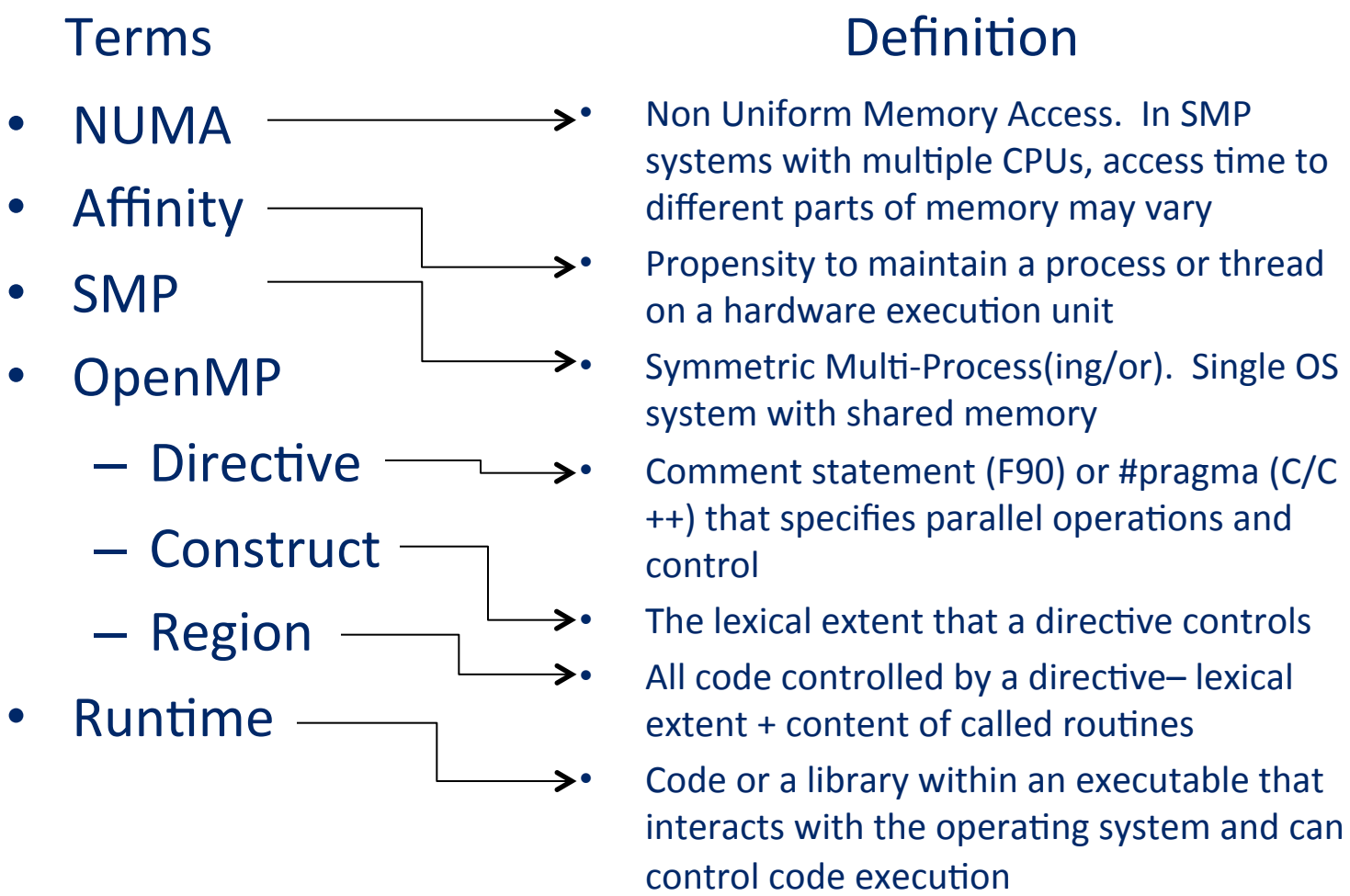
Version 2013-1
as of 31 May 2013

Parallel Computing for Science and Engineering

OpenMP

Scientific Computing Terminology



OpenMP-- Overview

- Standard developed in the late 1990s
- The “language” is easily grasp. You can start simple and expand.
- Lightweight from system perspective
- Very portable – GNU and vendor compilers
- Time spent 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.

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:
  - Compiler Directives
  - 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

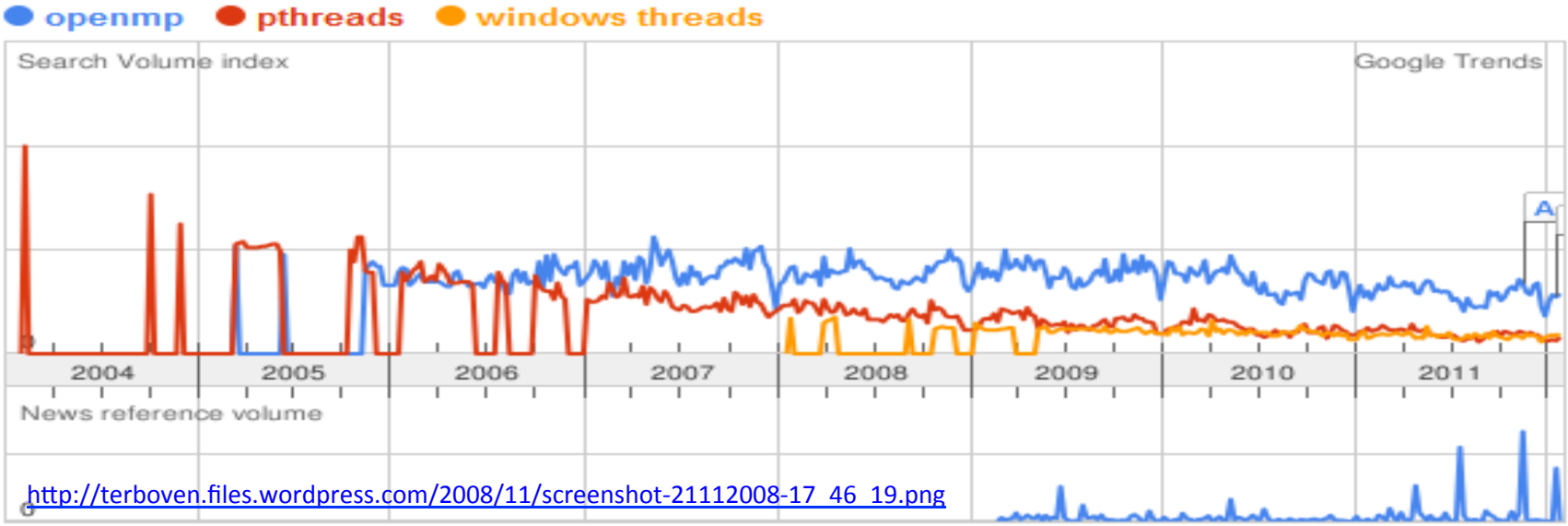
- Primary OpenMP participants

  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 — Published October 1997
- OpenMP C API, Version 1.0 — Published October 1998
- OpenMP 2.0 API for Fortran — Published 2000
- OpenMP 2.0 API for C/C++ — Published 2002
- OpenMP 2.5 API for C/C++ & F90 — Published 2005
- OpenMP 3.0 Tasks — Published May 2008
- OpenMP 3.1 — Published July 2011
- OpenMP 4.0 Affinity, Accelerator Support -- coming soon

OpenMP History



http://terboven.files.wordpress.com/2008/11/screenshot-21112008-17_46_19.png

- 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?

- Thread binding is coming in OpenMP 4.0
- (other techniques are already available)

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
    - 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

OpenMP Parallel Directive

- Supports parallelism by Directives in FORTRAN, C/C++,…
- Unlike others that require base language changes and constructs
- Unlike MPI which supports parallelism through communications library

Processes on an SMP System

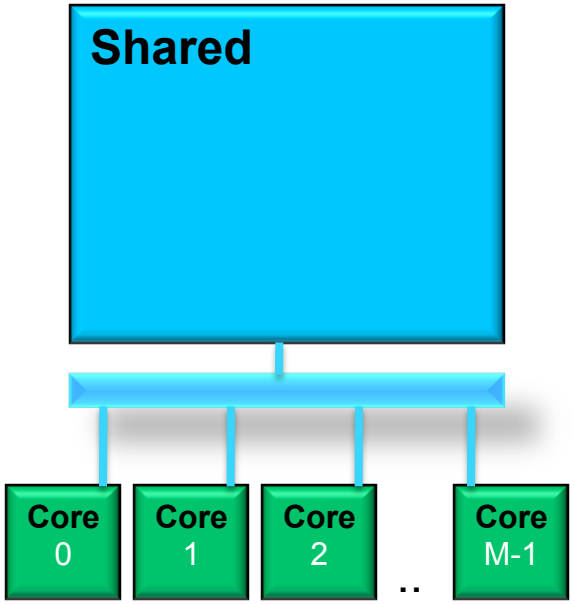
- The OS starts a process
  - One instance of your computer program, the “a.out”
- Many processes may be executed on a single core through “time sharing” (time slicing)
  - The OS allows each process to run for awhile
- The OS may run multiple processes concurrently on different cores
- Security considerations
  - 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

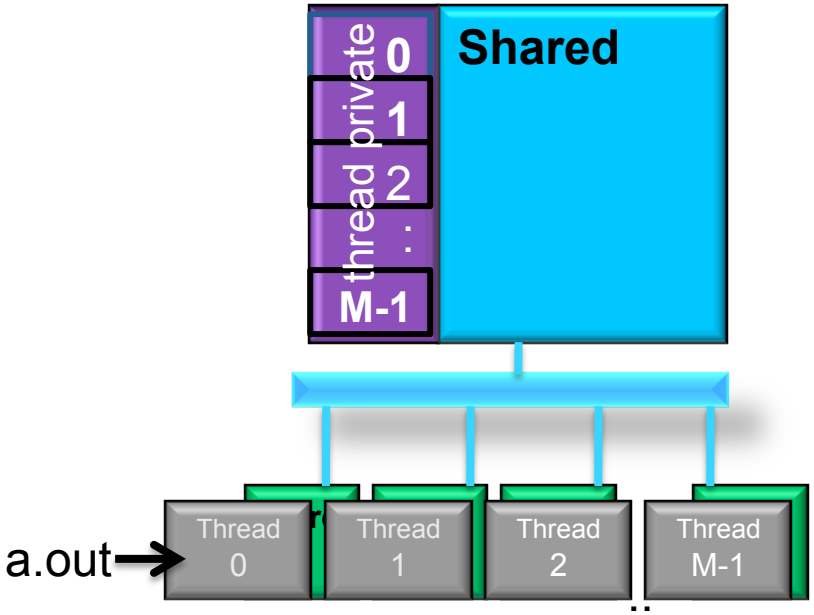
- Threads are instantiated (forked) in a program
- Threads run concurrently*
- All threads (forked from the same process) can read the memory allocated to the process
- Each thread is given some private memory only seen by the thread
- *When the # of threads forked exceeds the # of cores, time sharing (TS) will occur. Usually a bad idea. (But TS with user threads is less expensive than TS with processes)
- Implementation of threads differs from one OS to another

Programming with OpenMP on Shared Memory Systems

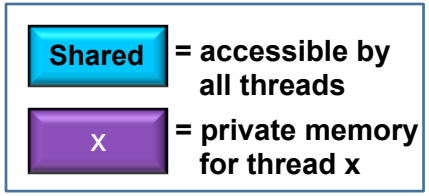
Hardware Model: Multiple Cores



Software Model: Threads in Parallel Region



M threads are usually mapped to M cores.



What is Parallel Computing

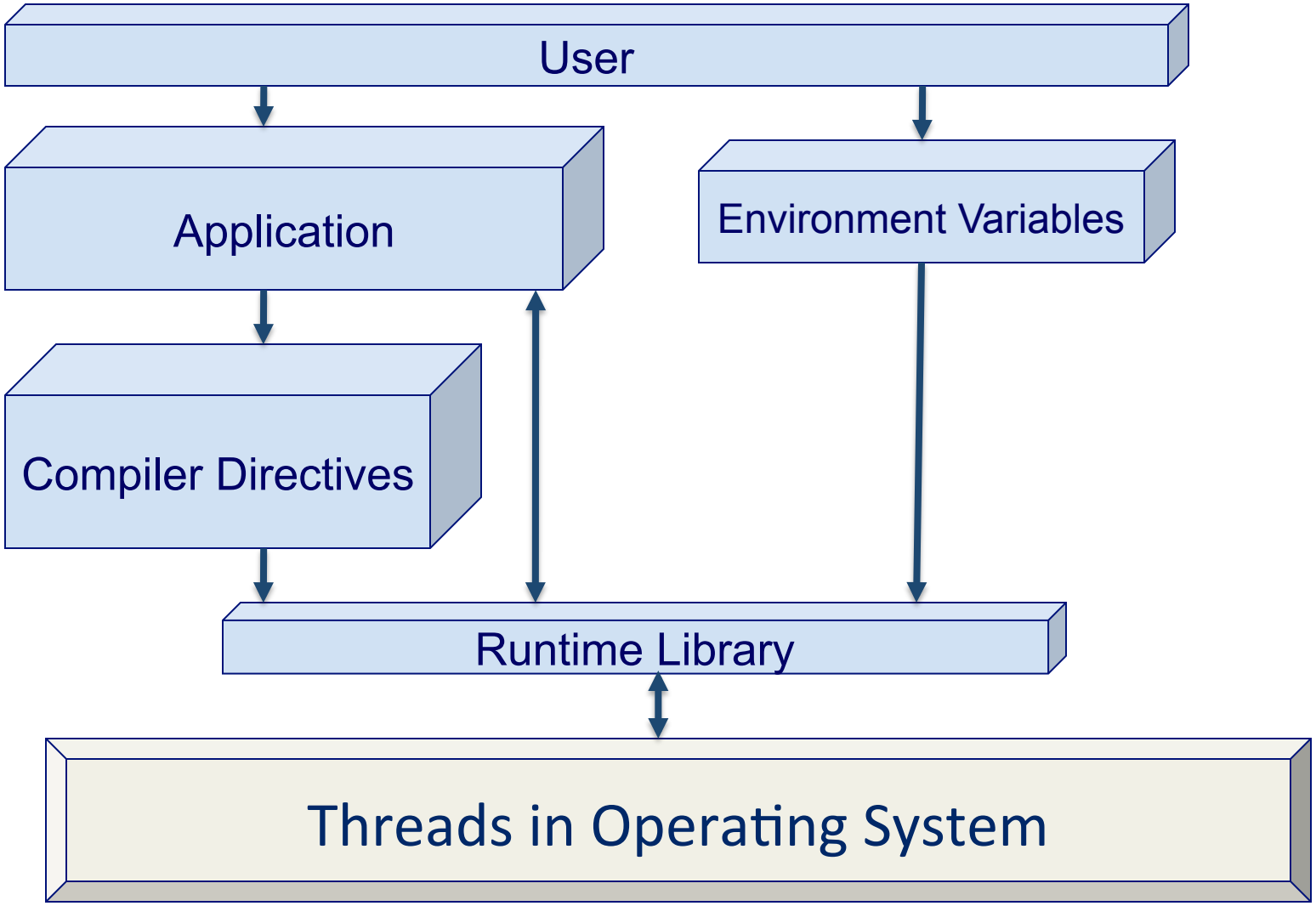
- Concurrent execution of computational work (tasks).
  - Tasks execute independently
  - Variable updates must be mutually exclusive
  - Synchronization through barriers

```
1   // We use loops for
    // repetitive tasks

2   for (i=0; i<N; i++){
3
4     a[i] = b[i] + c[i];
5   }
```

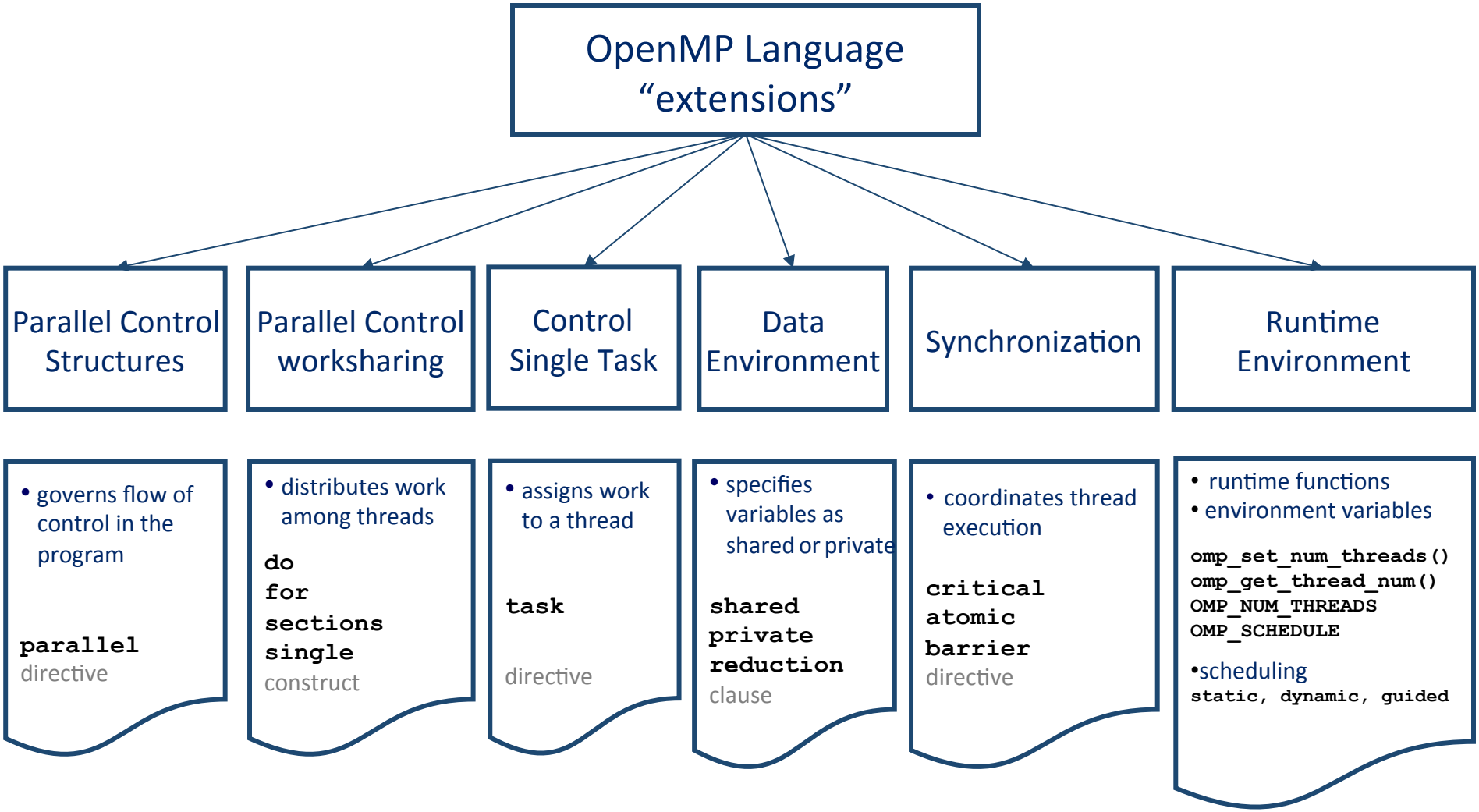
```
1   // We often update
    // variable(s)

2   for (i=0; i<N; i++){
3
4     prod = prod + b[i]*c[i];
5   }
```

Parallel Directives …

OpenMP Architecture



OpenMP Constructs



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

OpenMP Syntax

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

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

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

- Example

  C `#pragma omp parallel num_threads(4)`

  F90 `!$omp parallel num_threads(4)`

- Function prototypes and types are in the file:

  C `#include <omp.h>`

  F90 `use omp_lib`

- 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 a OpenMP-aware compiler can interpret. Directive sentinels are:

  F90 `!$OMP`

  C/C++ `# pragma omp`

Syntax: *sentinel* parallel *clauses* *uses defaults when clauses not present*

Fortran

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

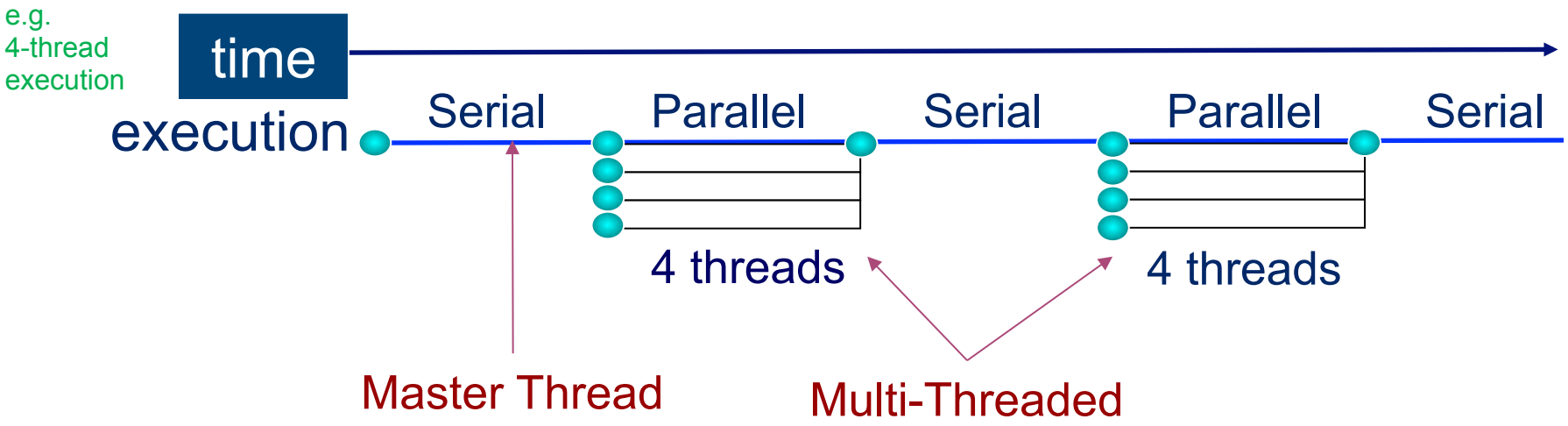
C/C++

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

- Fortran parallel regions are enclosed by enclosing directives.
- C/C++ parallel regions are enclosed by curly brackets.

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



Parallel Region

Fortran

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

C/C++

```
...
#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

Fortran

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

C/C++

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

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

Parallel Region & Thread Number

Fortran

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

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

C/C++

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

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

Parallel Region & Worksharing

Use OpenMP directives to specify Parallel Region, worksharing constructs, and Mutual Exclusion

```
parallel


end parallel
```

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

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

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

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

Parallel Region

Fortran

```
    ...
!$omp parallel

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

C/C++

```
...
#pragma omp parallel
{
   #pragma omp do
   for(i=0;i<n;i++){
      work(i);
   }
}
```

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

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
```

Fortran

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

C/C++

trip count required
no exit
cycle ok

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

Parallel Region

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

- WS Threads execution their "share" of statements in a PARALLEL region.
- **do/for** worksharing may require run-time work distribution and scheduling

Fortran

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

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

C/C++

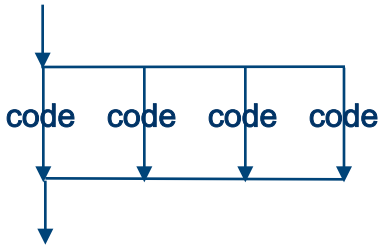
Line 1: Team of threads formed (parallel region).
Line 2-4: Loop iterations are split among threads.
Implied barrier at "**enddo**" and "**}**".
Line 5: (Optional) end of parallel loop.

- Each loop iteration must be independent of other iterations.

Replicated and Workshare Constructs

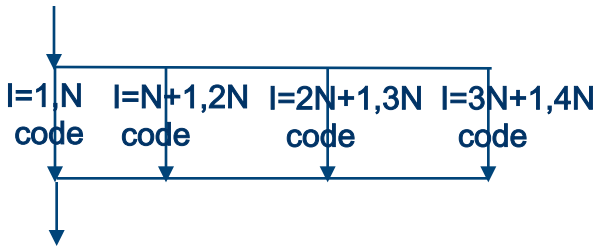
- **Replicated: Work blocks are executed by all threads.**
- **worksharing: Work is divided among threads.**

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



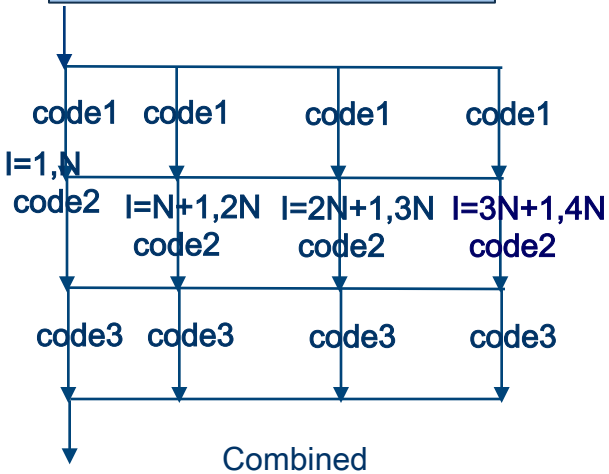
Replicated

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



worksharing

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



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

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

Fortran

```
!$OMP SECTIONS

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

!$OMP END SECTIONS
```

C/C++

```
#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)

Fortran

```
!$OMP single

   glob_count = glob_count + 1
   print *, glob_count

!$OMP end single
```

C/C++

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

}
```

OpenMP Clauses - Scoping

F90 `#pragma omp directive-name [clause [ [,]clause]…]`
C/C++ `!$omp directive-name [clause [ [,]clause]…]`

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

  `private`(variable list)
  - Each thread has its own copy of the specified variable
  - Variables are undefined after worksharing 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 worksharing constructs are always **`private`** by default; everything else is **`shared`** by default

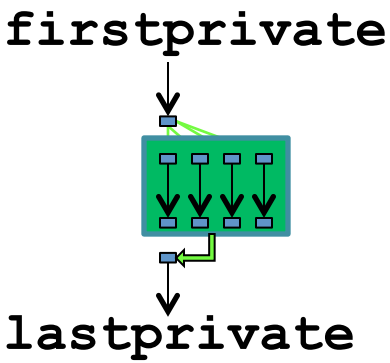
OpenMP Data Scoping

- 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 declared in a scope inside the construct (e.g. ordinary local variables declared inside functions)

OpenMP worksharing - Single

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

- Defaults: All do indices are private, all other variables are shared. (OMP workshare for indices have private indices.)

Fortran

```
!$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
```

C/C++

```
#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);
}
```

OpenMP Data Scoping

Fortran

```
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
```

OpenMP Data Scoping

C Code

```
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
```

OpenMP worksharing Directives

- **`nowait`** clause
  - Threads encounter a barrier synchronization at end of worksharing constructs.
  - Specifies that threads completing assigned work can proceed.
    - Fortran: append the **`nowait`** to the end statement:

```
end do nowait
end sections nowait
end single nowait
```

    - C/C++: **`nowait`** occurs in pragma clause:

```
#pragma omp   ...   nowait
```

OpenMP worksharing - Single

- Fortran: Always add "**use omp_lib**"

Fortran

```
program hello
use omp_lib
! integer :: omp_get_thread_num
print*, "hello, from master"

!$omp  parallel

print*, "id",omp_get_thread_num()

!$omp  end parallel
end program
```

C/C++

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

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

PGI compiler

```
pgf90 -O3 -mp hello.f90
pgcc  -O3 -mp hello.c
```

Intel compiler

```
ifort -O3 -openmp hello.f90
icc   -O3 -openmp hello.c
```

GNU compiler

```
gfortran-O3 -fopenmp hello.f90
gcc       -O3 -fopenmp hello.c
```

We gratefully acknowledge the sponsorship of Chevron Corporation, whose generous support of TACC has made possible this Scientific Computing Curriculum and other student-focused initiatives.

License