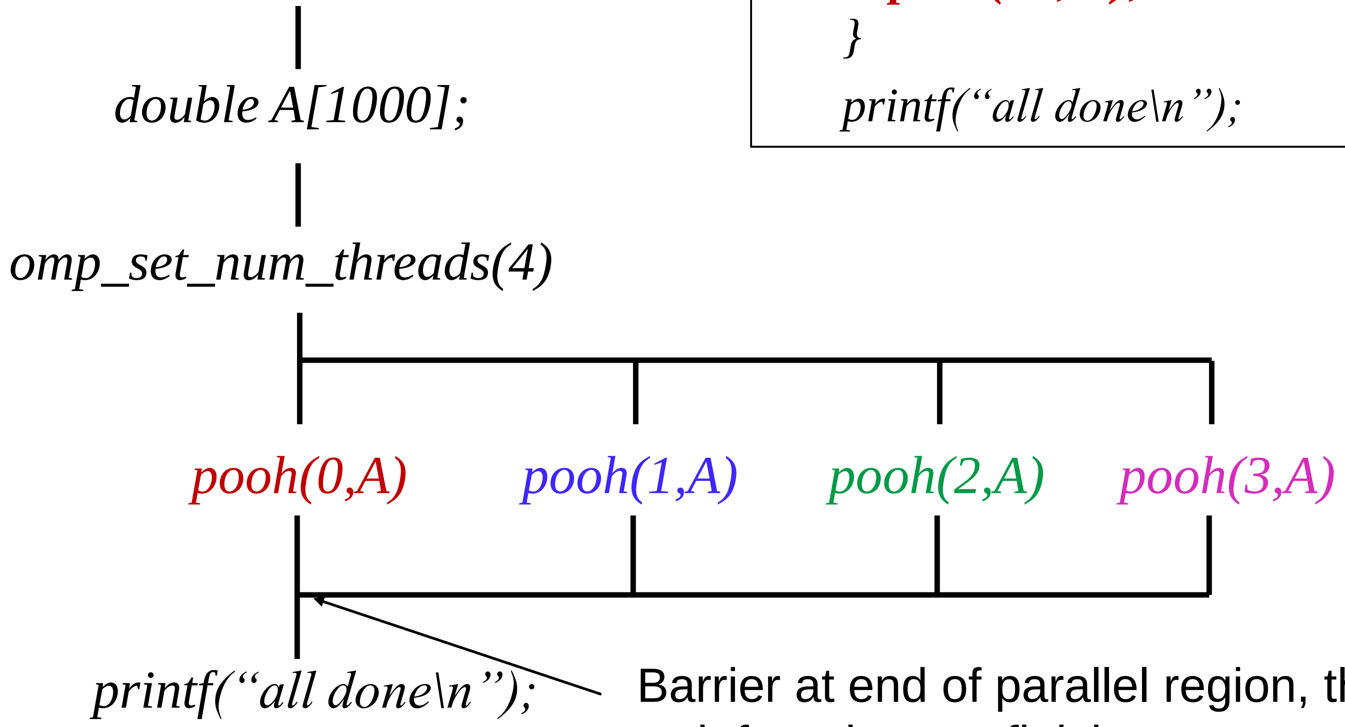
- Master thread spawns a team of threads as needed.
- Parallelism added incrementally until performance goals are met.
- Threads within a parallel region can spawn more threads – nested parallelism



Thread Creation: Parallel Region

- Use **PARALLEL** construct to create threads
- Same code executed on each thread, on different data (SPMD)

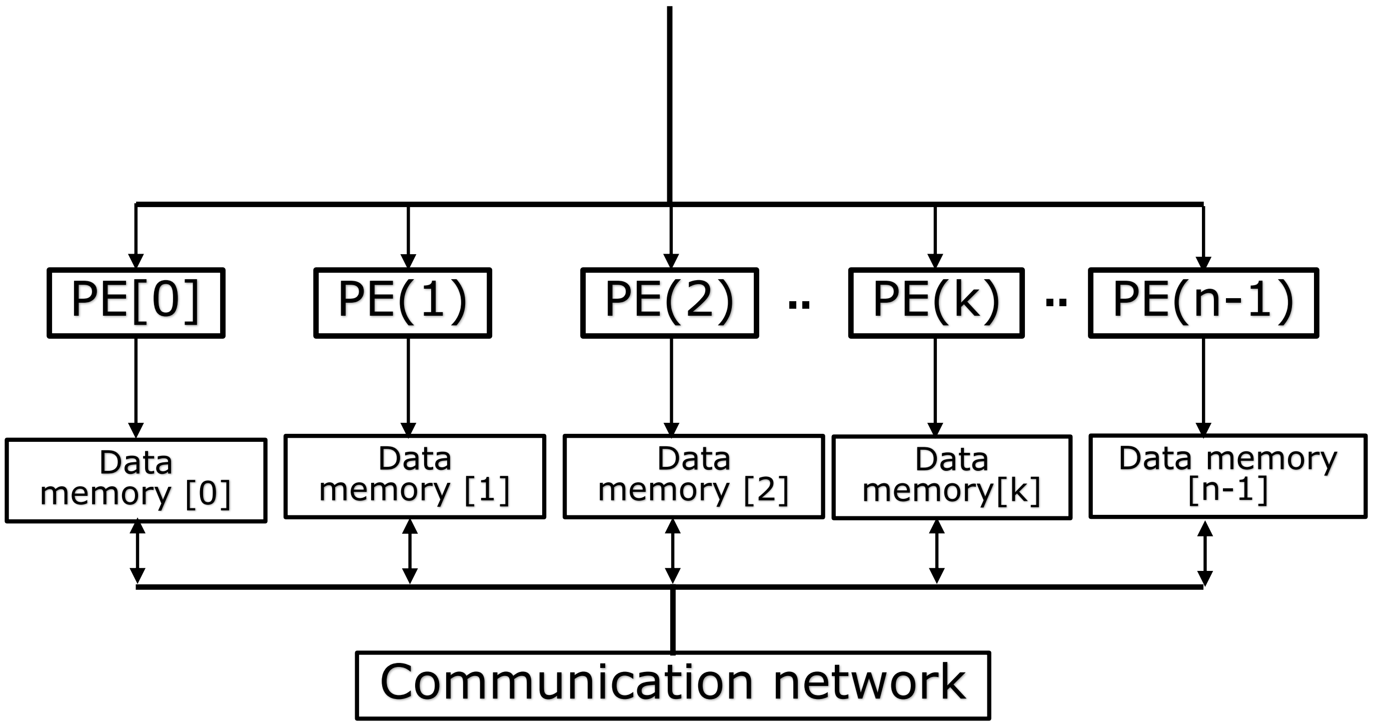
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
   int ID = omp_get_thread_num();
   pooh(ID, A);
}
printf("all done\n");
```



Barrier at end of parallel region, threads wait for others to finish.

What is SPMD?

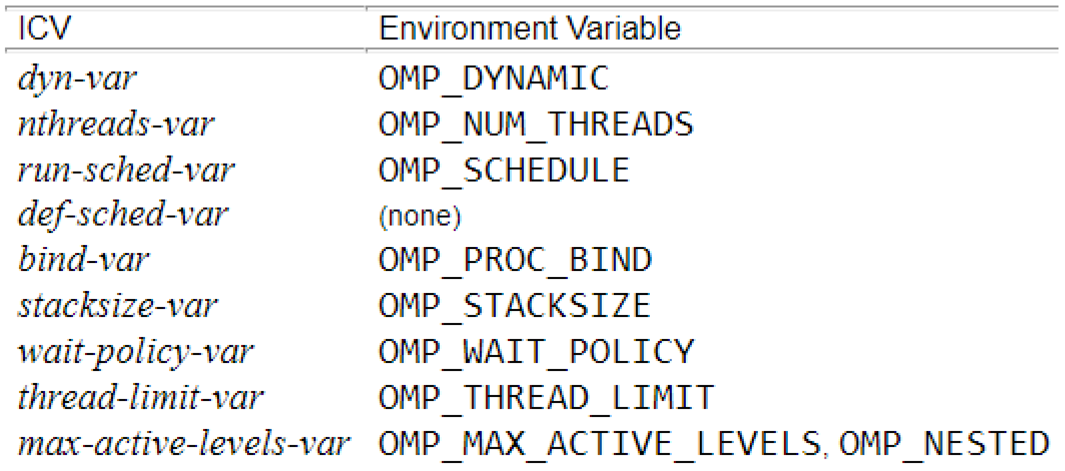
- SPMD – Single Program Multiple Data
- Part of the MIMD category in Flynn's taxonomy
- Multiple Processing Elements (PE) that run a copy of the same program and operate on different data elements



What happens behind the scenes

- Behavior of an OpenMP program can be controlled by Internal Control Variable (ICVs)

- ICVs are set at different stages of a program, initialized by implementation and values can be overridden through env variables or within the program.



ICV	Environment Variable
dyn-var	OMP_DYNAMIC
nthreads-var	OMP_NUM_THREADS
run-sched-var	OMP_SCHEDULE
def-sched-var	(none)
bind-var	OMP_PROC_BIND
stacksize-var	OMP_STACKSIZE
wait-policy-var	OMP_WAIT_POLICY
thread-limit-var	OMP_THREAD_LIMIT
max-active-levels-var	OMP_MAX_ACTIVE_LEVELS, OMP_NESTED

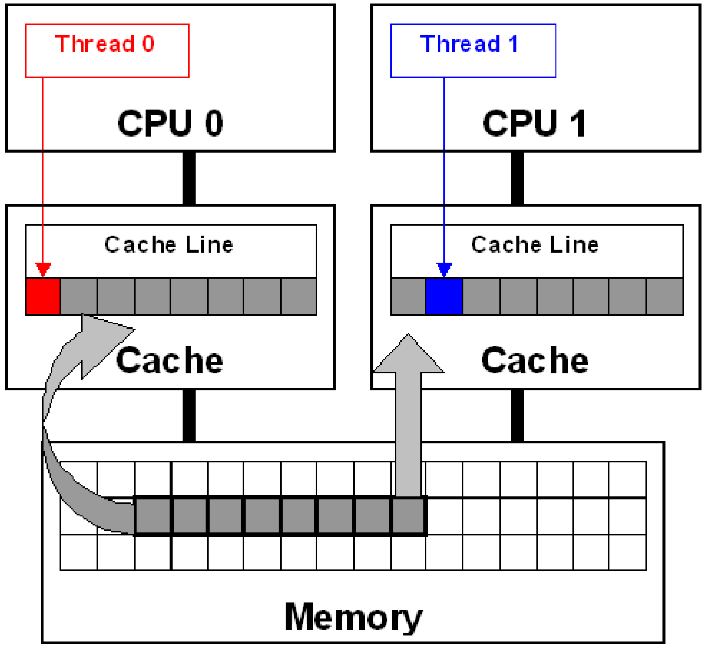
- Example - when the program starts up, query *OMP_NUM_THREADS* and update *nthreads-var* ICV

- Can override this using *omp_set_num_threads()* function call

- The *num_threads* clause can be used to request threads for a parallel region, ICV remains unchanged
  - *#pragma omp parallel num_threads(8)*

What happens when you run on multiple cores

- There are side effects to sharing resources among threads
- If independent data elements happen to sit on the same cache line, each modification will cause the cache line to be invalidated and forces a cache update to maintain coherency, hurting performance - **"false sharing"**



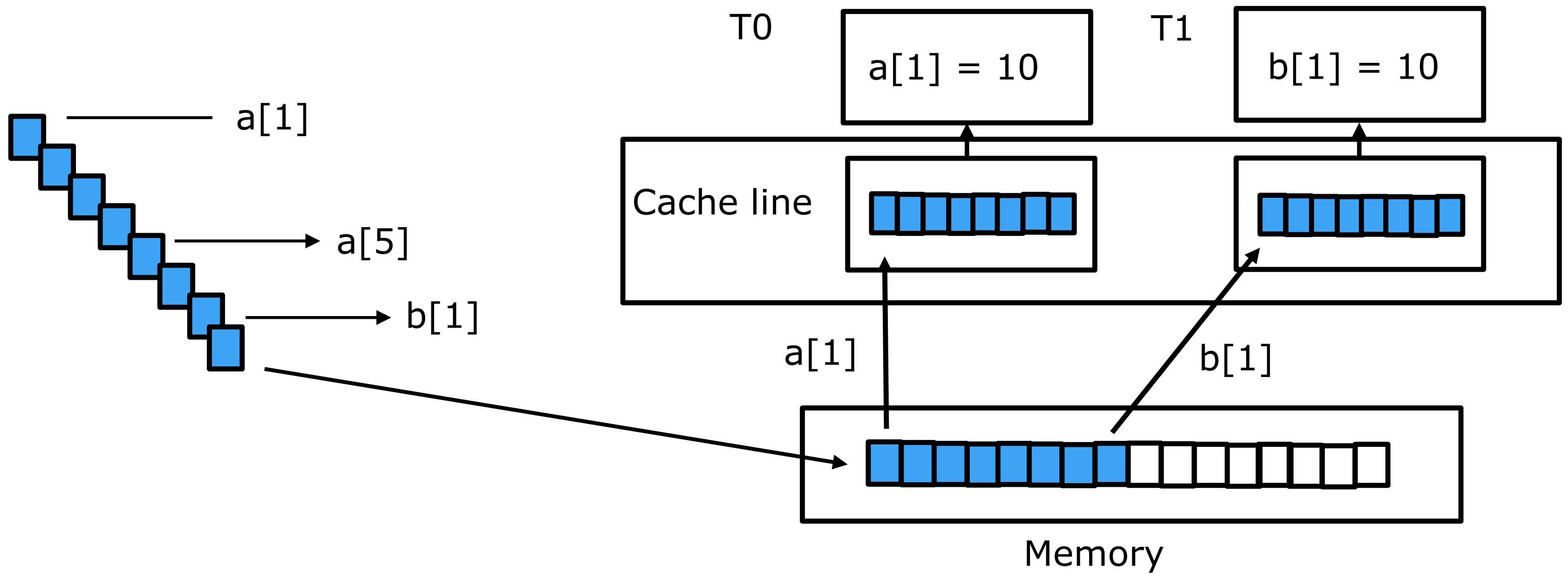
Q

- How do you then get performance?

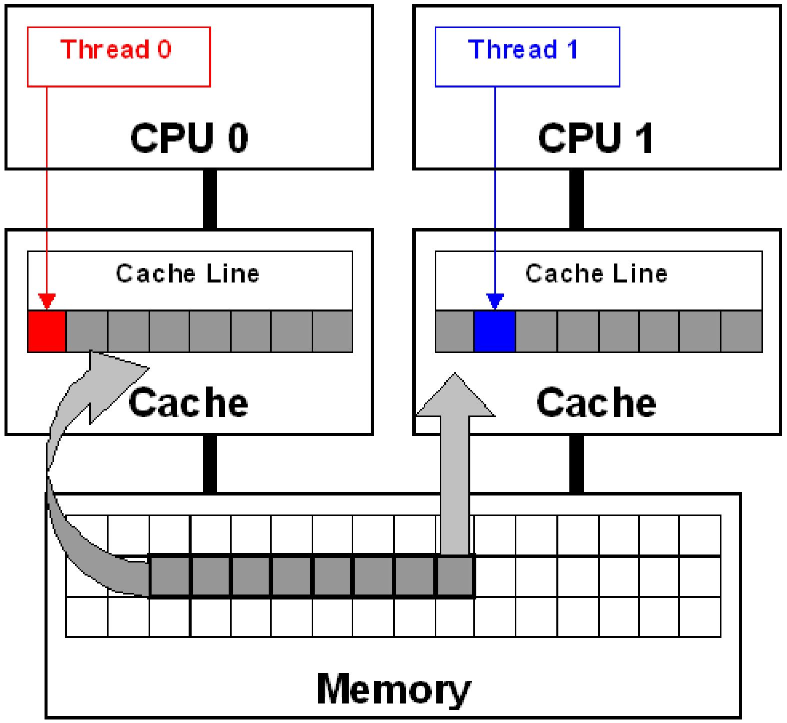
*https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

False sharing

- Caches get a chunk of data every time you request something from memory – cache line (64 bytes)
- If you ask for 8-byte element *a[1]*, the core will read 7 other elements along with it, into the cache.
- This helps with spatial locality, i.e., if you want variable *a[1]*, you **might** want the variables next to it in memory.
- When T0 updates *a[1]*, it forces T1 to load the cache line again.



How do you get performance with “false sharing”?



Solution

1. Pad arrays so elements used by separate threads are on distinct cache lines
2. Be careful while padding, and pad only how much you need. Assume L1 cache line is 64 bytes.
3. Compilers are now smart enough to recognize false sharing and can use thread-private temporary variables. Use optimization flags!

*https://software.intel.com/content/www/us/en/develop/articles/avoiding-and-identifying-false-sharing-among-threads.html

Outline

- Introduction to OpenMP
- Thread creation
- PARALLEL and work-sharing constructs
- Data scoping

Some terminology

- OpenMP has three major components
  - Compiler directives
  - Runtime routines - *omp_set_num_threads()*
  - Environment variables - *OMP_NUM_THREADS*

- User can decide what to use in their program

- Compiler directives have the format

  #pragma omp or !$omp   *DIRECTIVE-NAME*   *CLAUSE*

  where

  DIRECTIVE-NAME can be *PARALLEL, SIMD* – define program behavior

  CLAUSE can be *DEFAULT*, *PRIVATE*, *REDUCTION* – how data is shared among threads

- A construct is an OpenMP directive + directive-name + clauses + a structured block of code that does something – next slide

The OpenMP PARALLEL region construct

- Fundamental OpenMP parallel construct
- Thread that encounters the PARALLEL construct creates a team of threads and becomes the master
- Copy of the code executed by each thread – SPMD

```
#pragma omp parallel
{
#pragma omp for
        for (i=0;i<N;i++){
                do_something(i);
        }
}
```

loop control index *i* is "private" to each thread by default. More details in data scoping section.

Implicit barrier, threads wait here until all threads are finished.

Rules for the PARALLEL construct

- The number of threads in the PARALLEL region depends on some factors, we discussed this in an earlier slide
- Illegal to branch into or out of a parallel region – undefined behavior
- Only one IF clause permitted in the construct

```
$OMP  PARALLEL IF(EXECUTE==true)
10    sum(id) = next(id)
30    result(id) = sum(id) * 9.8
      if(converged(result(id)) goto 20
      go to 10
$OMP END PARALLEL
     if(not_FINISHED) goto 30
20    print *, id
```

- Not a good way to program, but then goto statements are not very acceptable anyway.
- Can use STOP or exit()

Work-sharing constructs

- Divide the work among threads, no new threads launched
- Must be within a PARALLEL construct
- Conditional statements for some threads to enter the construct not allowed

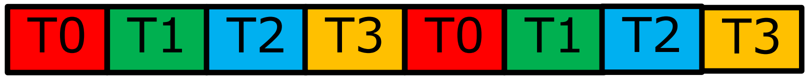
```
#pragma omp parallel
#pragma omp for
for(i=0;i<N;i++)
{
a[i] = a[i] + b[i];
}
```

No barrier upon entry into construct

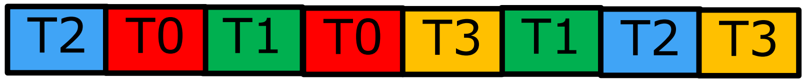
Implicit barrier upon exit from construct

Work-sharing constructs: SCHEDULE clause

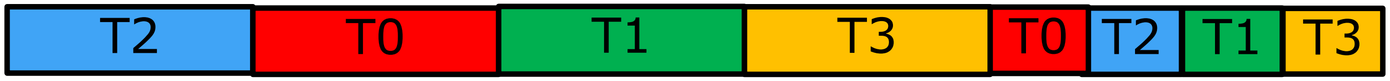
- Affects how loop iterations are mapped onto threads
  - SCHEDULE(STATIC,[chunk])
    - Deal-out blocks of iterations of size "chunk" to each thread. Done at compile time



  - SCHEDULE(DYNAMIC[,chunk])
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled. Scheduling logic used at run-time



  - SCHEDULE(GUIDED[,chunk])
    - Iterations dynamically assigned, with reducing chunk size, till all iterations have been handled.



  - More clauses like RUNTIME and AUTO

Work-sharing constructs: SCHEDULE clause

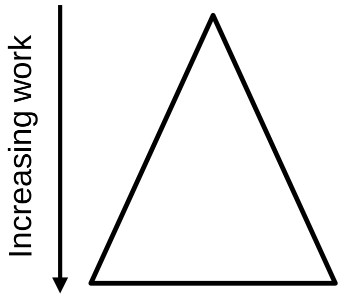
- New modifiers added *monotonic*, *nonmonotonic* (v4.5)

  - *monotonic*
    - Thread executes the assigned chunks in logical iteration order, *i=1, 4, 7* etc.

  - *nonmotonic*
    - Chunks executed in any order
    - Application result should not depend on iteration order – else unspecified behavior
    - Think of it as an iteration stealing scheduling scheme
    - A thread can run a “previous” iteration after the current one, *i=1,4,7,5*

Q

- Can you think of a scenario where this will help?
- How about loops that have a “triangular” shape of workload, is it always easy to determine how much work each thread will get?



Combined work-sharing constructs

Shortcut for specifying one construct immediately nested inside another construct

```
 double  res[MAX];  int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
         res[i] = huge();
    }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
        res[i] = huge();
   }
```

These are equivalent

```
 double ::  res(MAX)
 integer :: i
!$omp parallel do
do i=0, MAX
     res(i) = huge()
 end do
```

If an *end parallel do* directive is not given, it is assumed to be at the end of the *do-loops*.

SINGLE and MASTER construct

- SINGLE/MASTER – restrict execution to one thread
  - SINGLE – any one thread can execute the structured block
  - MASTER – master thread executes the structured block

- Implicit barrier for SINGLE, none for MASTER

- SINGLE construct useful for I/O, or check simulation time remaining and other book-keeping

```
#pragma omp parallel for(i=0; i<n;i++)
{
..
#pragma omp single/master
 {
   //calculate time in simulation
}
}
```

Q

- Can you think of a scenario where the MASTER construct can be used? How about in MPI + OpenMP applications?

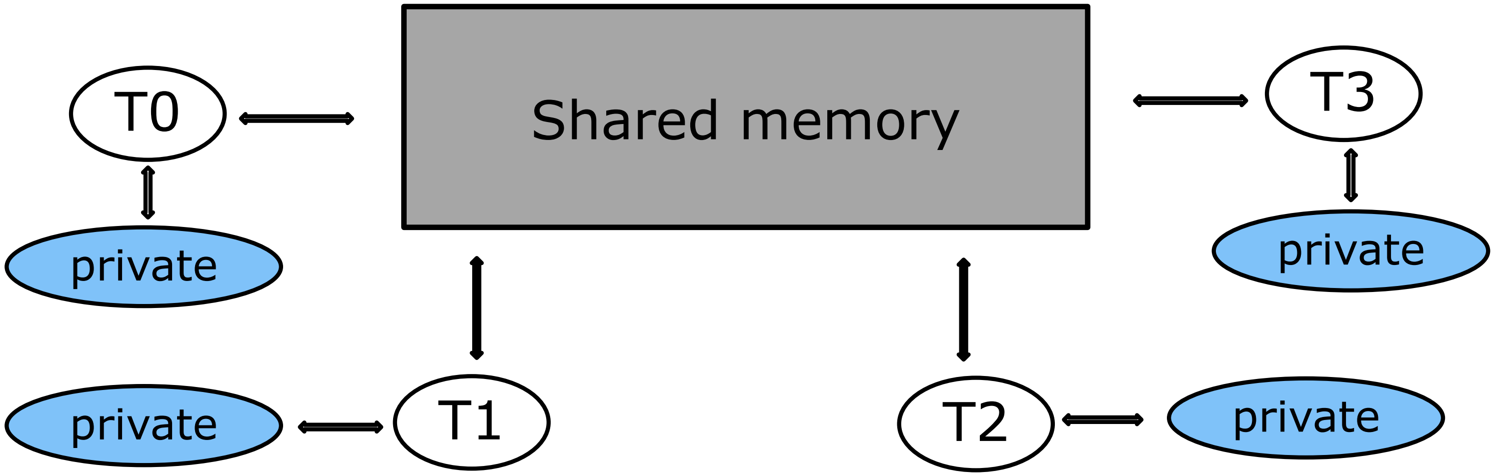
Outline

- Introduction to OpenMP
- Thread creation
- PARALLEL and work-sharing constructs
- Data scoping

Data scoping

- Important to know data scoping
  - how is data shared
  - how are updates to data made visible to other threads

- Shared memory programming model (SMP)
  - Most variables are shared by default
  - A thread can also have its private data



Data scoping

- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)

- Data **private** to a thread includes
  - Loop index variables
  - Stack variables in routines called from parallel regions
  - Automatic variables within a statement block (Fortran)

```
 double ::  res(MAX)
 integer :: i, index
!$omp parallel do
do i=0, MAX
      index = i + 1
     res[i] = huge(index)
 end do
```

```
function huge(index)
  integer :: index
  integer, automatic :: temp
….
end function huge
```

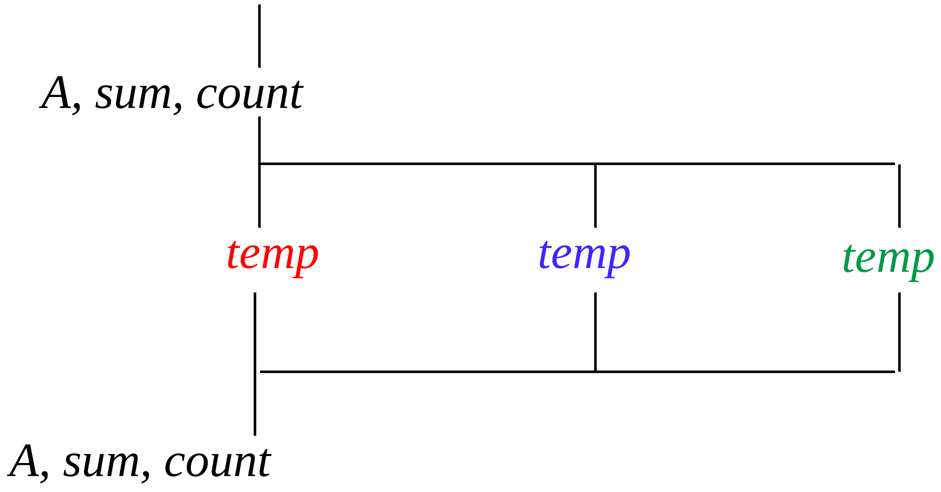
Data scoping: Examples

```
  double A[10];
  int main() {
  int sum[10];
 #pragma omp parallel
        calculate(sum);
  printf("%d\n", sum[0]);
}
```

```
extern double A[10];
void calculate(int *sum) {
  double temp[10];
  static int count;
  ...
}
```

A, *sum* and *count* are shared by all threads.

temp is local to each thread



Q

- Why is *count* shared?

OpenMP data scoping clauses

- OpenMP has additional clauses to explicitly define data scope
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - REDUCTION
  - DEFAULT - force programmer to declare attributes for all data
- Used in conjunction with PARALLEL or DO/FOR directives

Data Scoping: SHARED Clause

- SHARED (list)
  - Variables in the list are shared among all threads
  - One copy which is accessed by all
  - User needs to ensure proper access to SHARED data
  - **The code below can show incorrect results – why?**

```
int total = 0;
void sum() {
#pragma omp parallel for SHARED(total)
    for (int j = 0; j < 1000; ++j)
            total += j;
    printf(“%d\n”, total);
}
```

Data scoping: PRIVATE Clause

- PRIVATE(list)
  - **Local copy** of variables maintained by each thread
  - Scope of PRIVATE variable only within the parallel region
  - Uninitialized for each thread
  - Variable value before parallel construct is the **original variable**

```
void sum() {
      int tmp = 0;
#pragma omp parallel for private(tmp)
      for (int j = 0; j < 1000; ++j)
                  tmp += j;
      printf("%d\n", tmp);
}
```

Q

- Is the code correct?
- What is output of the printf statement? Hint – look at the variable scope.

Data scoping

- The original variable's value is unspecified if it is referenced outside of the PARALLEL construct

temp has unspecified value

```
int temp;
void result() {
    temp = 10;
#pragma omp parallel private(temp)
    work();
   printf("%d\n", temp);
}
```

- Threads also do not see this value within the construct

- How to ensure consistency inside and outside the construct?

```
extern int temp;
void work() {
    temp = 5;
}
```

which copy of *temp*?

FIRSTPRIVATE and LASTPRIVATE clause

- FIRSTPRIVATE - private variables initialized from the global counterpart outside the PARALLEL construct

```
weight = 20;
#pragma omp parallel for firstprivate(weight)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) weight++;
        A[i] = weight;
}
```

Each thread gets its own copy of ***weight*** with an initial value of 20

- LASTPRIVATE – value of private variable preserved in the environment outside the PARALLEL construct

```
weight = 20;
#pragma omp parallel for lastprivate(weight)
for (i = 0; i <= MAX; i++) {
        //set weight value
}
finalWeight = weight;
```

Last iteration value of ***weight*** is visible outside

Data scoping - REDUCTION clause

What if you want all threads to gather their results?

```
double  sum=0.0, avg=0.0, [MAX]; int i;
 for (i=0;i< MAX; i++) {
       sum + = A[i];
 }
 avg = sum/MAX;
```

- We need the value of *sum* from all threads - use the REDUCTION clause
- Private copy of reduction variable for all threads
- At end of parallel construct, reduction operation applied to all private copies of shared variable and reflected to global shared variable

REDUCTION clause

```
#pragma omp parallel for reduction(+:sum)
  for (i=0;i< MAX; i++) {
       sum + = A[i];  }
  avg = sum/MAX;  //sum has sum from all threads
```

Operation	Fortran or C/C++	Initialization
Addition	+	0
Multiplication	*	1
Subtraction	-	0

- These operations might not be associative for real numbers
- Cannot declare reduction variable as shared or private
- See OpenMP standard for additional reduction operators

Data scoping : DEFAULT clause

DEFAULT(shared | none)

- Applies the attribute to all variables inside the parallel construct
- Compiler will complain if a variable attribute is not specified

```
#include <omp.h>
int main()
{
    int i, j=5;     double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
       for(j=0; j<3; j++)
            x+= evolve(i, j, y);
    }
    printf(“ x is %f\n”,(float)x);
}
```

No attributes specified for *j* and *y*, compiler will complain

Quick test

- Consider this example of PRIVATE, FIRSTPRIVATE and DEFAULT

```
 int A = 1, B = 1, C = 1;
#pragma omp parallel default(none) private(B) firstprivate(C)
```

- Is the code snippet correct?
- What is the data sharing attributes for B and C?
- What are the initial values of B and C inside and values after the parallel region?

Recap

- History of OpenMP and why it is so relevant today
- Shared Memory
  - Creating a parallel region and distributing work among threads
- Data scoping
  - How is data shared and made visible across threads
  - PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE