



Overview of OpenMP
OpenMP directives
OpenMP data sharing clauses
OpenMP performance

Parallel Programming with OpenMP

OpenMP

What is OpenMP?

- Application Program Interface (API)
- Used for shared-memory parallel programming
 - Extensions to devices such as GPUs
- Defined for Fortran and C/C++
- Implemented on multiprocessors running Unix and Windows NT
- Standard agreed upon by major hardware and software vendors

Advantages

- Portable, Simple, Flexible interface, Scalable programming model

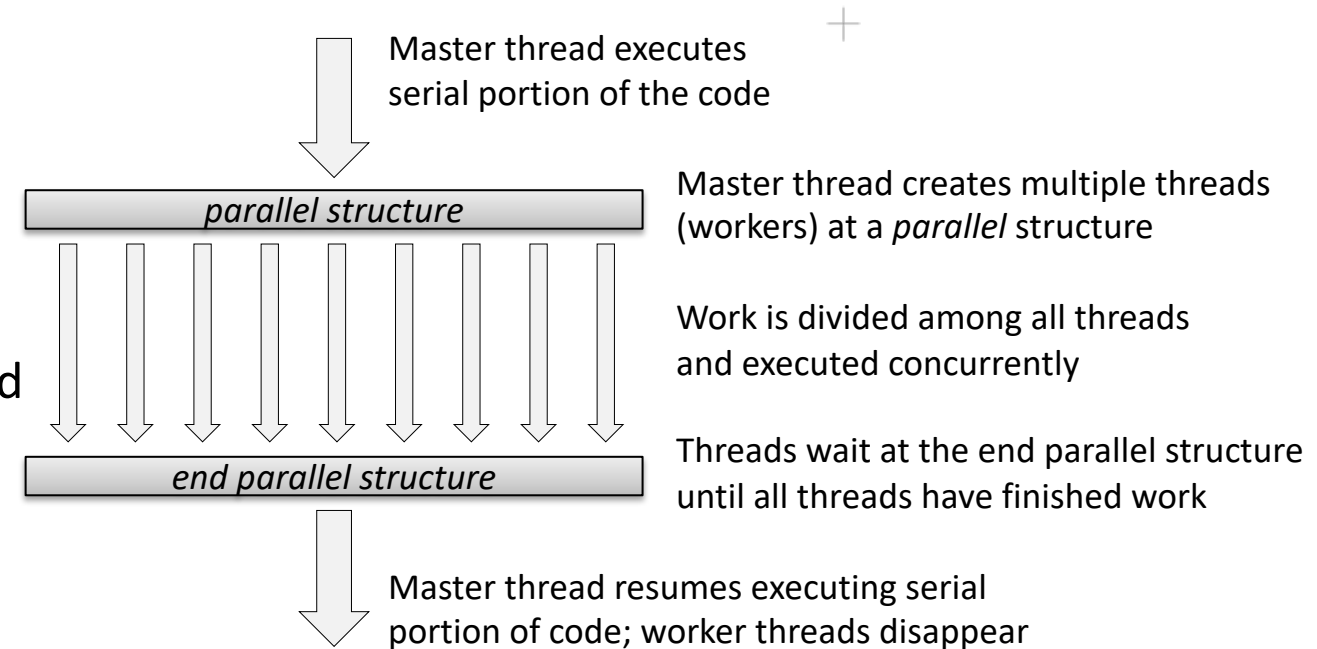
Caveats

- User-directed parallelization
- Explicit directives to compiler and run-time system
- No check for dependencies, conflicts, deadlocks, race conditions, etc.



Programming Model

- Fork-join model: fork starts parallel region, join ends parallel region
- Master thread begins serial execution
- Master creates team of threads at PARALLEL directive
- PARALLEL directives define parallel region
- Statement function calls in parallel region executed in parallel



Components of OpenMP 4.0

- Directives
 - Parallel regions, Work sharing, Synchronization, Data sharing and Data scope attributes
- Environment variables
 - Number of threads, scheduling, dynamic thread adjustment, nested parallelism, number of nested active parallel regions, size of a thread's stack, wait policy, limit on number of threads
- Runtime environment
 - Number of threads, thread ID, number of processors, scheduling, dynamic thread adjustment, nested parallelism, number of levels, active level, team size, thread's ancestor, timer routines, routines to manipulate locks

OpenMP Directive Format

Format

sentinel directive_name [clause[[,] clause] ...]

+

C/C++ Example:

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

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

Fortran Example:

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

Sample Program

```
main () {
    int var1, var2, var3;
    Serial code . . .
    #pragma omp parallel
    {
        int np = omp_get_num_threads();
        int myid = omp_get_thread_num();
        if (myid == 0)
            printf("I am master of %d threads\n", np);
        else
            printf("I am thread # %d\n", myid );
    }
    Serial code . . .
}
```

OpenMP Directives

- Parallel region
- Worksharing constructs
 - for, sections, single, workshare
- Synchronization
 - master, barrier, critical, ordered, atomic, flush
- SIMD
- Device
 - target, teams, distribute
- Task
 - Task, taskyield, taskwait, taskgroup

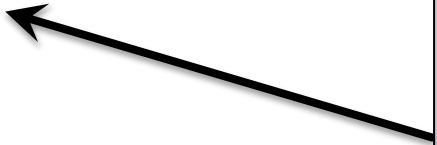


Parallel Region

- Parallel region executed concurrently by existing threads
- Number of threads set by environment variables and/or library calls
- Threads are numbered consecutively from 0 (master) onwards
- Number of processors hosting threads is implementation dependent
- Cannot branch into or out of structured block
- Implied BARRIER at the end of PARALLEL section
- Nested PARALLEL regions are allowed, but are serialized by default

+

```
#pragma omp parallel default(none) shared(x,n)
{
    iam = omp_get_thread_num();
    np = omp_get_num_threads();
    ipoints = n/np;
    subdomain(x, iam, ipoints);
}
```



```
IF(scalar-expression)
NUM_THREADS(integer-expression)
DEFAULT(shared|none)
PRIVATE(list)
FIRSTPRIVATE(list)
SHARED(list)
COPYIN(list)
REDUCTION(operator: list)
PROC_BIND(master|close|spread)
```

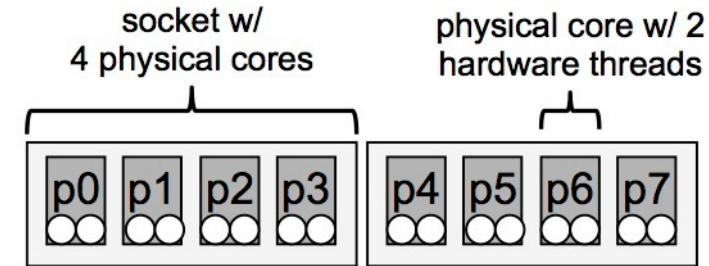

Thread Affinity: proc bind

PROC_BIND Clause

- Binds OpenMP threads to processor cores
- Three types of bindings are supported
 - master: assign every thread in the team to the same place as the master thread
 - close: assign threads to places closest to the place of the parent thread in order of thread ids; threads in the team execute on places that are consecutive starting from the parent's position, with wrap around
 - spread: assign threads to places such that a sparse distribution of the team is achieved

Example

- Consider mapping of 8 threads – T0 (master), T1, ... , T3; assume master T0 is mapped to p2




- master – all threads are mapped to p2
- close – T0 on p2; T1 on p3; T2 on p4; T3 on p5
- spread – T0 on p2; T1 on p4; T2 on p6; T3 on p0

Work Sharing Construct: for

- Specifies that for loop iterations must be executed in parallel
- Iterations are distributed across threads that already exist
- SCHEDULE: specifies division of iterations among threads
 - STATIC, DYNAMIC, GUIDED, AUTO, RUNTIME
- NOWAIT: threads continue without synchronizing at the end of the parallel loop

```
#pragma omp parallel
{
  #pragma omp for
  for (i=2; i<= n; i++)
    b[i]=(a[i]+a[i-1])/2.0;
  #pragma omp for nowait
  for (i=1; i<= m; i++) y(i)=sqrt(z[i]);
}
```

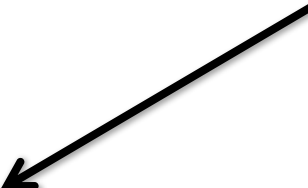


```
PRIVATE(list)
FIRSTPRIVATE(list)
LASTPRIVATE(list)
REDUCTION(operator: list)
SCHEDULE(kind[, chunk_size])
COLLAPSE(n)
ORDERED
NOWAIT
```

Work Sharing Construct: for

- COLLAPSE specifies the number of nested loops that must be parallelized; iterations of the nested loops are collapsed into one larger iteration space (iteration counts are computed **before** entry to the outermost loop)

Only k and j loops are collapsed and parallelized




```
#pragma omp for collapse(2) private(i, k, j)
  for (k=1; k<=2; k=k+1)
    for (j=10; j<=20; j=j+2)
      for (i=100; i<=200; i=i+5)
        bar(a,i,j,k);
```

Work Sharing Construct: sections

- Non-iterative work-sharing construct that specifies that the enclosed sections of code are to be divided among threads in the team
- Each section is executed once by a thread in the team
- Threads that complete execution of their sections wait at a barrier at the end unless a `nowait` is specified
- In the following example, XAXIS, YAXIS, and ZAXIS can be executed concurrently. All SECTION directives must be within the lexical extent of the SECTIONS construct.

```
#pragma omp parallel sections
{
  #pragma omp section
    xaxis();
  #pragma omp section
    yaxis();
  #pragma omp section
    zaxis();
}
```



```
PRIVATE(list)
REDUCTION(operator:list)
NOWAIT
```

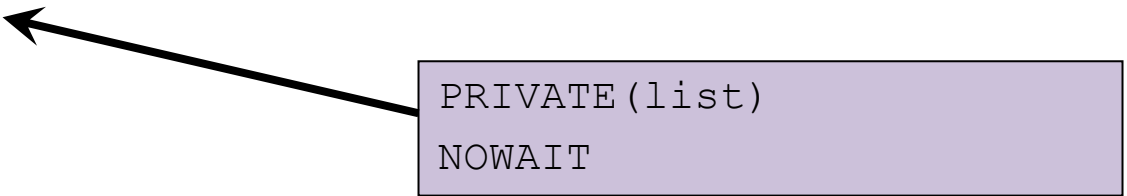
Parallel regions with only one work-sharing construct can be specified by shortcuts:

```
#pragma omp parallel sections
#pragma omp parallel for
```

Work Sharing Construct: single

- Specifies that the enclosed code is to be executed by only one thread
- Threads in the team that are not executing the SINGLE directive wait at the end unless `nowait` is specified
- In the following example, the first thread that encounters the SINGLE directive executes OUTPUT and INPUT. The user must not make any assumptions as to which thread will execute the SINGLE section. All other threads will skip the SINGLE section and stop at the barrier at the end of the SINGLE construct.

```
#pragma omp parallel default(shared)
{
    work(x);
    #pragma omp single
    {
        output(x);
        input(y);
    }
    work(y);
}
```



PRIVATE(list)
NOWAIT

Synchronization Constructs: master, barrier, ordered

MASTER directive

- Code enclosed in the master section is executed by the master thread
- Other threads in the team skip the section and continue execution

+

BARRIER directive

- Synchronizes all the threads in a team
- When encountered, each thread waits until all of the others threads in that team have reached this point

ORDERED directive

- Code in the ordered section is executed in the order in which iterations would be executed in a sequential execution of the loop
- Exactly one thread is allowed in an ordered section at a time
- No thread can enter an ordered section until it is guaranteed that all previous iterations have completed or will never execute an ordered section

Synchronization Constructs: critical

CRITICAL directive

- Restricts access to only one thread at a time
- A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section with the same name
- The example illustrates a queuing model in which a task is dequeued and worked on. A critical section is used to guard against multiple threads dequeuing the same task. Independent queues are protected by CRITICAL directives with different names, x and y.

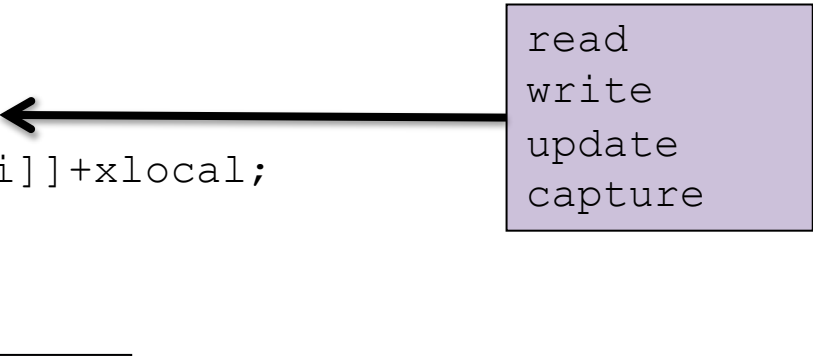
```
#pragma omp parallel default(private) shared(x,y)
{
    #pragma omp critical xaxis
    ix = dequeue(x);
    work(ix);
    #pragma omp critical yaxis
    iy = dequeue(y);
    work(iy);
}
```

Synchronization Constructs: atomic

ATOMIC directive

- Atomic directive ensures that a specific memory location is to be updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads
- This directive *applies only to the immediately following statement*
- Load and store of x are atomic; evaluation of expr is not atomic
- To avoid race conditions, all parallel updates of the location must be protected with the ATOMIC directive.

```
#pragma omp parallel for default(private) shared(x,y,index,n)
  for (i=1; i<= n; i++) {
    work(xlocal,ylocal);
    #pragma omp atomic update
    x[index[i]] = x[index[i]]+xlocal;
    y[i] = y[i] + ylocal;
  }
```



In this example, `y` is not updated atomically!

Synchronization Constructs: flush

FLUSH directive

- Executes the OpenMP flush operation, which makes a thread's temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables
- Expensive; use the optional *list* to flush specific variables
- The FLUSH directive is implied for: barrier; parallel, critical, ordered (upon entry & exit); for, sections, single (upon exit only)
- The directive is not implied if a NOWAIT clause is present
- Example: point-to-point synchronization between a pair of threads

```
#pragma omp parallel default(private) shared(isync)
    iam = omp_get_thread_num();
    isync[iam] = 0;
#pragma omp barrier
    isync[iam] = 1;
#pragma omp flush isync
    while (isync[neighbor] == 0) {
#pragma omp flush isync
    }
```

Flush updates the memory with the new value of ISYNC[IAM]

Flush needed to access the latest value of ISYNC[NEIGH] that resides in the memory.

Tasks

- The task construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

```
struct node {
    struct node *left;
    struct node *right;
}
extern void process(struct node *);
void traverse( struct node *p ) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

IF(*scalar-expression*)
FINAL(*scalar-expression*)
UNTIED
DEFAULT(shared|none)
MERGEABLE
PRIVATE(*list*)
FIRSTPRIVATE(*list*)
SHARED(*list*)

Data Sharing Attribute Clauses

- `default(shared|none)`
Controls the default data-sharing attributes of variables that are referenced in a parallel or task construct.
- `shared(list)`
Declares one or more list items to be shared by tasks generated by a parallel or task construct.
- `private(list)`
Declares one or more list items to be private to a task.
- `firstprivate(list)`
Declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.
- `lastprivate(list)`
Declares one or more list items to be private to an implicit task, and causes the corresponding original item to be updated after the end of the region.
- `reduction(operator:list)`
Declares accumulation into the list items using the indicated associative operator. Accumulation occurs into a private copy for each list item which is then combined with the original item.

Data Sharing Attribute Clauses: shared, private

```
int x[], y[], index[];
int i, j, k, m, n;
i=-1; k=1; j=-1;
#pragma omp parallel for \
default(private) \
shared(x,y,index,n) \
private(j) \
firstprivate(k) \
lastprivate(m)
for (i=0; i<= n; i++) {
    xlocal = i*i; ylocal = 2*i+1;
    x[index[i]] += xlocal;
    y[i] += ylocal;
    j = 2*i; m = j-1;
}
What are the values of i,j,k,m at this point?
```

Threadprivate Directive

- The threadprivate directive specifies that variables are replicated, with each thread having its own copy.

+

```
#pragma omp threadprivate(list)
```

Data Sharing Attribute Clauses: reduction

REDUCTION clause

- A private copy of the reduction variable is created and initialized for each thread +
- At the end of REDUCTION, the shared variable is updated with result of combining the original value of the (shared) reduction variable with the final value of all the private copies
- The reduction operators are all associative
- Reduction variables must be SHARED

```
#pragma omp parallel for default(private) reduction(+:A,B)
for (i=1; i<= n; i++){
    work(alocal,blocal);
    A = A + alocal;
    B = B + blocal;
}
```

Syntax

REDUCTION({operator|intrinsic}:list)

operator: arithmetic(+, *, -,/) or logical (and,or,...)

intrinsic: MAX, MIN, IAND, IOR, or IEO

list: list of variables to perform reduction

Data Copying Clauses

- `copyin(list)`

Copies the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the parallel region.

- `copyprivate(list)`

Broadcasts a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the parallel region.

```
#pragma omp threadprivate(work,size,tol)
void build() {
    int i;
    work = (float*)malloc( sizeof(float)*size );
    for( i = 0; i < size; ++i ) work[i] = tol;
}
void example() {
    #pragma omp single copyprivate(tol,size)
    {
        scanf("%f %d", &tol, &size);
    }
    #pragma omp parallel copyin(tol,size)
    {
        build();
    }
}
```

Directive Binding

- The FOR, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL directive, if one exists. +
- The ORDERED directive binds to the dynamically enclosing FOR directive.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL.

Directive Nesting

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled. +
- FOR, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive are not allowed to be nested one inside the other.
- FOR, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of FOR, SECTIONS, SINGLE, MASTER, and CRITICAL directives.
- MASTER directives are not permitted in the dynamic extent of FOR, SECTIONS, and SINGLE directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region.

SIMD Constructs

- Compilers may not vectorize loops when they are complex or possibly have dependencies, even though the programmer is certain the loop will execute correctly as a vectorized loop. +
- SIMD construct is used to declare that a loop can be transformed into an SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

```
void star( double *a, double *b, double *c, int n, int *ioff )
{
    int i;
    #pragma omp simd
    for ( i = 0; i < n; i++ )
        a[i] *= b[i] * c[i+ *ioff];
}
```

- DECLARE SIMD construct is used to declare a function is SIMD

Target devices

- OpenMP allows execution of instructions on target devices on the machine that are different from the host device
- Target constructs are:
 - declare target: specify device for mapping data and code
 - target data: create data environment on a device; map clause used to specify device action on data
 - target: create data environment and execute code on a device
 - target update: update data on device or host
 - teams: construct a league of threads and a master thread
 - distribute: execute target region by a team

+

Environment Variables

- `OMP_SCHEDULE` *type[,chunk]*
Runtime schedule type and chunk size; *type* is static, dynamic, guided, or auto; *chunk* is a positive integer that specifies chunk size.
- `OMP_NUM_THREADS` *list*
Number of threads to use for parallel regions.
- `OMP_DYNAMIC` *dynamic*
Enable or disable dynamic adjustment of threads to use for parallel regions; *dynamic* is true or false.
- `OMP_NESTED` *nested*
Enable or to disable nested parallelism; *nested* is true or false.
- `OMP_STACKSIZE` *size[B | K | M | G]*
Size of the stack for threads
- `OMP_WAIT_POLICY` *policy*
Controls the behavior of waiting threads; *policy* is ACTIVE (waiting threads consume processor cycles while waiting) or PASSIVE.
- `OMP_MAX_ACTIVE_LEVELS` *levels*
Maximum number of nested active parallel regions.
- `OMP_THREAD_LIMIT` *limit*
Maximum number of threads participating in the OpenMP program.

Environment Variables ...

- OMP_DEFAULT_DEVICE *device*
Default device number to be used for device constructs
- OMP_PLACES *places*
Set the place-partition variable that defines the OpenMP places available to the execution environment; places is an abstract name (threads, cores, sockets, or implementation-defined), or a list of non-negative numbers
- OMP_PROC_BIND *policy*
Set the thread affinity policy to be used for parallel regions at the corresponding nested level. *policy* can be the values true, false, or a comma-separated list of master, close, or spread in quotes.

+

Run-time Library Routines: Execution Environment

- *omp_set_num_threads*: sets the number of threads for parallel regions that do not specify a num_threads clause. +
- *omp_get_num_threads*: returns the number of threads in the current team.
- *omp_get_max_threads*: returns maximum number of threads that could be used to form a new team using a parallel construct without a num_threads clause.
- *omp_get_thread_num*: returns thread id
- *omp_get_num_procs*: returns the number of processors available to the program.
- *omp_in_parallel*: returns true if the call to the routine is enclosed by an active parallel region; otherwise, it returns false
- *omp_set_dynamic*: enables or disables dynamic adjustment of the number of threads
- *omp_get_dynamic*: determine if dynamic adjustment of the number of threads is enabled or not

Run-time Library Routines: Execution Environment ...

- *omp_set_nested*: enables or disables nested parallelism
- *omp_get_nested*: determine if nested parallelism is enabled or not
- *omp_set_schedule*: set the schedule that will be used: *kind* may be static, dynamic, guided, auto, or an implementation-defined schedule
- *omp_get_schedule*: determine the *kind* of schedule to be used.
- *omp_get_thread_limit*: maximum number of threads available to the program
- *omp_set_max_active_levels*: limits the number of nested active parallel regions
- *omp_get_max_active_levels*: determines the maximum number of nested active parallel regions
- *omp_get_level*: number of nested parallel regions enclosing the task that contains the call
- *omp_get_ancestor_thread_num(int level)*: for a given nested level of the current thread, the thread number of the ancestor or the current thread

+

Run-time Library Routines: Execution Environment ...

- *omp_get_team_size*: for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs
- *omp_get_active_level*: number of nested, active parallel regions enclosing the task
- *omp_in_final*: returns true if the routine is executed in a final or included task region; otherwise, it returns false
- *omp_get_proc_bind*: returns the thread affinity policy
- *omp_set_default_device*
- *omp_get_default_device*
- *omp_get_num_devices*: returns the number of target devices
- *omp_get_num_teams*: returns the number of teams in the current teams region
- *omp_get_team_num*: returns the team number of calling thread
- *omp_is_initial_device*: returns *true* if the current task is executing on the host device; otherwise, it returns *false*

+

Examples

Specify fixed number of threads within the program

```
omp_set_dynamic(0);
omp_set_num_threads(16);
#pragma parallel default(private) shared(x,np)
{
    iam=omp_get_thread_num();
    ipoints = np/16;
    do_by_16(x,iam,ipoints);
}
```

Determine number of threads

```
np=omp_get_num_threads();
#pragma parallel for
    for (i=0; i<np; i++) work(i);
#pragma parallel private(i)
{
    np=omp_get_num_threads();
    i=omp_get_thread_num();
    work(i);
}
```

Incorrect! np is always equal to 1 when `omp_get_num_threads()` is called from a serial region. It must be called from inside a parallel region

Correct! np is initialized to the number of threads in the team.

Locks

- Locks are special variables used for synchronization.
- Must be accessed only through lock routines
- Can be in one of the following states:
 - Uninitialized: need to initialize a lock before it can be used
 - Unlocked: a task can set the lock, which changes its state to locked; the task that sets the lock acquires ownership of the lock
 - Locked: a task that owns a lock can unset that lock, returning it to the unlocked state
- Locks are of two types:
 - Simple lock: cannot be set if it already owned by a task
 - Nested locks: can be set multiple times by the same task before being unset
- Lock routines access a lock variable in such a way that they always read and update the most current value of the lock variable. Explicit **flush** directives is not necessary to ensure that the lock variable's value is consistent among different tasks.

+

Run-time Library Routines: Lock Routines

Simple Lock Routines

- `void omp_init_lock(omp_lock_t *lock);`
Initializes a lock associated with lock variable *var* for use in subsequent calls
- `void omp_destroy_lock(omp_lock_t *lock);`
Disassociates the given lock variable *var* from any lock; uninitialized the lock
- `void omp_set_lock(omp_lock_t *lock);`
Forces the executing thread to wait until the specified lock is available; the thread is granted ownership of the lock when it is available;
- `void omp_unset_lock(omp_lock_t *lock);`
Releases the executing thread from ownership of the lock

- `int omp_test_lock(omp_lock_t *lock);`
Attempt to set a lock but do not suspend execution of the task executing the routine. Returns non-zero if the lock was set successfully, otherwise it returns zero; allows thread to test and set the lock as an atomic operation

Nested Lock Routines


- `void omp_init_nest_lock(omp_nest_lock_t *lock);`
- `void omp_destroy_nest_lock(omp_nest_lock_t *lock);`
- `void omp_set_nest_lock(omp_nest_lock_t *lock);`
- `void omp_unset_nest_lock(omp_nest_lock_t *lock);`
- `int omp_test_nest_lock(omp_nest_lock_t *lock);`

Example: Using Simple Locks

```
#include <omp.h>
omp_lock_t *lck;
omp_init_lock(lck);
#pragma parallel shared(lck) private(id)
{
    omp_set_lock(lck);
    /* We have lock, do work ... */
    omp_unset_lock(lck);

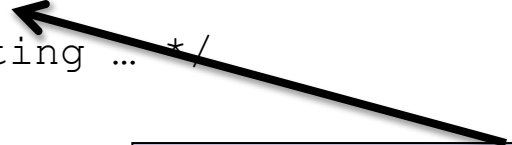
    while (! omp_test_lock(lck)) {
        /* Do useful work while waiting ... */
    }
    /* We have lock, do work ... */
    omp_unset_lock(lck);
}
omp_destroy_lock(lck);
```

Must include `omp.h` and define
locks of type `omp_lock_t`



`OMP_SET_LOCK` and
`OMP_TEST_LOCK`
allow thread to test
and set the lock as an
atomic operation

`OMP_TEST_LOCK` allows thread to do
useful work while waiting for lock to be
released; however, the thread is not
guaranteed a place in the queue waiting
for the lock to be released



Example: Deadlocks

- BARRIER directive inside a for, single, or critical can result in deadlock

+

```
#pragma parallel for default(shared)
  for (i=0; i < N; i++) {
    work(i);
    #pragma barrier
  }
#pragma parallel default(shared)
# pragma critical
  work(N);
  #pragma barrier
  more_work(N);
}
```

The diagram illustrates two instances of the `#pragma barrier` directive causing deadlocks. In the first instance, the barrier is placed inside a `for` loop. An arrow points from a box labeled "Causes deadlock!" to the `#pragma barrier` line. In the second instance, the barrier is placed inside a `critical` section. An arrow points from a box labeled "Causes deadlock!" to the `#pragma barrier` line.

Run-time Library Routines: Timing Routines

- *double omp_get_wtime(void);*

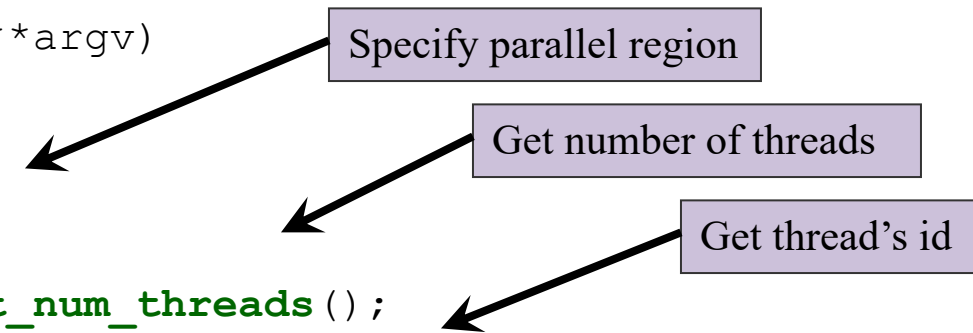
Returns elapsed wall clock time in seconds. Use in "pairs" with the value of the first call subtracted from the value of the second call to obtain the elapsed time for a block of code.

- *double omp_get_wtick(void);*

Returns the precision of the timer used by *omp_get_wtime*. Returns a double-precision floating point value equal to the number of seconds between successive clock ticks

Example: first.c

```
#include <stdio.h>
#include <string.h>
main(int argc, char **argv)
{
#pragma omp parallel
{
    int np = omp_get_num_threads();
    int myid = omp_get_thread_num();
    printf("Hello from process # %d out of %d \n", myid, np);
}
}
```



Number of threads are specified via
an environment variable
export OMP_NUM_THREADS=4

Grace

- General Information: <https://hprc.tamu.edu/resources/>
- User Guide: <https://hprc.tamu.edu/wiki/Grace>
- Batch submission system, including example batch submission job file:
<https://hprc.tamu.edu/wiki/Grace:Batch>
- Useful batch system commands
https://hprc.tamu.edu/wiki/Grace:Batch#Job_Monitoring_and_Control_Commands
 - Submit a job: `sbatch <script_file>`
 - Cancel a job: `scancel <job_id>`
 - Check status of a job: `squeue --job <job_id>`
 - Check status of all jobs of a user: `squeue -u <user_name>`
 - Check CPU and memory efficiency for a job: `seff <job_id>`

+

Grace: Usage

- Computing Environment: https://hprc.tamu.edu/wiki/Grace:Computing_Environment
 - Module setup commands (example)
- Compiling and running programs: <https://hprc.tamu.edu/wiki/Terra:Compile:All> (Link to Terra at present!)

- Compiling OpenMP-based C and C++ programs:

https://hprc.tamu.edu/wiki/Terra:Compile:All#OpenMP_Programs

```
icc -qopenmp -o first_omp.exe first_omp.c
```

```
icpc -qopenmp -o first_omp.exe first_omp.C
```

```
export OMP_NUM_THREADS=8
```

```
./first_omp.exe
```

Sample Output

```
Hello from process # 0 out of 8
```

```
Hello from process # 3 out of 8
```

```
Hello from process # 4 out of 8
```

```
Hello from process # 5 out of 8
```

```
Hello from process # 1 out of 8
```

```
Hello from process # 6 out of 8
```

```
Hello from process # 2 out of 8
```

```
Hello from process # 7 out of 8
```

Grace: OpenMP Thread Control via Environment Variables

OMP_NUM_THREADS	Number of threads to be used; can be changed by calling <code>omp_set_num_threads</code> from within the program
OMP_SCHEDULE	Schedule type for parallel do loops. Can be STATIC (default), DYNAMIC, GUIDED. Can override inside the program
OMP_DYNAMIC	Enable (=TRUE) or disable (=FALSE) dynamic adjustment of the number of threads.
OMP_STACKSIZE	Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread.
OMP_PLACES	Defines an ordered list of places where threads can execute. Every place is a set of hardware (HW) threads. Can be defined as an explicit list of places described by nonnegative numbers or an abstract name. Abstract name can be 'threads' (every place consists of exactly one hw thread), 'cores' (every place contains all the HW threads of the core), 'socket' (every places contains all the HW threads of the socket)
OMP_PROC_BIND	Sets the thread affinity policy to be used for parallel regions at the corresponding nesting level. Acceptable values are true, false, or a comma separated list, each element of which is one of the following values: master (all threads will be bound to same place as master thread), close (all threads will be bound to successive places close to place of master thread), spread (all threads will be distributed among the places evenly).
KMP_AFFINITY	<p>Maps OpenMP threads to physical threads that execute on specific sockets and cores on a node.</p> <ul style="list-style-type: none">• <code>export KMP_AFFINITY="verbose,scatter"</code> [Assigns consecutive (0, 1, ...) threads on alternating sockets. Works best when there is minimal sharing of data between threads.]• <code>export KMP_AFFINITY="verbose,compact,1"</code> [Assigns consecutive (0, 1, ...) threads on different physical cores on the same socket. Works best when there is significant sharing of data between threads.]

Grace: first.job

```
#!/bin/bash
#SBATCH --job-name=JobExample
#SBATCH --time=1:30:00
#SBATCH --mem=8G
#SBATCH --nodes=1
#SBATCH --ntasks-per-node=8
#SBATCH --output=output.%j
#
# <--- at this point the current working directory is the one you submitted the job from.
#
# First Executable Line
#
module load intel/2020a          # load Intel software stack
#
# Set number of OpenMP threads to match requested tasks/cores
export OMP_NUM_THREADS=8
./first_omp.exe
```

#Set the job name to "JobExample"

#Set the wall clock limit in hr:min:sec

#Request 8GB per node

#Request 1 node

#Request 8 tasks/cores per node

#Send stdout/err to "output.[jobID]"

Set wall clock limit and mem to the smallest possible values that are sufficient for your experiments.

Set nodes to 1 since OpenMP works only on the shared-memory of a single node. Set ntasks-per-node to the number of cores you want to use.