Parallel Computing for Science & Engineering

1/29/08

Instructors:
Dr. Bill Barth, Research Associate, TACC
Dr. Kent Milfeld, Research Associate, TACC

Programming with OpenMP on Shared Memory Systems

Reminders

- Enhanced performance and solving larger problems are the main reasons to use parallel computers
- Application developers have to design and program correct and efficient parallel codes for parallel computers to realize these benefits
- Achieving good single-processor performance is hard; achieving scalable parallel performance is harder
- But the payoff can be huge

OpenMP – what is it

- **De facto open standard for Scientific Parallel Programming on Symmetric MultiProcessor (SMP) Systems.**
- **Extensions to Fortran, C and C++ for portable SMP programming**
    - Bases on threads, but
    - Higher-level than POSIX threads (Pthreads) (http://www.llnl.gov/computing/tutorials/pthreads/#Abstract)
- **Implemented by:**
    - **Compiler Directives**
    - **Runtime Library (an API, Application Program Interface)**
    - **Environment Variables**
- **Compiler option required to detect code directives.**
- **http://www.openmp.org/ has tutorials and description.**

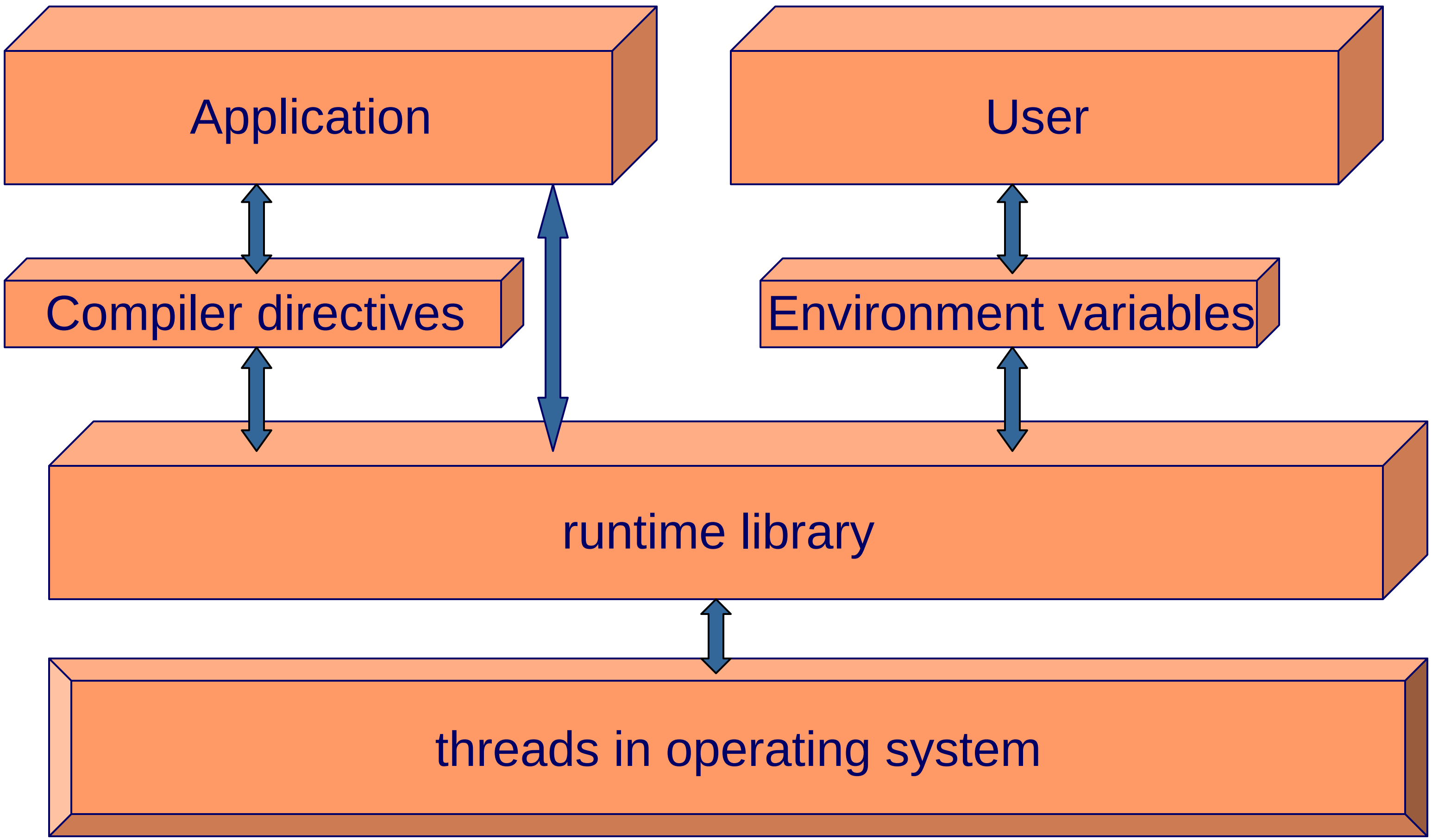
OpenMP History

- Primary OpenMP participants
    - Compaq, HP, IBM, Intel, KAI, SGI, SUN
- U.S. Dept. of Energy ASCI Program
- OpenMP Fortran API, Version 1.0, published Oct. 1997
- OpenMP C API, Version 1.0, published Oct. 1998
- OpenMP 2.0 API for Fortran, published in 2000
- OpenMP 2.0 API for C/C++, published in 2002
- OpenMP 2.5 API for C/C++ & F90 published in 2005

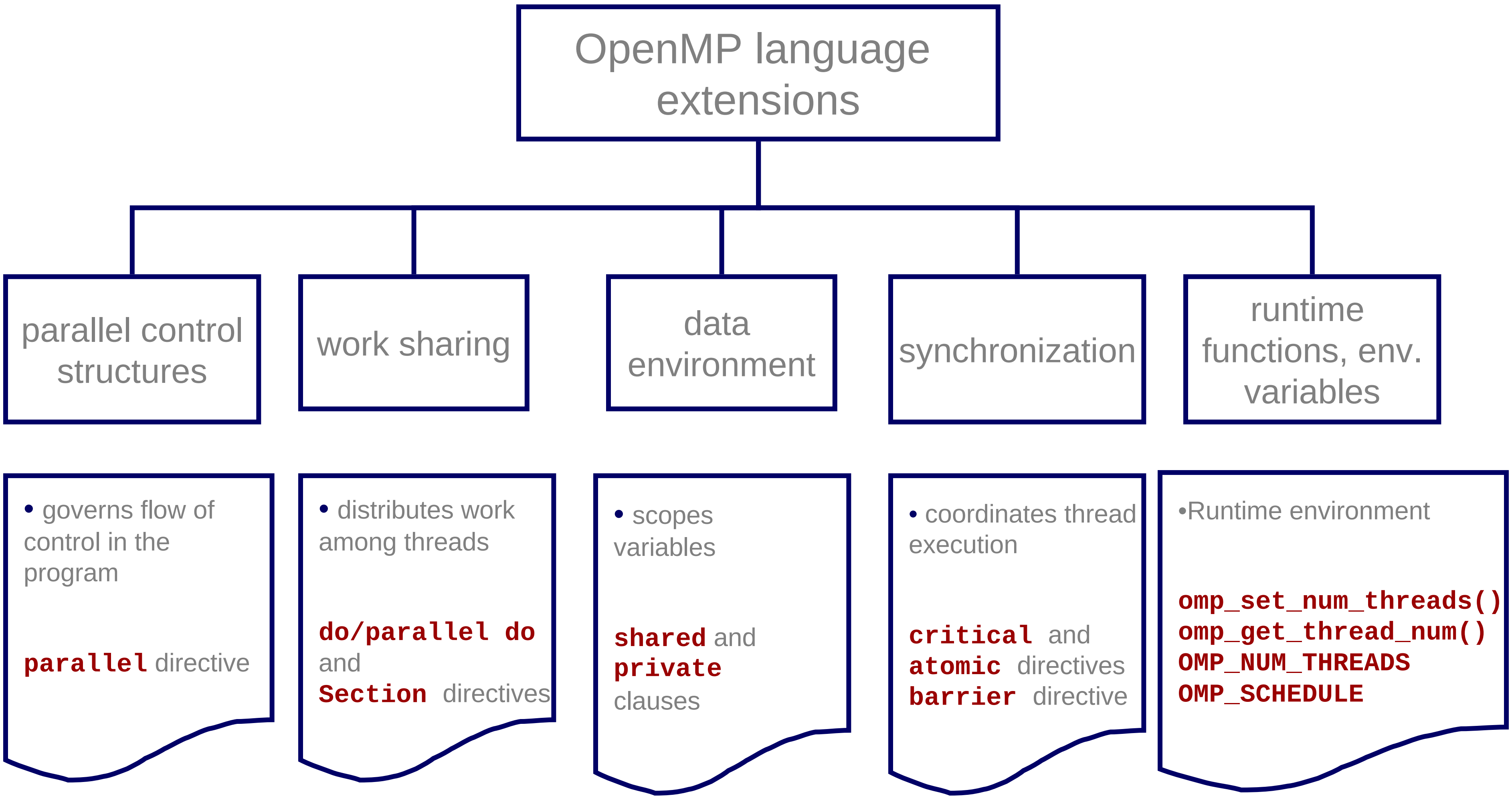
Advantages/Disadvantages of OpenMP

- Pros
  - Shared Memory Parallelism is easier to learn.
  - Coarse-grained or fine-grained parallelism
  - Parallelization can be incremental
  - Widely available, portable
- Cons
  - Scalability limited by memory architecture
  - Available on SMP systems only

OpenMP Architecture



OpenMP Constructs



OpenMP Parallel Directive

Fortran directives begin with: !OMP, COMP or *$OMP sentinel.
C/C++ directives begin with: # pragma omp sentinel.

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

Syntax: *sentinel* parallel *clauses* ***uses defaults w.o. clauses***

Fortran

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

C/C++

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

Parallel Region

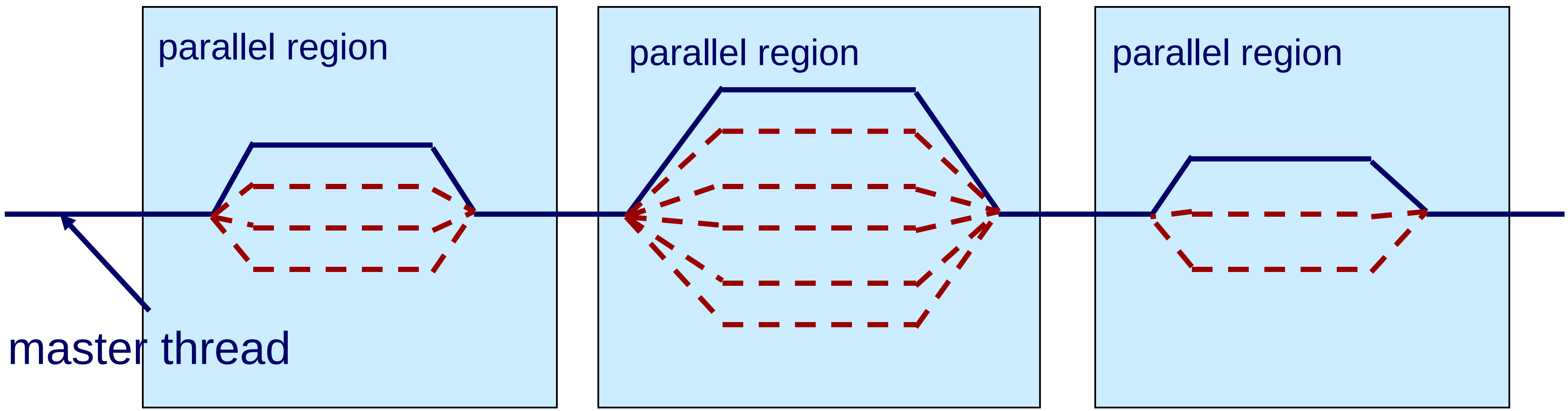
```
1  !$OMP PARALLEL
2        code block
3        call work(…)
4  !$OMP END PARALLEL
```

```
#pragma omp
{ code block
  i=work(…)
}
```

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

OpenMP fork-join parallelism

- Parallel Regions are basic “blocks” within code.
- A master thread is instantiated at run-time & persists throughout execution.
- Master thread assembles team of threads at parallel regions.



OpenMP Clauses
Clauses control the behavior of directives

Syntax: ***sentinel*** **parallel** **clauses** ***uses defaults w.o. clauses***

Data Scoping

private(list), shared(list), default(private|shared|none)

Scheduling

schedule(type,chunk)

IF Control

if(expression)

Ordering

ordered

Initialization

copyin (list), firstprivate(list), lastprivate(list)

Reduction

reduction (operator |intrinsic : list)

Barrier

nowait

How do threads communicate?

- Every thread has access to “global” memory (shared).
- Each thread has access to a stack memory (private).
- Use shared memory to communicate between threads.
- Simultaneous updates to shared memory can create a *race condition. R*esults change with different thread scheduling.
- Use mutual exclusion to avoid data sharing --- but don’t use too many because this will “serialize” performance.

Constructs within Parallel Regions

- **Use OpenMP directives to specify Parallel Region and Work-Sharing constructs.**

```
Parallel

End Parallel
```

Code block	**Each Thread Executes**
DO	**Work-Sharing**
SECTIONS	**Work Sharing**
SINGLE	**Work Sharing, 1 Thread**
CRITICAL	**One Thread at a time**

Parallel DO/for	**Stand-alone**
Parallel SECTIONS	**Parallel Constructs**

OpenMP Combined Directives

- Combined directives
  - PARALLEL DO/FOR and PARALLEL SECTIONS
  - Same as PARALLEL region containing only DO or SECTIONS work-sharing construct

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

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

Work Sharing – do/for

Work-sharing (WS) constructs: do/for, section, and single

- WS Threads execution their "share" of statements in a PARALLEL region.
- Do/for Work Sharing 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 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.
Line 5 (Optional) end of parallel loop (implied barrier at enddo, }).

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

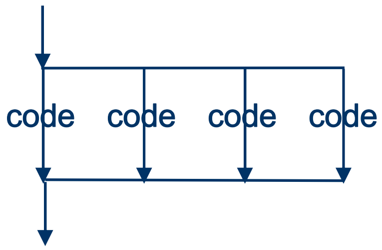
Replicated and Work Share Constructs

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

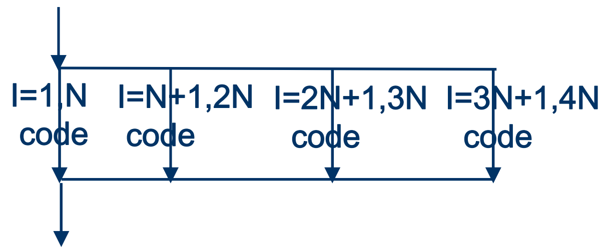
```
PARALLEL
  {code}
END PARALLEL
```

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

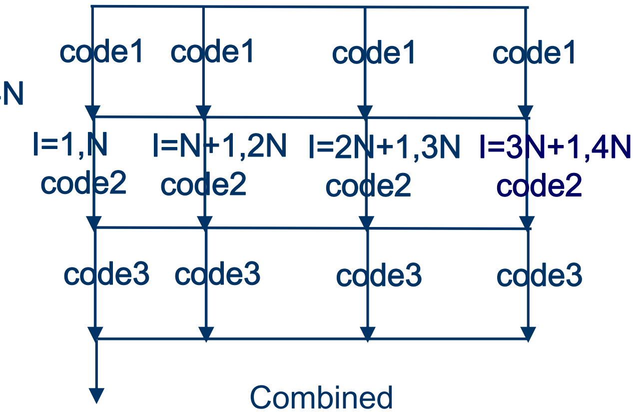
```
PARALLEL
    {code1}
DO
  do I = 1,N*4
    {code2}
  end do
    {code3}
END PARALLEL
```



Replicated



Work Sharing



Combined

OpenMP Work-Sharing Scheduling

Clause Syntax: SCHEDULE(*schedule-type* [,*chunk-size*])

Schedule Type

- STATIC
  - Threads receive chunks of iterations in thread order, round-robin
  - 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.
- DYNAMIC
  - Thread receives chunks as thread becomes available for more work
  - Default chunk size is 1
  - Good for load-balancing

OpenMP Work-Sharing Scheduling

- GUIDED
  - 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
- RUNTIME
  - Schedule is determined at run-time by OMP_SCHEDULE
  - Useful for experimentation

OpenMP Work-Sharing Scheduling

For example, loop with 100 iterations and four 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 Work-Sharing -- 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 Work-Sharing -- Single

- SINGLE
  - Block of code is executed by a single thread
  - Implied barrier

```
!$OMP single

   glob_count = glob_count + 1
   print *, glob_count

!$OMP end single
```

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

}
```

OpenMP Clauses -- Scoping

- Data scoping
  - PRIVATE
    - Each thread has its own copy of the specified variable
    - Variables are undefined after work sharing region
  - SHARED
    - Threads share a single copy of the specified variable
  - DEFAULT
    - A default of PRIVATE, SHARED or NONE can be specified
    - Note that loop counters of work sharing constructs are always PRIVATE by default; everything else is SHARED by default

OpenMP Data Scoping

- Data scoping (continued)
  - REDUCTION
    - 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
  - FIRSTPRIVATE
    - Like PRIVATE, but copies are initialized using value from master thread's copy
  - LASTPRIVATE
    - Like PRIVATE, but final value is copied out to master thread's copy
    - For DO: last iteration; SECTIONS: last section

OpenMP Data Scoping

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

Defaults: All DO LOOP indices are private, all other variables are shared. (OMP DO/FOR loops have private indices.)

```
!$omp parallel do shared(a), &
                        private(t1,t2)
  do i = 1,100
    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<100; 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 Data Scoping

```
SUM = 0
!$OMP PARALLEL DO REDUCTION(+:SUM)
      DO I = 1, 100
        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, 100
        TEMP = F(I)
      END DO
      PRINT *, 'F(100) == ', TEMP
      ! TEMP is equal to F(100) at end of loop
```

OpenMP Work-Sharing Directives

- NOWAIT clause
  - Normally threads encounter a barrier synchronization at end of work-sharing construct.
  - Specifies that threads completing assigned work can proceed.
  - NOWAIT on END DO/END SECTIONS/END SINGLE line for FORTRAN. Include as clause in C/C++.