COMP 422, Lecture 7: Shared-Memory Parallel Programming with OpenMP (Section 7.10)

Vivek Sarkar

Department of Computer Science
Rice University

vsarkar@rice.edu

Recap of Lecture 6 (Advanced Cilk Features)

- **Inlet**
- **Abort**
- **Cilk_alloca**
- **SYNCHED**
    - **—Matrix Multiply example**
- **Cilk_lockvar**

Programming Assignment #1: Parallel Graph Coloring in Cilk

Program inputs (see http://www.cs.princeton.edu/~appel/graphdata/):

- Number of colors, K
- Adjacency list for each node. If there is an edge between nodes x and y, y will apear in x's adjacency list and vice versa
- List of move pairs. Your goal is to find a legal solutions in which as many move pairs as possible are "eliminated" i.e., are given the same color

Program outputs:

- Number of legal solutions
- Minimum number of move-pairs that still remain in best legal solution (if any)

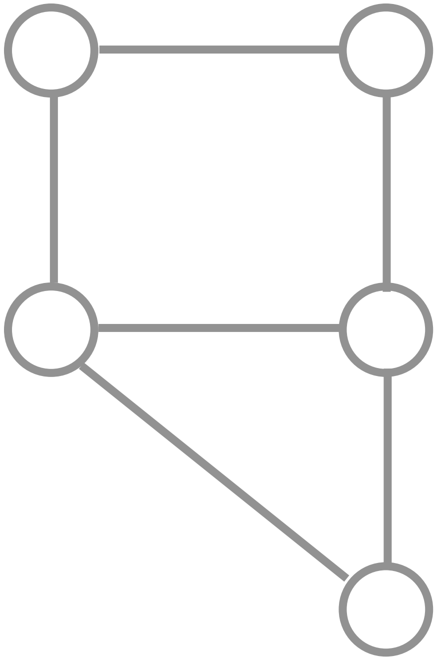
What you need to submit:

- Single source file containing your Cilk program.
- Write-up summarizing parallelizing approach used, and performance data for sample input file for 1 - 4 processors on an Ada node.

More details to follow

Example: Graph coloring

Given *k* colors, does there exist a coloring of the nodes such that adjacent nodes are assigned different colors



Source: http://ai.uwaterloo.ca/~vanbeek/Courses/Slides/introduction.ppt

Example: 3-coloring

variables:

v_1, v_2, v_3, v_4, v_5

domains:

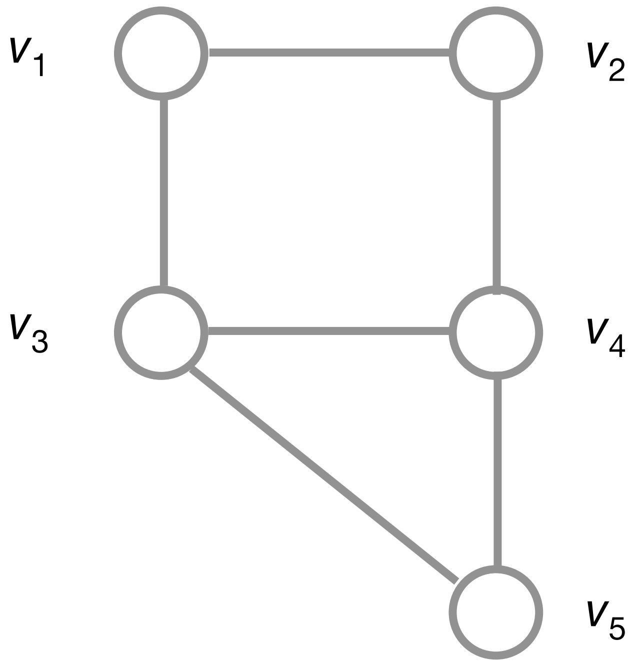
{1, 2, 3}

constraints:

$v_i \neq v_j$ *if* v_i *and* v_j *are adjacent*

move pairs:

$(v_1, v_4), (v_2, v_3)$



Source: http://ai.uwaterloo.ca/~vanbeek/Courses/Slides/introduction.ppt

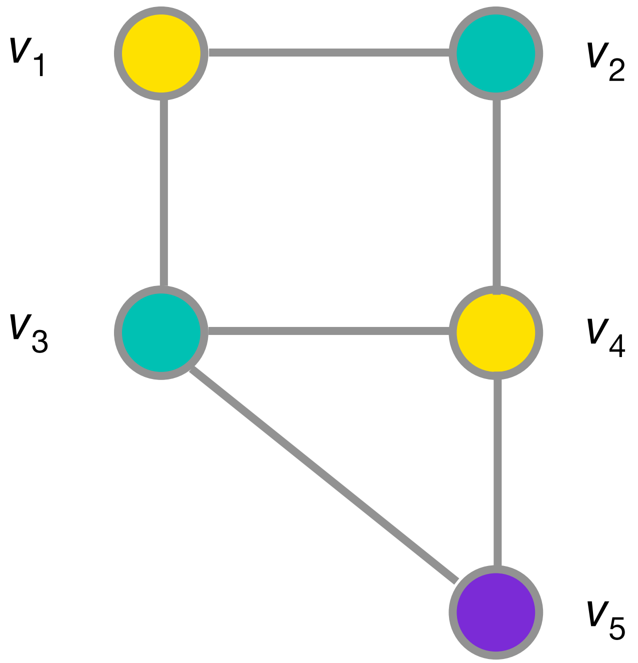
Example: 3-coloring

A solution

$v_1 \leftarrow 1$ ●
$v_2 \leftarrow 2$ ●
$v_3 \leftarrow 2$ ●
$v_4 \leftarrow 1$ ●
$v_5 \leftarrow 3$ ●

Note that both move pairs have been eliminated:

(v_1, v_4), (v_2, v_3)



Source: http://ai.uwaterloo.ca/~vanbeek/Courses/Slides/introduction.ppt

Acknowledgments for today’s lecture

- Slides from OpenMP tutorial given by Ruud van der Paas at HPCC 2007
  - http://www.tlc2.uh.edu/hpcc07/Schedule/OpenMP
- Slides accompanying course textbook
  - http://www-users.cs.umn.edu/~karypis/parbook/
- OpenMP 2.5 specification
  - http://www.openmp.org/mp-documents/spec25.pdf

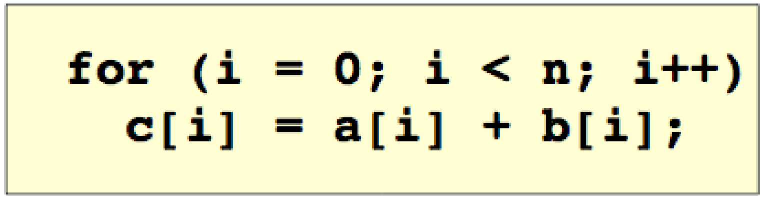
What is OpenMP?

- ❑ *De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran*
- ❑ *Consists of:*
  - *Compiler directives*
  - *Run time routines*
  - *Environment variables*
- ❑ *Specification maintained by the OpenMP Architecture Review Board* **(http://www.openmp.org)**
- ❑ *Latest Specification: Version 2.5*

- ❑ *Version 3.0 has been in the works since September 2007, final specification expected late 2007/early 2008*

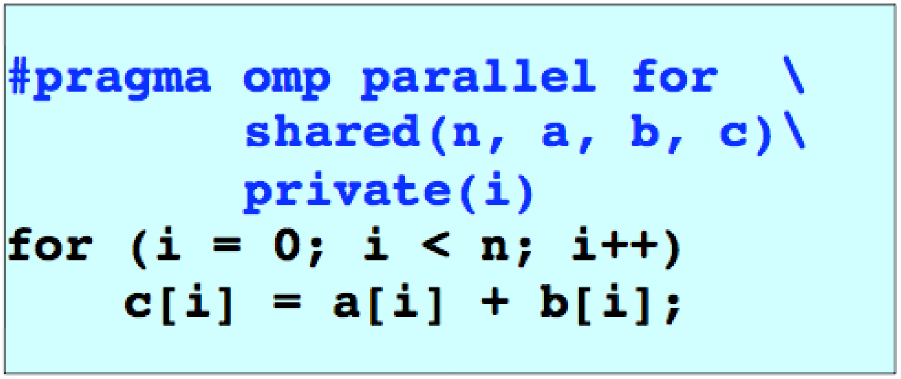
A first OpenMP example

For-loop with independent iterations



```
for (i = 0; i < n; i++)
  c[i] = a[i] + b[i];
```

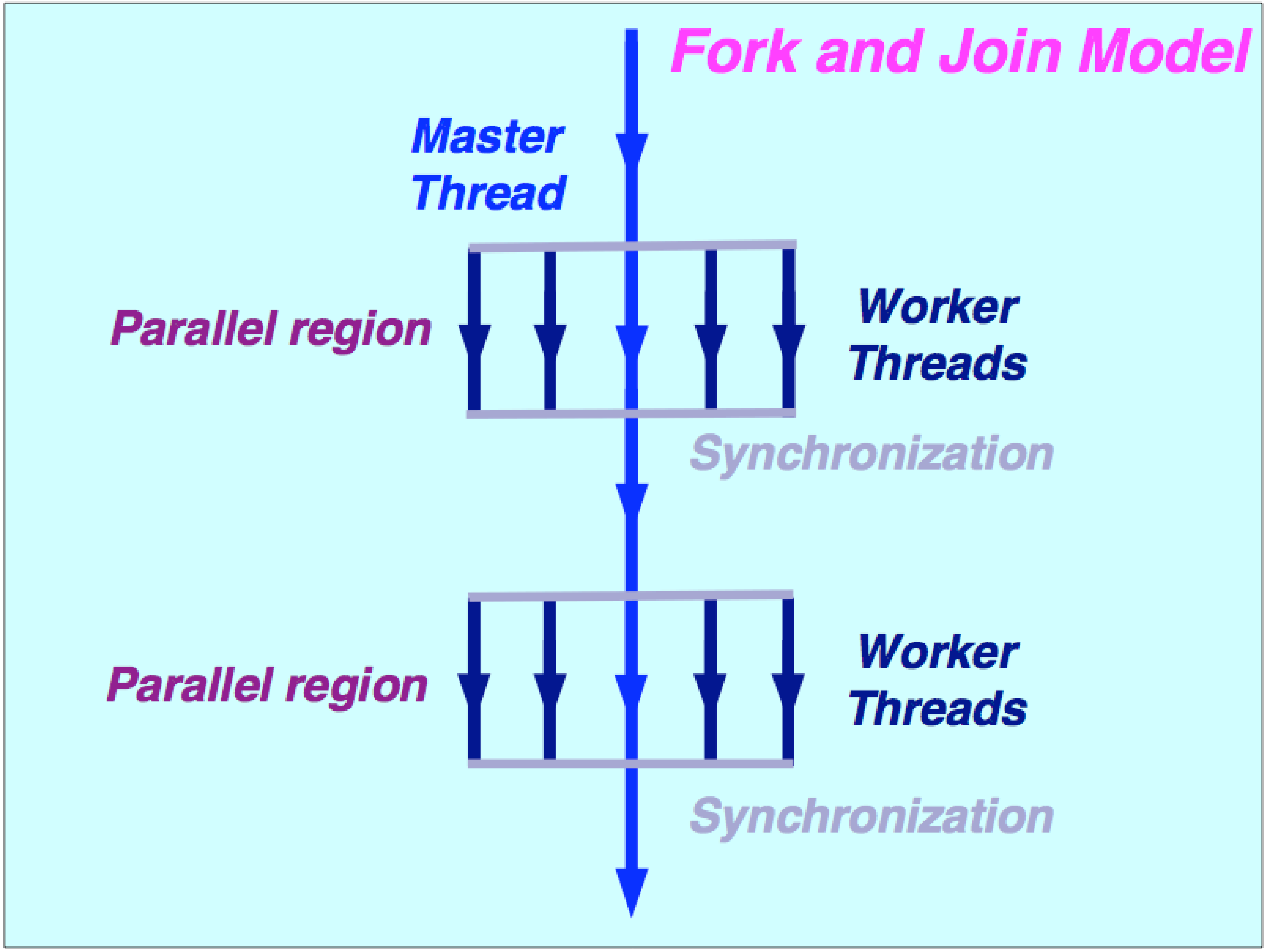
For-loop parallelized using an OpenMP pragma



```
#pragma omp parallel for   \
        shared(n, a, b, c)\
        private(i)
for (i = 0; i < n; i++)
    c[i] = a[i] + b[i];
```

```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 4
% a.out
```

The OpenMP Execution Model



Terminology

- *OpenMP Team := Master + Workers*
- *A Parallel Region is a block of code executed by all threads simultaneously*
  - *The master thread always has thread ID 0*
  - *Thread adjustment (if enabled) is only done before entering a parallel region*
  - *Parallel regions can be nested, but support for this is implementation dependent*
  - *An "if" clause can be used to guard the parallel region; in case the condition evaluates to "false", the code is executed serially*
- *A work-sharing construct divides the execution of the enclosed code region among the members of the team; in other words: they split the work*

Components of OpenMP

Directives

- *Parallel regions*
- *Work sharing*
- *Synchronization*
- *Data-sharing attributes*
  - ☞ *private*
  - ☞ *firstprivate*
  - ☞ *lastprivate*
  - ☞ *shared*
  - ☞ *reduction*
- *Orphaning*

Environment variables

- *Number of threads*
- *Scheduling type*
- *Dynamic thread adjustment*
- *Nested parallelism*

Runtime environment

- *Number of threads*
- *Thread ID*
- *Dynamic thread adjustment*
- *Nested parallelism*
- *Timers*
- *API for locking*

OpenMP directives and clauses

- ❑ *C: directives are case sensitive*
  - *Syntax:* #pragma omp directive [clause [clause] ...]
- ❑ *Continuation: use \ in pragma*
- ❑ *Conditional compilation:* _OPENMP *macro is set*

if (scalar expression)

- ✓ *Only execute in parallel if expression evaluates to true*
- ✓ *Otherwise, execute serially*

```
#pragma omp parallel if (n > threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

private (list)

- ✓ *No storage association with original object*
- ✓ *All references are to the local object*
- ✓ *Values are undefined on entry and exit*

shared (list)

- ✓ *Data is accessible by all threads in the team*
- ✓ *All threads access the same address space*

firstprivate (list)

- ✓ *All variables in the list are initialized with the value the original object had before entering the parallel construct*

lastprivate (list)

- ✓ *The thread that executes the <u>sequentially last</u> iteration or section updates the value of the objects in the list*

Parallel Region

```
#pragma omp parallel [clause[[,] clause] ...]
{
    "this is executed in parallel"

} (implied barrier)
```

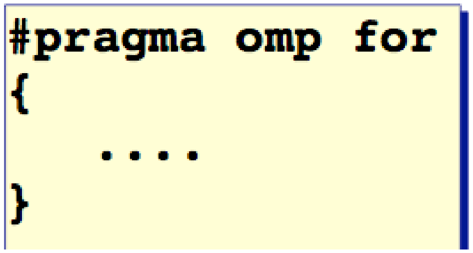
A parallel region is a block of code executed by multiple threads simultaneously, and supports the following clauses:

if	*(scalar expression)*	
private	*(list)*	
shared	*(list)*	
default	*(nonelshared)*	*(C/C++)*
default	*(nonelsharedlprivate)*	*(Fortran)*
reduction	*(operator: list)*	
copyin	*(list)*	
firstprivate	*(list)*	
num_threads	*(scalar_int_expr)*	

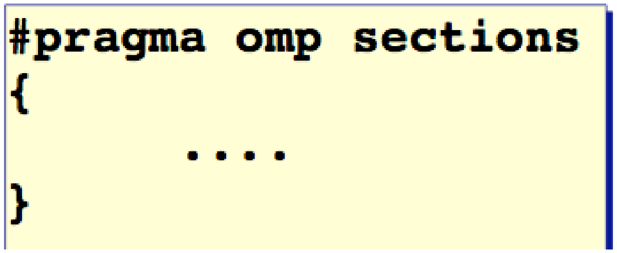
OpenMP Programming Model

- **The clause list is used to specify conditional parallelization, number of threads, and data handling.**
  - — Conditional Parallelization: **The clause `if (scalar expression)` determines whether the parallel construct results in creation of threads.**
  - — Degree of Concurrency: **The clause `num_threads(integer expression)` specifies the number of threads that are created.**
  - — Data Handling: **The clause `private (variable list)` indicates variables local to each thread. The clause `firstprivate (variable list)` is similar to the `private`, except values of variables are initialized to corresponding values before the parallel directive. The clause `shared (variable list)` indicates that variables are shared across all the threads.**

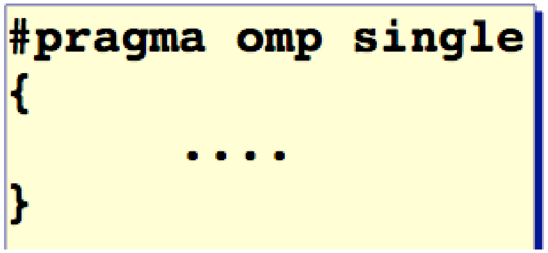
Work-sharing constructs in a Parallel Region



```
#pragma omp for
{
    ....
}
```

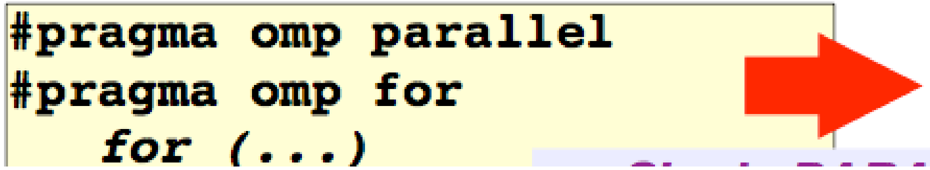


```
#pragma omp sections
{
        ....
}
```

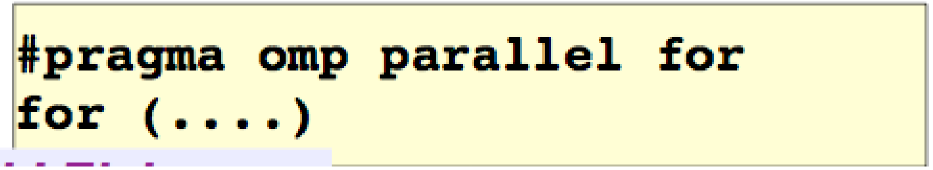


```
#pragma omp single
{
        ....
}
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless nowait is specified)
- A work-sharing construct does not launch any new threads
- Shorthand syntax supported for parallel region with single work-sharing construct e.g.,

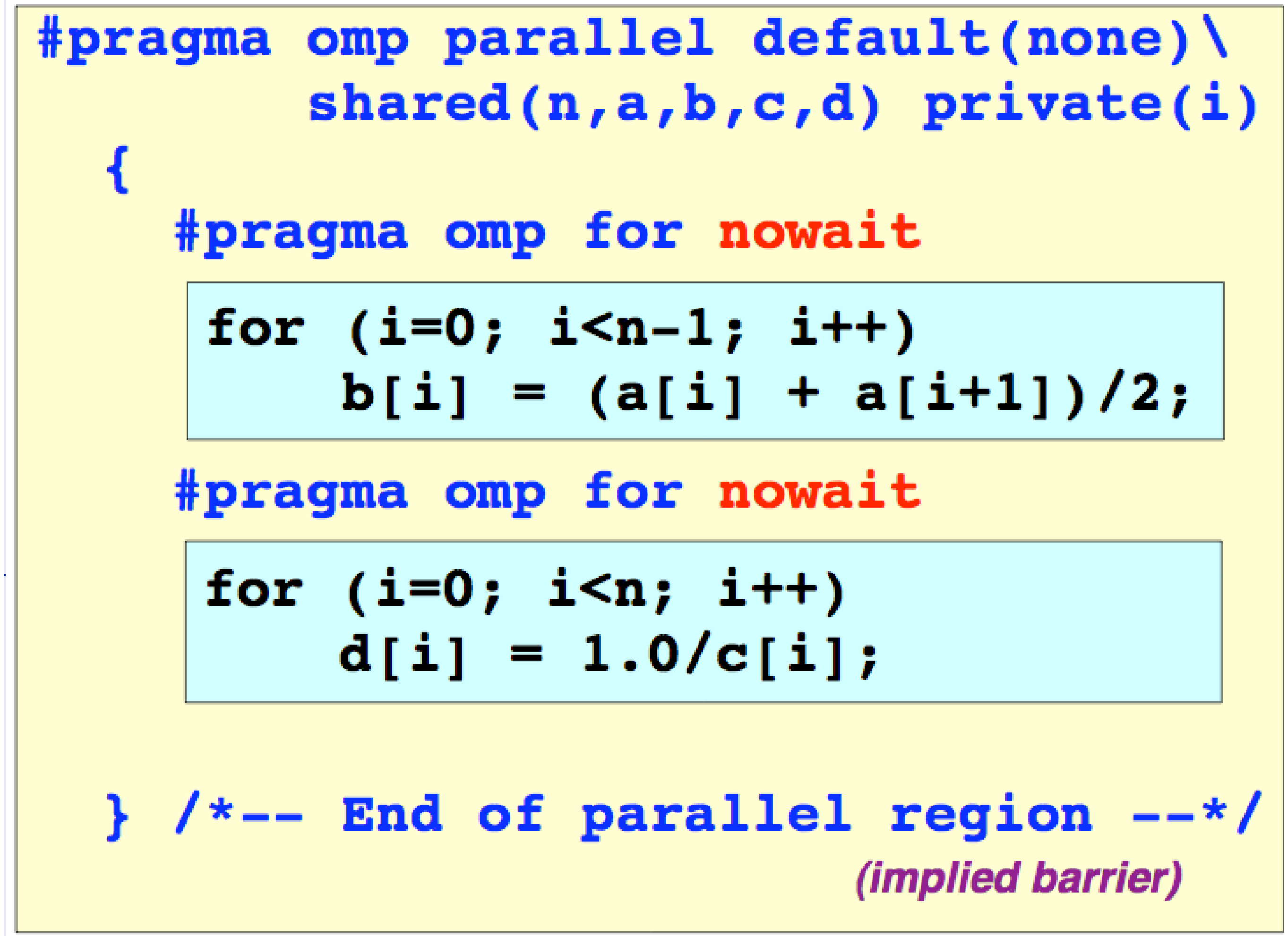


```
#pragma omp parallel
#pragma omp for
   for (...)
```



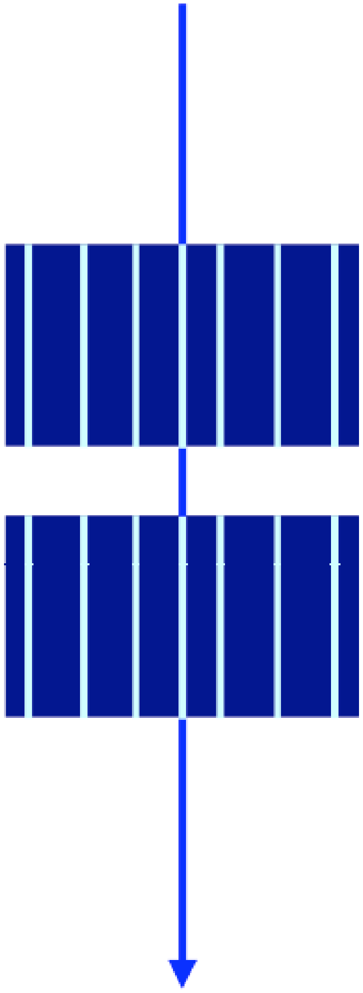
```
#pragma omp parallel for
for (....)
```

Example of work-sharing “omp for” loop



```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

  } /*-- End of parallel region --*/
```

(implied barrier)



Reduction Clause in OpenMP

- The `reduction` clause specifies how multiple local copies of a variable at different threads are combined into a single copy at the master when threads exit.
- The usage of the `reduction` clause is `reduction (operator: variable list)`.
- The variables in the list are implicitly specified as being private to threads.
- The `operator` can be one of `+, *, -, &, |, ^, &&,` and `||`.

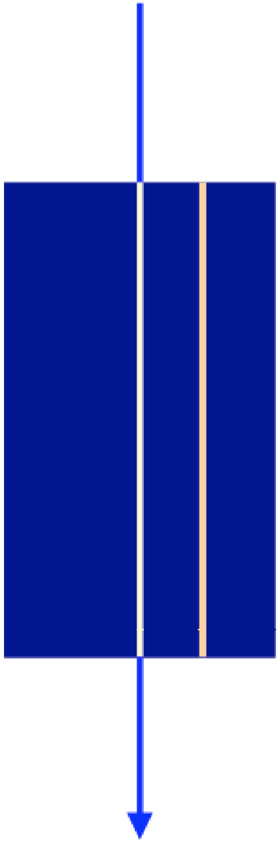
```
#pragma omp parallel reduction(+: sum) num_threads(8) {
/* compute local sums here */
}
/*sum here contains sum of all local instances of sums */
```

OpenMP Programming: Example

```
/* ******************************************************
An OpenMP version of a threaded program to compute PI.
****************************************************** */
#pragma omp parallel default(private) shared (npoints) \
    reduction(+: sum) num_threads(8)
{
    num_threads = omp_get_num_threads();
    sample_points_per_thread = npoints / num_threads;
    sum = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
       rand_no_x =(double)(rand_r(&seed))/(double)((2<<14)-1);
       rand_no_y =(double)(rand_r(&seed))/(double)((2<<14)-1);
       if (((rand_no_x - 0.5) * (rand_no_x - 0.5) +
           (rand_no_y - 0.5) * (rand_no_y - 0.5)) < 0.25)
           sum ++;
    }
}
```

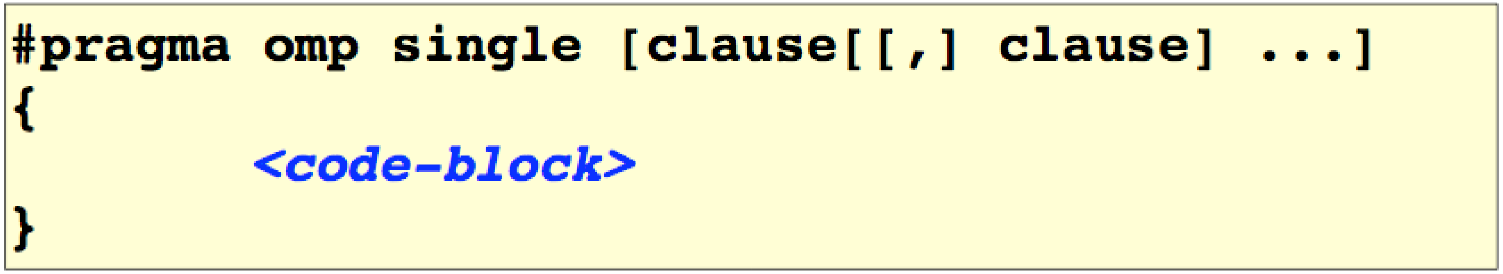
Example of work-sharing “sections”

```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
       for (i=0; i<n-1; i++)
           b[i] = (a[i] + a[i+1])/2;

      #pragma omp section
       for (i=0; i<n; i++)
           d[i] = 1.0/c[i];

    } /*-- End of sections --*/

  } /*-- End of parallel region --*/
```



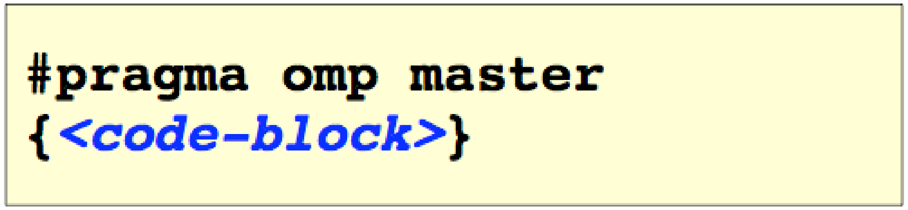
“single” and “master” constructs in a parallel region

Only one thread in the team executes the code enclosed



```
#pragma omp single [clause[[,] clause] ...]
{
        <code-block>
}
```

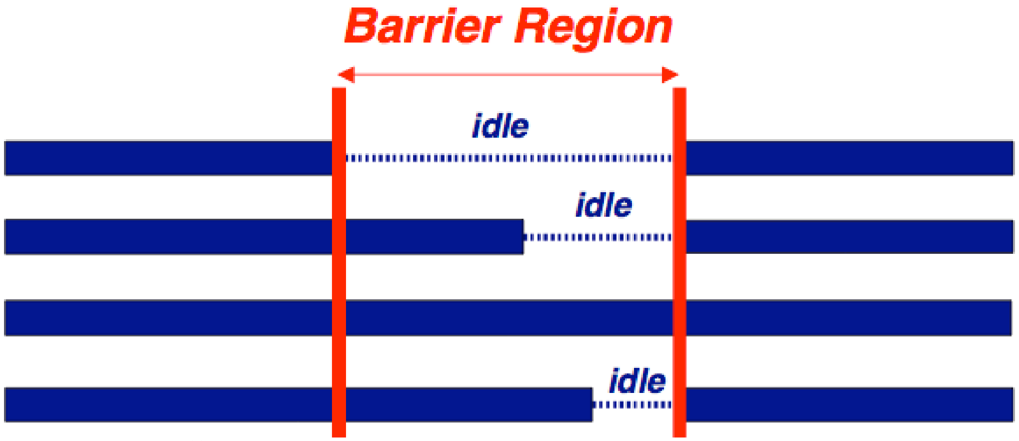
Only the master thread executes the code block



```
#pragma omp master
{<code-block>}
```

- Single and master are useful for computations that are intended for single-processor execution e.g., I/O and initializations
- There is no implied barrier on entry or exit of a single or master construct

Implicit barrier



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

Implicit barrier

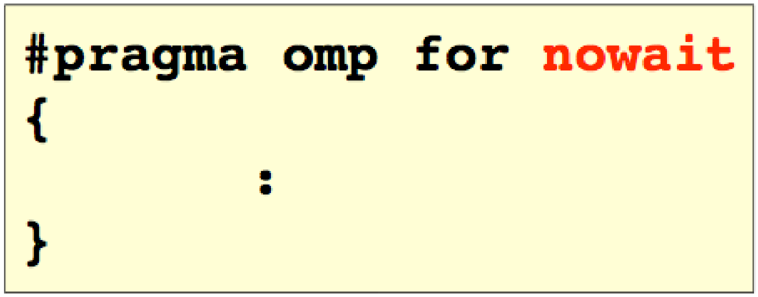
wait !

barrier

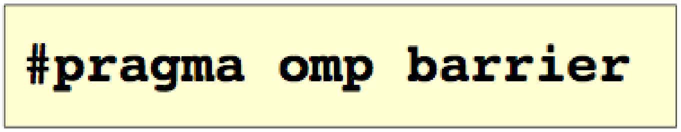
```
#pragma omp for
for (i=0; i < N; i++)
   d[i] = a[i] + b[i];
```

NOTE: barrier is redundant if there is a guarantee that the mapping of iterations onto threads is identical in both loops

nowait clause & explicit barrier



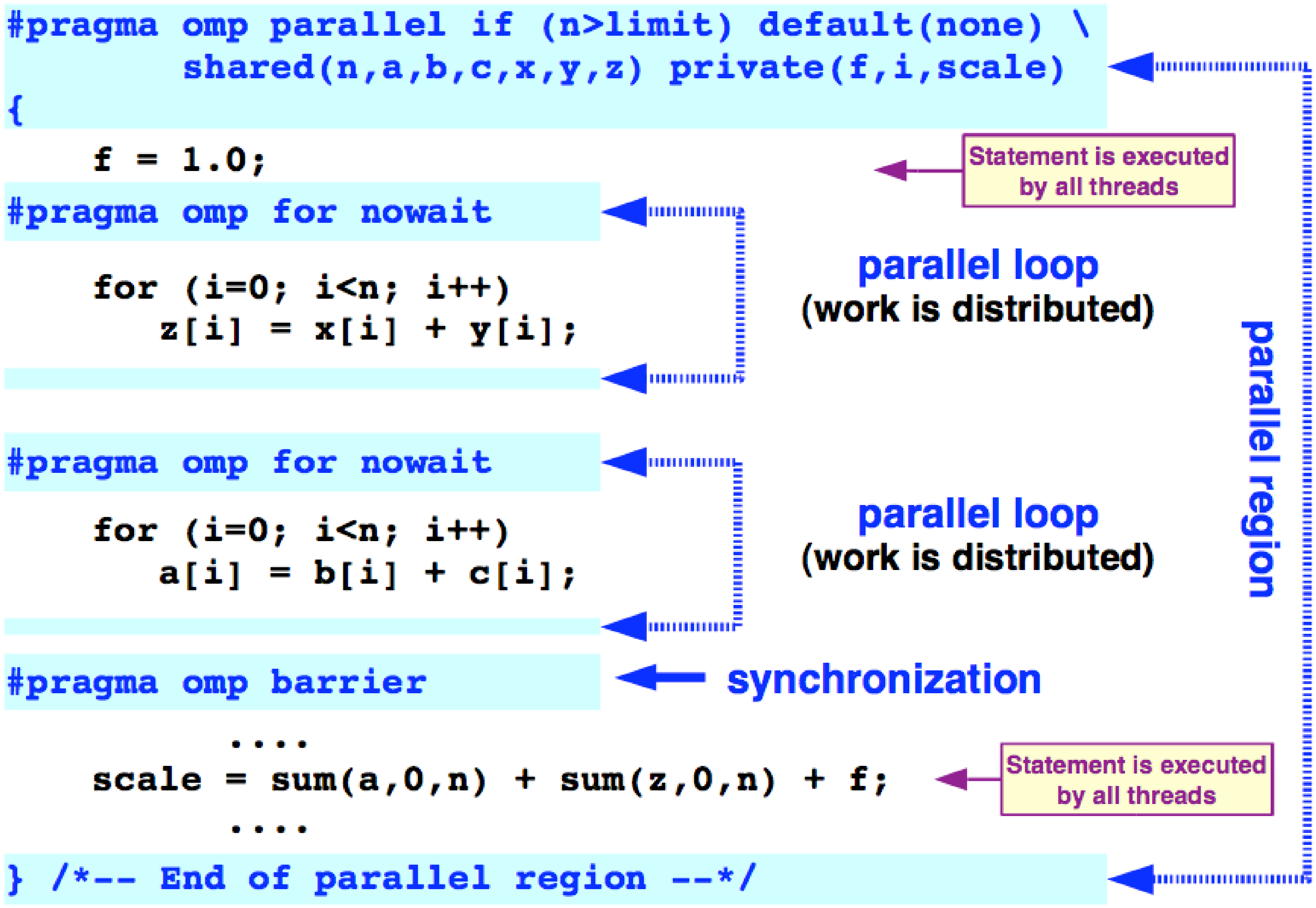
```
#pragma omp for nowait
{
        :
}
```



```
#pragma omp barrier
```

- To minimize synchronization, some OpenMP directives/pragmas support the optional *nowait* clause
- If present, threads do not synchronize/wait at the end of that particular construct
- An explicit barrier can then be inserted at only the desired program points

A more elaborate example



```
#pragma omp parallel if (n>limit) default(none) \
        shared(n,a,b,c,x,y,z) private(f,i,scale)
{
    f = 1.0;
#pragma omp for nowait

    for (i=0; i<n; i++)
       z[i] = x[i] + y[i];


#pragma omp for nowait

    for (i=0; i<n; i++)
       a[i] = b[i] + c[i];


#pragma omp barrier
              ....
    scale = sum(a,0,n) + sum(z,0,n) + f;
              ....
} /*-- End of parallel region --*/
```

Statement is executed by all threads

parallel loop
(work is distributed)

parallel loop
(work is distributed)

synchronization

Statement is executed by all threads

parallel region

schedule clause for parallel loops

schedule (static | dynamic | guided [, chunk])
schedule (runtime)

static [, chunk]

- ✓ *Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion*
- ✓ *In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads*

Example: *Loop of length 16, 4 threads:*

TID	0	1	2	3
no chunk	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

schedule clause for parallel loops (contd)

dynamic [, chunk]

- ✓ *Fixed portions of work; size is controlled by the value of chunk*
- ✓ *When a thread finishes, it starts on the next portion of work*

guided [, chunk]

- ✓ *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially*

runtime

- ✓ *Iteration scheduling scheme is set at runtime through environment variable OMP_SCHEDULE*

Assigning Iterations to Threads

- The `schedule` clause of the `for` directive deals with the assignment of iterations to threads.
- The general form of the `schedule` directive is

  `schedule(scheduling_class[, parameter]).`
- OpenMP supports four scheduling classes: `static`, `dynamic`, `guided`, and `runtime`.

Assigning Iterations to Threads: Example

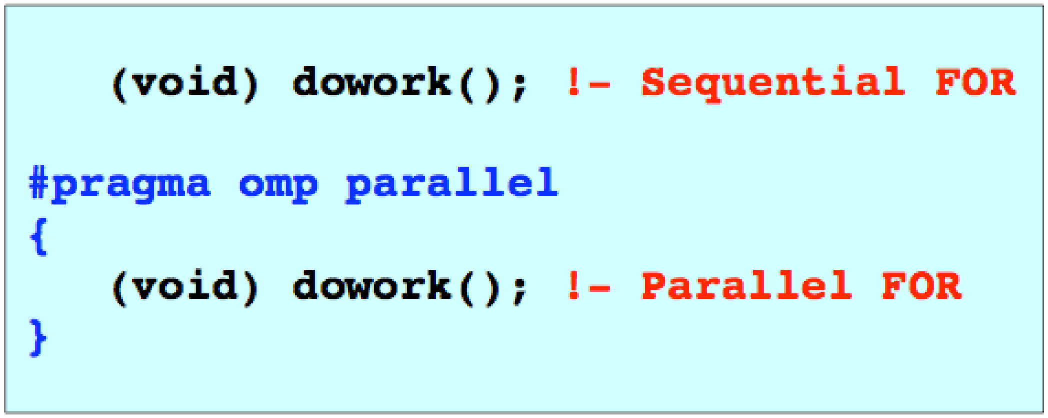
```
/* static scheduling of matrix multiplication loops */
#pragma omp parallel default(private) shared (a, b, c, dim) \
    num_threads(4)
    #pragma omp for schedule(static)
    for (i = 0; i < dim; i++) {
       for (j = 0; j < dim; j++) {
           c(i,j) = 0;
           for (k = 0; k < dim; k++) {
              c(i,j) += a(i, k) * b(k, j);
           }
       }
    }
```

Nesting `parallel` Directives

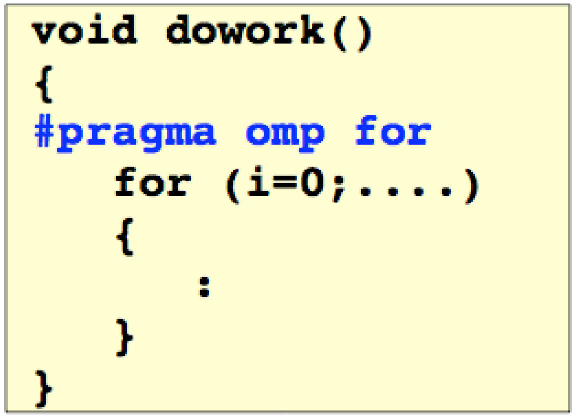
- Nested parallelism can be enabled using the `OMP_NESTED` environment variable.
- If the `OMP_NESTED` environment variable is set to `TRUE`, nested parallelism is enabled.
- In this case, each parallel directive creates a new team of threads.

Out-of-line (“orphaned”) directives

- *The OpenMP standard does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned*
- *That is, they can appear outside the lexical extent of a parallel region*



```
   (void) dowork(); !- Sequential FOR

#pragma omp parallel
{
   (void) dowork(); !- Parallel FOR
}
```



```
void dowork()
{
#pragma omp for
   for (i=0;....)
   {
      :
   }
}
```

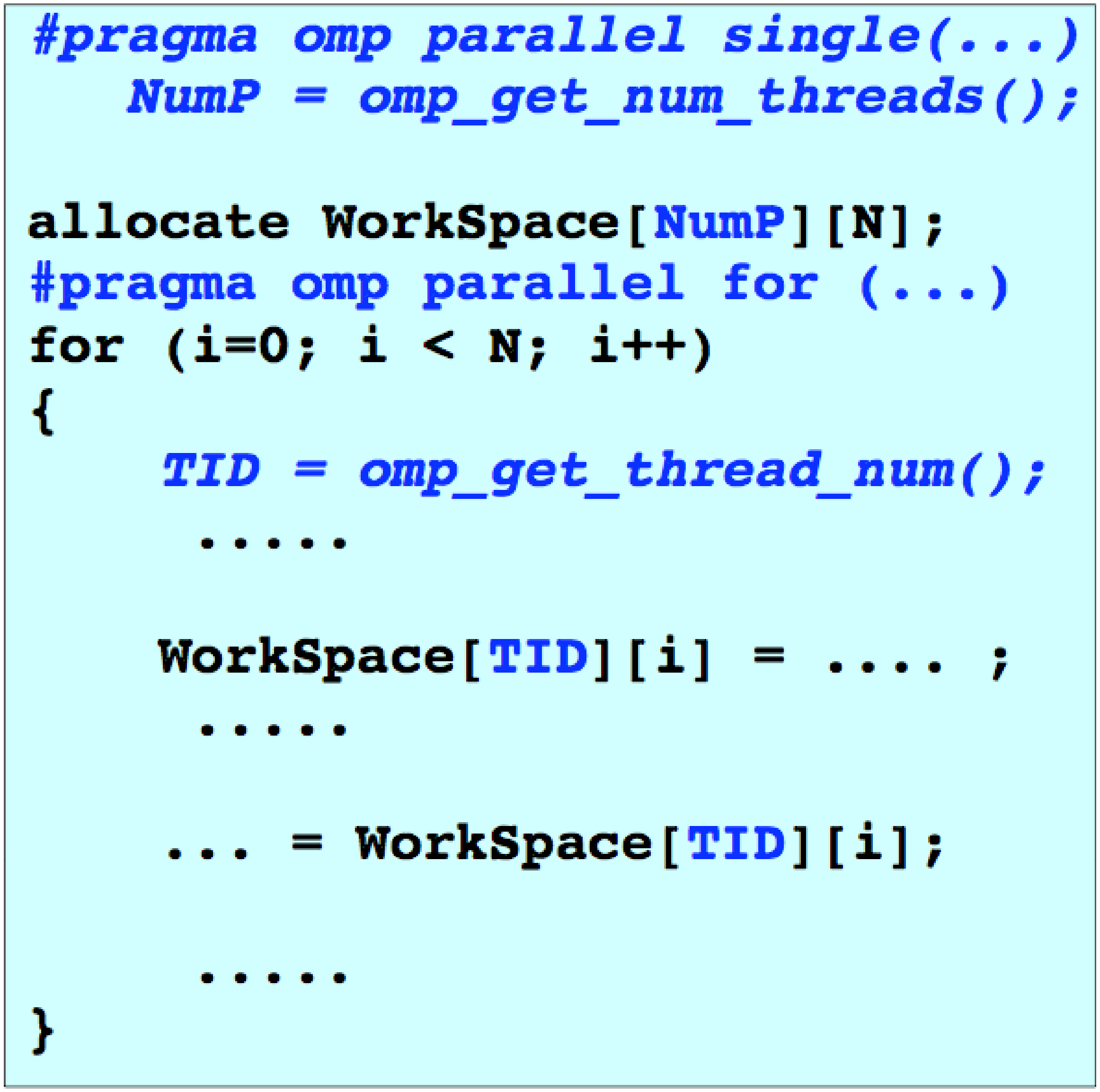
- *When an orphaned worksharing or synchronization directive is encountered in the sequential part of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored*

OpenMP Library Functions

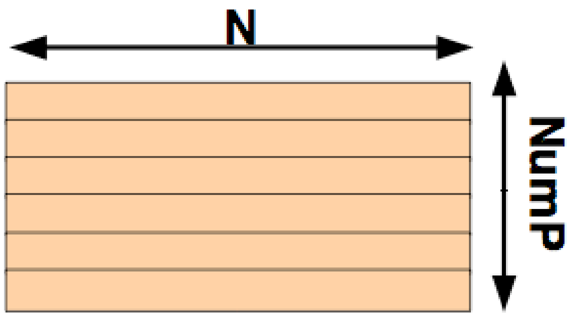
- In addition to directives, OpenMP also supports a number of functions that allow a programmer to control the execution of threaded programs.

```
/* thread and processor count */
void omp_set_num_threads (int num_threads);
int omp_get_num_threads ();
int omp_get_max_threads ();
int omp_get_thread_num ();
int omp_get_num_procs ();
int omp_in_parallel();
```

Example



```
#pragma omp parallel single(...)
    NumP = omp_get_num_threads();

allocate WorkSpace[NumP][N];
#pragma omp parallel for (...)
for (i=0; i < N; i++)
{
     TID = omp_get_thread_num();
       .....

     WorkSpace[TID][i] = .... ;
       .....

     ... = WorkSpace[TID][i];

       .....
}
```



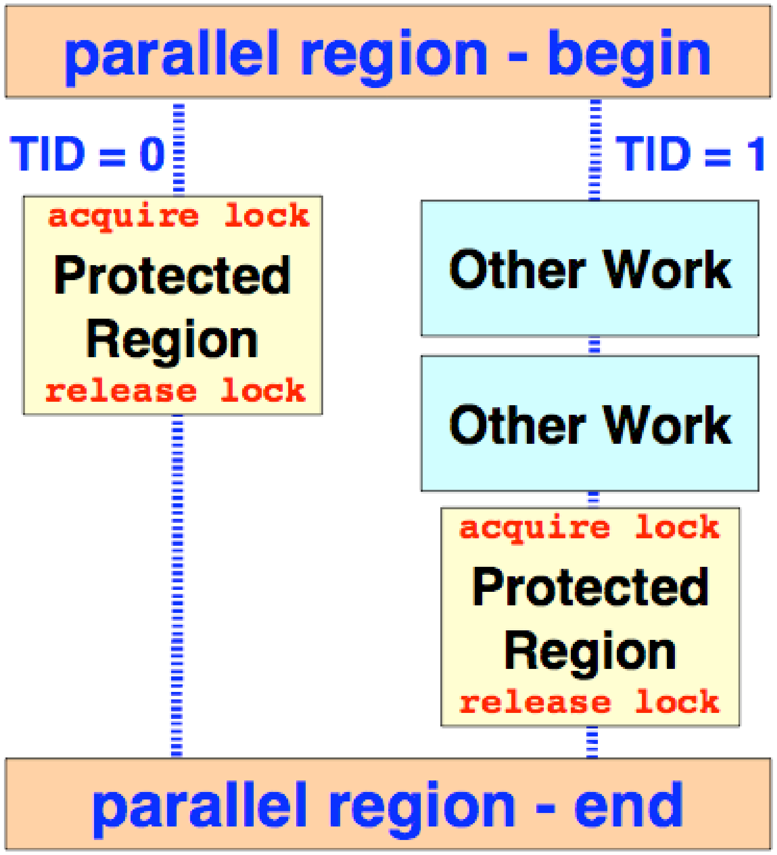
OpenMP Locks

- *Simple locks: may not be locked if already in a locked state*
- *Nestable locks: may be locked multiple times by the same thread before being unlocked*
- *In the remainder, we discuss simple locks only*
- *The interface for functions dealing with nested locks is similar (but using nestable lock variables):*

Simple locks	Nestable locks
`omp_init_lock`	`omp_init_nest_lock`
`omp_destroy_lock`	`omp_destroy_nest_lock`
`omp_set_lock`	`omp_set_nest_lock`
`omp_unset_lock`	`omp_unset_nest_lock`
`omp_test_lock`	`omp_test_nest_lock`

- **OpenMP also supports “critical” and “atomic” constructs that can be used in lieu of locks**

OpenMP Locking Example



- *The protected region contains the update of a shared variable*
- *One thread acquires the lock and performs the update*
- *Meanwhile, the other thread performs some other work*
- *When the lock is released again, the other thread performs the update*

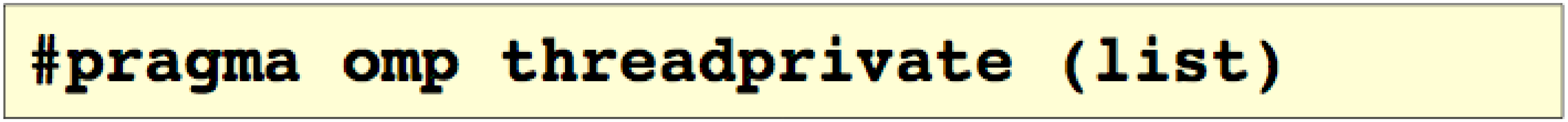
Environment Variables in OpenMP

- **`OMP_NUM_THREADS`: This environment variable specifies the default number of threads created upon entering a parallel region.**
- **`OMP_SET_DYNAMIC`: Determines if the number of threads can be dynamically changed.**
- **`OMP_NESTED`: Turns on nested parallelism.**
- **`OMP_SCHEDULE`: Scheduling of for-loops if the clause specifies runtime**

Shared Data in OpenMP

- ❑ ***Global data is shared and requires special care***
- ❑ ***A problem may arise in case multiple threads access the same memory section simultaneously:***
  - *Read-only data is no problem*
  - *Updates have to be checked for race conditions*
- ❑ ***It is your responsibility to deal with this situation***
- ❑ ***In general one can do the following:***
  - *Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel*
  - *Manually create thread private copies of the latter*
  - *Use the thread ID to access these private copies*
- ❑ ***Alternative: Use OpenMP's threadprivate directive***

threadprivate directive



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

- Thread private copies of the designated global variables created
- Several restrictions and rules apply when doing this:
  - —The number of threads has to remain the same for all the parallel regions (i.e. no dynamic threads)
  - —Initial data is undefined, unless copyin is used
  - —......
- Check the documentation when using threadprivate !

OpenMP Performance Tips

- ❑ *Parallelize at the highest level possible*
  - ✓ *Outer loop preferred over inner loop*
    - ◆ *If it is sufficiently long*
- ❑ *Parallel Regions*
  - • *Use as few parallel regions as possible*
    - ✓ *Enclose multiple loops in one parallel region*
  - • *Avoid a parallel region in a inner loop*
    - ✓ *Can often be moved up*
- ❑ *Reduce barrier usage to the bare minimum*
  - • *Use nowait where possible*
    - ✓ *Be careful not to introduce a data race though!*

OpenMP Performance Tips (contd)

- ❑ *Minimize the size of a* ***critical*** *region*
- ❑ *Avoid the* ***ordered*** *construct*
  - *It is slow*
- ❑ *Avoid, or minimize,* ***false sharing*** *from the start*
  - *Use private data as much as possible*
  - *Experiment with different values for the chunk size*
  - *Try a non-static iteration scheme*
- ❑ *Things To Experiment With:*
  - *Master versus Single*
  - *Read-only data - shared or private ?*

Reading List for Next Lecture (Jan 31st)

- Task construct proposed in OpenMP 3.0
  - Section 2.7: task construct
  - http://www.openmp.org/mp-documents/spec30_draft.pdf
- Memory Models
  - OpenMP 2.5
    - Section 1.4: Memory Model
    - Section 2.7.5: Flush construct
    - http://www.openmp.org/mp-documents/spec25.pdf
  - Cilk 5.4.6
    - Section 2.5: Shared Memory (Cilk_fence construct)
    - http://supertech.csail.mit.edu/cilk/manual-5.4.6.pdf