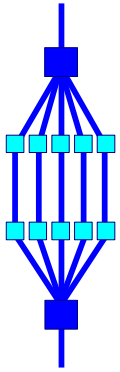


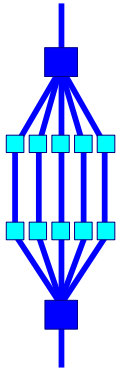
An Overview of OpenMP 3.0

Ruud van der Pas

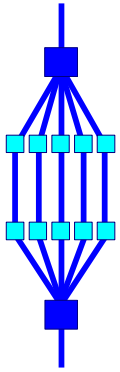
Senior Staff Engineer
Sun Microsystems
Menlo Park, CA, USA

IWOMP 2009
TU Dresden
Dresden, Germany
June 3-5, 2009

Outline



- *OpenMP Guided Tour*
- *OpenMP In-depth Overview*



OpenMP™

http://www.openmp.org

http://www.compunity.org

File Edit View History Bookmarks Tools Help

http://openmp.org/wp/

Google

Getting Started Latest Headlines Developer Guide

OpenMP.org

http://www.openmp.org

OpenMP™

THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING

RSS

What's Here:

» API Specs

- »About OpenMP.org
- »OpenMP Compilers
- »OpenMP Resources
- »OpenMP Forum

Input Register

Alert the OpenMP.org webmaster about new products or updates and we'll post it here.

»webmaster@openmp.org

Search OpenMP.org

Google™ Custom Search

Search

Archives

- June 2008
- May 2008
- April 2008

Admin

- Log in

OpenMP News

»Christian's First Experiments with Tasking in OpenMP 3.0

From Christian Terboven's blog:

> OpenMP 3.0 is out, maybe a bit later than we hoped for, but I think that we got a solid standard document. At IWOMP 2008 a couple of weeks ago, there was an OpenMP tutorial which included a talk by Alex Duran (from UPC in Barcelona, Spain) on what is new in OpenMP 3.0 - which is really worth a look! My talk was on some OpenMP application experiences, including a case study on Windows, and I really think that many of our codes can profit from Tasks. Motivated by Alex' talk I tried the updated Nanos compiler and prepared a couple of examples for my lectures on Parallel Programming in Maastricht and Aachen. In this post I am walking through the simplest one: Computing the Fibonacci number in parallel.

Read more…

Posted on June 6, 2008

»New Forum Created

The **OpenMP 3.0 API Specifications forum** is now open for discussing the specs document itself.

Posted on May 31, 2008

»New Links

New links and information have been added to the **OpenMP Compilers** and the **OpenMP Resources** pages.

Posted on May 23, 2008

»Recent Forum Posts

- **strange behavior of C function strcmp() With OPENMP**
- **virtual destructor not called with first private clause**

OpenMP.org

The OpenMP Application Program Interface (API) supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

»Read about OpenMP

Get It

»OpenMP specs

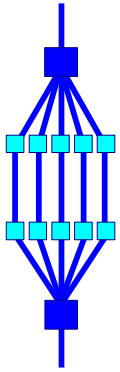
Use It

»OpenMP Compilers

Learn It

Done

Shameless Plug - "Using OpenMP"



"Using OpenMP"

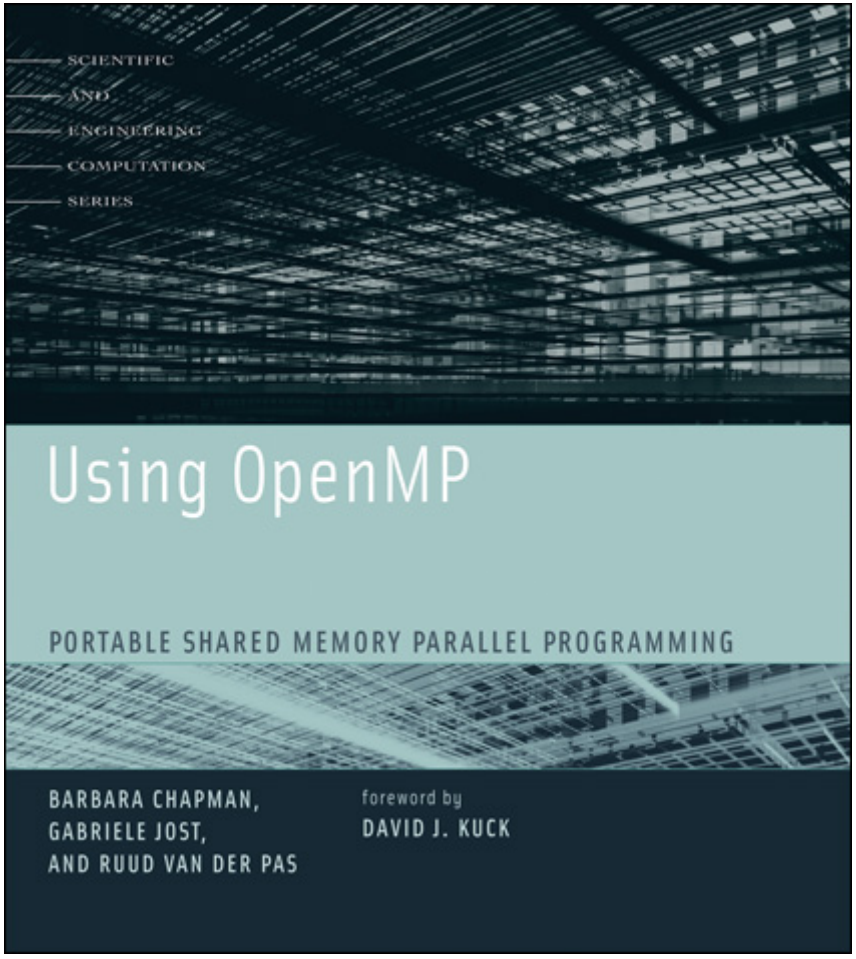
Portable Shared Memory Parallel Programming

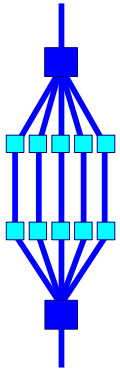
Chapman, Jost, van der Pas

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 News

»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.

You are encouraged to try out these examples and perhaps use them as a starting

book.

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.

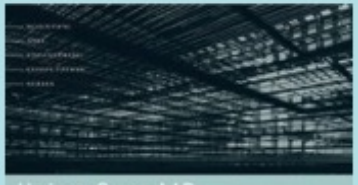
Also, we have created a new forum, **»Using OpenMP - The Book and Examples**, for discussion and feedback.

The OpenMP API supports multi-platform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model with a simple and flexible interface for developing parallel applications on platforms from the desktop to the supercomputer.

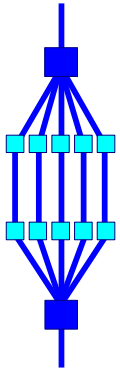
Download the examples and discuss in forum:
http://www.openmp.org/wp/2009/04/download-book-examples-and-discuss

»OpenMP Compilers

Learn

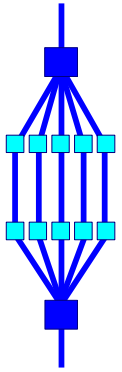


What is OpenMP?



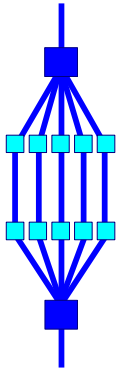
- *De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran*
- *Consists of:*
  - *Compiler directives*
  - *Run time routines*
  - *Environment variables*
- *Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)*
- *Version 3.0 has been released May 2008*

When to consider OpenMP?



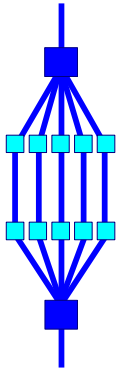
- ***The compiler may not be able to do the parallelization in the way you like to see it:***
  - ***It can not find the parallelism***
    - ✔ ***The data dependence analysis is not able to determine whether it is safe to parallelize or not***
  - ***The granularity is not high enough***
    - ✔ ***The compiler lacks information to parallelize at the highest possible level***
- ***This is when explicit parallelization through OpenMP directives comes into the picture***

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

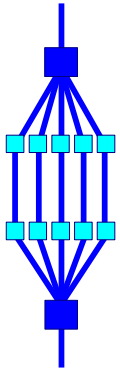
Memory and threading model map naturally

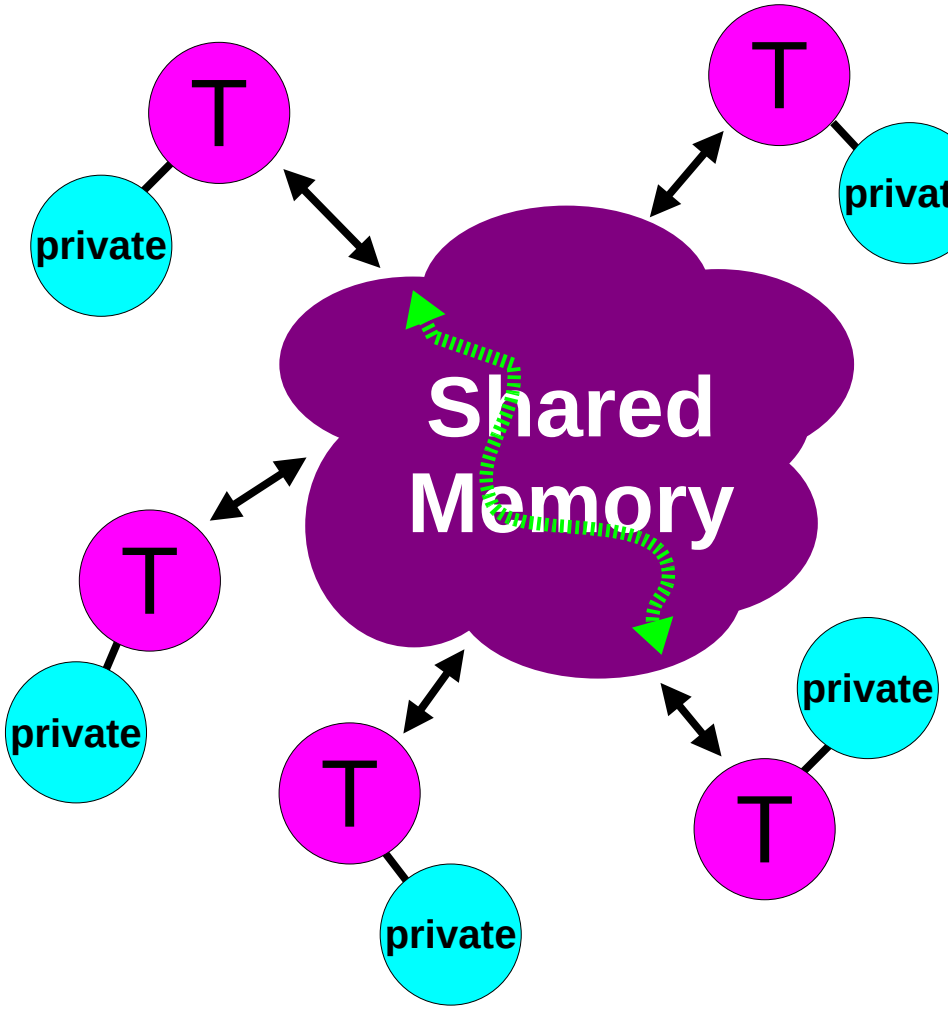
Lightweight

Mature

Widely available and used

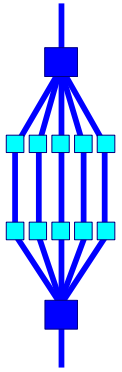
The OpenMP Memory Model





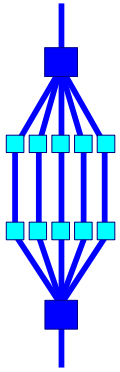
- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can only be accessed by the thread that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place,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

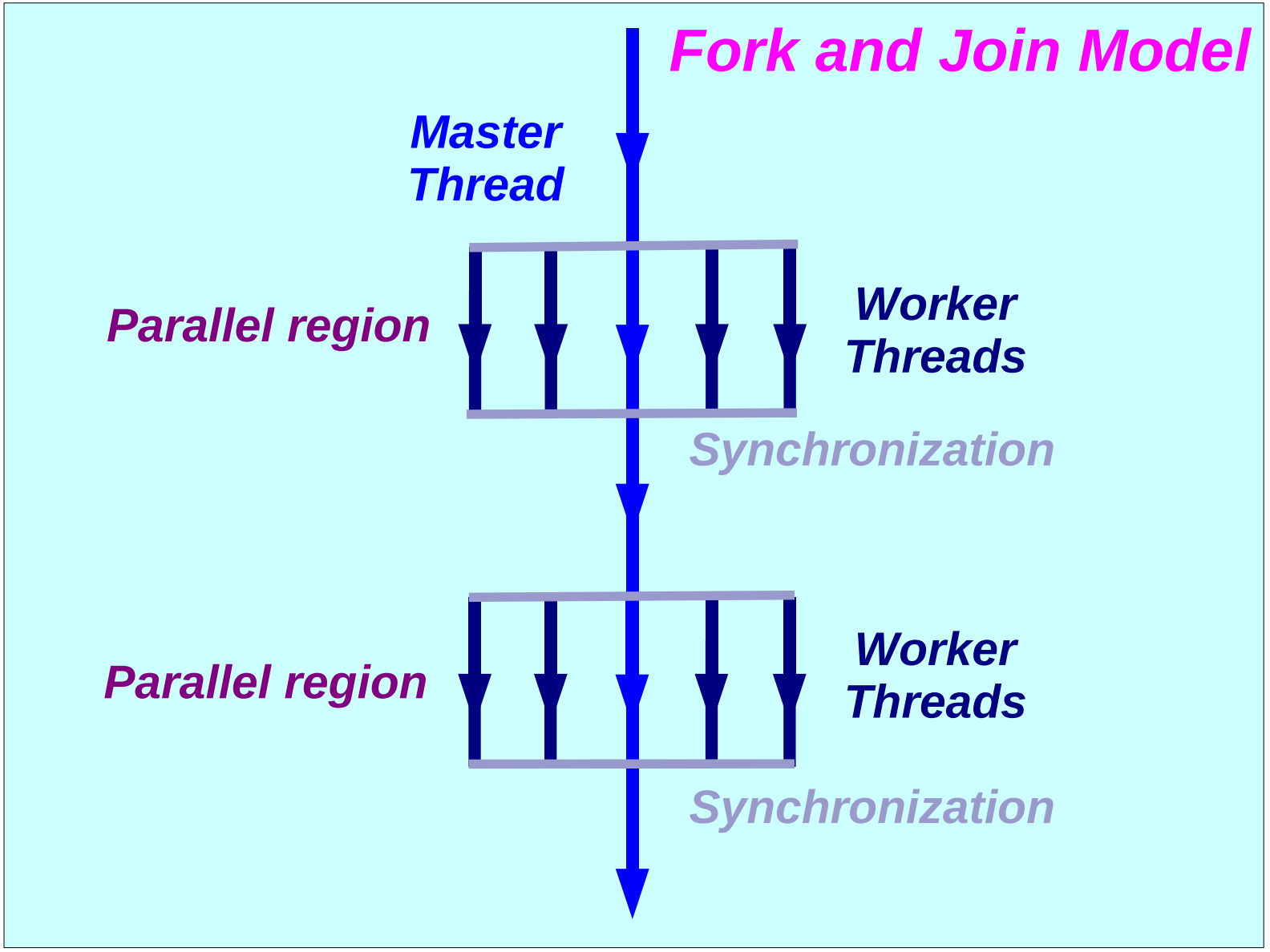
Data-Sharing Attributes



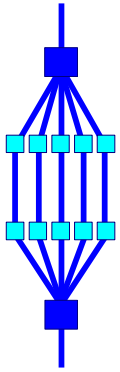
- *In an OpenMP program, data needs to be "labelled"*
- *Essentially there are two basic types:*
  - *Shared*
    - ✔ *There is only instance of the data*
    - ✔ *All threads can read and write the data simultaneously, unless protected through a specific OpenMP construct*
    - ✔ *All changes made are 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*

The OpenMP Execution Model





An OpenMP example



For-loop with independent iterations

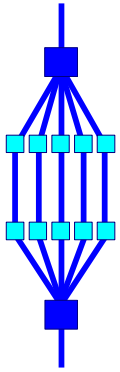
```
for (int i=0; i<n; i++)
  c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

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

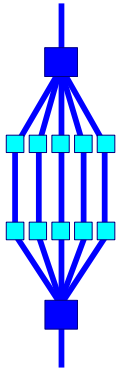
```
% cc -xopenmp source.c
% setenv OMP_NUM_THREADS 5
% a.out
```

Example parallel execution



Thread 0	Thread 1	Thread 2	Thread 3	Thread 4
i=0-199	i=200-399	i=400-599	i=600-799	i=800-999
a[i]	a[i]	a[i]	a[i]	a[i]
+	+	+	+	+
b[i]	b[i]	b[i]	b[i]	b[i]
=	=	=	=	=
c[i]	c[i]	c[i]	c[i]	c[i]

Components of OpenMP 2.5



Directives

- Parallel region
- Worksharing
- Synchronization
- Data-sharing attributes
  - private
  - firstprivate
  - lastprivate
  - shared
  - reduction
- Orphaning

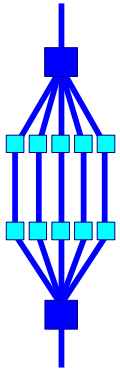
Runtime environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Wallclock timer
- Locking

Environment variables

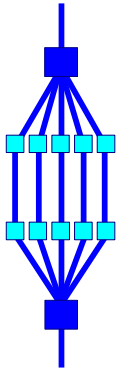
- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism

Added with OpenMP 3.0

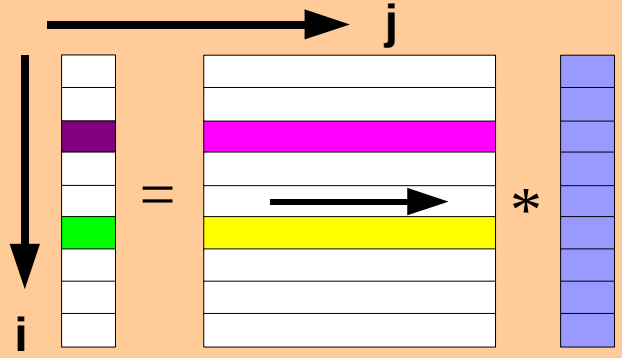


Directives	Runtime environment	Environment variables
◆ *Tasking*	◆ *Schedule* ◆ *Active levels* ◆ *Thread limit* ◆ *Nesting level* ◆ *Ancestor thread* ◆ *Team size*	◆ *Stacksize* ◆ *Idle threads* ◆ *Active levels* ◆ *Thread limit*

Example - Matrix times vector



```
#pragma omp parallel for default(none) \
            private(i,j,sum) shared(m,n,a,b,c)
for (i=0; i<m; i++)
{
   sum = 0.0;
   for (j=0; j<n; j++)
     sum += b[i][j]*c[j];
   a[i] = sum;
 }
```



TID = 0

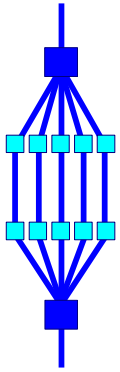
```
for (i=0,1,2,3,4)
 i = 0
sum = Σ  b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum = Σ  b[i=1][j]*c[j]
   a[1] = sum
```

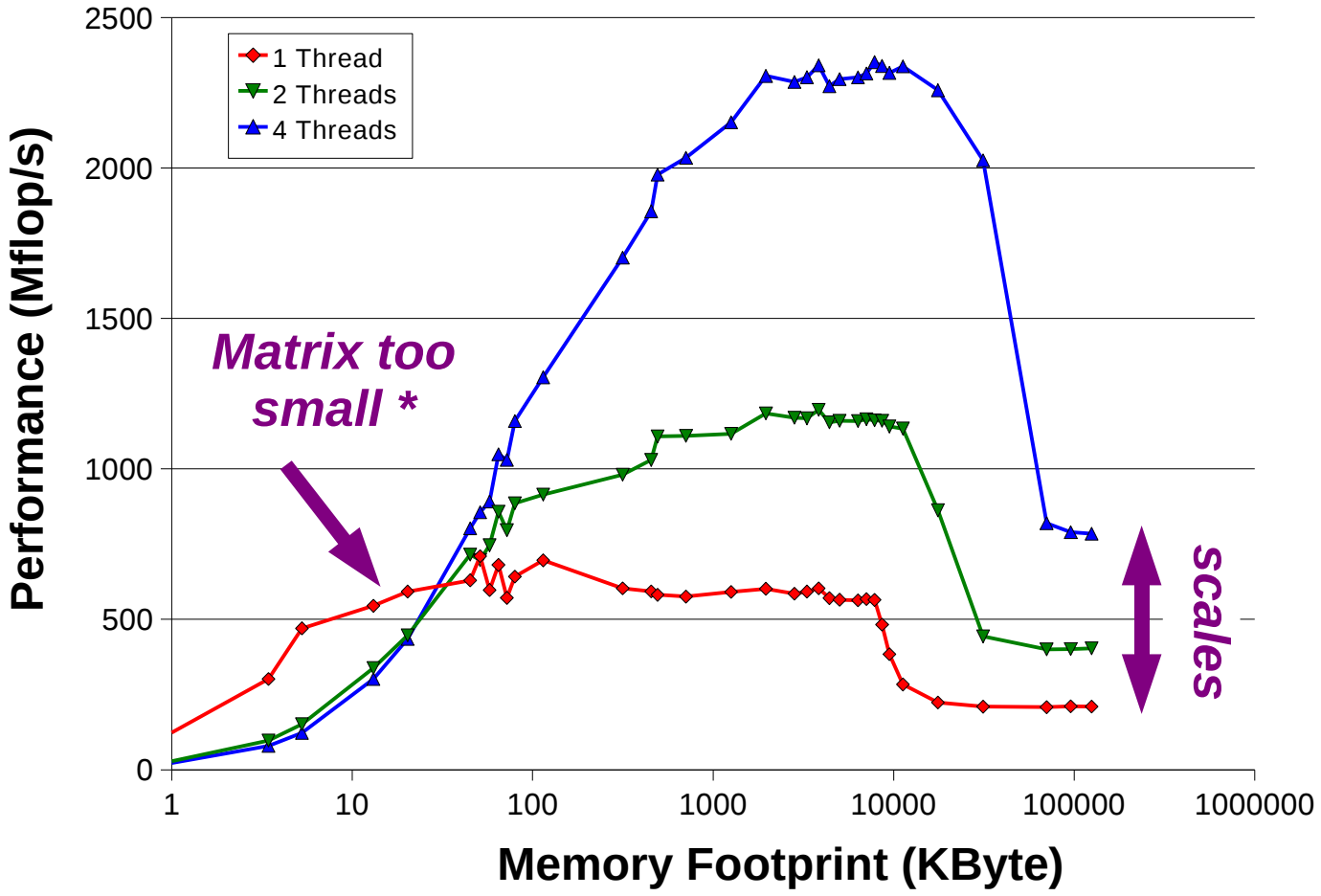
TID = 1

```
for (i=5,6,7,8,9)
 i = 5
sum = Σ  b[i=5][j]*c[j]
   a[5] = sum
 i = 6
sum = Σ  b[i=6][j]*c[j]
   a[6] = sum
```

... etc ...

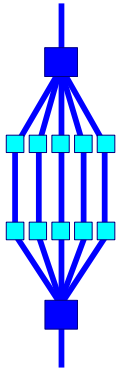
OpenMP Performance Example





**) With the IF-clause in OpenMP this performance degradation can be avoided*

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

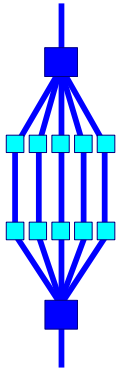
parallel loop
(work is distributed)

parallel loop
(work is distributed)

synchronization

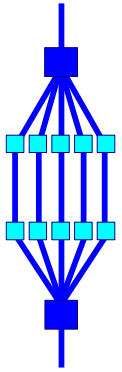
Statement is executed by all threads

parallel region



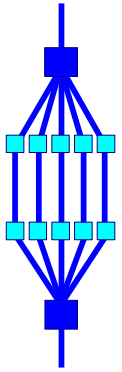
OpenMP Overview

Terminology and behavior



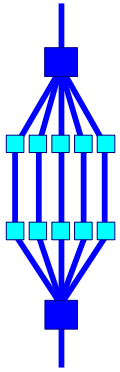
- *OpenMP Team := Master + Workers*
- *A Parallel Region is a block of code executed by all threads simultaneously*
  - *The master thread always has thread ID 0*
  - *Thread adjustment (if enabled) is only done before entering a parallel region*
  - *Parallel regions can be nested, but support for this 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*

About OpenMP clauses



- *Many OpenMP directives support clauses*
- *These clauses are used to specify additional information with the directive*
- *For example,* ***private(a)*** *is a clause to the for directive:*
  - **#pragma omp for private(a)**
- *Before we present an overview of all the directives, we discuss several of the OpenMP clauses first*
- *The specific clause(s) that can be used, depends on the directive*

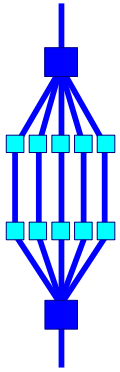
Directive format



- *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 if/private/shared clauses



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 --*/
```

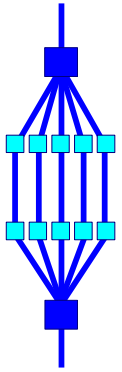
private (list)

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

shared (list)

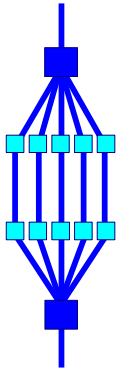
- *Data is accessible by all threads in the team*
- *All threads access the same address space*

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 first/last private clause to override this behavior***
- ***We illustrate these concepts with an example***

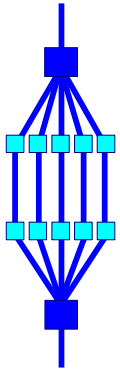
Example private variables



```
main()
{
  A = 10;

#pragma omp parallel
{
 #pragma omp for private(i) firstprivate(A) lastprivate(B)...
  for (i=0; i<n; i++)
  {
      ....
      B = A + i;        /*-- A undefined, unless declared
                             firstprivate --*/

      ....
  }

  C = B;                /*-- B undefined, unless declared
                             lastprivate --*/

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

Disclaimer: This code fragment is not very meaningful and only serves to demonstrate the clauses

The first/last private clauses



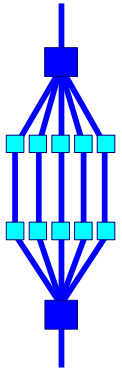
firstprivate (list)

- ✔ ***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***

The default clause



default (none | shared | private)

default (none | shared)

Fortran

C/C++

Note: default(private) is not supported in C/C++

none

- ✔ *No implicit defaults*
- ✔ *Have to scope all variables explicitly*

shared

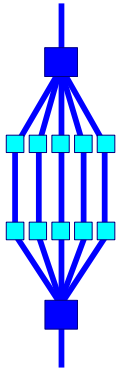
- ✔ *All variables are shared*
- ✔ *The default in absence of an explicit "default" clause*

private

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

Additional default clause

3.0



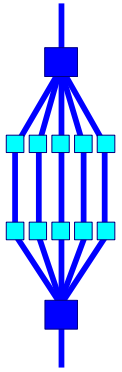
default (firstprivate)

Fortran only

firstprivate

- ✔ *All variables are private to the thread*
- ✔ *Pre-initialized*

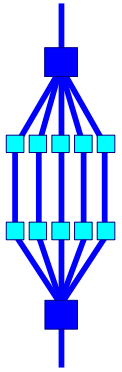
Fortran - Allocatable Arrays



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

```
      PARAMETER (n = 200)
      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
```

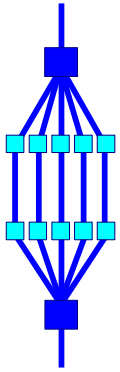
C++: Threadprivate



- *Clarify 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)
  ...
};
```

The reduction clause - Example

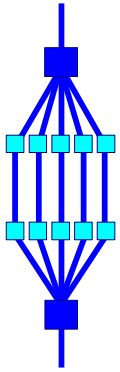


```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

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 | intrinsic]) : list) *Fortran*

reduction (operator : list) *C/C++*

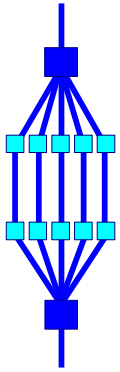
- ✔ *Reduction variable(s) must be shared variables*
- ✔ *A reduction is defined as:*

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*

Barrier/1



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

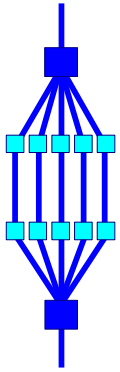
```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

```
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 updated all of a[ ] first, before using a[ ] *

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

wait !

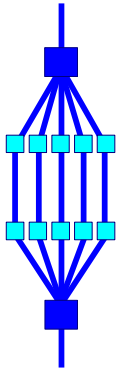
barrier

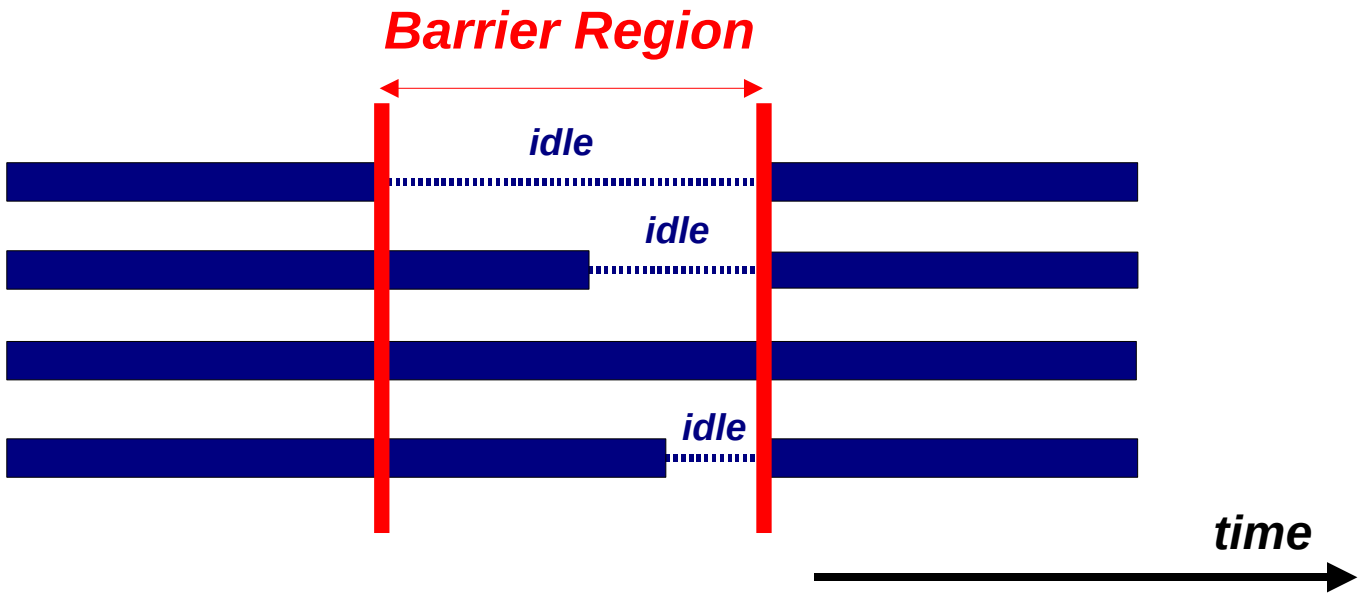
```
for (i=0; i < N; i++)
  d[i] = a[i] + b[i];
```

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

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

Barrier/3



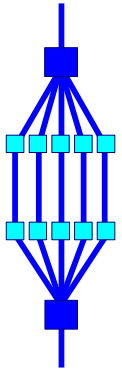


Barrier syntax in OpenMP:

```
#pragma omp barrier
```

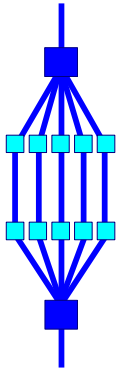
```
!$omp barrier
```

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

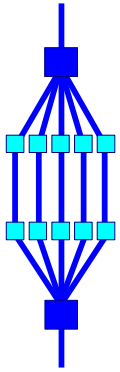


- *To minimize synchronization, some OpenMP directives/ pragmas support the optional* **nowait** *clause*
- *If present, threads do not synchronize/wait at the end of that particular construct*
- *In Fortran the* **nowait** *clause is appended at the closing part of the construct*
- *In C, it is one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

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

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

The Parallel Region

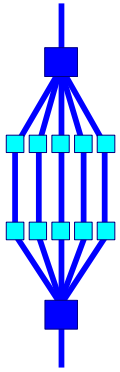


A parallel region is a block of code executed by multiple threads simultaneously

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

   "this is executed in parallel"

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

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

} (implied barrier)
```

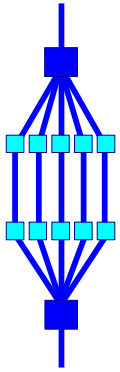
The Parallel Region - Clauses



A parallel region supports the following clauses:

if	***(scalar expression)***	
private	***(list)***	
shared	***(list)***	
default	***(none|shared)***	***(C/C++)***
default	***(none|shared|private/firstprivate)***	***(Fortran)***
reduction	***(operator: list)***	**3.0**
copyin	***(list)***	
firstprivate	***(list)***	
num_threads	***(scalar_int_expr)***	

Work-sharing constructs



The OpenMP work-sharing constructs

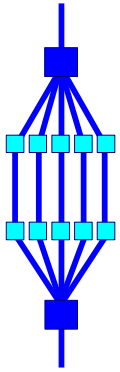
```
#pragma omp for
{
   ....
}

!$OMP DO
   ....
!$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 specified)*
- *A work-sharing construct does not launch any new threads*

The workshare construct



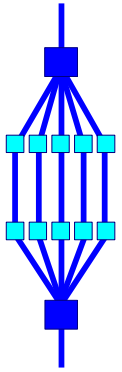
Fortran has a fourth worksharing construct:

```
!$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



The iterations of the loop are distributed over the threads

```
#pragma omp for [clause[[,] clause] ...]
   <original for-loop>
```

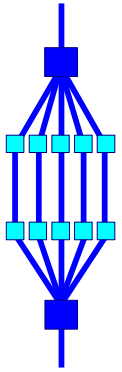
```
!$omp do [clause[[,] clause] ...]
        <original do-loop>
!$omp end do [nowait]
```

Clauses supported:

private **firstprivate**
lastprivate **reduction**
ordered* ***schedule*** ← *covered later*
nowait
collapse 3.0

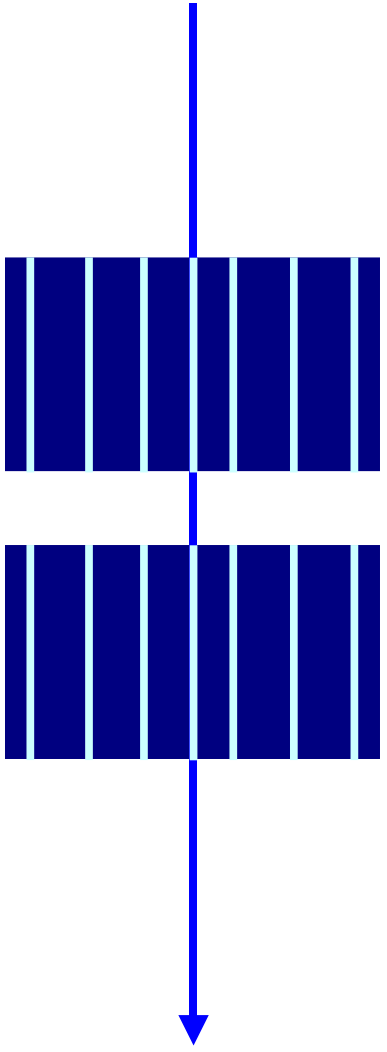
) Required if ordered sections are in the dynamic extent of this construct

The omp for directive - Example

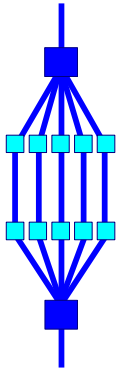


```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
      for (i=0; i<n-1; i++)
          b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
      for (i=0; i<n; i++)
          d[i] = 1.0/c[i];

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

(implied barrier)



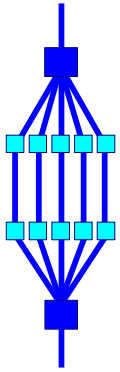
C++: Random Access Iterator Loops



- *Allow parallelization of random access iterator loops*


```
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 //
  }
}
```

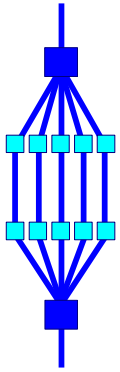
Loop Collapse



- *Allows parallelization of perfectly nested loops without using nested parallelism*
- **collapse** *clause on for/do loop indicates how many loops should be collapsed*
- *Compiler should form a single loop and then parallelize that*

```
!$omp parallel do collapse(2)
      do i = il, iu, is
         do j = jl, ju, js
            do k = k1, ku, ks
               ...
            end do
         end do
      end do
```

The sections directive



The individual code blocks are distributed over the threads

```
#pragma omp sections [clause(s)]
{
#pragma omp section
          <code block1>
#pragma omp section
          <code block2>
#pragma omp section
             :
}
```

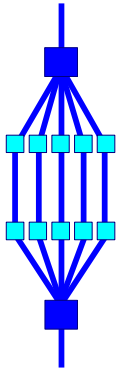
```
!$omp sections [clause(s)]
!$omp section
          <code block1>
!$omp section
          <code block2>
!$omp section
             :
!$omp end sections [nowait]
```

Clauses supported:

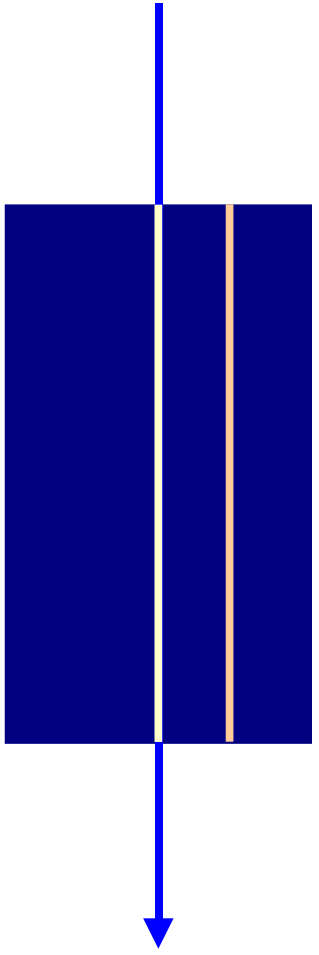
private **firstprivate**
lastprivate **reduction**
nowait

Note: The SECTION directive must be within the lexical extent of the SECTIONS/END SECTIONS pair

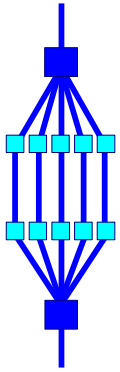
The sections directive - Example

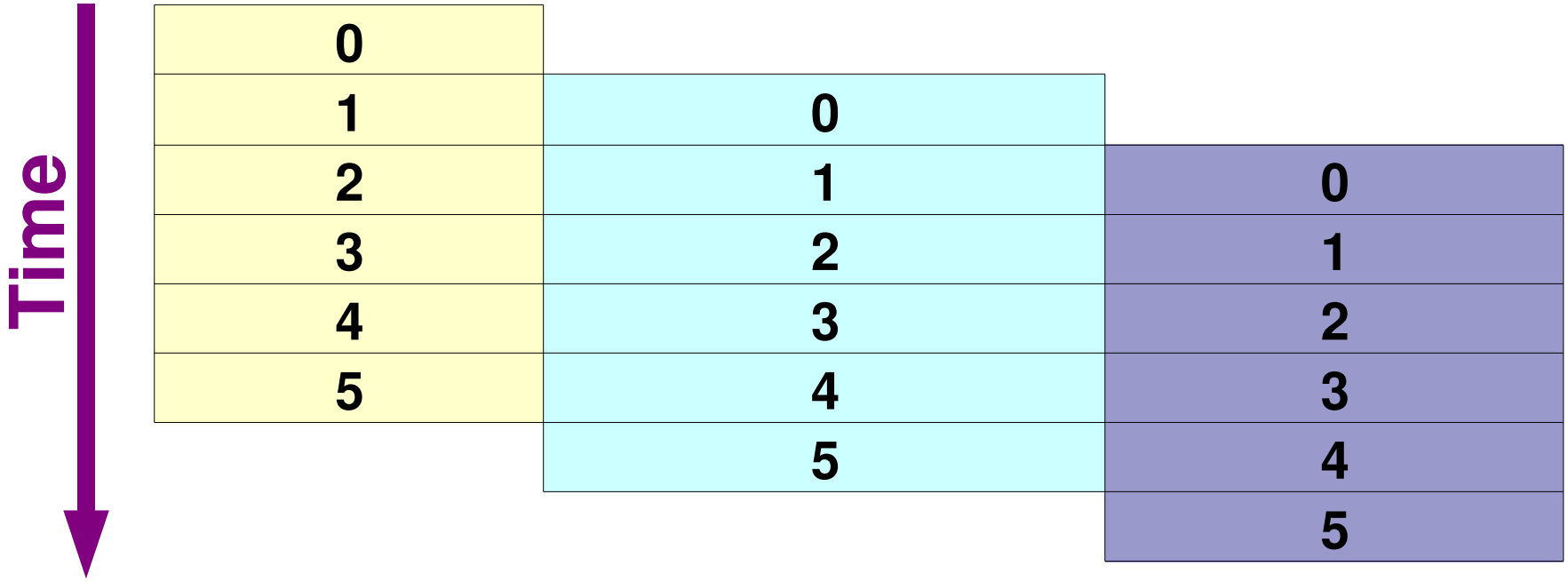


```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp sections nowait
    {
      #pragma omp section
        for (i=0; i<n-1; i++)
            b[i] = (a[i] + a[i+1])/2;

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

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

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



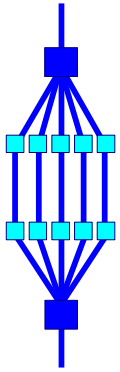
Overlap I/O and Processing/1





A complete example program, implementing this idea, can be found on http://www.openmp.org as part of the "Using OpenMP" example set

Overlap I/O and Processing/2



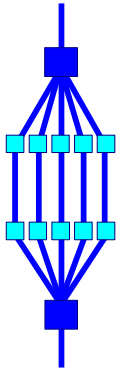
```
#pragma omp parallel sections
{
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
        (void) read_input(i);
        (void) signal_read(i);
     }
   }
   #pragma omp section
   {
     for (int i=0; i<N; i++) {
        (void) wait_read(i);
        (void) process_data(i);
        (void) 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

Single processor region/1



This construct is ideally suited for I/O or initializations

Original Code

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

"declare A to be be shared"

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

one volunteer requested

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

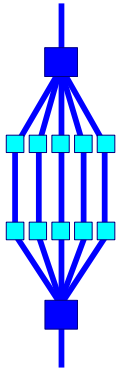
thanks, we're done

```
      .....
}
```

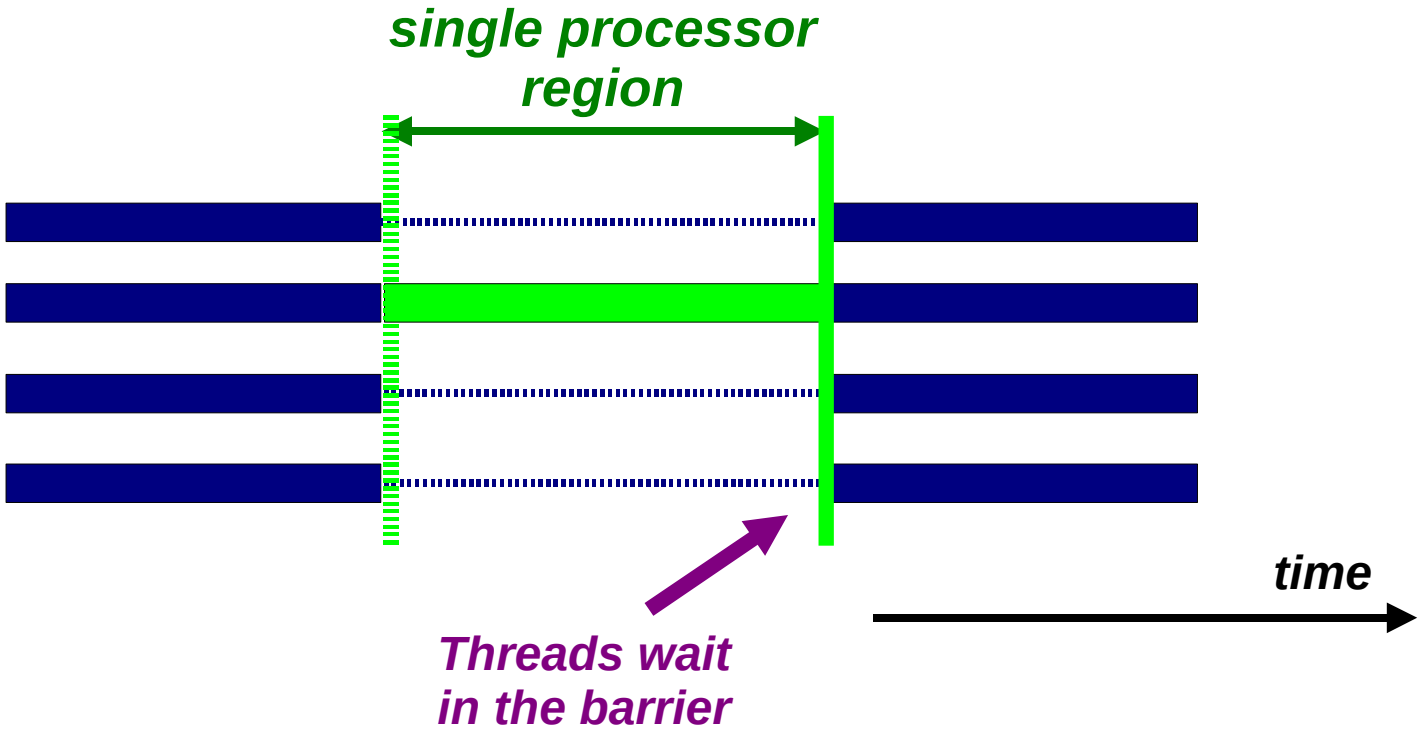
Parallel Version

May have to insert a barrier here

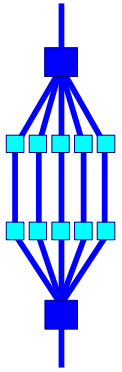
Single processor region/2



- *Usually, there is a barrier at the end of the region*
- *Might therefore be a scalability bottleneck (Amdahl's law)*



SINGLE and MASTER construct



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]
```

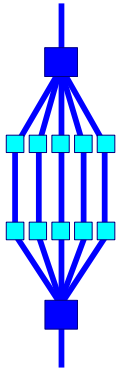
Only the master thread executes the code block:

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

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

There is no implied barrier on entry or exit !

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
{ ... }
```

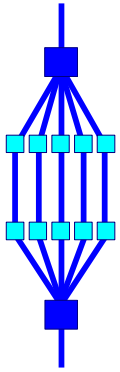
Single PARALLEL sections

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

→

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

Orphaning



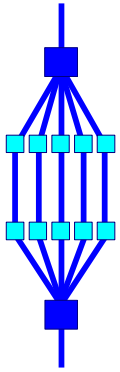
```
          :
#pragma omp parallel
{
          :
      (void) dowork();
          :
}
          :
```

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

orphaned work-sharing directive

- *The OpenMP specification 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

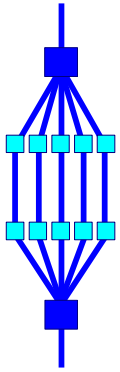


```
(void) dowork(); !- Sequential FOR

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

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

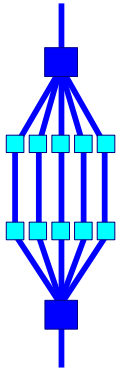
- ***When an orphaned worksharing or synchronization directive is encountered in the sequential part of the program (outside the dynamic extent of any parallel region), it is executed by the master thread only. In effect, the directive will be ignored***

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

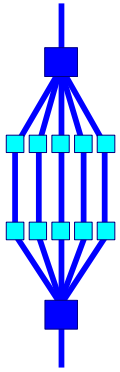
Should behave identically

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



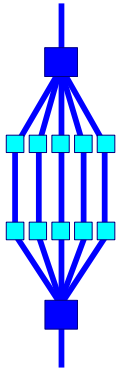
```
#pragma omp parallel for private(i) shared(m,c,..)
  for (i=0; i<n; i++) /* Parallel loop */
  {
      (void) FuncPar(i,m,c,...)
  } /*-- End of parallel for --*/
```

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>
    }
         ......
}
```

Critical Region/1



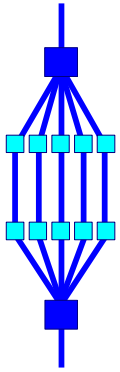
If sum is a shared variable, this loop can not run in parallel

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

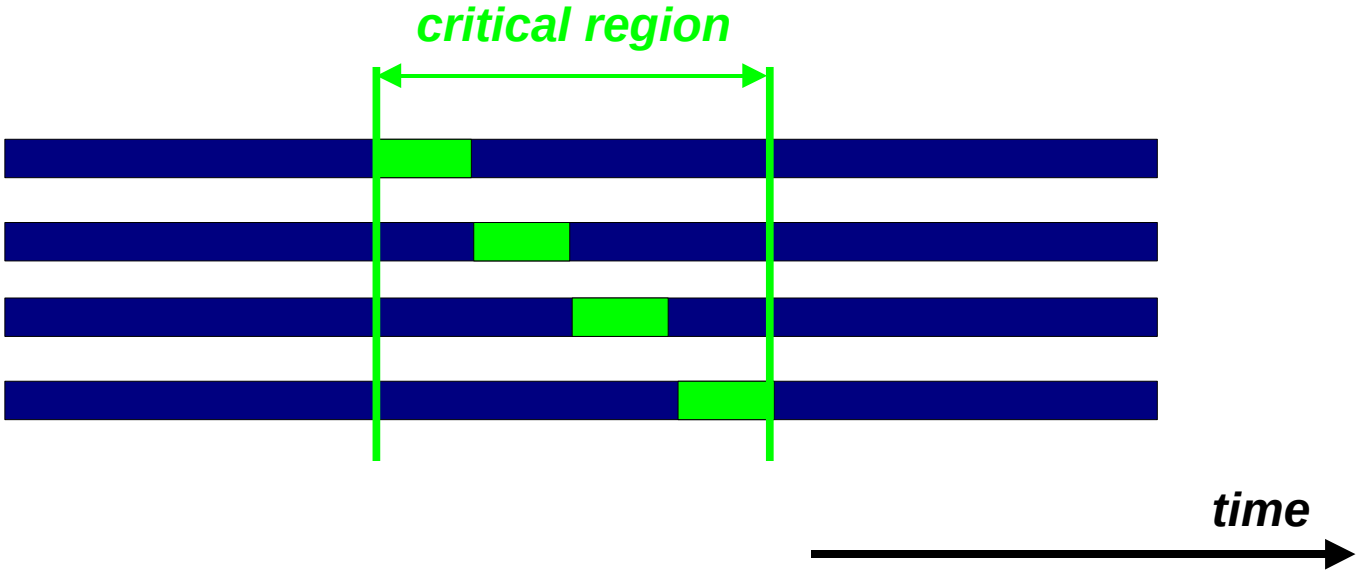
We can use a critical region for this:

```
for (i=0; i < N; i++){
      .....                    one at a time can proceed
   sum += a[i];
      .....                    next in line, please
}
```

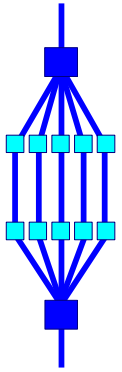
Critical Region/2



- *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 ....

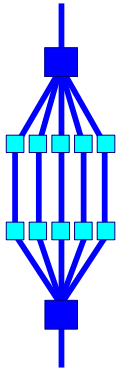
```
#pragma omp atomic
   <statement>
```

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

This is a lightweight, special form of a critical section

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

More synchronization constructs



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)

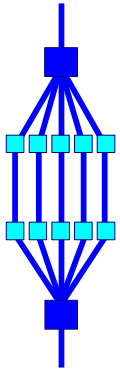
Ensure that all threads in a team have a consistent view of certain objects in memory:

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

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

In the absence of a list, all visible variables are flushed

The schedule clause/1



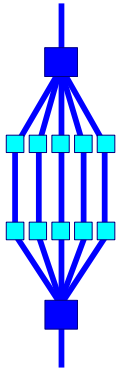
schedule (static | dynamic | guided | auto [, chunk])
schedule (runtime)

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***

3.0

The schedule clause/2



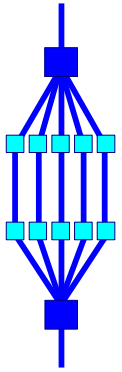
Example static schedule

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 defined*

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*

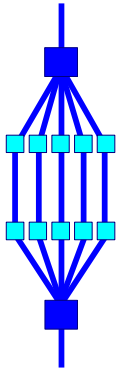
guided [, chunk]

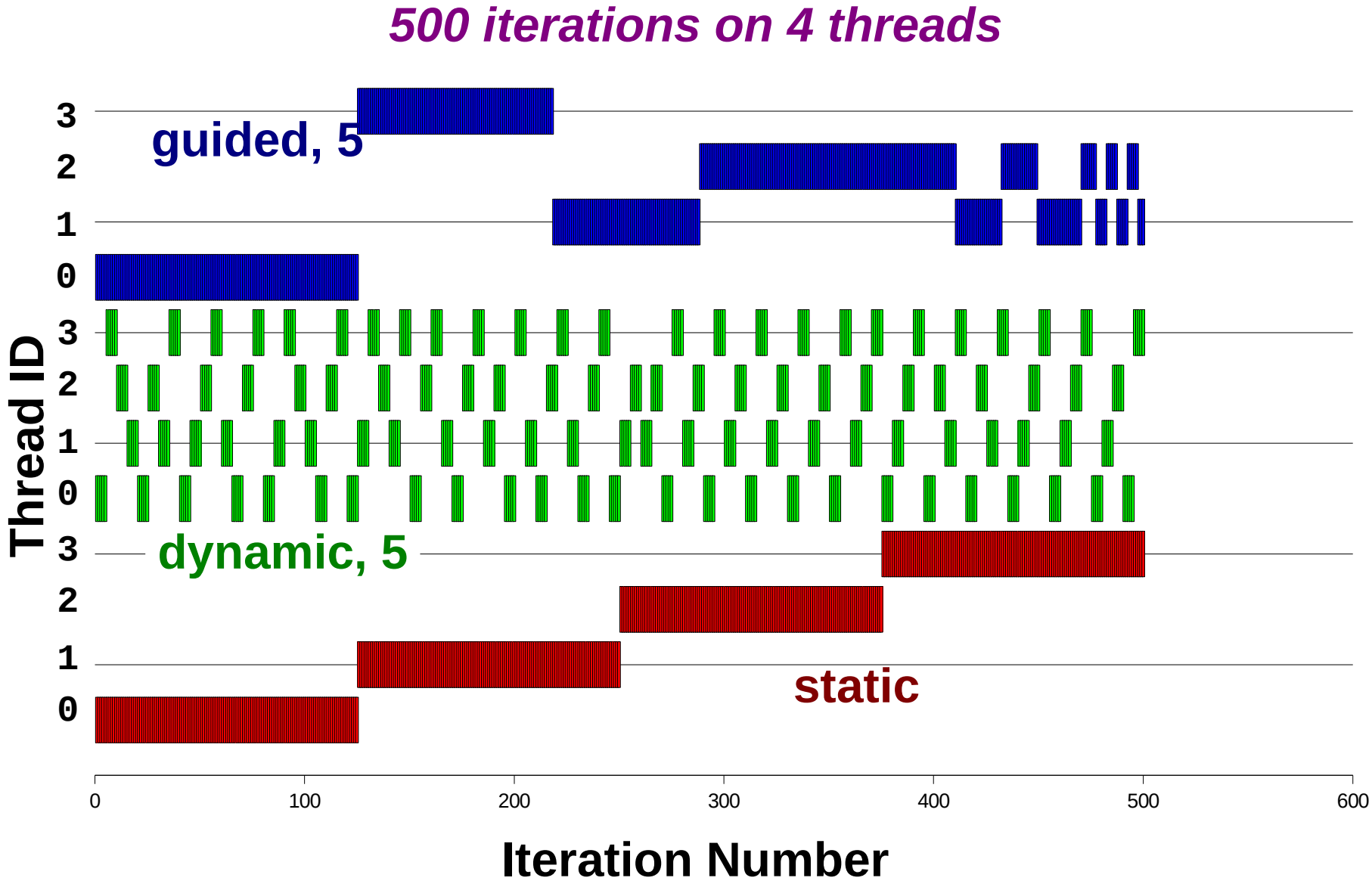
- ✔ *Same dynamic behavior as "dynamic", but size of the portion of work decreases exponentially*

runtime

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

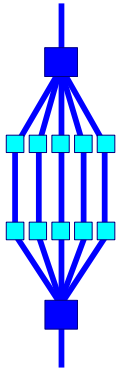
The experiment





Additional schedule clause

3.0

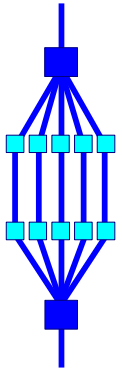


auto

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

Schedule Kinds

3.0



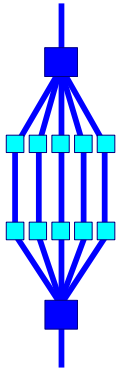
- *Made* `schedule(runtime)` *more useful*
  - *Can set/get it with library routines*

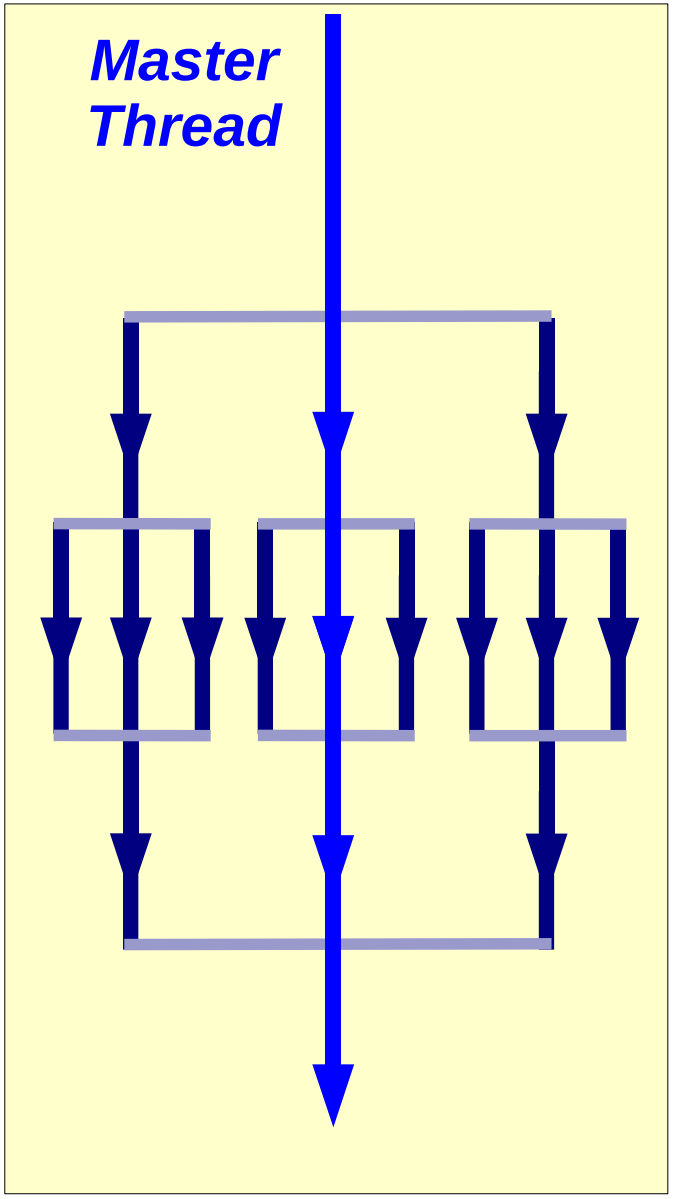
    `omp_set_schedule()`
    `omp_get_schedule()`

  - *Allow implementations to add their own schedule kinds*
- *Added a new schedule kind auto which gives full freedom to the implementation to determine the scheduling of iterations to threads*

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

  {.....}
```

Nested Parallelism





3-way parallel

9-way parallel

3-way parallel

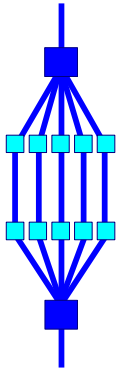
Outer parallel region

Nested parallel region

Outer parallel region

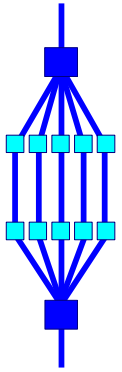
Note: nesting level can be arbitrarily deep

Improved Nesting Support



- *Better support for nested parallelism*
- *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)`

Improved Nesting Support



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

  `OMP_MAX_ACTIVE_LEVELS`

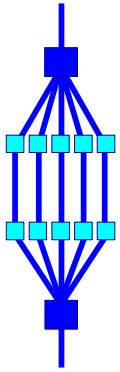
  `omp_set_max_active_levels()`

  `omp_get_max_active_levels()`

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

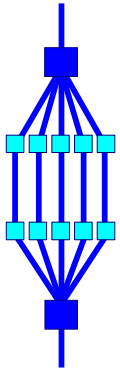
  `OMP_THREAD_LIMIT`

  `omp_get_thread_limit()`



OpenMP Environment Variables

OpenMP Environment Variables



OpenMP environment variable	Default for Sun OpenMP
OMP_NUM_THREADS n	1
OMP_SCHEDULE "schedule,[chunk]"	static, "N/P" (1)
OMP_DYNAMIC { TRUE | FALSE }	TRUE (2)
OMP_NESTED { TRUE | FALSE }	FALSE (3)

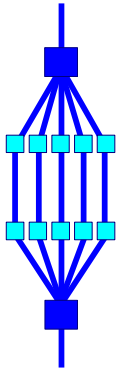
(1) The chunk size approximately equals the number of iterations (N) divided by the number of threads (P)

(2) The number of threads is limited to the number of on-line processors in the system. This can be changed by setting OMP_DYNAMIC to FALSE.

(3) Multi-threaded execution of inner parallel regions in nested parallel regions is supported as of Sun Studio 10

Note: The names are in uppercase, the values are case insensitive

Additional Environment Variables

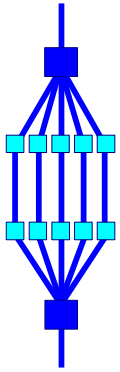


OpenMP environment variable	Default for Sun OpenMP
OMP_STACKSIZE size [B|K|M|G]	4 MB (32 bit) / 8 MB (64-bit)
OMP_WAIT_POLICY [ACTIVE | PASSIVE]	PASSIVE
OMP_MAX_ACTIVE_LEVELS	4
OMP_THREAD_LIMIT	1024

(1) The default unit for the stack size is KBytes

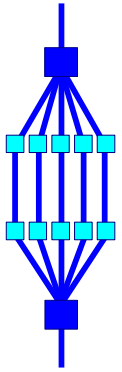
(2) With Sun's OpenMP implementation, idle threads may spin-wait for a short while first, before switching to sleep mode

Note: The names are in uppercase, the values are case insensitive



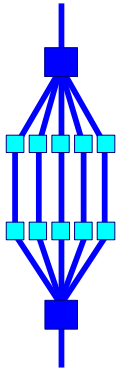
OpenMP Run-time Environment

OpenMP run-time environment



- *OpenMP provides several user-callable functions*
  - *To control and query the parallel environment*
  - *General purpose semaphore/lock routines*
    - ✔ *OpenMP 2.0: supports nested locks*
    - ✔ *Nested locks are not covered in detail here*
- *The run-time functions take precedence over the corresponding environment variables*
- *Recommended to use under control of an #ifdef for _OPENMP (C/C++) or conditional compilation (Fortran)*
- *C/C++ programs need to include* ***<omp.h>***
- *Fortran: may want to use* ***"USE omp_lib"***

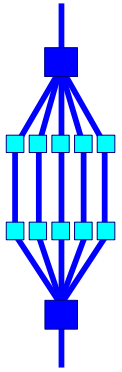
Run-time library overview



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”

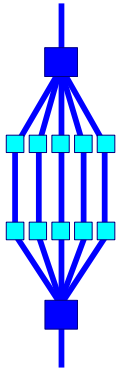
Additional run-time functions



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

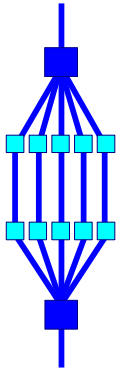
C/C++ : Need to include file <omp.h>
Fortran : Add “use omp_lib” or include file “omp_lib.h”

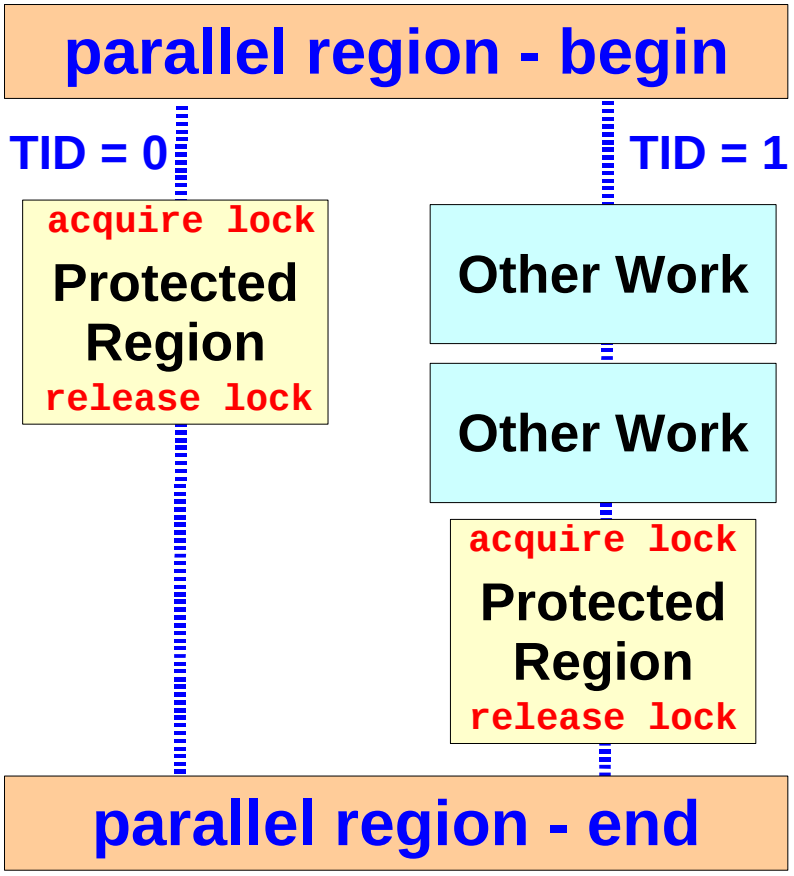
OpenMP locking routines



- ***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:***
  - ***Fortran: type INTEGER and of a KIND large enough to hold an address***
  - ***C/C++: type omp_lock_t and omp_nest_lock_t for nested locks***
- ***Lock variables should be manipulated through the API only***
- ***It is illegal, and behavior is undefined, in case a lock variable is used without the appropriate initialization***

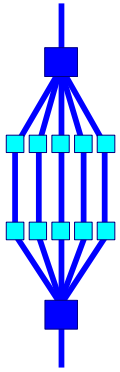
OpenMP locking example





- *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
```

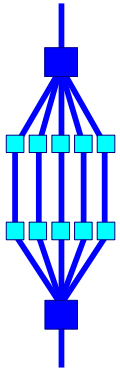
Initialize lock variable

Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

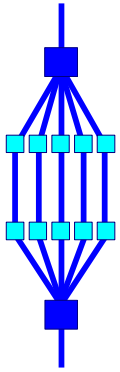
Example output for 2 threads



```
 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
```

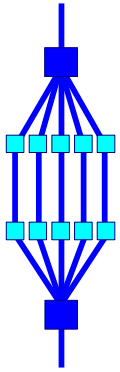
Used to check the answer

Note: program has been instrumented to get this information



Global Data

Global data - An example



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

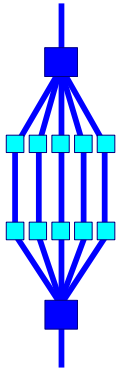
file global.h

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

```
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!



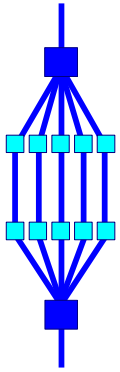
Thread 1

Thread 2

```
call suba(1)                    call suba(2)

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

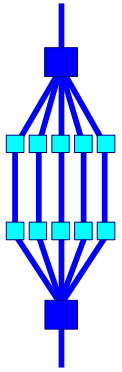
file global_ok.h

```
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
```

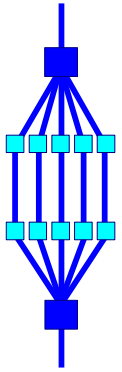
- *By expanding array B, we can give each thread unique access to it's storage area*
- *Note that this can also be done using dynamic memory (allocatable, malloc, ...)*

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



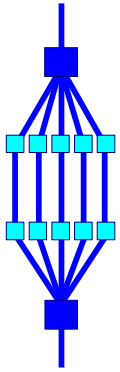
- *OpenMP's threadprivate directive*

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

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

- *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)*
    - ✔ *Sun 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
```

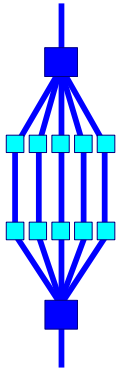
file global_ok2.h

```
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

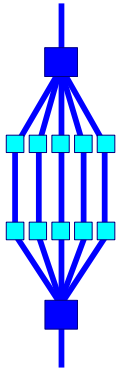


copyin (list)

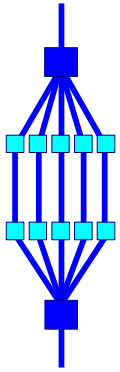
- ✔ *Applies to THREADPRIVATE common blocks only*
- ✔ *At the start of the parallel region, data of the master thread is copied to the thread private copies*

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 )
```



A First Glimpse Into Tasking

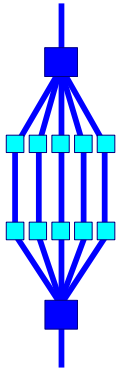


Why The Excitement About OpenMP 3.0 ?

Support for TASKS !

With this new feature, a wider range of applications can now be parallelized

Task Construct Syntax



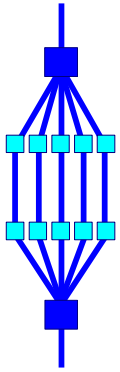
***C/C++*:**

```
#pragma omp task [clause [[,]clause] ...]
   structured-block
```

***Fortran*:**

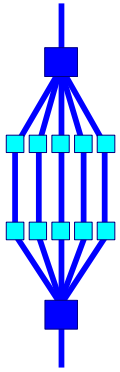
```
!$omp task[clause [[,]clause] ...]
   structured-block
!$omp end task
```

Task Synchronization



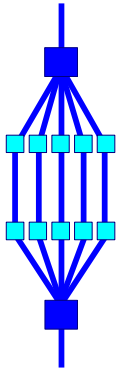
- *Syntax:*
  - *C/C++:* `#pragma omp taskwait`
  - *Fortran:* `!$omp taskwait`
- *Current task suspends execution until all children tasks, generated within the current task up to this point, are complete*

When are Tasks Complete?



- *At implicit thread barrier*
- *At explicit thread barrier*
  - *C/C++:* `#pragma omp barrier`
  - *Fortran:* `!$omp barrier`
- *At task barrier*
  - *C/C++:* `#pragma omp taskwait`
  - *Fortran:* `!$omp taskwait`

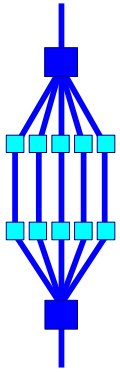
Example - A Linked List



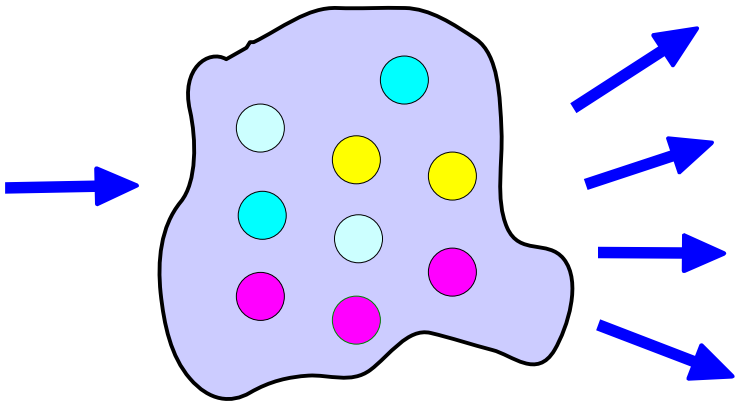
```
        ........

  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



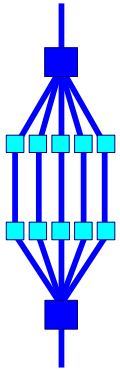
Encountering thread adds task to pool



Threads execute tasks in the pool

Developer specifies tasks in application
Run-time system executes tasks

Example - A Linked List With Tasking

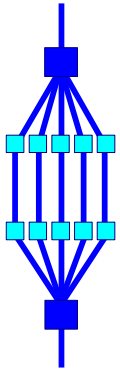


OpenMP Task is specified here (executed in parallel)

```
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
```

3.0

Example 2 – Fibonacci Numbers



The Fibonacci Numbers are defined as follows:

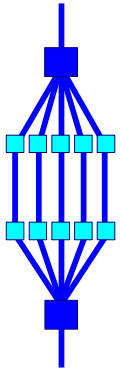
```
F(0) = 1
F(1) = 1

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

Sequence:

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

Credit goes to Christian Terboven (RWTH Aachen) for his work on the OpenMP tasking version of this algorithm

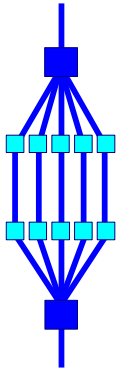
Recursive Algorithm*



```
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);
}
```

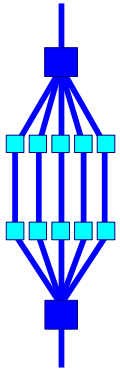
) Not very efficient, used for demo purposes only

Parallel Recursive Algorithm



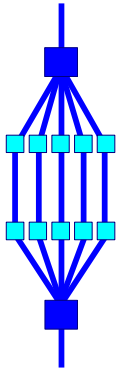
```
long comp_fib_numbers(int n){

   // 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



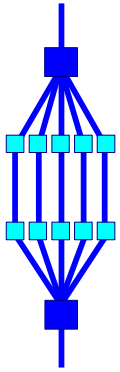
```
#pragma omp parallel shared(nthreads)
{
   #pragma omp single nowait
   {

      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);
}
```

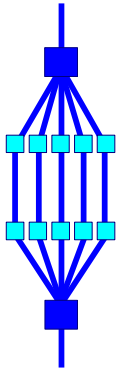
Performance Example*



```
$ 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*

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*
  - *Support for OpenMP 3.0 is on the rise*
- *The Tasking feature in OpenMP 3.0 opens up opportunities to parallelize a wider range of applications*
- *Sun Studio has extensive support for OpenMP developers*
  - *And there is more to come*