An Overview of OpenMP

Ruud van der Pas

Senior Staff Engineer
SPARC Microelectronics
Oracle
Santa Clara, CA, USA

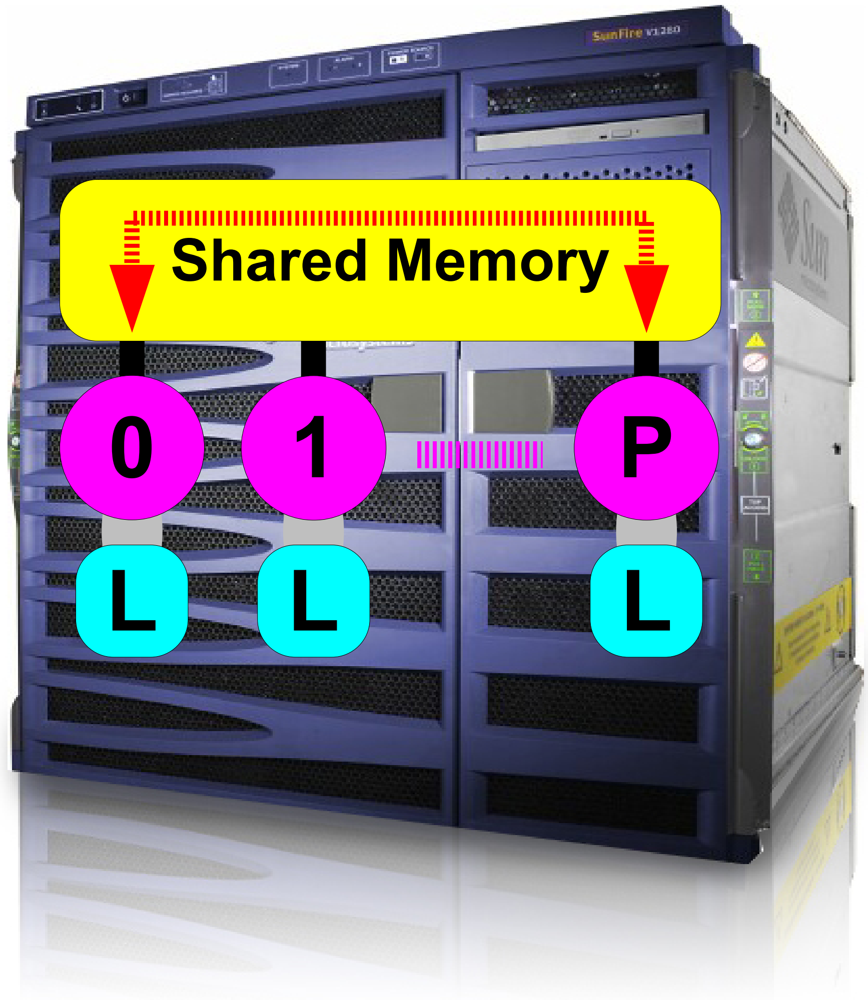
IWOMP 2011

Chicago, IL, USA
June 13-15, 2011

Outline

- Getting Started with OpenMP
- Using OpenMP
- What's New in OpenMP 3.1

Getting Started With OpenMP


Shared Memory
0
1
P
L
L
L

OpenMP™

http://www.openmp.org

http://www.compunity.org

http://www.openmp.org

Shameless Plug - “Using OpenMP”

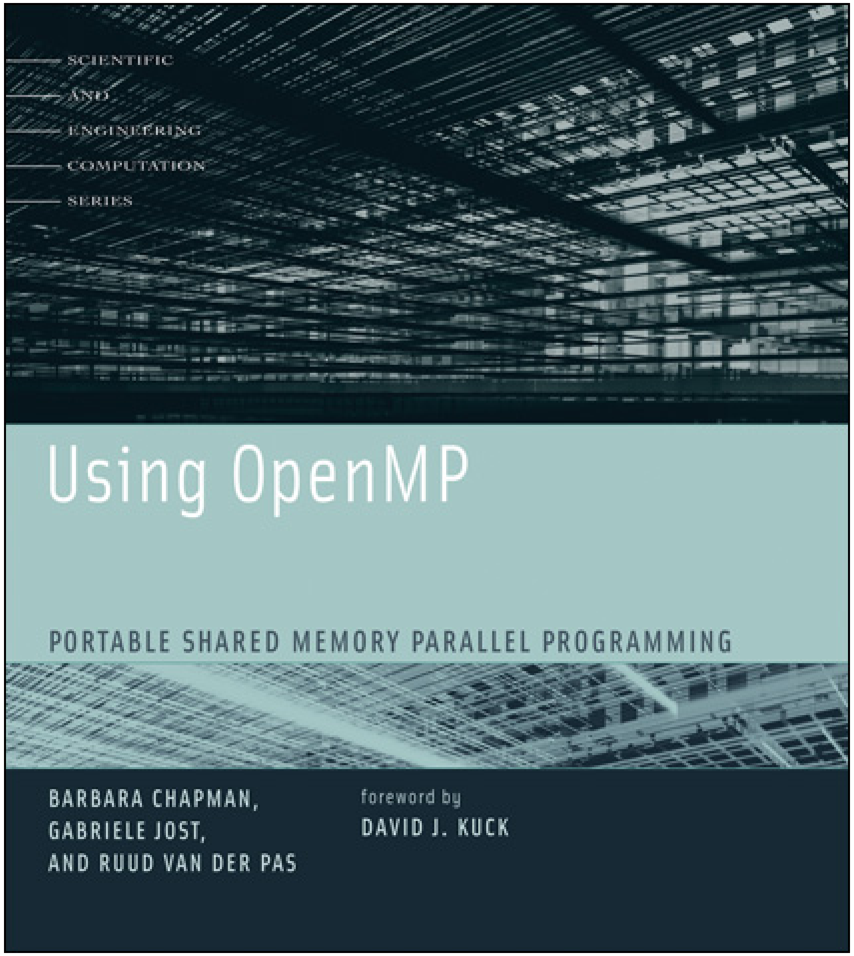
“Using OpenMP”
Portable Shared Memory Parallel Programming

Chapman, Jost, van der Pas

MIT Press, 2008

ISBN-10: 0-262-53302-2
ISBN-13: 978-0-262-53302-7

List price: 35 $US



All 41 examples are available NOW!

As well as a forum on http://www.openmp.org

OpenMP™ THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

Subscribe to the News Feed

»» OpenMP Specifications
»About OpenMP
»Compilers
»R
»D

Download Book Examples and Discuss

Ruud van der Pas, one of the authors of the book ***Using OpenMP - - Portable Shared Memory Parallel Programming*** by Chapman, Jost, and van der Pas, has made 41 of the examples in the book available for download and your use.

These source examples are available as a free download **»here** (a zip file) under the BSD license. Each source comes with a copy of the license. Please do not remove this.

Download the examples and discuss in forum:

http://www.openmp.org/wp/2009/04/download-book-examples-and-discuss

To make things easier, each source directory has a make file called "Makefile". This file can be used to build and run the examples in the specific directory. Before you do so, you need to activate the appropriate include line in file Makefile. There are include files for several compilers and Unix based Operating Systems (Linux, Solaris and Mac OS to precise).

These files have been put together on a best effort basis. The User's Guide that is bundled with the examples explains this in more detail.

Also, we have created a new forum, **»Using OpenMP - The Book and Examples**, for discussion and feedback.

Posted on April 2, 2009

Ev
»M
(p
W
13

Input Register
Alert the OpenMP.org webmaster about new products, events, or updates and we'll post it here.
»webmaster@openmp.org

Search OpenMP.org
Google™ Custom Search
Search

Get
»OpenMP specs

Use
»OpenMP Compilers

Learn

»*Using OpenMP* – the book
»*Using OpenMP* – the examples
»*Using OpenMP* – the forum
»Wikipedia
»OpenMP Tutorial
»More Resources

Discuss

What is OpenMP?

- ***De-facto standard Application Programming Interface (API) to write shared memory parallel applications in C, C++, and Fortran***
- ***Consists of Compiler Directives, Run time routines and Environment variables***
- ***Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)***
- ***Version 3.0 has been released May 2008***
  - ***The upcoming 3.1 release will be released soon***

OpenMP.org » About the OpenMP ARB and OpenMP.org

http://openmp.org/wp/about-openmp/

Members

OpenMP is widely supported by industry, as well as the academic community

Permanent Members of the ARB:

- **AMD** (Roy Ju)
- **Cray** (James Beyer)
- **Fujitsu** (Matthijs van Waveren)
- **HP** (Uriel Schafer)
- **IBM** (Kelvin Li)
- **Intel** (Sanjiv Shah)
- **NEC** (Kazuhiro Kusano)
- **The Portland Group, Inc.** (Michael Wolfe)
- **Oracle Corporation** (Nawal Copty)
- **Microsoft** (-)
- **Texas Instruments** (Andy Fritsch)
- **CAPS-Entreprise** (Francois Bodin)

Auxiliary Members of the ARB:

- **ANL** (Kalyan Kumaran)
- **ASC/LLNL** (Bronis R. de Supinski)
- **cOMPunity** (Barbara Chapman)
- **EPCC** (Mark Bull)
- **LANL** (John Thorp)
- **NASA** (Henry Jin)
- **RWTH Aachen University** (Dieter an Mey)

Directors

When to consider OpenMP?

- ***Using an automatically parallelizing compiler:***
  - ***It can not find the parallelism***
    - ✔ ***The data dependence analysis is not able to determine whether it is safe to parallelize (or not)***
  - ***The granularity is not high enough***
    - ✔ ***The compiler lacks information to parallelize at the highest possible level***
- ***Not using an automatically parallelizing compiler:***
  - ***No choice, other than doing it yourself***

Advantages of OpenMP

- *Good performance and scalability*
  - *If you do it right ....*
- *De-facto and mature standard*
- *An OpenMP program is portable*
  - *Supported by a large number of compilers*
- *Requires little programming effort*
- *Allows the program to be parallelized incrementally*

OpenMP and Multicore

OpenMP is ideally suited for multicore architectures

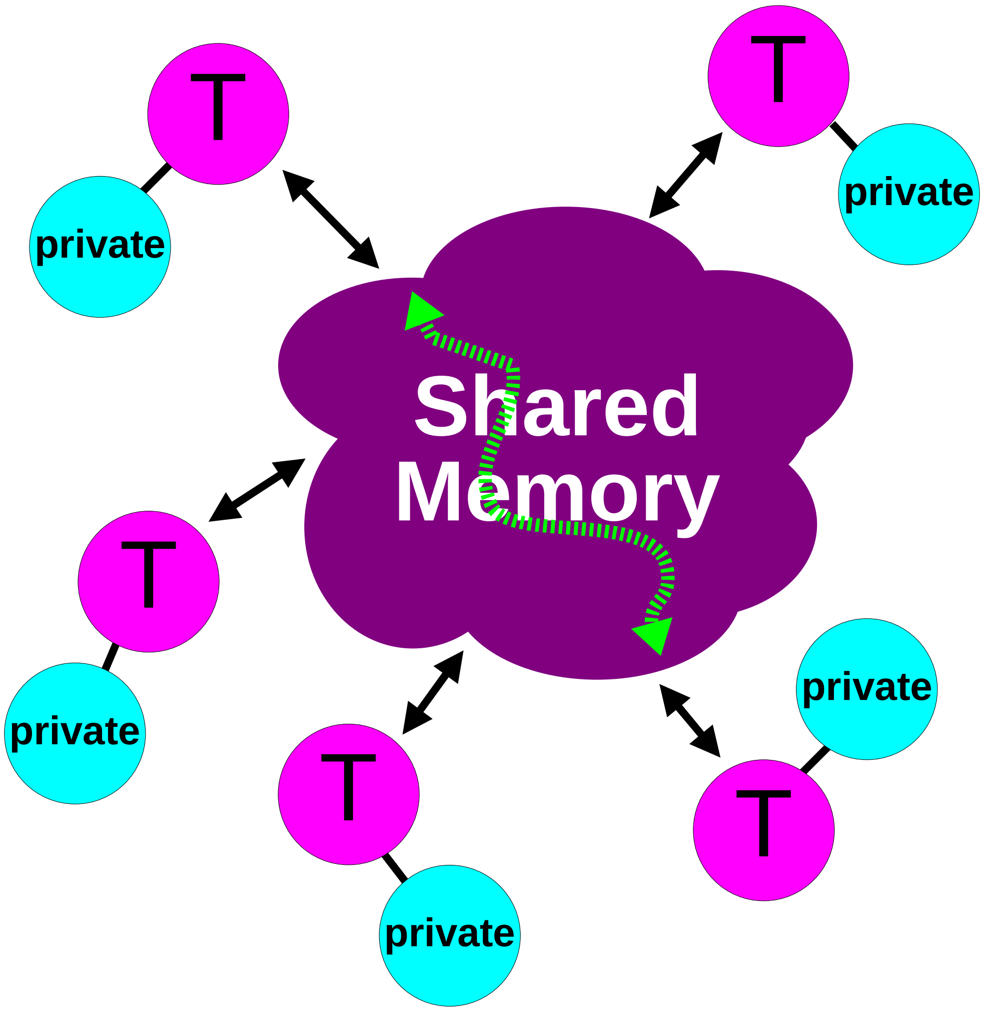
Memory and threading model map naturally

Lightweight

Mature

Widely available and used

The OpenMP Memory Model



- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can only be accessed by the thread that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place, but it is mostly implicit*

Data-sharing Attributes

- In an OpenMP program, data needs to be “labeled”
- Essentially there are two basic types:
  - Shared - There is only one instance of the data
    - Threads can read and write the data simultaneously unless protected through a specific construct
    - All changes made are visible to all threads
      - But not necessarily immediately, unless enforced ......
  - Private - Each thread has a copy of the data
    - No other thread can access this data
    - Changes only visible to the thread owning the data

Private and shared clauses

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*

About storage association

- Private variables are undefined on entry and exit of the parallel region
- A private variable within a parallel region has no storage association with the same variable outside of the region
- Use the **firstprivate** and **lastprivate** clauses to override this behavior
- We illustrate these concepts with an example

The firstprivate and lastprivate clauses

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 sequentially last iteration or section updates the value of the objects in the list*

Example firstprivate


```
n = 2; indx = 4;
#pragma omp parallel default(none) private(i,TID) \
        firstprivate(indx) shared(n,a)
{ TID = omp_get_thread_num();
  indx = indx + n*TID;
  for(i=indx; i<indx+n; i++)
    a[i] = TID + 1;
} /*-- End of parallel region --*/
```

					TID = 0		TID = 1		TID = 2	
index	0	1	2	3	4	5	6	7	8	9
value					1	1	2	2	3	3

Example lastprivate


```
#pragma omp parallel for default(none) lastprivate(a)
for (int i=0; i<n; i++)
{ .......
   a = i + 1;
   .......
} // End of parallel region

b = 2 * a; // value of b is 2*n
```

The default clause

default (none | shared) *C/C++*

default (none | shared | private | threadprivate) *Fortran*

none

- ✔ *No implicit defaults; have to scope all variables explicitly*

shared

- ✔ *All variables are shared*
- ✔ *The default in absence of an explicit "default" clause*

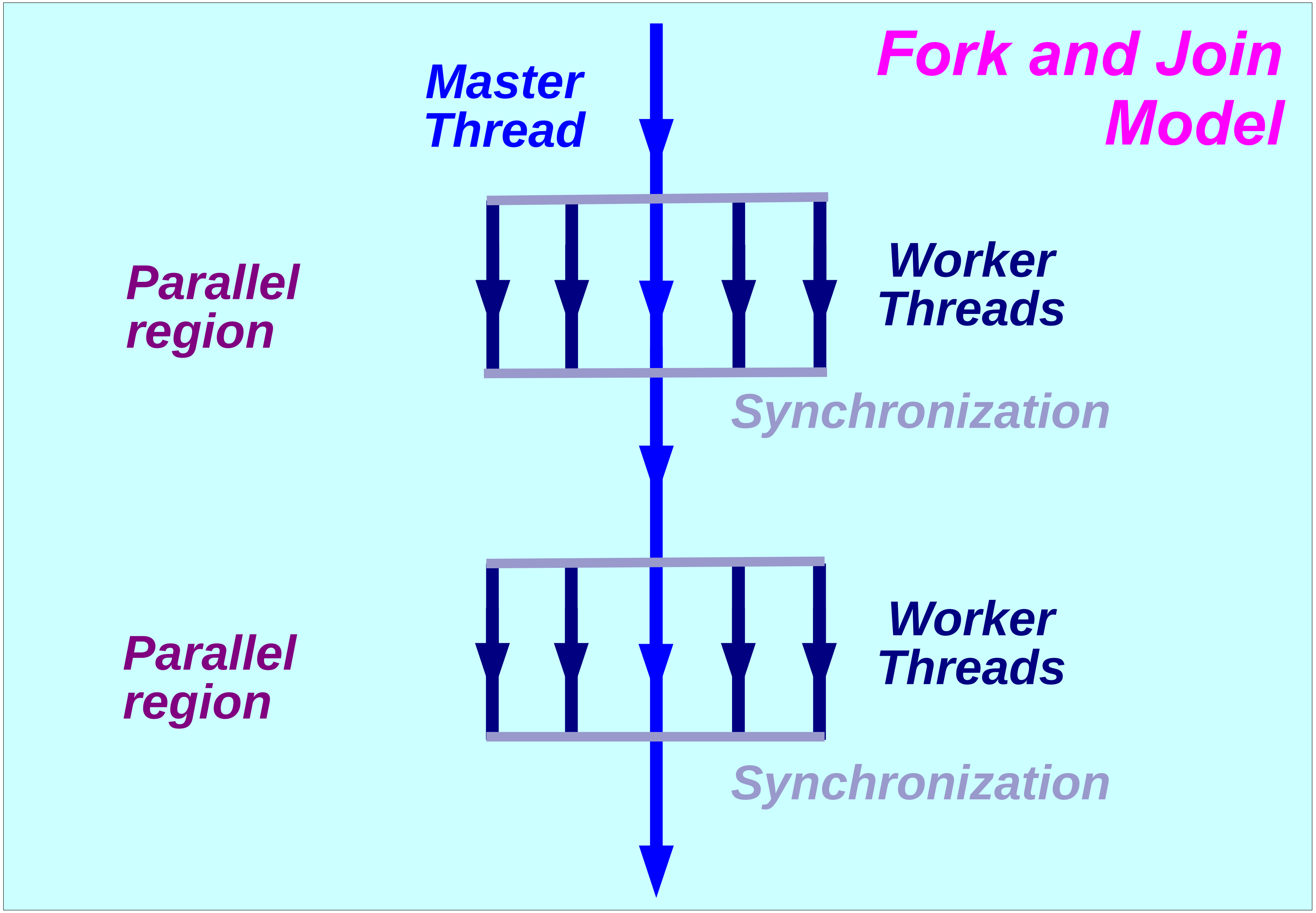
private

- ✔ *All variables are private to the thread*
- ✔ *Includes common block data, unless THREADPRIVATE*

f rstprivate

- ✔ *All variables are private to the thread; pre-initialized*

The OpenMP Execution Model



Defining Parallelism in OpenMP

- ***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*

The Parallel Region

A parallel region is a block of code executed by all threads in the team

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

} // End of parall section (note: implied barrier)
```

```
!$omp parallel [clause[[,] clause] ...]

   "this code is executed in parallel"

!$omp end parallel (implied barrier)
```

Parallel Region - An Example/1

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {



          printf("Hello World\n");



   return(0);
}
```

Parallel Region - An Example/1

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

Parallel Region - An Example/2

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

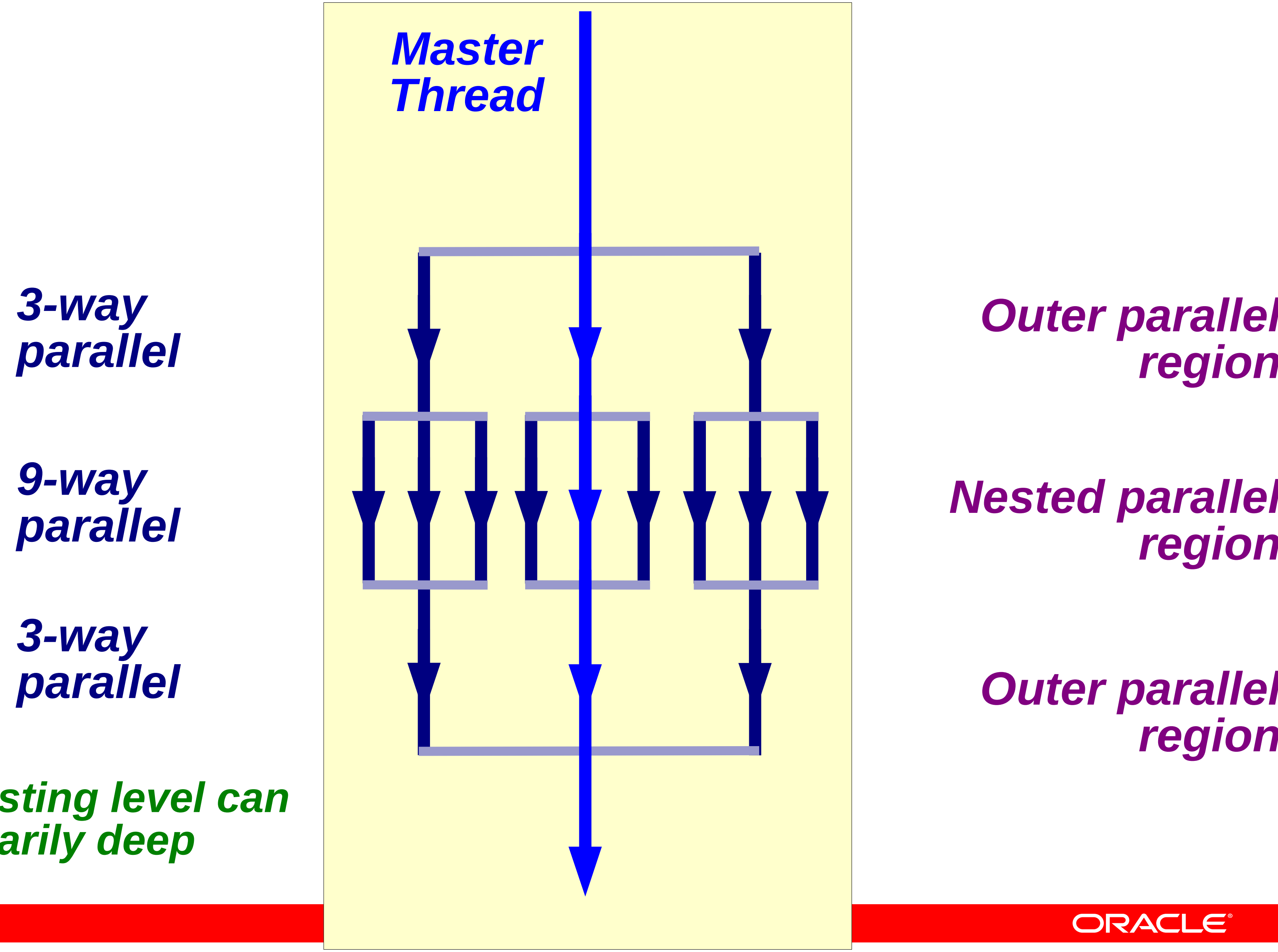
The if clause

if (scalar expression)

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

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

Nested Parallelism



Note: nesting level can be arbitrarily deep

Nested Parallelism Support/1

- ***Environment variable and runtime routines to set/get the maximum number of nested active parallel regions***

```
OMP_MAX_ACTIVE_LEVELS
omp_set_max_active_levels()
omp_get_max_active_levels()
```

- ***Environment variable and runtime routine to set/get the maximum number of OpenMP threads available to the program***

```
OMP_THREAD_LIMIT
omp_get_thread_limit()
```

Nested Parallelism Support/2

- ***Per-task internal control variables***
  - *Allow, for example, calling* **`omp_set_num_threads()`** *inside a parallel region to control the team size for next level of parallelism*
- ***Library routines to determine***
  - *Depth of nesting*

```
omp_get_level()
omp_get_active_level()
```

  - *IDs of parent/grandparent etc. threads*

```
omp_get_ancestor_thread_num(level)
```

  - *Team sizes of parent/grandparent etc. teams*

```
omp_get_team_size(level)
```

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

Using OpenMP

Using OpenMP

- We have just seen a glimpse of OpenMP
- To be practically useful, much more functionality is needed
- Covered in this section:
  - Many of the language constructs
  - Features that may be useful or needed when running an OpenMP application
- Note that the tasking concept is covered in a separate section

Components of OpenMP

IWOMP International Workshop on OpenMP

Directives

- ***Parallel region***
- ***Worksharing constructs***
- ***Tasking***
- ***Synchronization***
- ***Data-sharing attributes***

Runtime environment

- ***Number of threads***
- ***Thread ID***
- ***Dynamic thread adjustment***
- ***Nested parallelism***
- ***Schedule***
- ***Active levels***
- ***Thread limit***
- ***Nesting level***
- ***Ancestor thread***
- ***Team size***
- ***Wallclock timer***
- ***Locking***

Environment variables

- ***Number of threads***
- ***Scheduling type***
- ***Dynamic thread adjustment***
- ***Nested parallelism***
- ***Stacksize***
- ***Idle threads***
- ***Active levels***
- ***Thread limit***

Directive format

- *C: directives are case sensitive*
  - *Syntax:* **#pragma omp directive [clause [clause] ...]**
- *Continuation: use \ in pragma*
- *Conditional compilation:* **_OPENMP** *macro is set*
- *Fortran: directives are case insensitive*
  - *Syntax:* **sentinel directive [clause [[,] clause]...]**
  - *The sentinel is one of the following:*
    - ✔ ***!$OMP*** *or* ***C$OMP*** *or* ****$OMP*** *(fixed format)*
    - ✔ ***!$OMP*** *(free format)*
- *Continuation: follows the language syntax*
- *Conditional compilation:* **!$** *or* **C$ -> 2 spaces**

The reduction clause - Example

```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

Variable SUM is a shared variable

- ***Care needs to be taken when updating shared variable SUM***
- ***With the reduction clause, the OpenMP compiler generates code such that a race condition is avoided***

The reduction clause

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

- ✔ *Reduction variable(s) must be shared variables*
- ✔ *A reduction is defined as:*

Check the docs for details

Fortran	C/C++
`x = x operator expr`	`x = x operator expr`
`x = expr operator x`	`x = expr operator x`
`x = intrinsic (x, expr_list)`	`x++, ++x, x--, --x`
`x = intrinsic (expr_list, x)`	`x <binop> = expr`

- ✔ *Note that the value of a reduction variable is undefined from the moment the first thread reaches the clause till the operation has completed*
- ✔ *The reduction can be hidden in a function call*

Fortran - Allocatable Arrays

- Fortran allocatable arrays whose status is "currently allocated" are allowed to be specified as private, lastprivate, firstprivate, reduction, or copyprivate

```
integer, allocatable,dimension (:) :: A
integer i

allocate (A(n))

!$omp parallel private (A)
      do i = 1, n
         A(i) = i
      end do
      ...
!$omp end parallel
```

Barrier/1

Suppose we run each of these two loops in parallel over i:

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

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

This may give us a wrong answer (one day)

Why ?

Barrier/2

*We need to have <u>updated all of a[ ]</u> first, before using a[ ] **

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

wait !

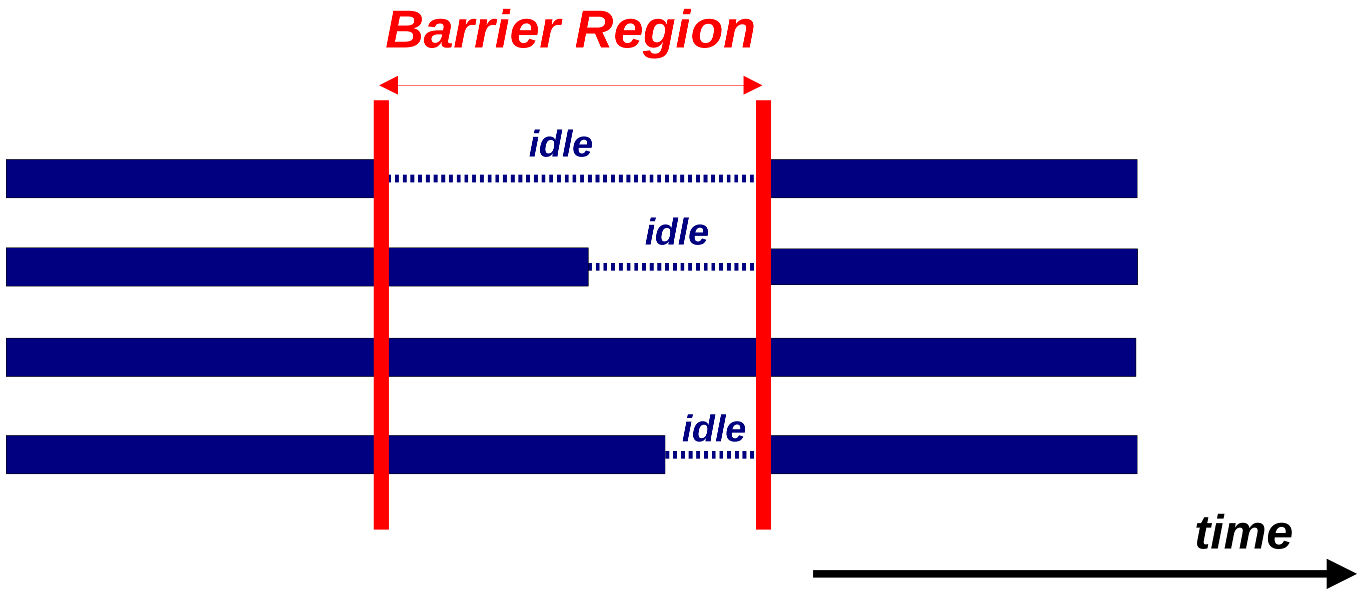
barrier

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

All threads wait at the barrier point and only continue when all threads have reached the barrier point

****) If there is the <u>guarantee</u> that the mapping of iterations onto threads is identical for both loops, there will not be a data race in this case***

Barrier/3



Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```

When to use barriers ?

- *If data is updated asynchronously and data integrity is at risk*
- *Examples:*
  - *Between parts in the code that read and write the same section of memory*
  - *After one timestep/iteration in a solver*
- *Unfortunately, barriers tend to be expensive and also may not scale to a large number of processors*
- *Therefore, use them with care*

The nowait clause

- *To minimize synchronization, some directives support the optional* nowait *clause*
    - *If present, threads do not synchronize/wait at the end of that particular construct*
- *In C, it is one of the clauses on the pragma*
- *In Fortran, it is appended at the closing part of the construct*

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

```
!$omp do
        :
        :
!$omp end do nowait
```

The Worksharing Constructs


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

!$OMP DO
    ....
!$OMP END DO
```

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

!$OMP SECTIONS
       ....
!$OMP END SECTIONS
```

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

!$OMP SINGLE
       ....
!$OMP END 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 specif ed)*
- *A work-sharing construct does not launch any new threads*

The Workshare construct

Fortran has a fourth worksharing construct:

```
!$OMP WORKSHARE

    <array syntax>

!$OMP END WORKSHARE [NOWAIT]
```

Example:

```
!$OMP WORKSHARE
    A(1:M) = A(1:M) + B(1:M)
!$OMP END WORKSHARE NOWAIT
```

The omp for/do directive

```
#pragma omp for [clauses]
   for (.....)
   {
         <code-block>
   }
```

```
!$omp do [clauses]
   do ...
       <code-block>
   end do
!$omp end do[nowait]
```

The iterations of the loop are distributed over the threads

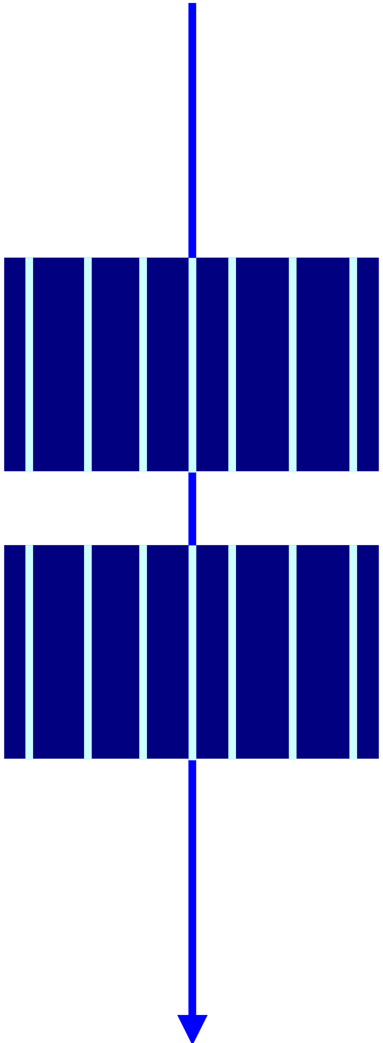
The omp for directive - Example

```
#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)



C++: Random Access Iterator Loops

Parallelization of random access iterator loops is supported

```
void iterator_example()
{
  std::vector vec(23);
  std::vector::iterator it;

  #pragma omp for default(none)shared(vec)
  for (it = vec.begin(); it < vec.end(); it++)
  {
    // do work with *it //
  }
}
```

Loop Collapse

- Allows parallelization of perfectly nested loops without using nested parallelism
- The collapse clause on for/do loop indicates how many loops should be collapsed
- Compiler forms a single loop and then parallelizes it

```
!$omp parallel do collapse(2) ...
  do i = il, iu, is
     do j = jl, ju, js
        do k = kl, ku, ks
           .....
        end do
     end do
  end do
!$omp end parallel do
```

The schedule clause/1

schedule (static | dynamic | guided | auto [, 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*
  - *Details are implementation defined*
- ✔ *Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region*

The schedule clause/2

Example static schedule

Loop of length 16, 4 threads:

Thread	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

**) The precise distribution is implementation def ned*

The schedule clause/3

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*

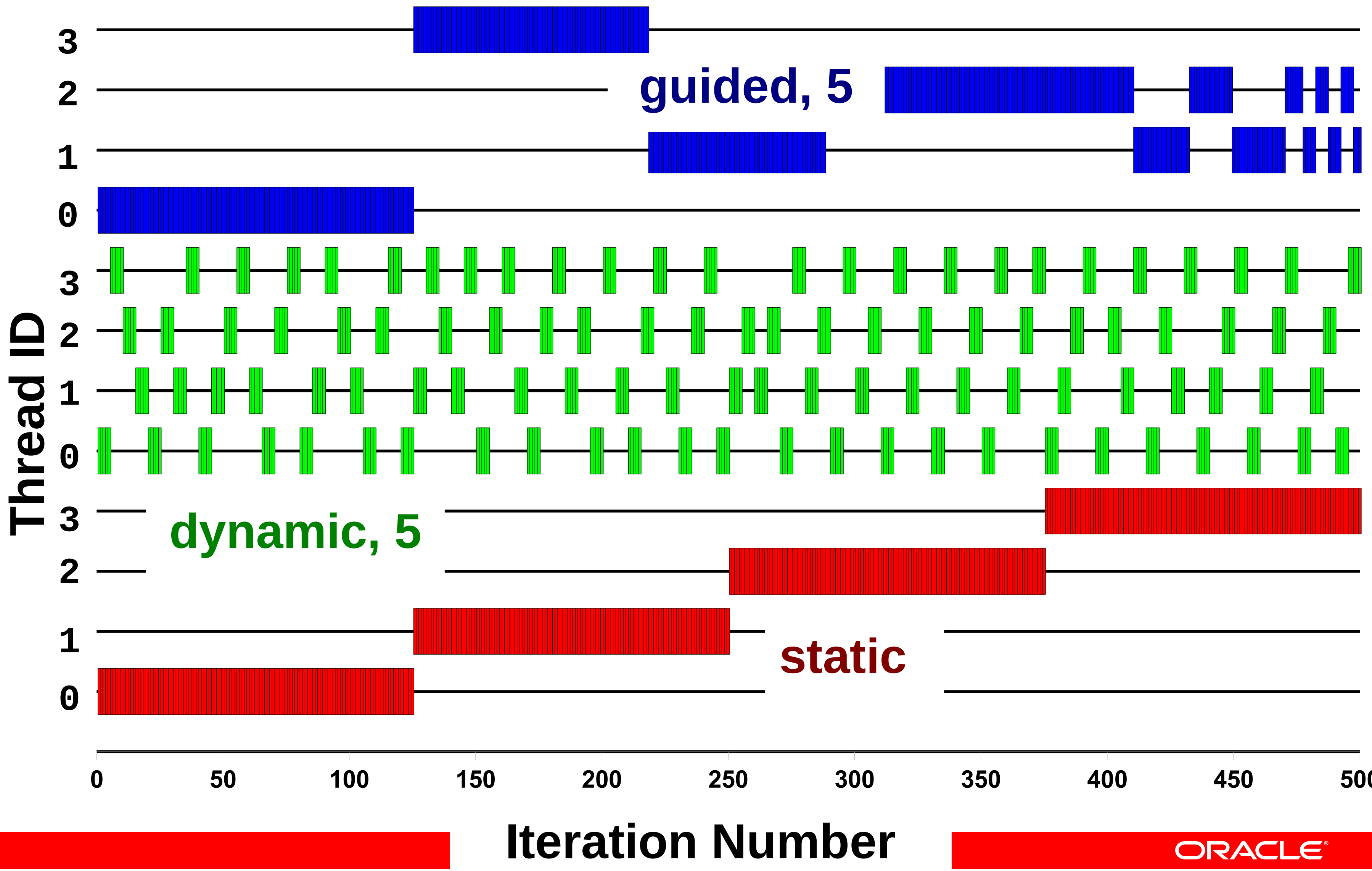
auto

- ✔ *The compiler (or runtime system) decides what is best to use; choice could be implementation dependent*

runtime

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

Experiment - 500 iterations, 4 threads



ORACLE

Schedule Kinds Functions

- *Makes* `schedule(runtime)` *more general*
- *Can set/get schedule it with library routines:*

```
omp_set_schedule()
omp_get_schedule()
```

- *Also allows implementations to add their own schedule kinds*

Parallel sections


```
#pragma omp sections [clauses]
{
  #pragma omp section
     {....}
  #pragma omp section
     {....}
  ....
}
```

```
!$omp sections [clauses]

  !$ omp section
     {....}
  !$ omp section
     {....}
  ....
!$omp end sections [nowait]
```

Individual section blocks are executed in parallel

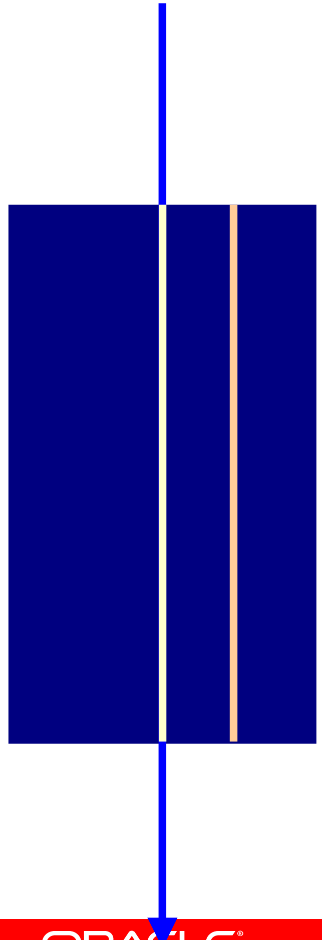
The Sections Directive - Example

```
#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 --*/
```



Overlap I/O and Processing/1

Time

Input Thread	Processing Thread(s)	Output Thread
0		
1	0	
2	1	0
3	2	1
4	3	2
5	4	3
	5	4
		5

Overlap I/O and Processing/2

```
#pragma omp parallel sections
{
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) read_input(i);
         (void) signal_read(i);
     }
   }
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) wait_read(i);
         (void) process_data(i);
         (void) signal_processed(i);
     }
   }
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
         (void) wait_processed(i);
         (void) write_output(i);
     }
   }
} /*-- End of parallel sections --*/
```

Input Thread

Processing Thread(s)

Output Thread

The Single Directive

Only one thread in the team executes the code enclosed

```
#pragma omp single [private][firstprivate] \
                   [copyprivate][nowait]
{
       <code-block>
}
```

```
!$omp single [private][firstprivate]
   <code-block>
!$omp end single [copyprivate][nowait]
```

Single processor region/1

Original Code

```
.....
"read A[0..N-1]";
.....
```

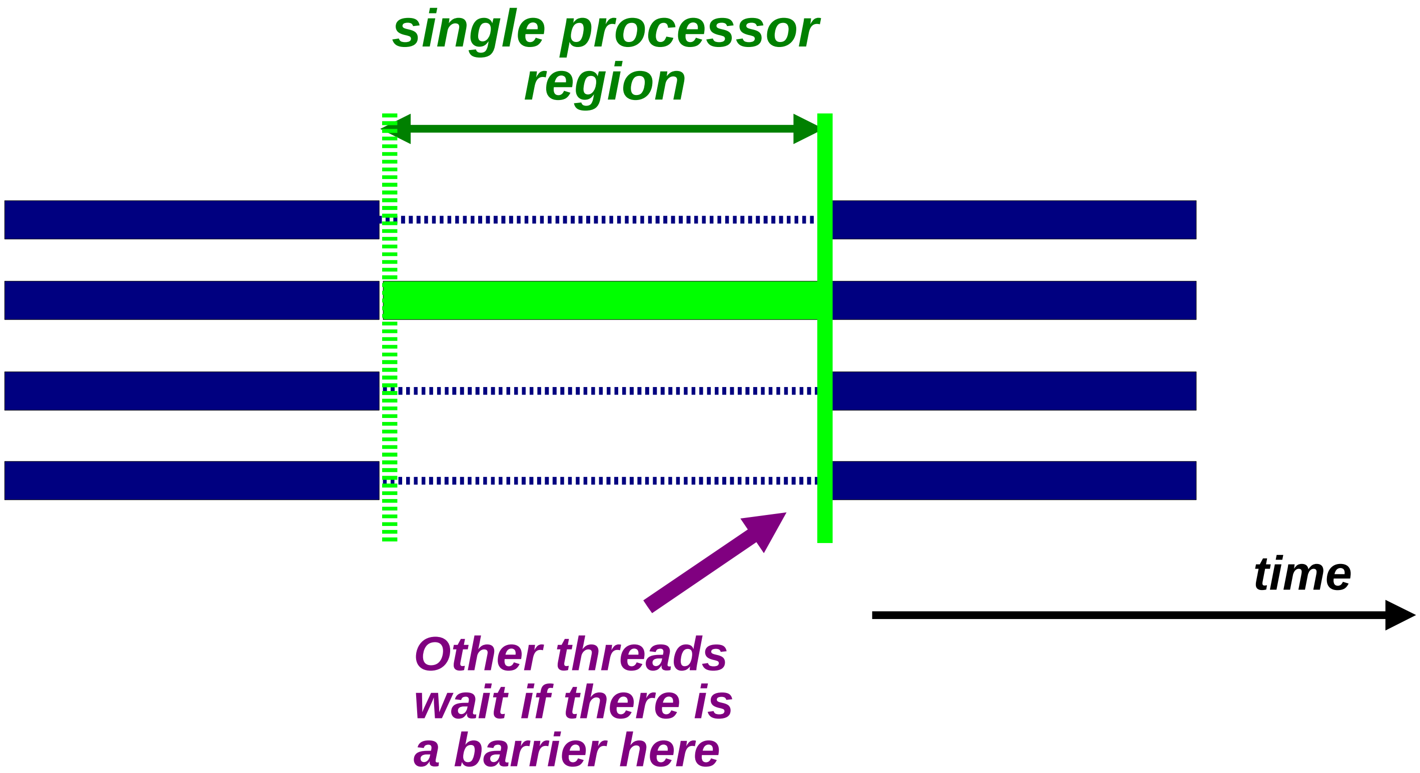
```
#pragma omp parallel \
            shared (A)
{
        .....
    #pragma omp single nowait
        {"read A[0..N-1]";}
        .....
    #pragma omp barrier
        "use A"
}
```

Parallel Version

Only one thread executes the single region

This construct is ideally suited for I/O or initializations

Single processor region/2



Combined work-sharing constructs


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

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

Single PARALLEL loop

```
!$omp parallel
!$omp do
       ...
!$omp end do
!$omp end parallel
```

```
!$omp parallel do
       ...
!$omp end parallel do
```

Single WORKSHARE loop

```
!$omp parallel
!$omp workshare
       ...
!$omp end workshare
!$omp end parallel
```

```
!$omp parallel workshare
       ...
!$omp end parallel workshare
```

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

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

Single PARALLEL sections

```
!$omp parallel
!$omp sections
       ...
!$omp end sections
!$omp end parallel
```

```
!$omp parallel sections
       ...
!$omp end parallel sections
```

Orphaning


```
         :
#pragma omp parallel
{
         :
      (void) dowork();
         :
}
         :
```

```
void dowork()
{
         :
  #pragma omp for
   for (int i=0;i<n;i++)
   {
         :
   }
         :
}
```

orphaned work-sharing directive

- *The OpenMP specif cation 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*

More on orphaning

```
(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*

Example - Parallelizing Bulky Loops

```
for (i=0; i<n; i++) /* Parallel loop */
{
    a = ...
    b = ... a ..
    c[i] = ....
          ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
          ......
}
```

Step 1: "Outlining"

```
for (i=0; i<n; i++) /* Parallel loop */
{
     (void) FuncPar(i,m,c,...)
}
```

- *Still a sequential program*
- *Should behave identically*
- *Easy to test for correctness*
- *But, parallel by design*

```
void FuncPar(i,m,c,....)
{
    float a, b; /* Private data */
    int   j;
    a = ...
    b = ... a ..
    c[i] = ....
         ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
         ......
}
```

Step 2: Parallelize

```
#pragma omp parallel for private(i) shared(m,c,..)
```

```
for (i=0; i<n; i++) /* Parallel loop */
{
     (void) FuncPar(i,m,c,...)
} /*-- End of parallel for --*/
```

Minimal scoping required

Less error prone

```
void FuncPar(i,m,c,....)
{
    float a, b; /* Private data */
    int   j;
    a = ...
    b = ... a ..
    c[i] = ....
         ......
    for (j=0; j<m; j++)
    {
      <a lot more code in this loop>
    }
         ......
}
```

Additional Directives/1


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

```
!$omp master
        <code-block>
!$omp end master
```

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

```
#pragma omp atomic
```

```
!$omp atomic
```

The Master Directive

Only the <u>master thread</u> executes the code block:

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

```
!$omp master
        <code-block>
!$omp end master
```

There is no implied barrier on entry or exit !

Critical Region/1

If sum is a shared variable, this loop can not run in parallel by simply using a "#pragma omp for"

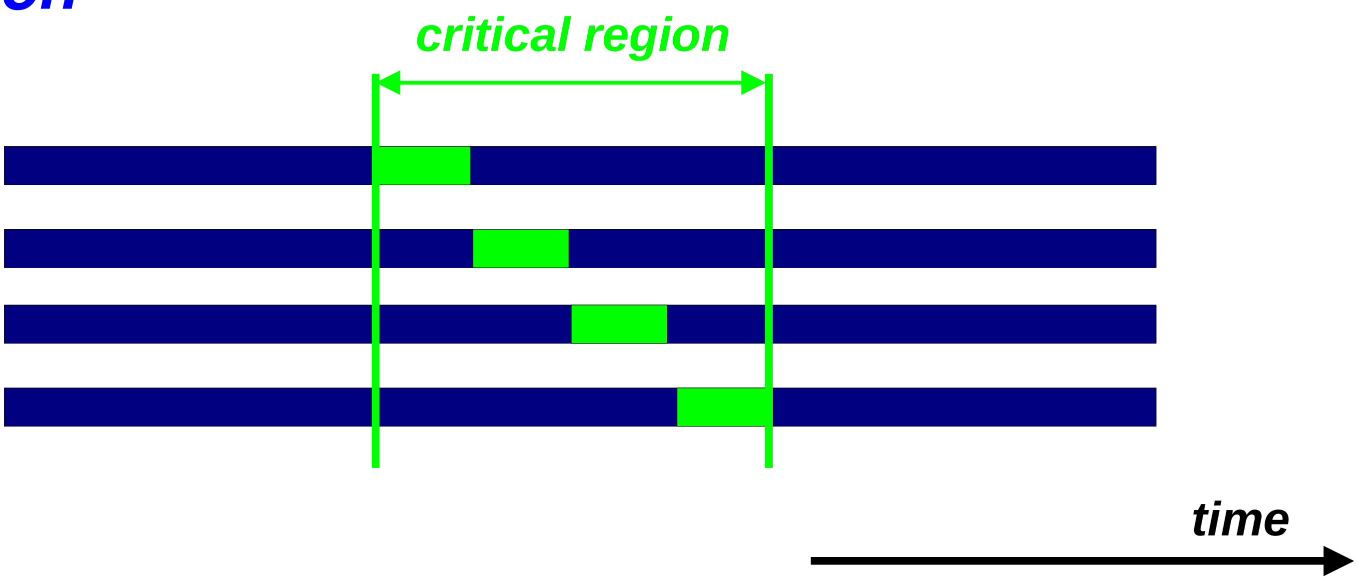
```
for (i=0; i < n; i++){
      .....
    sum += a[i];
      .....
}
```

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

All threads execute the update, but only one at a time will do so

Critical Region/2

- *Useful to avoid a race condition, or to perform I/O (but that still has random order)*
- *Be aware that there is a cost associated with a critical region*



Critical and Atomic constructs

Critical: All threads execute the code, but only one at a time:

```
#pragma omp critical [(name)]
{<code-block>}
```

```
!$omp critical [(name)]
        <code-block>
!$omp end critical [(name)]
```

There is no implied barrier on entry or exit !

Atomic: only the loads and store are atomic ....

```
#pragma omp atomic
   <statement>
```

```
!$omp atomic
   <statement>
```

This is a lightweight, special form of a critical section

```
#pragma omp atomic
     a[indx[i]] += b[i];
```

Additional Directives/2


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

```
!$omp ordered
        <code-block>
!$omp end ordered
```

```
#pragma omp flush [(list)]
```

```
!$omp flush [(list)]
```

Additional Directives/2

The enclosed block of code is executed in the order in which iterations would be executed sequentially:

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

```
!$omp ordered
        <code-block>
!$omp end ordered
```

May introduce serialization (could be expensive)

Ensure that all threads in a team have a consistent view of certain objects in memory:

```
#pragma omp flush
[(list)]
```

```
!$omp flush [(list)]
```

In the absence of a list, all visible variables are f ushed

The flush directive

Thread A	Thread B

`X = 0`
 `:`
 `:`
 `X = 1`
 `:`
 `:`
	`while (X == 0)`
	 `{`
	 `"wait"`
	 `}`

If shared variable X is kept within a register, the modification may not be made visible to the other thread(s)

Implied Flush Regions/1

- During a barrier region
- At exit from worksharing regions, unless a nowait is present
- At entry to and exit from parallel, critical, ordered and parallel worksharing regions
- During omp_set_lock and omp_unset_lock regions
- During omp_test_lock, omp_set_nest_lock, omp_unset _nest_lock and omp_test_nest_lock regions, if the region causes the lock to be set or unset
- Immediately before and after every task scheduling point

Implied Flush Regions/2

- At entry to and exit from atomic regions, where the list contains only the variable updated in the atomic construct
- A flush region is <u>not implied</u> at the following locations:
  – At entry to a worksharing region
  – At entry to or exit from a master region

OpenMP and Global Data

Global data - An example

```fortran
      program global_data
            ....
      include "global.h"
         ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
       ......
```

f le global.h

```fortran
common /work/a(m,n),b(m)
```

```fortran
subroutine suba(j)
  .....
include "global.h"
  .....
do i = 1, m
   b(i) = j
end do

do i = 1, m
    a(i,j) =
func_call(b(i))
end do

return
end
```

Data Race !

Global data - A Data Race!

Thread 1	*Thread 2*
`call suba(1)`	`call suba(2)`

Thread 1:

```
subroutine suba(j=1)

do i = 1, m
   b(i) = 1
end do
      ....
do i = 1, m
  a(i,1)=func_call(b(i))
end do
```

Thread 2:

```
subroutine suba(j=2)

do i = 1, m
   b(i) = 2
end do
      ....
do i = 1, m
  a(i,2)=func_call(b(i))
end do
```

Shared

Example - Solution


```
      program global_data
              ....
      include "global_ok.h"
           ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
         ......
```

file global_ok.h

```
integer, parameter::
nthreads=4
common /work/a(m,n)
common /tprivate/b(m,nthreads)
```

```
subroutine suba(j)
   .....
include "global_ok.h"
   .....

TID = omp_get_thread_num()+1
do i = 1, m
   b(i,TID) = j
end do

do i = 1, m
   a(i,j)=func_call(b(i,TID))
end do

return
end
```

- ***By expanding array B, we can give each thread unique access to it's storage area***
- ***Note that this can also be done using dynamic memory (allocatable, malloc, ...)***

About global data

- 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

The threadprivate directive

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

```
!$omp threadprivate (/cb/ [,/cb/] ...)
```

- ❑ *Thread private copies of the designated global variables and common blocks are 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)*
    - ✔ *Oracle implementation supports changing the number of threads*
  - *Initial data is undefined, unless* **copyin** *is used*
  - *......*
- ❑ *Check the documentation when using threadprivate !*

Example - Solution 2

```
      program global_data
              ....
      include "global_ok2.h"
           ....
!$omp parallel do private(j)
      do j = 1, n
         call suba(j)
      end do
!$omp end parallel do
         ......
      stop
      end
```

file global_ok2.h

```
common /work/a(m,n)
common /tprivate/b(m)
!$omp
threadprivate(/tprivate/)
```

```
subroutine suba(j)
  .....
include "global_ok2.h"
  .....

do i = 1, m
   b(i) = j
end do

do i = 1, m
   a(i,j) = func_call(b(i))
end do

return
end
```

- ***The compiler creates thread private copies of array B, to give each thread unique access to it's storage area***
- ***Note that the number of copies is automatically adjusted to the number of threads***

The copyin clause

copyin (list)

- *Applies to THREADPRIVATE common blocks only*
- *At the start of the parallel region, data of the master thread is copied to the thread private copies*

Example:

```
       common /cblock/velocity
       common /fields/xfield, yfield, zfield

 ! create thread private common blocks

 !$omp threadprivate (/cblock/, /fields/)

 !$omp parallel          &
 !$omp default (private) &
 !$omp copyin ( /cblock/, zfield )
```

C++ and Threadprivate

- *As of OpenMP 3.0, it has been clarified where/how threadprivate objects are constructed and destructed*
- *Allow C++ static class members to be threadprivate*

```
class T {
  public:
  static int i;
  #pragma omp threadprivate(i)
  ...
};
```

OpenMP Runtime Routines

OpenMP Runtime Functions/1

Name	Functionality
omp_set_num_threads	Set number of threads
omp_get_num_threads	Number of threads in team
omp_get_max_threads	Max num of threads for parallel region
omp_get_thread_num	Get thread ID
omp_get_num_procs	Maximum number of processors
omp_in_parallel	Check whether in parallel region
omp_set_dynamic	Activate dynamic thread adjustment (but implementation is free to ignore this)
omp_get_dynamic	Check for dynamic thread adjustment
omp_set_nested	Activate nested parallelism (but implementation is free to ignore this)
omp_get_nested	Check for nested parallelism
omp_get_wtime	Returns wall clock time
omp_get_wtick	Number of seconds between clock ticks

C/C++ : Need to include file <omp.h>
Fortran : Add "use omp_lib" or include file "omp_lib.h"

OpenMP Runtime Functions/2

Name	*Functionality*
omp_set_schedule	*Set schedule (if “runtime” is used)*
omp_get_schedule	*Returns the schedule in use*
omp_get_thread_limit	*Max number of threads for program*
omp_set_max_active_levels	*Set number of active parallel regions*
omp_get_max_active_levels	*Number of active parallel regions*
omp_get_level	*Number of nested parallel regions*
omp_get_active_level	*Number of nested active par. regions*
omp_get_ancestor_thread_num	*Thread id of ancestor thread*
omp_get_team_size (level)	*Size of the thread team at this level*

C/C++ : Need to include file <omp.h>
Fortran : Add “use omp_lib” or include file “omp_lib.h”

OpenMP locking routines

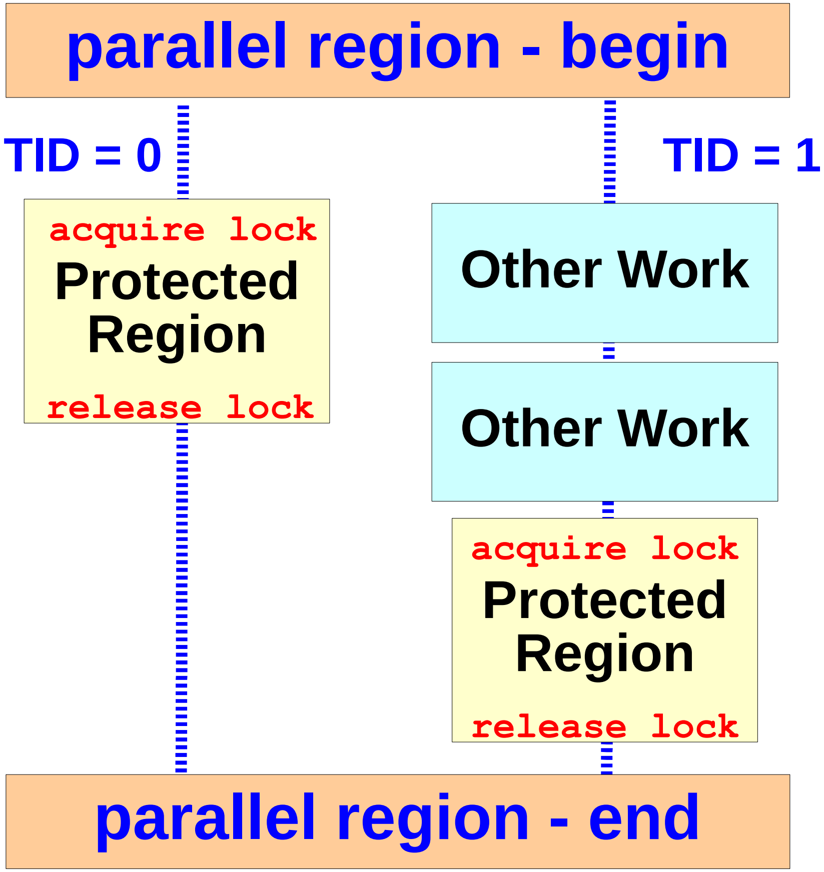
- ***Locks provide greater flexibility over critical sections and atomic updates:***
  - *Possible to implement asynchronous behavior*
  - *Not block structured*
- ***The so-called lock variable, is a special variable:***
  - *C/C++: type omp_lock_t and omp_nest_lock_t for nested locks*
  - *Fortran: type INTEGER and of a KIND large enough to hold an address*
- ***Lock variables should be manipulated through the API only***
- ***It is illegal, <u>and behavior is undefined</u>, in case a lock variable is used without the appropriate initialization***

Nested locking

- *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 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*

Locking Example - The Code

```
      Program Locks
            ....
      Call omp_init_lock (LCK)

!$omp parallel shared(LCK)

       Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
          Call Do_Something_Else()
       End Do

       Call Do_Work()

       Call omp_unset_lock (LCK)

!$omp end parallel

      Call omp_destroy_lock (LCK)

      Stop
      End
```

Initialize lock variable

Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

Example output for 2 threads

```
 TID:  1 at 09:07:27 => entered parallel region
 TID:  1 at 09:07:27 => done with WAIT loop and has the lock
 TID:  1 at 09:07:27 => ready to do the parallel work
 TID:  1 at 09:07:27 => this will take about 18 seconds
 TID:  0 at 09:07:27 => entered parallel region
 TID:  0 at 09:07:27 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:32 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:37 =>   WAIT for lock - will do something else for  5 seconds
 TID:  0 at 09:07:42 =>   WAIT for lock - will do something else for  5 seconds
 TID:  1 at 09:07:45 => done with my work
 TID:  1 at 09:07:45 => done with work loop - released the lock
 TID:  1 at 09:07:45 => ready to leave the parallel region
 TID:  0 at 09:07:47 => done with WAIT loop and has the lock
 TID:  0 at 09:07:47 => ready to do the parallel work
 TID:  0 at 09:07:47 => this will take about 18 seconds
 TID:  0 at 09:08:05 => done with my work
 TID:  0 at 09:08:05 => done with work loop - released the lock
 TID:  0 at 09:08:05 => ready to leave the parallel region
Done at 09:08:05 - value of SUM is 1100
```

Used to check the answer

Note: program has been instrumented to get this information

OpenMP Environment Variables

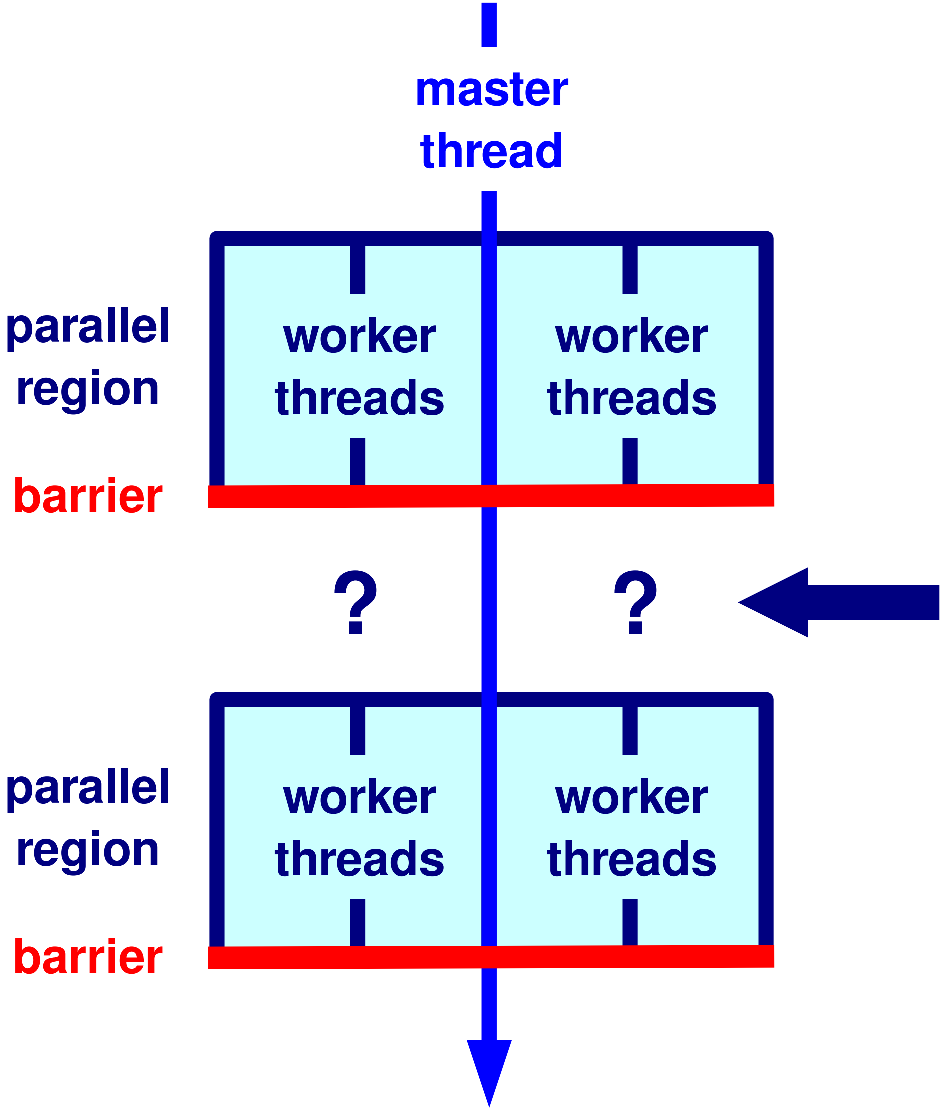
OpenMP Environment Variables

OpenMP environment variable	Default for Oracle Solaris Studio
OMP_NUM_THREADS n	1
OMP_SCHEDULE "schedule,[chunk]"	static, "N/P"
OMP_DYNAMIC { TRUE | FALSE }	TRUE
OMP_NESTED { TRUE | FALSE }	FALSE
OMP_STACKSIZE size [B|K|M|G]	4 MB (32 bit) / 8 MB (64-bit)
OMP_WAIT_POLICY [ACTIVE | PASSIVE]	PASSIVE
OMP_MAX_ACTIVE_LEVELS	4
OMP_THREAD_LIMIT	1024

Note:

The names are in uppercase, the values are case insensitive

Implementing the Fork-Join Model



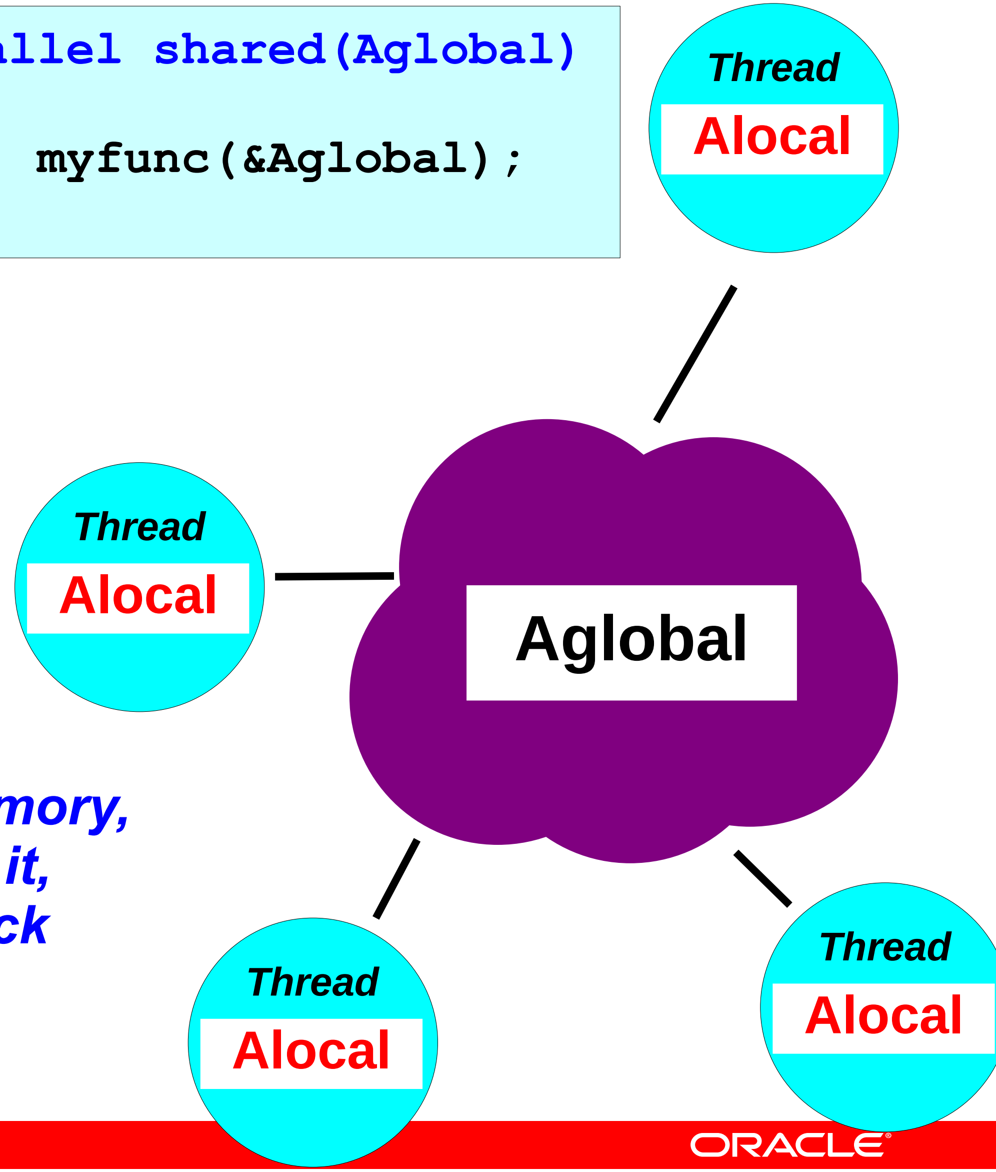
Use the OMP_WAIT_POLICY environment variable to control the behaviour of idle threads

About the Stack

```
#omp parallel shared(Aglobal)
{
    (void) myfunc(&Aglobal);
}
```

```
void myfunc(float *Aglobal)
{
    int Alocal;

    .......
}
```



Variable Alocal is in private memory, managed by the thread owning it, and stored on the so-called stack

Tasking In OpenMP

Tasking in OpenMP

- When any thread encounters a **task construct**, a new explicit task is generated
  - Tasks can be nested
- Execution of explicitly generated tasks is assigned to one of the threads in the current team
  - This is subject to the thread's availability and thus could be immediate or deferred until later
- Completion of the task can be guaranteed using a **task synchronization** construct

The Tasking Construct

Define a task:

```
#pragma omp task
```

```
!$omp task
```

*A **task** is a specific instance of executable code and its data environment, generated when a thread encounters a task construct or a parallel construct.*

*A **task region** is a region consisting of all code encountered during the execution of a task.*

*The **data environment** consists of all the variables associated with the execution of a given task. The data environment for a given task is constructed from the data environment of the generating task at the time the task is generated.*

Task Completion in OpenMP

- **Task completion** occurs when the end of the structured block associated with the construct that generated the task is reached
- Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of **task synchronization constructs**
  - Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits
- A task synchronization construct is a **taskwait** or a **barrier** construct

Task Completion

Explicitly wait on the completion of child tasks:

```
#pragma omp taskwait
```

```
!$omp flush taskwait
```

Example/1


```
$ cc -fast hello.c
$ ./a.out
A race car
$
```

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

          printf("A ");
          printf("race ");
          printf("car ");

   printf("\n");
   return(0);
}
```

What will this program print ?

Example/2

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
          printf("A ");
          printf("race ");
          printf("car ");

   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/3

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car
```

Note that this program could for example also print
"A A race race car car " or
"A race A car race car", or
"A race A race car car", ...
although I have not observed this (yet)

Example/4


```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
          printf("A ");
          printf("race ");
          printf("car ");
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/5

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
```

But now only 1 thread executes ....

Example/6


```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
        printf(“A “);
        #pragma omp task
          {printf("race ");}
        #pragma omp task
          {printf("car ");}
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/7

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
$ ./a.out
A race car
$ ./a.out
A car race
$
```

Tasks can be executed in arbitrary order

Example/8

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
        printf("A ");
        #pragma omp task
          {printf("race ");}
        #pragma omp task
          {printf("car ");}
        printf("is fun to watch ");
     }
   } // End of parallel region

   printf("\n");
   return(0);
}
```

What will this program print using 2 threads ?

Example/9

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch race car
$ ./a.out

A is fun to watch car race
$
```

Tasks are executed at a task execution point

Example/10

```
int main(int argc, char *argv[]) {

   #pragma omp parallel
   {
     #pragma omp single
     {
        printf(“A “);
        #pragma omp task
          {printf("car ");}
        #pragma omp task
          {printf("race ");}
        #pragma omp taskwait
        printf(“is fun to watch “);
     }
   } // End of parallel region

   printf("\n");retu
}
```

What will this program print using 2 threads ?

Example/11

```
$ cc -xopenmp -fast hello.c
$ export OMP_NUM_THREADS=2
$ ./a.out
$
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$
```

Tasks are executed first now

Clauses on the task directive

if(*scalar-expression*)	if false, create an undeferred task, encountering thread must suspend the encountering task region, resume execution of the current task region until the task is completed
untied	any task can resume after suspension
default(shared | none)	
private(*list*)	
firstprivate(*list*)	
shared(*list*)	
New in OpenMP 3.1:	
final(*scalar-expression*)	if true, the generated task is a final task
mergeable	if the task is an undeferred task or an included task, the implementation may generate a merged task

Task Scheduling Points in OpenMP/1

- Threads are allowed to suspend the current task region at a **task scheduling point** in order to execute a different task
  - If the suspended task region is for a **tied** task, the initially assigned thread resumes execution of the suspended task
  - If it is **untied**, any thread my resume its execution

Task Scheduling Points/2

- Whenever a thread reaches a **task scheduling point**, the implementation may cause it to perform a *task switch*, beginning or resuming execution of a different task bound to the current team
- Task scheduling points are implied at:
  - The point immediately following the generation of an explicit task
  - After the last instruction of a task region
  - In taskwait and taskyield regions
  - In implicit and explicit barrier regions
- In addition to this, the implementation may insert task scheduling points in untied tasks

Task Scheduling Points/3

- Task scheduling points dynamically divide task regions into parts
- When a thread encounters a task scheduling point, it may do of the following:
  – Begin execution of a tied task bound to the current team
  – Resume any suspended task region bound to the current team to which it is tied
  – Begin execution of an untied task bound to the current team
  – Resume any suspended untied task region bound to the current team
- If more than one of these choices is available, the behavior is undetermined

Tasking Examples Using OpenMP 3.0

Example - A Linked List

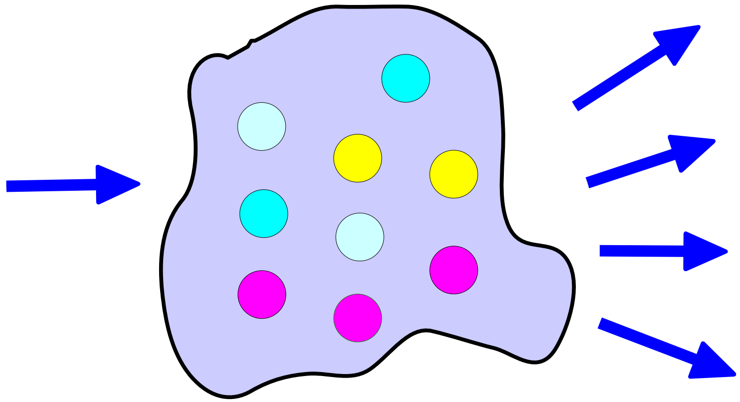
```
        ........
   while(my_pointer) {

     (void) do_independent_work (my_pointer);
     my_pointer = my_pointer->next ;
   } // End of while loop
        ........
```

Hard to do before OpenMP 3.0:
First count number of iterations, then convert while loop to for loop

The Tasking Example

Encountering thread adds task(s) to pool



Threads execute tasks in the pool

Developer specifies tasks in application
Run-time system executes tasks

Example - A Linked List With Tasking

OpenMP Task is specif ed here (executed in parallel)

```
my_pointer = listhead;

#pragma omp parallel
{
   #pragma omp single nowait
   {
      while(my_pointer) {
        #pragma omp task firstprivate(my_pointer)
        {
            (void) do_independent_work (my_pointer);
        }
        my_pointer = my_pointer->next ;
      }
   } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier
```

Example – Fibonacci Numbers

The Fibonacci Numbers are defined as follows:

```
F(0) = 1
F(1) = 1

F(n) = F(n-1) + F(n-2) (n=2,3,4,.....)
```

Sequence:

1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Recursive Algorithm*

```
long comp_fib_numbers(int n){
    // Basic algorithm: f(n) = f(n-1) + f(n-2)
    long fnm1, fnm2, fn;
    if ( n == 0 || n == 1 ) return(n);

    fnm1 = comp_fib_numbers(n-1);

    fnm2 = comp_fib_numbers(n-2);

    fn   = fnm1 + fnm2;
    return(fn);
}
```

****) Not very efficient, used for demo purposes only***

Parallel Recursive Algorithm

```
long comp_fib_numbers(int n){
   // Basic algorithm: f(n) = f(n-1) + f(n-2)
   long fnm1, fnm2, fn;
   if ( n == 0 || n == 1 ) return(n);
#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}
#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}
#pragma omp taskwait
   fn   = fnm1 + fnm2;
   return(fn);
}
```

Driver Program

```
#pragma omp parallel shared(nthreads)
{
   #pragma omp single nowait
   {

      result = comp_fib_numbers(n);

   } // End of single
} // End of parallel region
```

Parallel Recursive Algorithm - V2

```
long comp_fib_numbers(int n){
   // Basic algorithm: f(n) = f(n-1) + f(n-2)
   long fnm1, fnm2, fn;
   if ( n == 0 || n == 1 ) return(n);
   if ( n < 20 ) return(comp_fib_numbers(n-1) +
                        comp_fib_numbers(n-2));

#pragma omp task shared(fnm1)
   {fnm1 = comp_fib_numbers(n-1);}

#pragma omp task shared(fnm2)
   {fnm2 = comp_fib_numbers(n-2);}

#pragma omp taskwait
   fn   = fnm1 + fnm2;

   return(fn);
}
```

Performance Example*

```
$ export OMP_NUM_THREADS=1
$ ./fibonacci-omp.exe 40
Parallel   result for n = 40: 102334155 (1 threads
                              needed 5.63 seconds)
$ export OMP_NUM_THREADS=2
$ ./fibonacci-omp.exe 40
Parallel   result for n = 40: 102334155 (2 threads
                              needed 3.03 seconds)
$
```

***) MacBook Pro Core 2 Duo**

What's New In OpenMP 3.1 ?

About OpenMP 3.1

- A brand new update on the specifications
  - Following the evolutionary model
- Public comment phase ended May 1, 2011
  - Started February 2011
- It takes time for compilers to support any new standard
  - OpenMP 3.1 is no exception
- OpenMP continues to evolve!

New OpenMP 3.1 Features/1

- The "reduction" clause for C/C++ now supports the "min" and "max" operators
- Data environment for "firstprivate" extended to allow "intent(in)" in Fortran and const qualified types in C/C++
- Addition of "omp_in_final" runtime routine
  - Supports specialization of final or included task regions
- New OMP_PROC_BIND environment variable
- Corrections and clarifications
  - Incorrect use of "omp_integer_kind" in Fortran interfaces in Appendix D has been corrected
  - Description of some examples expanded and clarified

New OpenMP 3.1 Features/2

- Additions to tasking
  - The “mergeable” clauses
    - When a mergeabe clause is present on a task construct, and the generated task is undeferred or included, the implementation may generate a merged task instead
      - A merged task is a task whose data environment, inclusive of ICVs, is the same as that of its generating task region

Note: ICV = Internal Control Variable

New OpenMP 3.1 Features/3

- Additions to tasking
  - The “final” clause
    - If true, the generated task will be a final task
    - All tasks generated encountered during execution of a final task will generate included tasks
      - An included task is a task for which execution is sequentially included in the generating task region; that is, it is undeferred and executed immediately by the encountering thread

New OpenMP 3.1 Features/4

- Additions to tasking
  - The "taskyield" construct has been added
    - Current task can be suspended in favor of execution of another task
    - Allows user defined task switching points
  - This construct includes an explicit task scheduling point

```
#pragma omp taskyield
```

```
!$omp taskyield
```

New OpenMP 3.1 Features/5

- Enhancements for the “atomic” construct
  - New “update” clause
  - New “read”, “write” and “capture” forms
  - Disallow closely nested parallel regions within “atomic”
    - Clarification of existing restriction

Summary OpenMP

Summary OpenMP

- OpenMP provides for a small, but yet powerful, programming model
- It can be used on a shared memory system of any size
  - This includes a single socket multicore system
- Compilers with OpenMP support are widely available
- The tasking concept opens up opportunities to parallelize a wider range of applications
- OpenMP continues to evolve!
  - The new 3.1 specifications are again a step forward

Thank You And .... Stay Tuned !

ruud.vanderpas@oracle.com

Hardware and Software

Engineered to Work Together