Introduction to **OpenMP**

Scientific Computing for Engineers: Spring 2017

Piotr Luszczek[1]

Innovative Computing Laboratory, University of Tennessee Knoxville

January 25, 2017

[1] luszczek (at) icl.utk.edu

Outline

1. Introduction
2. Models
3. Syntax and Semantics
4. Individual Directives
5. Work Sharing
6. Synchronization Primitives
7. Memory Consistency Model Considerations
8. Runtime
9. Mutual Exclusion, Timing, and Environment
10. Conclusions, Summary, Further Reading

Acknowledgment

These slides are adapted from the
Lawrence Livermore National Laboratory
OpenMP Tutorial
by
Blaise Barney
available at
https://computing.llnl.gov/tutorials/openMP/

What is **OpenMP**?

- A language extension and an Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism
- Comprises three primary API components
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables
- Portable
  - The API is specified for **C/C++** and **Fortran**
  - Has been implemented for most major platforms including Unix/Linux platforms and Windows NT derivatives (Visual Studio 2012)

What is **OpenMP**? (continued)

- Standardized
  - Jointly defined and endorsed by a group of major computer hardware and software vendors
  - Still hoping for an ANSI standard
- What does **OpenMP** stand for?
  - Short version: OpenMulti-Processing
  - Long version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

OpenMP is NOT

- Meant for **distributed memory** parallel systems (by itself)
- Necessarily implemented **identically** by all vendors
- Guaranteed to make the **most efficient** use of shared memory
- Required to **check** for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences that cause a program to be classified as non-conforming
- Meant to cover compiler-generated **automatic parallelization** and directives to the compiler to assist such parallelization
- Designed to guarantee that **input or output** to the same file is synchronous when executed in parallel
  - The programmer is responsible for **synchronizing** input and output.

History of **OpenMP**

- In the early 90's, vendors of shared-memory machines supplied similar, directive-based, **Fortran** programming extensions.
- The user would augment a serial **Fortran** program with directives specifying which loops were to be parallelized.
- The compiler would be responsible for automatically parallelizing such loops across the SMP processors.
- Implementations were all functionally similar, but were divergent.
- First attempt at a standard was the draft for ANSIX3H5 in 1994.
  - It was never adopted, largely due to waning interest as distributed memory machines became popular.

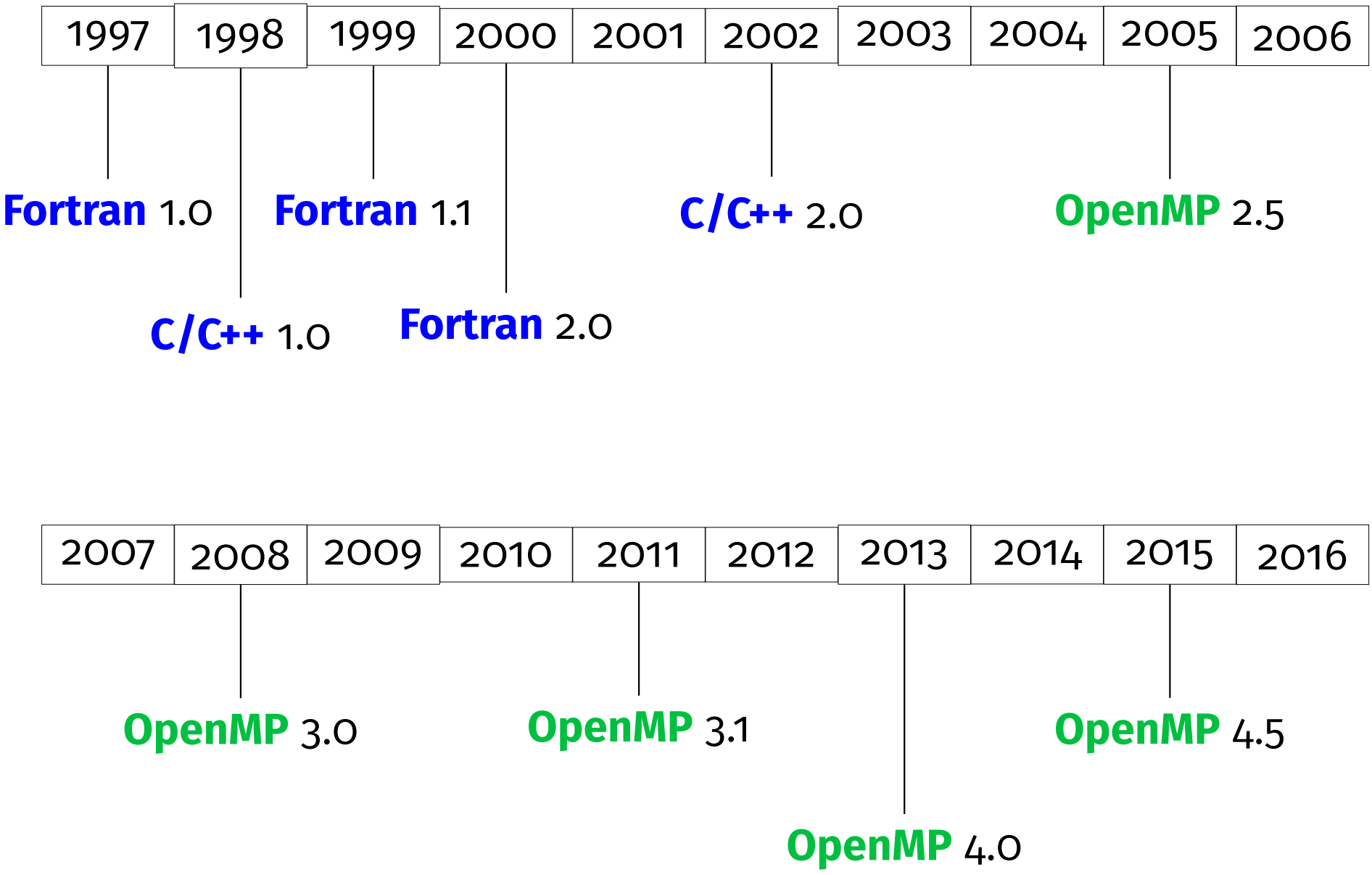
History of **OpenMP** (continued)

- The **OpenMP** standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off, as newer shared memory machine architectures started to become prevalent.
- Led by the OpenMP Architecture Review Board (ARB). Original ARB members included: (Disclaimer: all partner names derived from the OpenMP web site)
  - Compaq / Digital (acquired by HP)
  - Hewlett-Packard Company (split into HP-E and HP-PC)
  - Intel Corporation
  - International Business Machines (IBM)
  - Kuck & Associates, Inc. (KAI) acquired by Intel
  - Silicon Graphics, Inc. (acquired by Rackable and now HPE)
  - Sun Microsystems, Inc. (acquired by Oracle)
  - U.S. Department of Energy ASCI program

Other Contributors

- Endorsing application developers
  - ADINA R&D, Inc.
  - ANSYS, Inc.
  - Dash Associates
  - Fluent, Inc.
  - ILOG CPLEX Division
  - Livermore Software Technology Corporation (LSTC)
  - MECALOG SARL
  - Oxford Molecular Group PLC
  - The Numerical Algorithms Group Ltd.(NAG)
- Endorsing software vendors
  - Absoft Corporation
  - Edinburgh Portable Compilers
  - GENIAS Software GmBH
  - Myrias Computer Technologies, Inc.
  - The Portland Group, Inc. (PGI)

OpenMP Release History



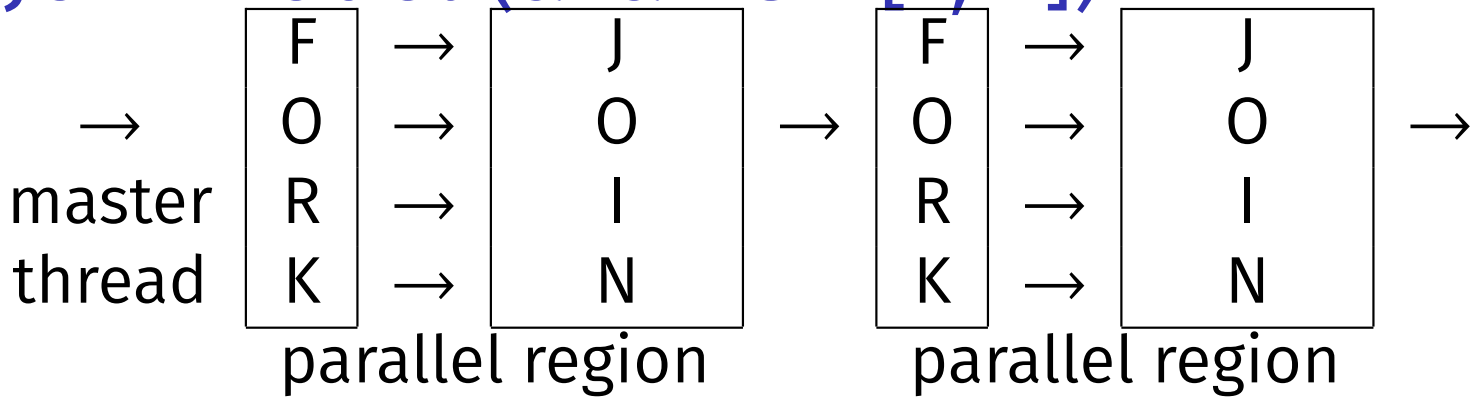
Goals of **OpenMP**

- Standardization
  - Provide a standard among a variety of shared memory architectures/platforms
- Lean and mean
  - Establish a simple and limited set of directives for programming shared memory machines
  - Significant parallelism can be implemented by using just 3 or 4 directives.
- Ease of Use
  - Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
  - Provide the capability to implement both coarse-grain and fine-grain parallelism
- Portability
  - Supports **Fortran** (77, 90, 95, 2003, 2008); C and C++
- Public forum for API and membership

OpenMP Programming Model

- Shared memory, thread-based parallelism
  - OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.
- Explicit Parallelism
  - OpenMP is an **explicit** (not automatic) programming model, offering the programmer full control over parallelization.
- OpenMP uses the **fork-join** model of parallel execution.
- Compiler directive-based
- Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in **C/C++** or **Fortran** source code.
- Nested parallelism support
  - The API provides for the placement of parallel constructs inside of other parallel constructs.
  - Implementations may or may not support this feature.

Fork-Join Model (aka BSP [1, 2])



- All **OpenMP** programs begin as a single process: the *master thread*. The master thread executes sequentially until the first parallel region construct is encountered.
- FORK: the master thread then creates a *team* of parallel threads
- The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the team threads.
- JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Input/Output

- **OpenMP** specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to read/write from/to the same file.
- If every thread conducts I/O to a different file, the issues are not as significant.
- It is entirely up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Memory Model

- **OpenMP** provides a *relaxed-consistency* and *temporary* view of thread memory (in their words). In other words, threads can *cache* their data and are not required to maintain exact consistency with real memory all of the time.
- When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.
- More on this later…

OpenMP Code Structure – **Fortran**

```
    PROGRAM HELLO
    INTEGER VAR1, VAR2, VAR3
Serial code
.
.
Beginning of parallel section. Fork a team of threads. Specify variable
scoping
    !$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)
Parallel section executed by all threads
.
.
All threads join master thread and disband
    !$OMP END PARALLEL
Resume serial code
. . .
    END
```

OpenMP Code Structure – C/C++

```
#include <omp.h>
int main (void) {
    int var1, var2, var3;
Serial code
. . .
Beginning of parallel section. Fork a team of threads. Specify variable
scoping
    #pragma omp parallel private(var1, var2) shared(var3)
    {
Parallel section executed by all threads
    . . .
All threads join master thread and disband
    }
Resume serial code
. . .
}
```

Fortran Directives Format

```
<sentinel> <directive-name> [clause...]
```

- All **Fortran** **OpenMP** directives must begin with a sentinel. The accepted sentinels depend on the type of **Fortran** source (see next two slides).
- A valid **OpenMP** directive must appear after the sentinel and before any clauses.
- Optional clauses can be in any order and repeated as necessary unless otherwise restricted.
- Example:

```
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)
```

Fortran Directives Format (continued)

- Fixed Form Source
  - **!$OMP C$OMP $OMP** are accepted sentinels and must start in column 1
  - All **Fortran** fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line
  - Initial directive lines must have a space/zero in column 6.
  - Continuation lines must have a non-space/zero in column 6.
- Free Form Source
  - **!$OMP** is the only accepted sentinel. Can appear in any column, but must be preceded by white space only.
  - All **Fortran** free form rules for line length, white space, continuation and comment columns apply for the entire directive line
  - Initial directive lines must have a space after the sentinel.
  - Continuation lines must have an ampersand as the last non-blank character in a line. The following line must begin with a sentinel and then the continuation directives.

Fortran Directives Format

- General Rules
  - Comments cannot appear on the same line as a directive
  - Only one directive-name may be specified per directive
  - **Fortran** compilers that are **OpenMP** enabled generally include a command line option that instructs the compiler to activate and interpret all **OpenMP** directives.
  - Several **Fortran** **OpenMP** directives come in pairs and have the form shown below. The “end” directive is optional but advised for readability.

```
!$OMP <directive> [structured block of code]
!$OMP end <directive>
```

C/C++ Directives Format

```
#pragma omp <directive-name> [clause,...]
```

- New line at the end required and precedes the structured block that follows the directive

Example:

```
#pragma omp parallel default(shared) private(beta,pi)
```

C/C++ Directives Format (continued)

- General Rules:
  - Case sensitive
  - Directives follow conventions of the C/C++ standards for compiler directives.
  - Only one directive-name may be specified per line.
  - Each directive applies to at most one succeeding statement, which must be a structured block.
  - Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash (“\”) at the end of a directive line.

Directive Scoping

- Static (Lexical) Extent
  - The code textually enclosed between the beginning and the end of a structured block following a directive.
  - The static extent of a directives does not span multiple routines or code files
- Orphaned Directive
  - An **OpenMP** directive that appears independently from another enclosing directive is said to be an orphaned directive. It exists outside of another directive's static (lexical) extent.
  - Will span routines and possibly code files
- Dynamic Extent
  - The dynamic extent of a directive includes both its static (lexical) extent and the extents of its orphaned directives.

Directive Scoping Example

```
      PROGRAM TEST
      ...
!$OMP PARALLEL
      ...
!$OMP DO
      DO I=. . .
      ...
      CALL SUB1 ...
      END DO
      ...
      CALL SUB2
      ...
!$OMP END PARALLEL
```

```
      SUBROUTINE SUB1
      ...
!$OMP CRITICAL
      ...
!$OMP END CRITICAL
      END

      SUBROUTINE SUB2 ...
      ...
!$OMP SECTIONS
      ...
!$OMP END SECTIONS
      ...
      END DO
```

STATIC EXTENT

The DO directive occurs within an enclosing parallel region.

ORPHANED DIRECTIVES

The CRITICAL and SECTIONS directives occur outside an enclosing parallel region.

DYNAMIC EXTENT

The CRITICAL and SECTIONS directives occur within the dynamic extent of the DO and PARALLEL directives.

PARALLEL Region Construct

- Block of code that will be executed by multiple threads
- Fundamental **OpenMP** parallel construct
- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

PARALLEL Region Syntax – **Fortran**

```
!$OMP PARALLEL [clause ...]
      IF (scalar_logical_expression)
      PRIVATE (list)
      SHARED (list)
      DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
      FIRSTPRIVATE (list)
      REDUCTION (operator: list)
      COPYIN (list)
      NUM_THREADS (scalar-integer-expression)
block
!$OMP END PARALLEL
```

PARALLEL Region Syntax – **C/C++**

```
#pragma omp parallel [clause ...]
      if (scalar_expression)
      private (list)
      shared (list)
      default (shared | none)
      firstprivate (list)
      reduction (operator: list)
      copyin (list)
      num_threads (integer-expression)
structured_block
```

PARALLEL Region – Number of Threads

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the **IF** clause
  2. Setting of the **NUM_THREADS** clause
  3. Use of the **omp_set_num_threads()** library function
  4. Setting of the **OMP_NUM_THREADS** environment variable
  5. Implementation default – usually the number of cores on a node.
- Threads are numbered from 0 (master thread) to $N - 1$

Nested Parallel Regions

- Use the **omp_get_nested()** library function to determine if nested parallel regions are enabled.
- The two methods available for enabling nested parallel regions (if supported) are:
  - The **omp_set_nested()** library routine
  - Setting of the **OMP_NESTED** environment variable to TRUE
- If not supported, a parallel region nested with in another parallel region results in the creation of a new team, consisting of one thread, by default.

PARALLEL Region Restrictions

- A parallel region must be a structured block that does not span multiple routines or code files
- It is illegal to branch into or out of a parallel region
- Only a single **IF** clause is permitted
- Only a single **NUM_THREADS** clause is permitted

PARALLEL Region Example – **Fortran**

```
      PROGRAM HELLO
      INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,
     + OMP_GET_THREAD_NUM
C Fork a team of threads with each thread having a private TID variable
!$OMP PARALLEL PRIVATE(TID)
C Obtain and print thread id
      TID = OMP_GET_THREAD_NUM()
      PRINT *, 'Hello World from thread = ', TID
C Only master thread does this
      IF (TID .EQ. 0) THEN
        NTHREADS = OMP_GET_NUM_THREADS()
        PRINT *, 'Number of threads = ', NTHREADS
      END IF
C All threads join master thread and disband
!$OMP END PARALLEL
      END
```

PARALLEL Region Example – **C/C++**

```
#include <omp.h>
int main(void) {
        int nthreads, tid;
/* Fork a team of threads with each thread having a private tid variable */
#pragma omp parallel private(tid)
        {
        /* Obtain and print thread id */
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
        /* Only master thread does this */
        if (tid == 0) {
          nthreads = omp_get_num_threads();
          printf("Number of threads = %d\n", nthreads);
         }
        }
/* All threads join master thread and terminate */
}
```

Work-sharing Constructs

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all
- Successive work-sharing constructs must be encountered in the same order by all members of a team.

Types of Work-sharing Constructs

- **DO/for**
  - Shares iterations of a loop across the team
  - Represents a type of “data parallelism”
- **SECTIONS**
  - Breaks work into separate, discrete sections
  - Each section is executed by a thread.
  - Can be used to implement a type of “functional parallelism”
- **SINGLE**
  - Serializes a section of code

Fortran DO Directive Syntax

```
!$OMP DO [clause ...]
        SCHEDULE (type [,chunk])
        ORDERED
        PRIVATE (list)
        FIRSTPRIVATE (list)
        LASTPRIVATE (list)
        SHARED (list)
        REDUCTION (operator | intrinsic : list)
        COLLAPSE (n)
do_loop
!$OMP END DO [ NOWAIT ]
```

C/C++ DO Directive Syntax

```
#pragma omp for [clause ...]
        schedule (type [,chunk])
        ordered
        private (list)
        firstprivate (list)
        lastprivate (list)
        shared (list)
        reduction (operator: list)
        collapse (n)
        nowait
for_loop
```

SCHEDULE Clause

Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

STATIC Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC Loop iterations are divided into pieces of size chunk, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.

RUNTIME The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.

Other DO/for Clauses

NOWAIT/nowait If specified, then threads do not synchronize at the end of the parallel loop.

ORDERED Specifies that the iterations of the loop must be executed as they would be in a serial program.

COLLAPSE Specifies how many loops in a nested loop should be collapsed into one large iteration space and divided according to the schedule clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Other clauses are described in detail later.

Fortran DO Directive Example

```
      PROGRAM VEC_ADD_DO
      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=1000, CHUNKSIZE=100)
      REAL A(N), B(N), C(N)
! Some initializations
      DO I = 1, N
        A(I) = I * 1.0
        B(I) = A(I)
      END DO
      CHUNK = CHUNKSIZE
!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
      DO I = 1, N
        C(I) = A(I) + B(I)
      END DO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
      END
```

C/C++ for Directive Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N 1000
int main (void) {
        int i, chunk;
        float a[N], b[N], c[N];
/* Some initializations */
        for (i=0; i < N; i++) a[i] = b[i] = i * 1.0;
        chunk = CHUNKSIZE;
#pragma omp parallel shared(a,b,c,chunk) private(i)
        {
        #pragma omp for schedule(dynamic,chunk) nowait
        for (i=0; i < N; i++) c[i] = a[i] + b[i];
        }
} /* end of parallel section */
```

SECTIONS Directive – **Fortran** Syntax

```
!$OMP SECTIONS [clause ...]
        PRIVATE (list)
        FIRSTPRIVATE (list)
        LASTPRIVATE (list)
        REDUCTION (operator | intrinsic : list)
!$OMP SECTION
        block
!$OMP SECTION
        block
!$OMP END SECTIONS [ NOWAIT ]
```

SECTIONS Directive – **C/C++** Syntax

```
#pragma omp sections [clause ...]
        private (list)
        firstprivate (list)
        lastprivate (list)
        reduction (operator: list)
        nowait
{
#pragma omp section newline
        structured_block
#pragma omp section newline
        structured_block
}
```

SECTIONS Directive Example – **Fortran**

```
      PROGRAM VEC_ADD_SECTIONS
      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N), D(N)
! Some initializations
      DO I = 1, N
        A(I) = I * 1.5
        B(I) = I + 22.35
      END DO
!$OMP PARALLEL SHARED(A,B,C,D), PRIVATE(I)
!$OMP SECTIONS
!$OMP SECTION
      DO I = 1, N
        C(I) = A(I) + B(I)
      END DO
!$OMP SECTION
      DO I = 1, N
        D(I) = A(I) * B(I)
      END DO
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
      END
```

SECTIONS Directive Example – **C/C++**

```
int main (void) {
        int i; float a[N], b[N], c[N], d[N];
        for (i=0; i < N; i++) { /* Some initializations */
          a[i] = i * 1.5; b[i] = i + 22.35;
        }
#pragma omp parallel shared(a,b,c,d) private(i)
        {
#pragma omp sections nowait
          {
#pragma omp section
            for (i=0; i < N; i++) c[i] = a[i] + b[i];
#pragma omp section
            for (i=0; i < N; i++) d[i] = a[i] * b[i];
          } /* end of sections */
        } /* end of parallel section */
}
```

Questions for Thought

- What happens if the number of threads and the number of SECTIONs are different?
- What if there are more threads than SECTIONs?
- Fewer threads than SECTIONs?
- Which thread executes which SECTION?

Synchronization Constructs

- Motivation: Consider a simple example where two threads on two different processors are both trying to increment a variable x at the same time (assume x is initially 0).

THREAD 1:	THREAD 2:
increment(x) {	increment(x) {
x = x + 1;	x = x + 1;
}	}
THREAD 1:	THREAD 2:
10 LOAD A, (x address)	10 LOAD A, (x address)
20 ADD A, 1	20 ADD A, 1
30 STORE A, (x address)	30 STORE A, (x address)

- The incrementation of x must be synchronized between the two threads to insure that the correct result is produced.
- **OpenMP** provides a variety of synchronization constructs that control how the execution of each thread proceeds relative to other team threads.

Synchronization Constructs

- The CRITICAL directive specifies a region of code that must be executed by only one thread at a time.

Fortran	!$OMP CRITICAL [name] block !$OMP END CRITICAL
C/C++	#pragma omp critical [name] structured_block

- If a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.
- The optional name enables multiple different CRITICAL regions to exist.
- Names act as global identifiers. Different CRITICAL regions with the same name are treated as the same region.
- All unnamed CRITICAL regions are treated as the same region.

CRITICAL Directive Example

```
      PROGRAM CRITICAL
      INTEGER X
      X = 0
!$OMP PARALLEL SHARED(X)
!$OMP CRITICAL
      X = X + 1
!$OMP END CRITICAL
!$OMP END PARALLEL
      END
```

```
#include <omp.h>
int main(void) {
      int x = 0;
#pragma omp parallel shared(x)
      {
#pragma omp critical
        x = x + 1;
      } /* end of parallel region */
}
```

Other Synchronization Constructs

- MASTER directive specifies a region that is to be executed only by the master thread
- All other threads on the team skip this section of code.

Fortran

```
!$OMP MASTER
  block
!$OMP END MASTER
```

C/C++

```
#pragma omp master
  structured block
```

- BARRIER directive synchronizes all threads in the team
  - When a BARRIER directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

Fortran

```
!$OMP BARRIER
```

C/C++

```
#pragma omp barrier
```

- ATOMIC directive designates a given memory location for atomic update, rather than letting multiple threads attempt to write
  - Applies only to a single, immediately following statement (only limited syntax supported)

Fortran

```
!$OMP ATOMIC
```

C/C++

```
#pragma omp atomic
```

FLUSH Directive

- Identifies a synchronization point at which the implementation must provide a consistent view of memory
- Thread-visible variables are written back to memory at this point.
- Necessary to instruct the compiler that a variable must be written to/read from the memory system, i.e. that a variable cannot be kept in a local CPU register
  - Keeping a variable in a register in a loop is very common when producing efficient machine language code for a loop.

FLUSH Directive (continued)

Fortran `!$OMP FLUSH (list)`

C/C++ `#pragma omp flush (list)`

- The optional list contains a list of named variables that will be flushed in order to avoid flushing all variables. For pointers in the list, the pointer itself is flushed, not the object to which it points.
- Implementations must ensure any prior modifications to thread-visible variables are visible to all threads after this point; i.e., compilers must restore values from registers to memory, hardware might need to flush write buffers, etc.
- The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present.

FLUSH Directive and the End of Other Directives

The FLUSH directive is implied for the directives shown in the table below. The directive is not implied if a NOWAIT clause is present.

Fortran	**C/C++**
BARRIER	barrier
END PARALLEL	parallel – upon entry and exit
CRITICAL and END CRITICAL	critical – upon entry and exit
END DO	for – upon exit
END SECTIONS	sections – upon exit
END SINGLE	single – upon exit
ORDERED and END ORDERED	ordered – upon entry and exit

ORDERED Directive

- Specifies that iterations of the enclosed loop will be executed in the same order as if they were executed on a serial processor
- Threads will need to wait before executing their chunk of iterations if previous iterations haven't completed yet.
- Used within a DO / for loop with an ORDERED clause
- The ORDERED directive provides a way to "fine tune" where ordering is to be applied within a loop. Otherwise, it is not required.
- An ORDERED directive can only appear in the dynamic extent of the following directives:

  Fortran DO or PARALLEL DO C/C++ for or parallel for
- Only one thread is allowed in an ordered section at any time
- It is illegal to branch into or out of an ORDERED block.
- An iteration of a loop must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.
- A loop that contains an ORDERED directive must be a loop with an ORDERED clause.

ORDERED Directive Syntax

Fortran

```
!$OMP DO ORDERED [clauses...]
  (loop region)
!$OMP ORDERED
  (block)
!$OMP END ORDERED
  (end of loop region)
!$OMP END DO
```

C/C++

```
#pragma omp for ordered [clauses...]
  (loop region)
#pragma omp ordered
  structured_block
  (endo of loop region)
```

ORDERED Directive Syntax

```
#include <stdio.h>
#include <omp.h>
static float a[1000 ], b[1000 ], c[1000
];
void test(int first, int last) {
        int i;
#pragma omp for schedule(static)
ordered
        for (i = first; i <= last; ++i) {
/* Do something here. */
        if (i % 2) {
#pragma omp ordered
        printf("test() iteration %d\n", i);
        }
}
```

```
void test2(int iter) {
#pragma omp ordered
        printf("test2() iteration %d\n",
iter);
}
int main(void) {
        int i;
#pragma omp parallel
        {
        test(1, 8);
#pragma omp for ordered
        for (i = 0 ; i < 5 ; i++)
          test2(i);
        }
}
```

Data Scope Attribute Clauses

- Also called data sharing attribute clauses
- Because **OpenMP** is based upon the shared memory programming model, most variables are shared by default.
- Global variables include:
  - **Fortran**: COMMON blocks, SAVE variables, MODULE variables
  - **C/C++**: File scope variables, static
- Private variables include:
  - Loop index variables
  - Stack variables in subroutines called from parallel regions
  - **Fortran**: Automatic variables within a statement block
- Clauses used to explicitly define how variables should be scoped include:
  - PRIVATE SHARED
  - FIRSTPRIVATE LASTPRIVATE
  - DEFAULT
  - REDUCTION
  - COPYIN

Data Scope Attribute Clauses (continued)

- Used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.
- Provide the ability to control the data environment during execution of parallel constructs
- Define how and which data variables in the serial section of the program are transferred to the parallel sections of the program (and back)
- Define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads
- Effective only with in their lexical/static extent

PRIVATE and SHARED Clauses

- PRIVATE Clause
  - Declares variables in its list to be private to each thread
  - A new object of the same type is declared once for each thread in the team.
  - All references to the original object are replaced with references to the new object.
  - Variables declared PRIVATE should be assumed to be uninitialized for each thread.
- SHARED Clause
  - Declares variables in its list to be shared among all threads in the team
  - A shared variable exists in only one memory location and all threads can read or write to that address
  - It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

FIRSTPRIVATE and LASTPRIVATE Clauses

- FIRSTPRIVATE Clause
  - Combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list
  - Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct.
- LASTPRIVATE Clause
  - Combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object
  - The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.
  - For example, the team member that executes the final iteration for a DO section, or the team member that executes the last SECTION of a SECTIONS context, performs the copy with its own values.

PRIVATE Variables Example

```
int main(void) {
        int A = 10;
        int B, C;
        int n = 20;
#pragma omp parallel
        {
#pragma omp for private(i) firstprivate(A) lastprivate(B)
        for (int i=0; i < n; i++) {
          /* .... */
          B = A + i; /* A undefined unless declared firstprivate */
          /* .... */
        }
        C = B; /* B undefined unless declared lastprivate */
        } /* end of parallel region */
}
```

DEFAULT Clause

- Allows the user to specify a default scope for all variables in the lexical extent of any parallel region
- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses.
- The **C/C++** **OpenMP** specification does not include **private** or **firstprivate** as a possible default. However, actual implementations may provide this option.
- Using NONE as a default requires that the programmer explicitly scope all variables.

Fortran	DEFAULT (PRIVATE | FIRSTPRIVATE | SHARED | NONE)
C/C++	default (shared | none)

REDUCTION Clause

- Performs a reduction on the variables that appear in its list.
- A private copy for each list variable is created for each thread. At the end of the reduction, the reduction operator is applied to all private copies of the shared variable, and the final result is written to the global shared variable.
- Syntax:

Fortran	REDUCTION (operator | intrinsic: list)
C/C++	reduction (operator: list)

REDUCTION Clause Example – **Fortran**

```
      PROGRAM DOT_PRODUCT
      INTEGER N, CHUNKSIZE, CHUNK, I
      PARAMETER (N=100, CHUNKSIZE=10)
      REAL A(N), B(N), RESULT
! Some initializations
      DO I = 1, N
        A(I) = I * 1.0 ; B(I) = I * 2.0
      END DO
      RESULT= 0.0
      CHUNK = CHUNKSIZE
!$OMP PARALLEL DO DEFAULT(SHARED) PRIVATE(I)
!$OMP& SCHEDULE(STATIC,CHUNK) REDUCTION(+:RESULT)
      DO I = 1, N
        RESULT = RESULT + (A(I) * B(I))
      END DO
!$OMP END PARALLEL DO NOWAIT
      PRINT *, 'Final Result= ', RESULT
      END
```

REDUCTION Clause Example – **C/C++**

```
#include <omp.h>
int main (void) {
        int i, n = 100, chunk = 10;
        float a[100 ], b[100 ], result = 0.0;
        /* Some initializations */
        for (i=0; i < n; i++) {
        a[i] = i * 1.0;
        b[i] = i * 2.0;
        }
        #pragma omp parallel for default(shared) private(i) \
        schedule(static,chunk) reduction(+:result)
        for (i=0; i < n; i++)
        result += (a[i] * b[i]);
        printf("Final result= %f\n",result);
}
```

THREADPRIVATE Directive

- Used to make global file scope variables (**C/C++**) or common blocks (Fortran) local and persistent to a thread through the execution of multiple parallel regions
- Must appear after the declaration of listed variables/common blocks
- Each thread then gets its own copy of the variable/common block, so data written by one thread is not visible to other threads.
- On first entry to a parallel region, data in THREADPRIVATE variables and common blocks should be assumed undefined, unless a COPYIN clause is specified in the PARALLEL directive
- THREADPRIVATE variables differ from PRIVATE ones because they are able to persist between different parallel sections of a code.
- Data in THREADPRIVATE objects is guaranteed to persist only if the dynamic threads mechanism is ”turned off” and the number of threads in different parallel regions remains constant. The default setting of dynamic threads is undefined.

Fortran	!$OMP THREADPRIVATE (/cb/, ...)
C/C++	#pragma omp threadprivate (list)

THREADPRIVATE Example – **Fortran**

```
      PROGRAM THREADPRIV
      INTEGER A, B, I, TID, OMP_GET_THREAD_NUM ; REAL*4 X
      COMMON /C1/ A
!$OMP THREADPRIVATE(/C1/, X)
C * * * Explicitly turn off dynamic threads * * *
      CALL OMP_SET_DYNAMIC(.FALSE.)
      PRINT *, '1st Parallel Region:'
!$OMP PARALLEL PRIVATE(B, TID)
      TID = OMP_GET_THREAD_NUM ()
      A = TID ; B = TID ; X = 1.1 * TID + 1.0
      PRINT *, 'Thread',TID,': A,B,X=',A,B,X
!$OMP END PARALLEL
      PRINT *, 'Master thread doing serial work here', '2nd Parallel Region: '
!$OMP PARALLEL PRIVATE(TID)
      TID = OMP_GET_THREAD_NUM ()
      PRINT *, 'Thread',TID,': A,B,X=',A,B,X
!$OMP END PARALLEL
      END
```

THREADPRIVATE Example - **C/C++**

```
#include <omp.h>
int a, b, i, tid; float x;
#pragma omp threadprivate(a, x)
int main (void) {
        omp_set_dynamic(0); /* Explicitly turn off dynamic threads */
        printf("1st Parallel Region:\n");
#pragma omp parallel private(b,tid)
        {
        tid = omp_get_thread_num(); a = tid; b = tid; x = 1.1 * tid +1.0;
        printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
        } /* end of parallel region */
        printf("Master thread doing serial work here\n");
        printf("2nd Parallel Region:\n");
#pragma omp parallel private(tid)
        {
        tid = omp_get_thread_num();
        printf("Thread %d: a,b,x= %d %d %f\n",tid,a,b,x);
        } /* end of parallel section */
```

COPYIN Clause

- Provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team
- List contains the names of variables to copy. In Fortran, the list can contain both the names of common blocks and named variables.
- The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

Fortran	COPYIN (list)
C/C++	copyin (list)

Runtime Library Routines

- The **OpenMP** standard defines an API for library calls that perform a variety of functions:
  - Query the number of threads/processors, set number of threads to use
  - General purpose locking routines (semaphores)
  - Portable wall clock timing routines
  - Set execution environment functions: nested parallelism, dynamic adjustment of threads.
  - For **C/C++**, it may be necessary to specify the include file "omp.h".

Note: Your implementation may or may not support nested parallelism and/or dynamic threads. If nested parallelism is supported, it is often only nominal, in that a nested parallel region may only have one thread.

OMP_SET_NUM_THREADS()

- Sets the number of threads that will be used in the next parallel region
- Must be a positive integer
- Can only be called from serial portion of the code
- Has precedence over the OMP_NUM_THREADS environment variable

Fortran	SUBROUTINE OMP_SET_NUM_THREADS(integer)
C/C++	#include <omp.h> void omp_set_num_threads(int num_threads)

OMP_GET_NUM_THREADS() and OMP_GET_THREAD_NUM()

- OMP_GET_NUM_THREADS()
  - Returns the number of threads that are currently in the team executing the parallel region from which it is called
- OMP_GET_THREAD_NUM()
  - Returns the thread number of the thread, within the team, making this call. This number will be between 0 and OMP_GET_NUM_THREADS-1. The master thread of the team is thread 0.
  - If called from a nested parallel region, or a serial region, this function will return 0.

Fortran	INTEGER FUNCTION OMP_GET_NUM_THREADS() INTEGER FUNCTION OMP_GET_THREAD_NUM()
C/C++	#include <omp.h> int omp_get_num_threads(void) int omp_get_thread_num(void)

OMP_GET_MAX_THREADS()

- Returns the maximum value that can be returned by a call to the OMP_GET_NUM_THREADS function
- Generally reflects the number of threads as set by the OMP_NUM_THREADS environment variable or the OMP_SET_NUM_THREADS() library routine.
- May be called from both serial and parallel regions of code

Fortran	INTEGER FUNCTION OMP_GET_MAX_THREADS()
C/C++	#include <omp.h> int omp_get_max_threads(void)

OMP_GET_THREAD_LIMIT() and OMP_GET_NUM_PROCS()

- OMP_GET_THREAD_LIMIT()
  - New with **OpenMP** 3.0
  - Returns the maximum number of **OpenMP** threads available to a program
- OMP_GET_NUM_PROCS()
  - Returns the number of processors that are available to the program

Fortran	INTEGER FUNCTION OMP_GET_THREAD_LIMIT() INTEGER FUNCTION OMP_GET_NUM_PROCS()
C/C++	#include <omp.h> int omp_get_thread_limit(void) int omp_get_num_procs(void)

OMP_IN_PARALLEL()

- May be called to determine if the section of code currently executing is parallel or not
- For **Fortran**, this function returns .TRUE. if it is called from the dynamic extent of a region executing in parallel, and .FALSE. otherwise. For **C/C++**, it will return a non- zero integer if parallel, and zero otherwise.

Fortran	LOGICAL FUNCTION OMP_IN_PARALLEL()
C/C++	#include <omp.h> int omp_in_parallel(void)

OMP_SET_DYNAMIC()

- Enables or disables dynamic adjustment (by the run time system) of the number of threads available for execution of parallel regions
- For **Fortran**, if called with .TRUE. then the number of threads available for subsequent parallel regions can be adjusted automatically by the run-time environment. If called with .FALSE., dynamic adjustment is disabled.
- For **C/C++**, if dynamic_threads evaluates to non-zero, then the mechanism is enabled, otherwise it is disabled.
- The OMP_SET_DYNAMIC subroutine has precedence over the OMP_DYNAMIC environment variable.
- The default setting is implementation dependent.
- Must be called from a serial section of the program

Fortran	SUBROUTINE OMP_SET_DYNAMIC(scalar_logical_expressio
C/C++	#include <omp.h> void omp_set_dynamic(int dynamic_threads)

OMP_GET_DYNAMIC()

- Used to determine if dynamic thread adjustment is enabled or not
- For **Fortran**, this function returns .TRUE. if dynamic thread adjustment is enabled, and .FALSE. otherwise.
- For **C/C++**, non-zero will be returned if dynamic thread adjustment is enabled, and zero otherwise.

Fortran	LOGICAL FUNCTION OMP_GET_DYNAMIC()
C/C++	#include <omp.h> int omp_get_dynamic(void)

OMP_SET_NESTED()

- Used to enable or disable nested parallelism
- For **Fortran**, calling this function with .FALSE. will disable nested parallelism, and calling with .TRUE. will enable it.
- For **C/C++**, if nested evaluates to non-zero, nested parallelism is enabled; otherwise it is disabled.
- The default is for nested parallelism to be disabled.
- This call has precedence over the OMP_NESTED environment variable.

Fortran	SUBROUTINE OMP_SET_NESTED(scalar_logical_expr)
C/C++	#include <omp.h> void omp_set_nested(int nested)

OMP_GET_NESTED()

- Used to determine if nested parallelism is enabled or not
- For **Fortran**, this function returns .TRUE. if nested parallelism is enabled, and .FALSE. otherwise.
- For **C/C++**, non-zero will be returned if nested parallelism is enabled, and zero otherwise.

Fortran	LOGICAL FUNCTION OMP_GET_NESTED()
C/C++	#include <omp.h> int omp_get_nested(void)

Locking Routines

- OMP_INIT_LOCK()
  - Initializes a lock associated with the lock variable
  - The initial state is unlocked.
  - For **Fortran**, var must be an integer large enough to hold an address, such as INTEGER *8 on 64-bit systems.

Fortran	SUBROUTINE OMP_INIT_LOCK(var)
C/C++	#include <omp.h> void omp_init_lock(omp_lock_t *lock)

- OMP_DESTROY_LOCK()
  - Disassociates the given lock variable from any locks

Fortran	SUBROUTINE OMP_DESTROY_LOCK(var)
C/C++	#include <omp.h> void omp_destroy_lock(omp_lock_t *lock)

Locking Routines (continued)

- OMP_SET_LOCK()
  - Forces the executing thread to wait until the specified lock is available

Fortran	SUBROUTINE OMP_SET_LOCK(var)
C/C++	#include <omp.h> void omp_set_lock(omp_lock_t *lock)

- OMP_UNSET_LOCK()
  - Releases the lock from the executing thread
- OMP_TEST_LOCK()
  - Attempts to set a lock, but does not block if the lock is unavailable

OMP_GET_WTIME()

- Provides a portable wall clock timing routine
- Returns a double-precision floating point value equal to the number of elapsed seconds since some point in the past
- Usually used in pairs with the value of the first call subtracted from the value of the second call to obtain the elapsed time for a block of code
- Designed to be per thread times, and therefore may not be globally consistent across all threads in a team

Fortran	DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
C/C++	#include <omp.h> double omp_get_wtime(void)

OpenMP Environment Variables

- OMP_SCHEDULE
  - Applies only to DO, PARALLEL DO (**Fortran**) and for, parallel for (**C/C++**) directives which have their schedule clause set to RUNTIME. The value of this variable determines how iterations of the loop are scheduled on processors.
  - For example:

    ```
    setenv OMP_SCHEDULE "guided, 4"
    setenv OMP_SCHEDULE "dynamic"
    ```
- OMP_NUM_THREADS
  - Sets the number of threads to use during execution
  - For example:

    ```
    setenv OMP_NUM_THREADS 8
    ```

OpenMP Environment Variables (cont.-d)

- OMP_DYNAMIC
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions
  - Valid values are TRUE or FALSE.
  - For example:

    ```
    setenv OMP_DYNAMIC TRUE
    ```
- OMP_NESTED
  - Enables or disables nested parallelism
  - Valid values are TRUE or FALSE.
  - For example:

    ```
    setenv OMP_NESTED TRUE
    ```

Thread Stack Size

- The **OpenMP** standard does not specify how much stack space a thread should have. Consequently, implementations will differ in the default thread stack size.
- Default thread stack size can be easy to exhaust. It can also be non-portable between compilers.
- Threads that exceed their stack allocation may or may not seg fault. An application may continue to run while data is being corrupted.
- Statically linked codes may be subject to further stack restrictions.
- A user's login shell may also restrict stack size.
- May need to increase thread stack size. How to do this is system-dependent.

Performance Expectations

- One might expect to get an N times speedup when running a program parallelized using **OpenMP** on an N processor/core platform. However, this is seldom the case due to the following reasons:
  - A large portion of the program may not be parallelized by **OpenMP**, which means that the theoretical upper limit of speedup is limited according to Amdahl's law.
  - N processors in a SMP may have N times the computation power, but the memory bandwidth usually does not scale up N times. Quite often, the original memory path is shared by multiple processors and performance degradation may be observed when they compete for the shared memory bandwidth.
  - Many other common problems affecting the final speedup in parallel computing also apply to **OpenMP**, such as load balancing and synchronization overhead.

References and Further Reading

1. **OpenMP** website: openmp.org
   API specifications, FAQ, presentations, discussions, media releases, calendar, membership application and more ...
2. Wikipedia: en.wikipedia.org/wiki/OpenMP
3. Barbara Chapman, Gabriele Jost, and Ruud van der Pas: *Using OpenMP*. The MIT Press, 2008.
4. Compiler documentation
   - IBM: www-4.ibm.com/software/ad/fortran
   - Cray: http://docs.cray.com/ (Cray Fortran Reference Manual)
   - Intel: www.intel.com/software/products/compilers/
   - PGI: www.pgroup.com
   - PathScale: http://www.pathscale.com/EKOPath-User-Guide
   - GNU: http://gcc.gnu.org/projects/gomp/
5. cOMPunity http://www.compunity.org/
6. International workshop on **OpenMP** www.iwomp.org

Advanced Usage, Alternative Views and Opinions on **OpenMP**

1. Tasks API
   - Compare with Apple's Grand Central Dispatch, Cilk, Cilk++, Intel Parallel Collections, ompSS, StarPU, QUARK, Thread Building Blocks, ...
2. Features specific to C++
   - Not covered: invocation of c-tors (constructors) and d-tors, ...
3. Thread local storage
4. EPCC micro benchmark
   - http://www.epcc.ed.ac.uk/software-products/openmpmpi-microbenchmarks
5. *Is **OpenMP** for Users?* by William Gropp
   - www.cs.illinois.edu/~wgropp/bib/talks/tdata/2004/openmpusers.pdf
6. OpenACC and **OpenMP** 4.0
   - Work in progress that depends on committee time, community feedback, and market pressures within the industry

References

L. Valiant.
A bridging model for parallel computation.
Communications of ACM, 33(8):103–111, 1990.

L. G. Valiant.
Bulk-synchronous parallel computers.
In M. Reeve, editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons, 1989.