An Overview Of OpenMP

Ruud van der Pas

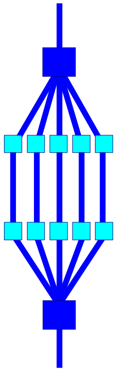
Senior Staff Engineer
Systems Group
Sun Microsystems
Menlo Park, CA, USA

IWOMP 2006
Reims, France
June 12-15, 2006

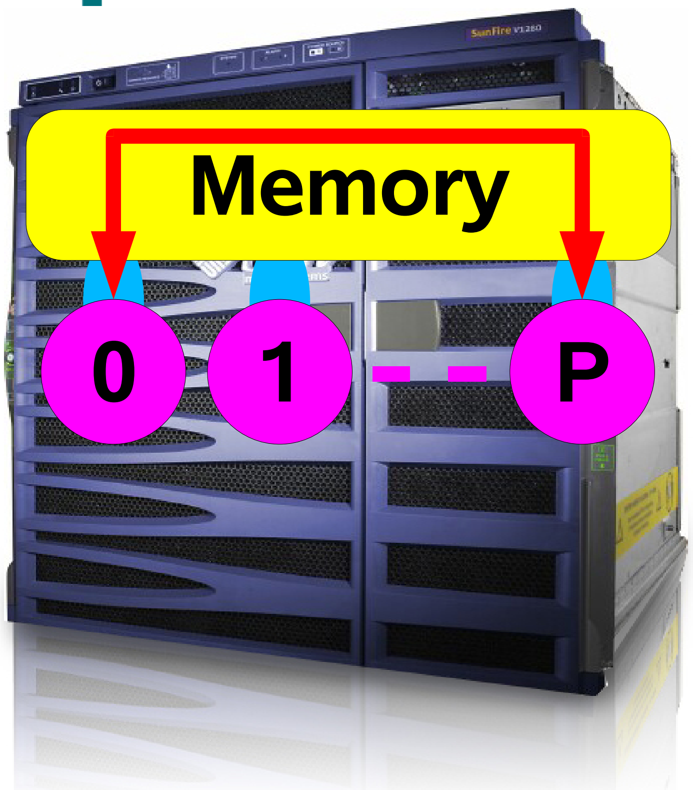
Outline

- ❑ ***The OpenMP Programming Model***
- ❑ ***OpenMP Guided Tour***
- ❑ ***OpenMP Overview***
  - ***Directives***
  - ***Environment variables***
  - ***Run-time environment***
- ❑ ***OpenMP and Global Data***
- ❑ ***Wrap-up***

The OpenMP Programming Model

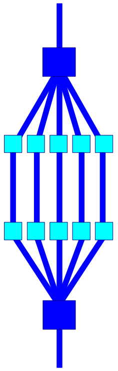


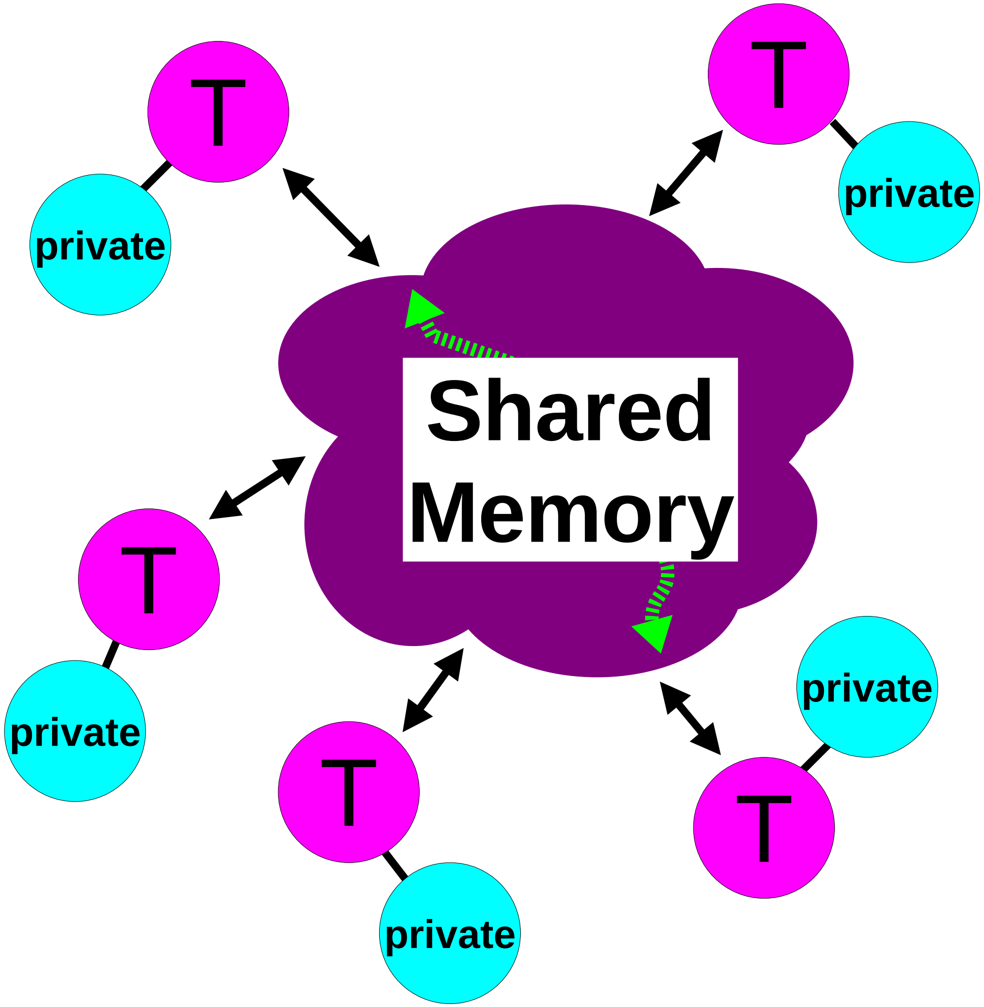
OpenMP™



http://www.openmp.org

Shared Memory Model

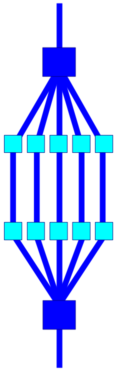




Programming Model

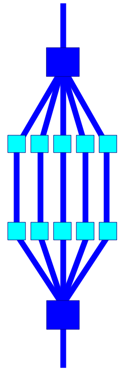
- ✔ *All threads have access to the same, globally shared, memory*
- ✔ *Data can be shared or private*
- ✔ *Shared data is accessible by all threads*
- ✔ *Private data can only be accessed by the thread that owns it*
- ✔ *Data transfer is transparent to the programmer*
- ✔ *Synchronization takes place,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

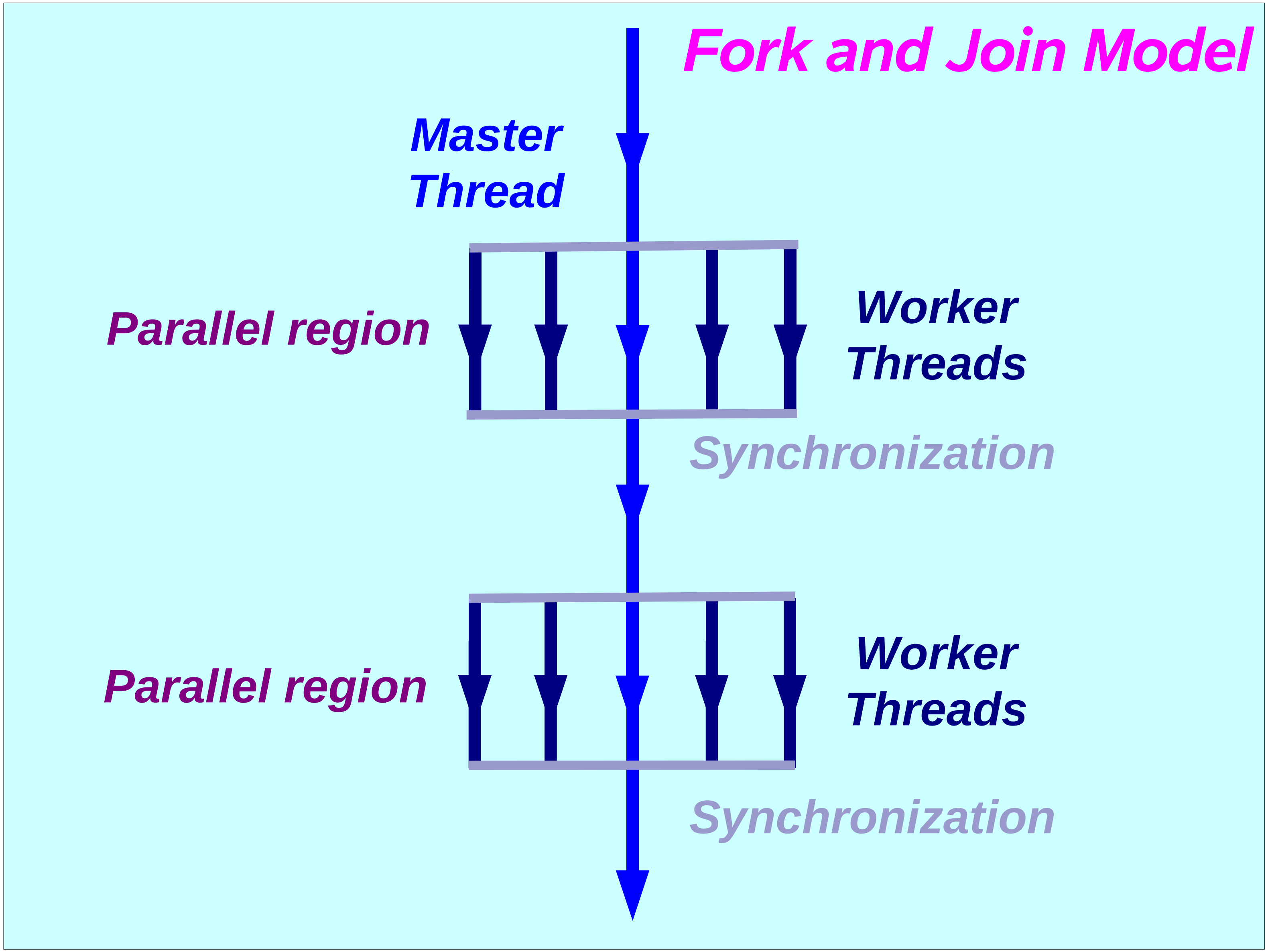
About Data



- *In a shared memory parallel program variables have a "label" attached to them:*
  - ☞ *Labelled "Private" ⇨ Visible to one thread only*
    - ✔ *Change made in local data, is not seen by others*
    - ✔ *Example - Local variables in a function that is executed in parallel*
  - ☞ *Labelled "Shared" ⇨ Visible to all threads*
    - ✔ *Change made in global data, is seen by all others*
    - ✔ *Example - Global data*

The OpenMP execution model





OpenMP Guided Tour

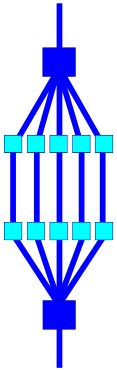
OpenMP™

http://www.openmp.org

cOMPunity

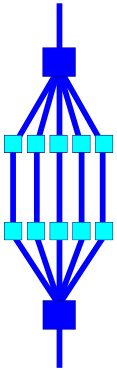
http://www.compunity.org

What is OpenMP?



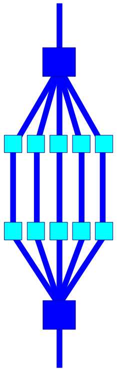
- ❑ *De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran*
- ❑ *Consists of:*
  - *Compiler directives*
  - *Run-time routines*
  - *Environment variables*
- ❑ *Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)*
- ❑ *Latest Specification: Version 2.5*
- ❑ *Language committee meetings for Version 3.0 have started in September 2005*

When to consider using OpenMP?



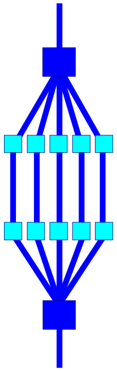
- ❑ *The compiler may not be able to do the parallelization in the way you like to see it:*
  - ● *A loop is not parallelized*
    - ✔ *The data dependence analysis is not able to determine whether it is safe to parallelize or not*
  - ● *The granularity is not high enough*
    - ✔ *The compiler lacks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

Advantages of OpenMP



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

A first OpenMP example



For-loop with independent iterations

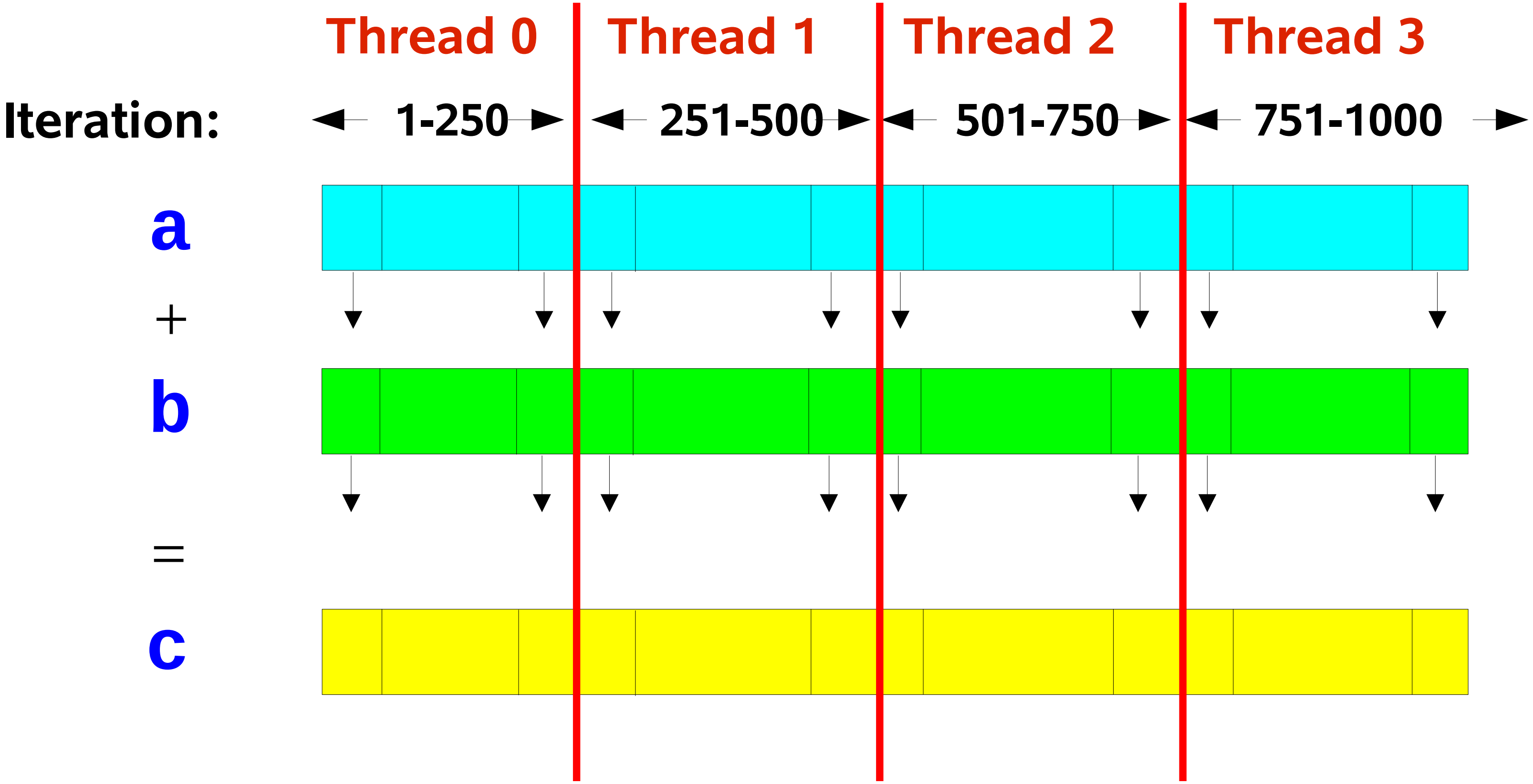
```
for (i = 0; i < n; i++)
   c[i] = a[i] + b[i];
```

For-loop parallelized using an OpenMP pragma

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

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

Example parallel execution



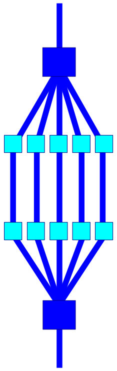
A loop parallelized with OpenMP

```
#pragma omp parallel default(none) \
            shared(n,x,y) private(i)
{
 #pragma omp for
   for (i=0; i<n; i++)
       x[i] += y[i];
} /*-- End of parallel region --*/
```

clauses

```
!$omp parallel default(none)     &
!$omp shared(n,x,y) private(i)
!$omp do
      do i = 1, n
         x(i) = x(i) + y(i)
      end do
!$omp end do
!$omp end parallel
```

Components of OpenMP



Directives

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

Environment variables

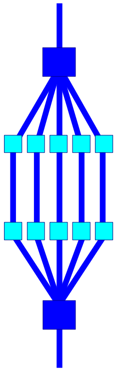
- ◆ *Number of threads*
- ◆ *Scheduling type*
- ◆ *Dynamic thread adjustment*
- ◆ *Nested parallelism*

Runtime environment

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

The fork-join execution model is used

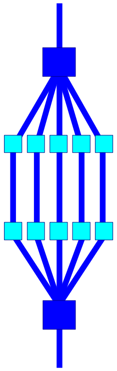
Directive format



- ❑ *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

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

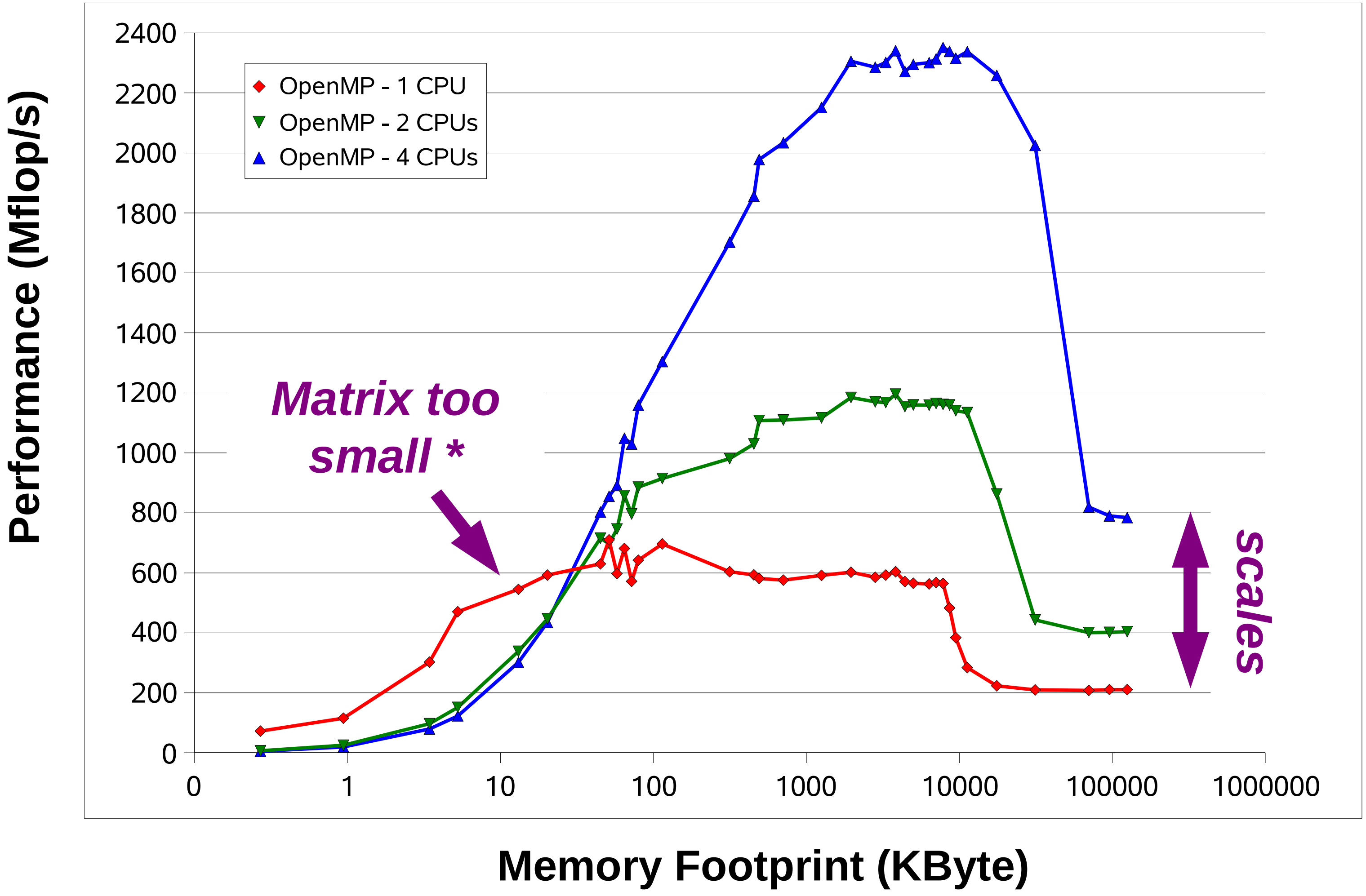
parallel region

Another OpenMP example

```
 1 void mxv_row(int m,int n,double *a,double *b,double *c)
 2 {
 3  int i, j;
 4  double sum;
 5
 6 #pragma omp parallel for default(none) \
 7             private(i,j,sum) shared(m,n,a,b,c)
 8  for (i=0; i<m; i++)
 9  {
10    sum = 0.0;
11    for (j=0; j<n; j++)
12      sum += b[i*n+j]*c[j];
13     a[i] = sum;
14  } /*-- End of parallel for --*/
15 }
```

```
% cc -c -fast -xrestrict -xopenmp -xloopinfo mxv_row.c
"mxv_row.c", line  8: PARALLELIZED, user pragma used
"mxv_row.c", line 11: not parallelized
```

OpenMP performance

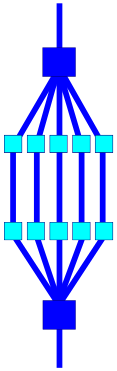


SunFire 6800
UltraSPARC III Cu @ 900 MHz
8 MB L2-cache

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

OpenMP Directives

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*

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 --*/
```

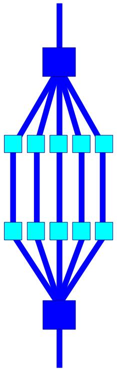
private (list)

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

shared (list)

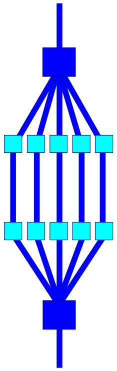
- ✔ *Data is accessible by all threads in the team*
- ✔ *All threads access the same address space*

About storage association



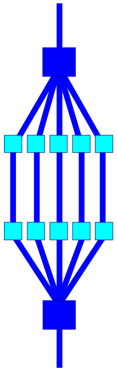
- ❑ ***Private variables are undefined on entry and exit of the parallel region***
- ❑ ***The value of the original variable (before the parallel region) is undefined after the parallel region !***
- ❑ ***A private variable within a parallel region has no storage association with the same variable outside 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 --*/

}
```

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) *Fortran*

default (none | shared) *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***

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++*

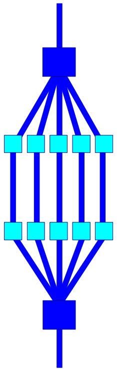
- ✔ *Reduction variable(s) must be shared variables*
- ✔ *A reduction is defined as:*

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:

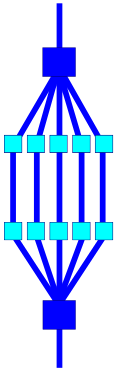
```
for (i=0; i < N; i++)
   a[i] = b[i] + c[i];
```

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

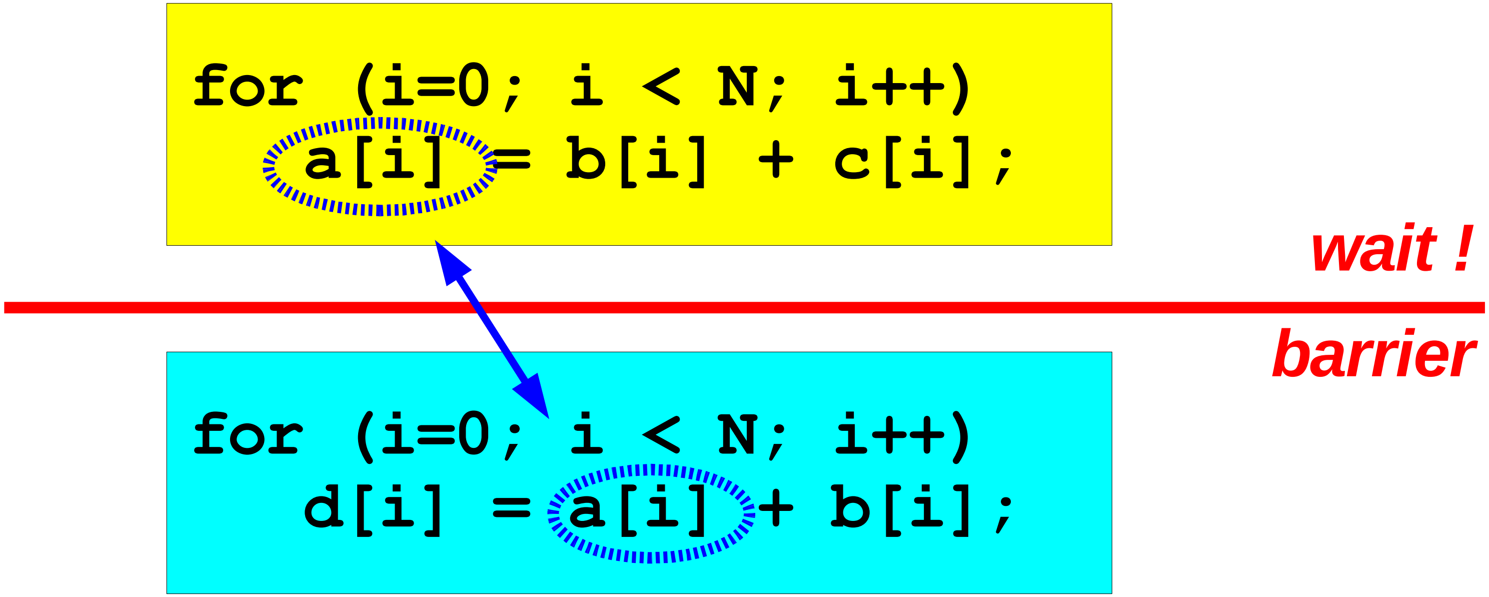
This may give us a wrong answer (one day)

Why ?

Barrier/2



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



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

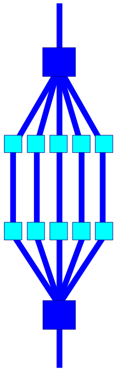
wait !

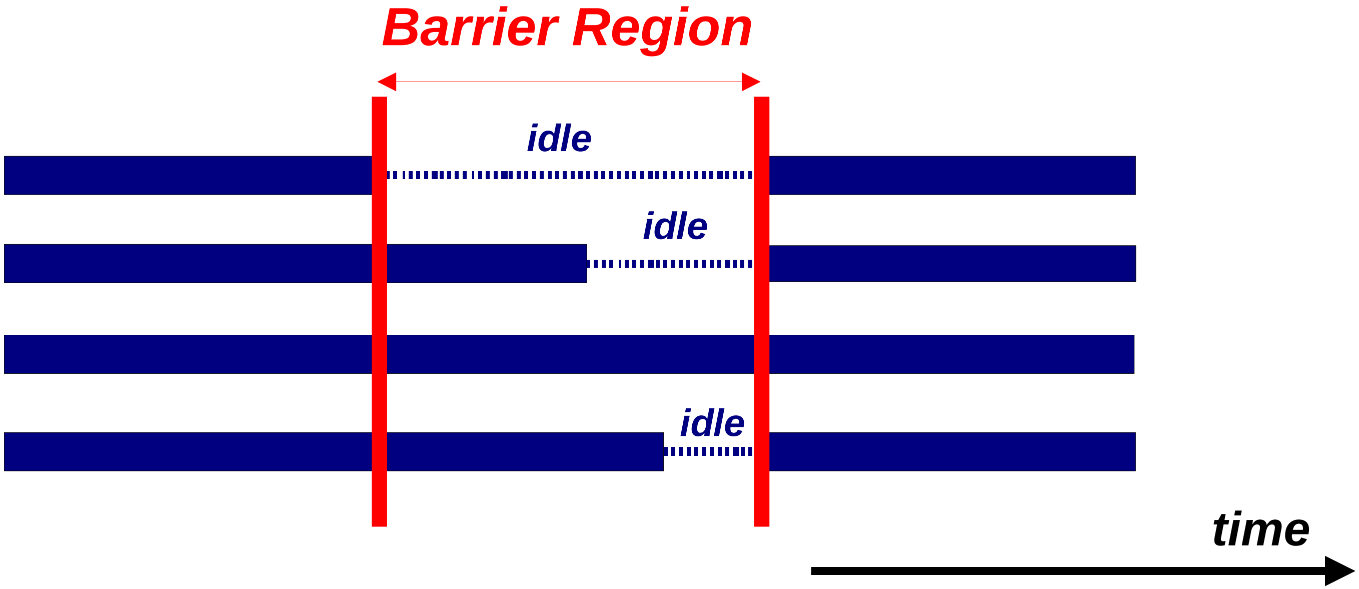
barrier

```
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

Barrier/3



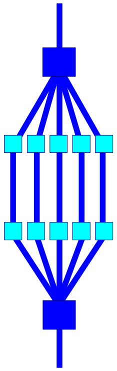


Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```

When to use barriers ?



- ❑ *When data is updated asynchronously and the 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*

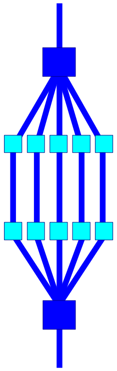
The nowait clause

- ❑ *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 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)
```

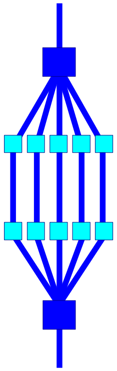
```
#pragma omp parallel [clause[[,] clause] ...]
{
   "this is executed in parallel"

} (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)*	*(Fortran)*
reduction	(*operator*: *list*)	
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>
```

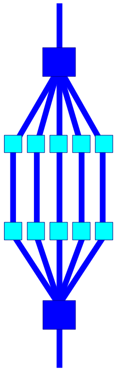
```
!$omp do [clause[[,] clause] ...]
        <original do-loop>
!$omp end do [nowait]
```

Clauses supported:

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

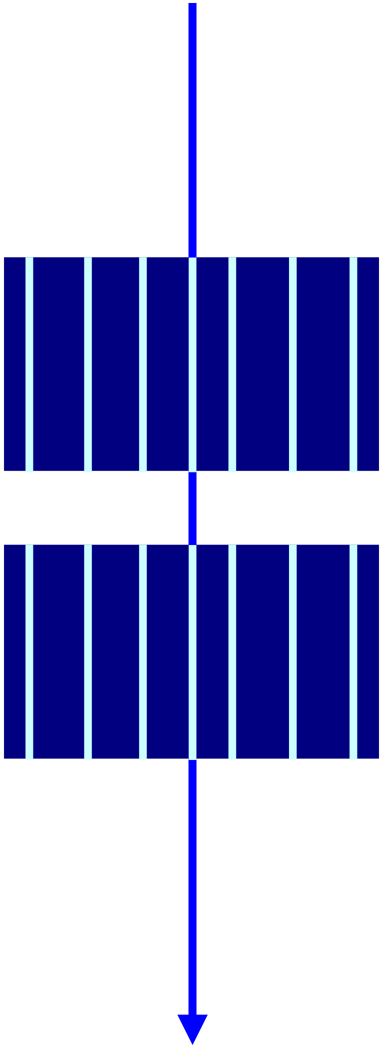
**) Required if ordered sections are in the dynamic extent of this construct*

The omp for directive - example



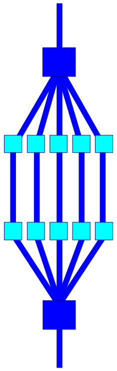
```
#pragma omp parallel default(none)\
        shared(n,a,b,c,d) private(i)
  {
    #pragma omp for nowait
     for (i=0; i<n-1; i++)
         b[i] = (a[i] + a[i+1])/2;
    #pragma omp for nowait
     for (i=0; i<n; i++)
         d[i] = 1.0/c[i];

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

(implied barrier)



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
           :
}
```

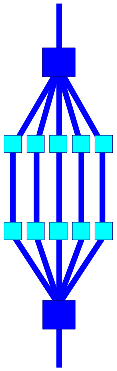
```
!$omp sections [clause(s)]
!$omp section
         <code block1>
!$omp section
         <code block2>
!$omp section
           :
!$omp end sections [nowait]
```

Clauses supported:

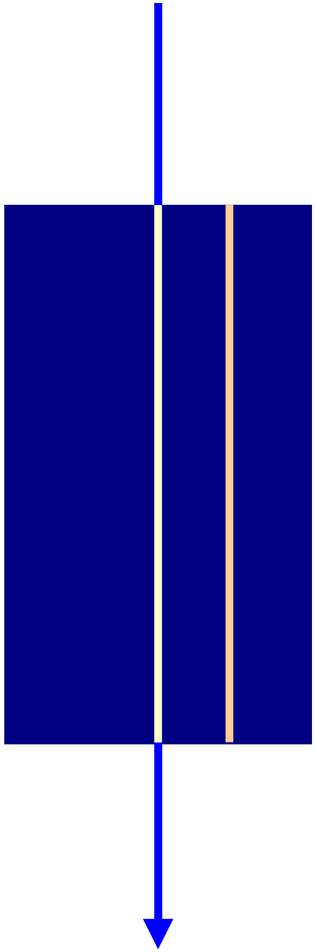
private	firstprivate
lastprivate	reduction
nowait	

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

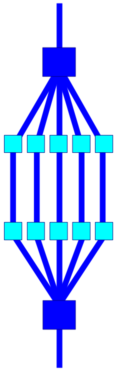
The sections directive - example



```
#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 --*/
```



Short-cuts



```
#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
{ ...}
```

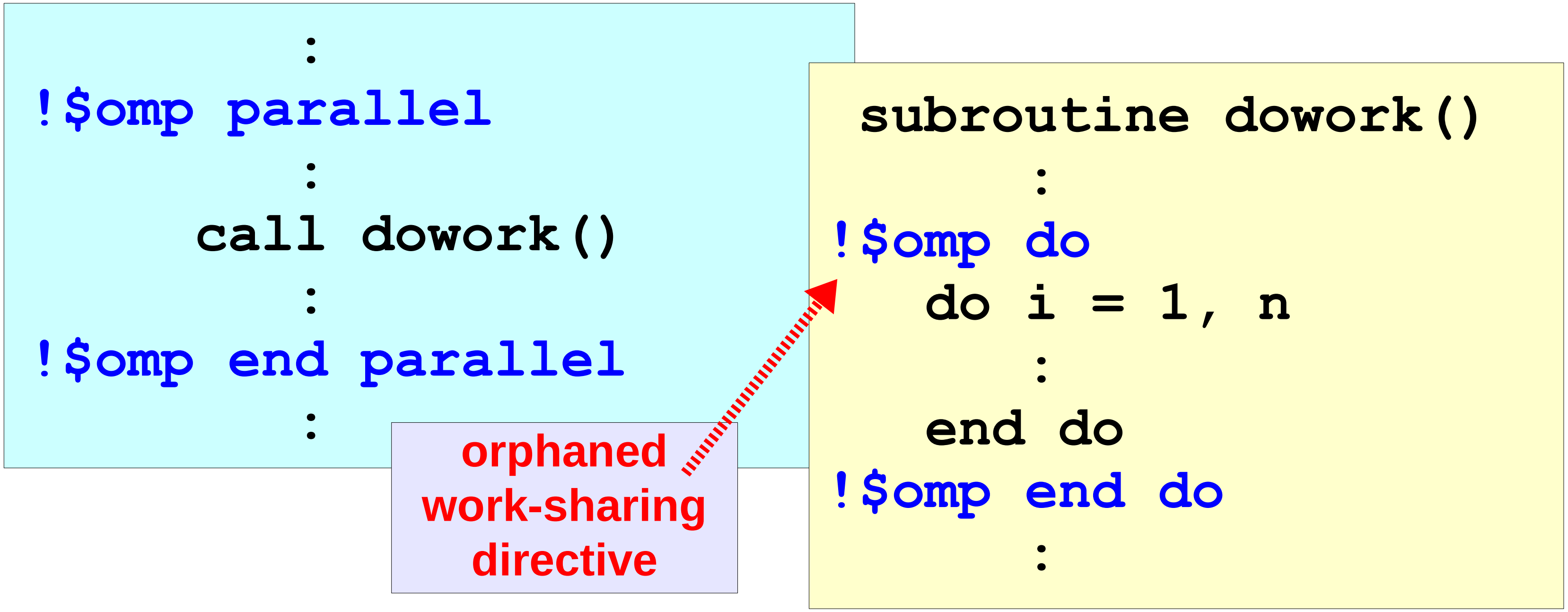
```
#pragma omp parallel sections
{ ... }
```

Single PARALLEL sections

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

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

Orphaning



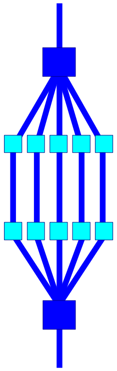
```
              :
!$omp parallel
              :
         call dowork()
              :
!$omp end parallel
              :
```

```
 subroutine dowork()
            :
!$omp do
    do i = 1, n
            :
    end do
!$omp end do
            :
```

orphaned
work-sharing
directive

- *The OpenMP standard does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be <u>orphaned</u>*
- *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

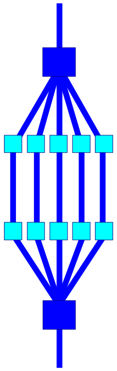
Should behave identically

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,..)
```

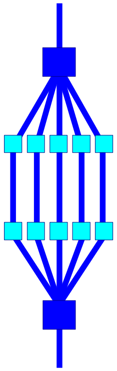
```
   for (i=0; i<n; i++) /* Parallel loop */
   {
        (void) FuncPar(i,m,c,...)
   } /*-- End of parallel for --*/
```

Minimal scoping required

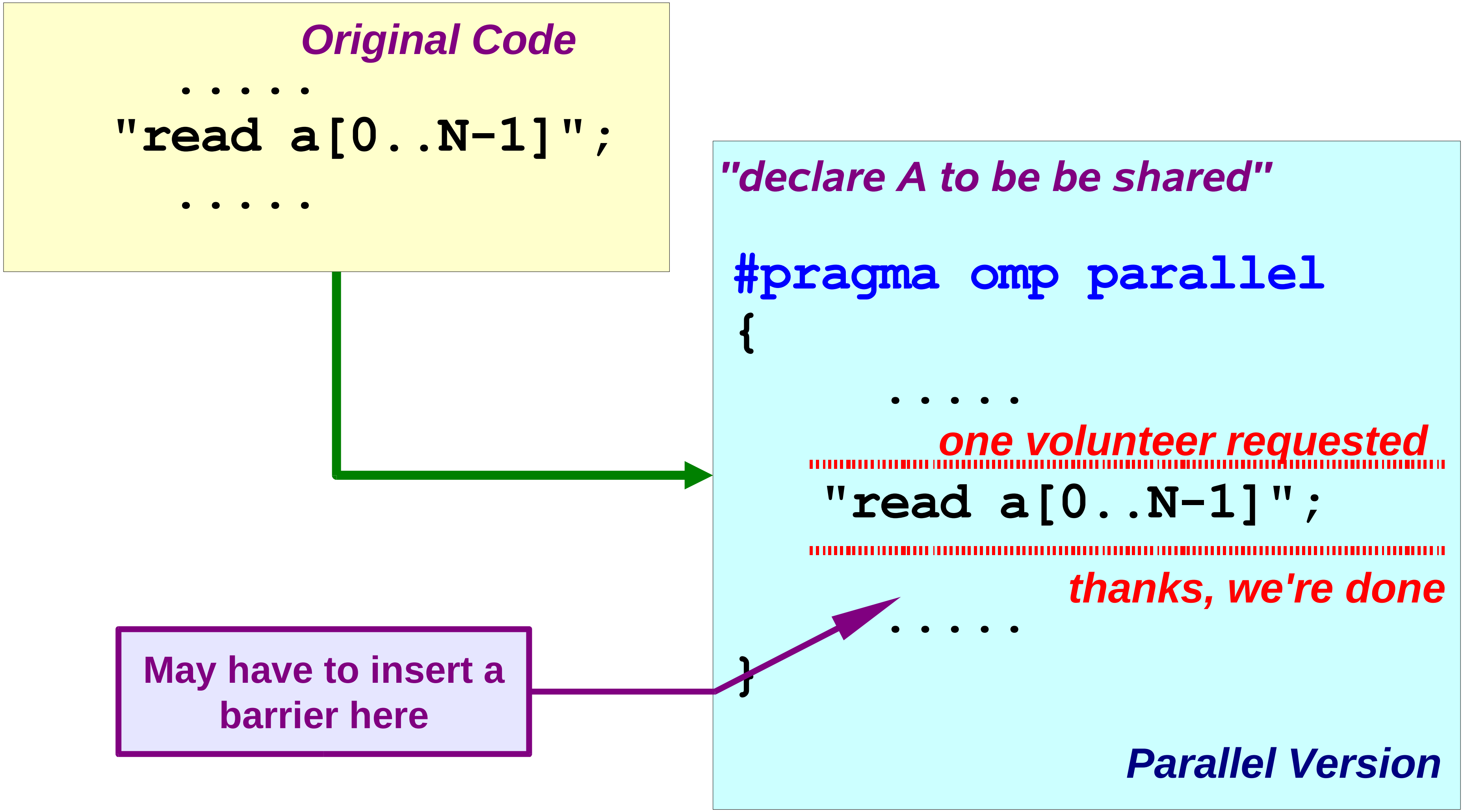
Less error prone

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

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"

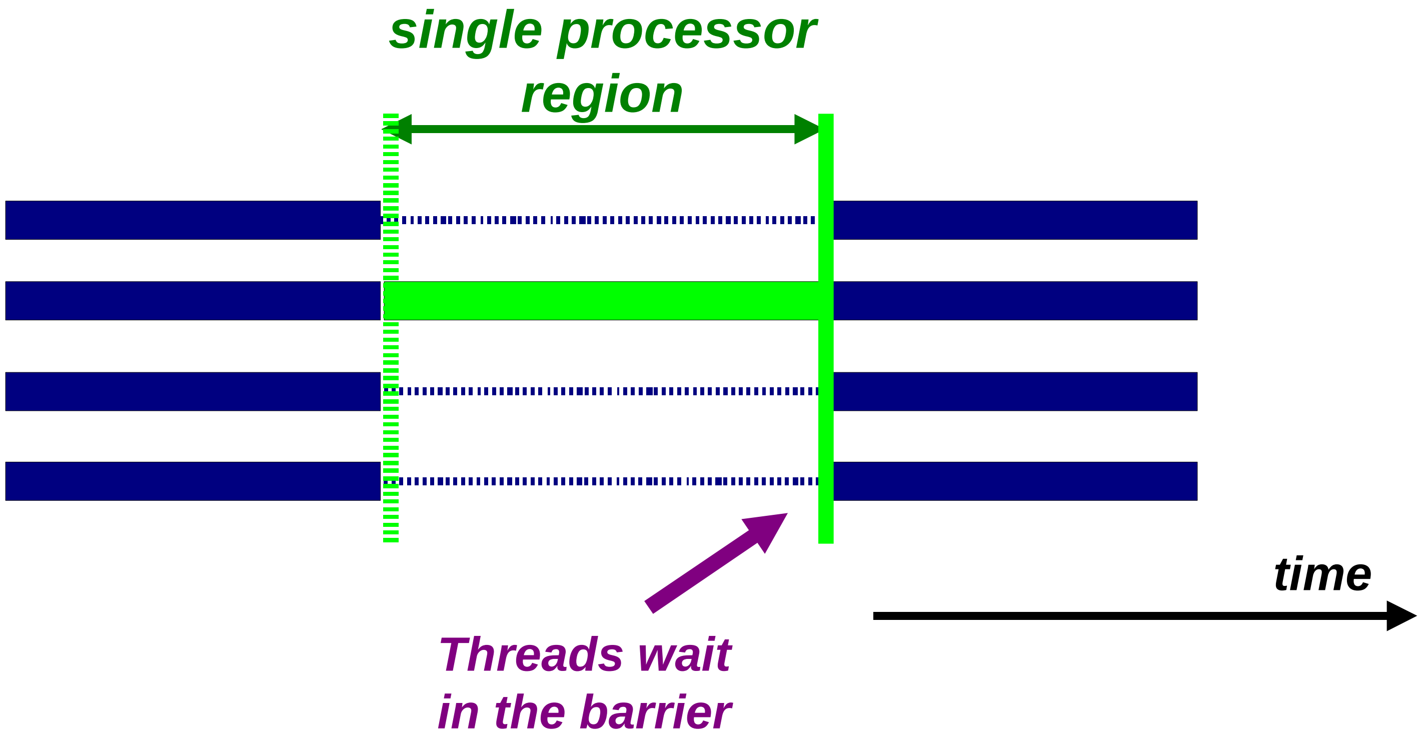
```
#pragma omp parallel
{
      .....
        one volunteer requested
   "read a[0..N-1]";
                    thanks, we're done
      .....
}
```

May have to insert a barrier here

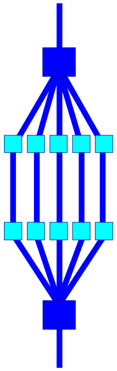
Parallel Version

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 [clause[[,] clause] ...]
{
        <code-block>
}
```

```
!$omp single [clause[[,] clause] ...]
   <code-block>
!$omp end single [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 !

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:

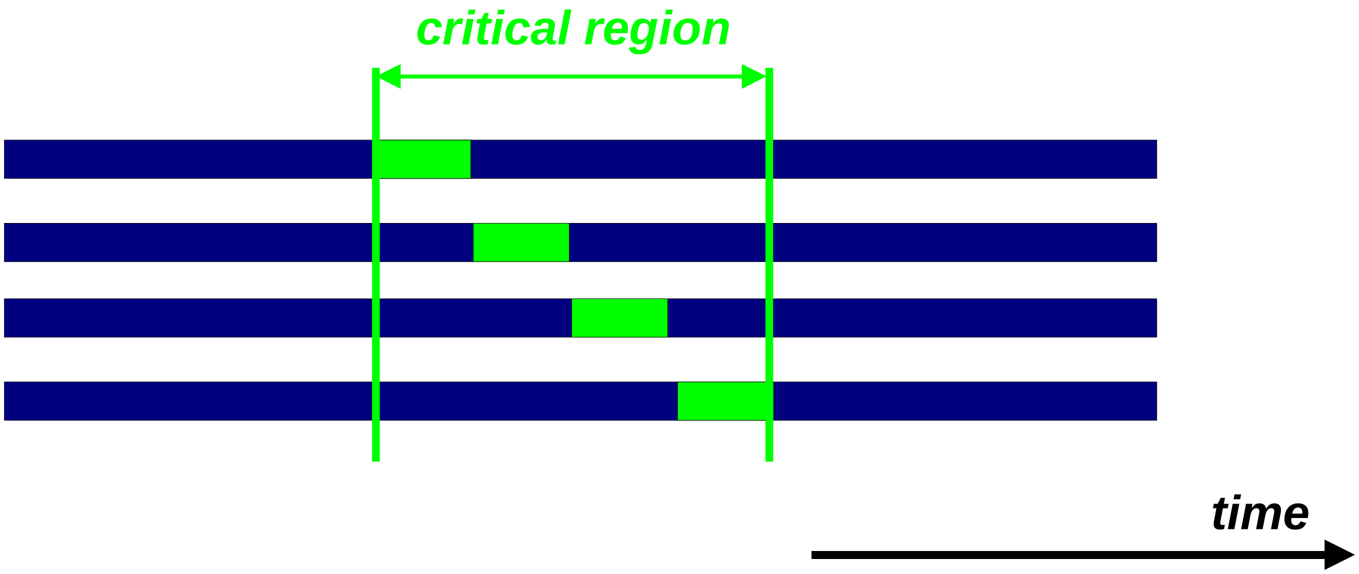
```
for (i=0; i < N; i++){
        .....
    sum += a[i];
        .....
}
```

one at a time can proceed

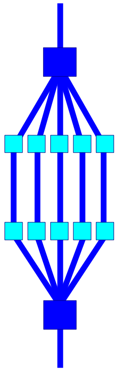
next in line, please

Critical Region/2

- ❑ *Useful to avoid a race condition, or to perform I/O (but which still has random order)*
- ❑ *Be aware that your parallel computation may be <u>serialized</u> and so this could introduce a scalability bottleneck (Amdahl's law)*



The Critical Construct



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 !

The Atomic Construct

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

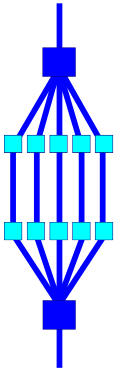
```
#pragma omp atomic
   <statement>
```

```
!$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)

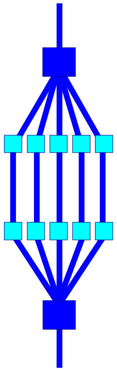
Ensure that all threads in a team have a consistent view of certain objects in memory:

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

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

In the absence of a list, all visible variables are flushed; this could be expensive

Load balancing



- Load balancing is an important aspect of performance
- For regular operations (e.g. a vector addition), load balancing is not an issue
- For less regular workloads, care needs to be taken in distributing the work over the threads
- Examples:
  - Transposing a matrix
  - Multiplication of triangular matrices
  - Parallel searches in a linked list
- For these irregular situations, the schedule clause supports various iteration scheduling algorithms

The schedule clause/1

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

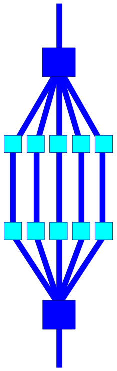
static [, chunk]

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

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

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

The schedule clause/2



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*

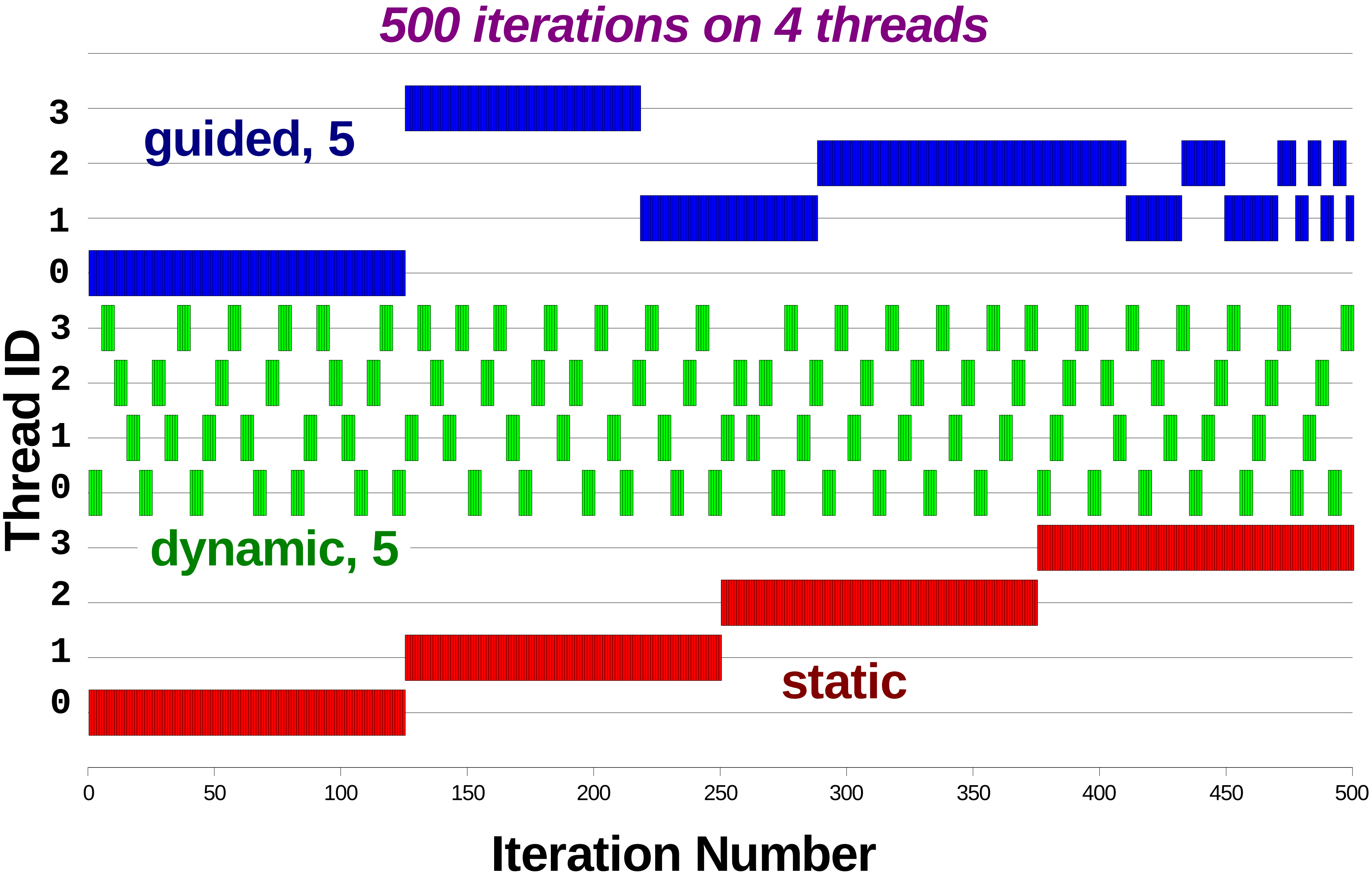
guided [, chunk]

- ✔ *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



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)

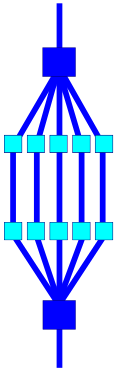
(2) The number of threads is limited to the number of on-line processors in the system. This can be changed by setting OMP_DYNAMIC to FALSE.

(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

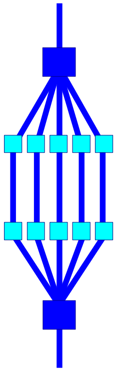
OpenMP Run-time Environment

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"*

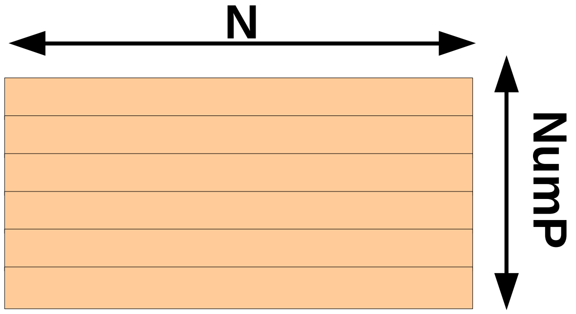
Run-time library overview



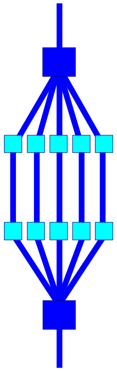
Name	Functionality
omp_set_num_threads	Set number of threads
omp_get_num_threads	Return number of threads in team
omp_get_max_threads	Return maximum number of threads
omp_get_thread_num	Get thread ID
omp_get_num_procs	Return maximum number of processors
omp_in_parallel	Check whether in parallel region
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

Example

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

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

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

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

     .....
}
```



OpenMP locking routines



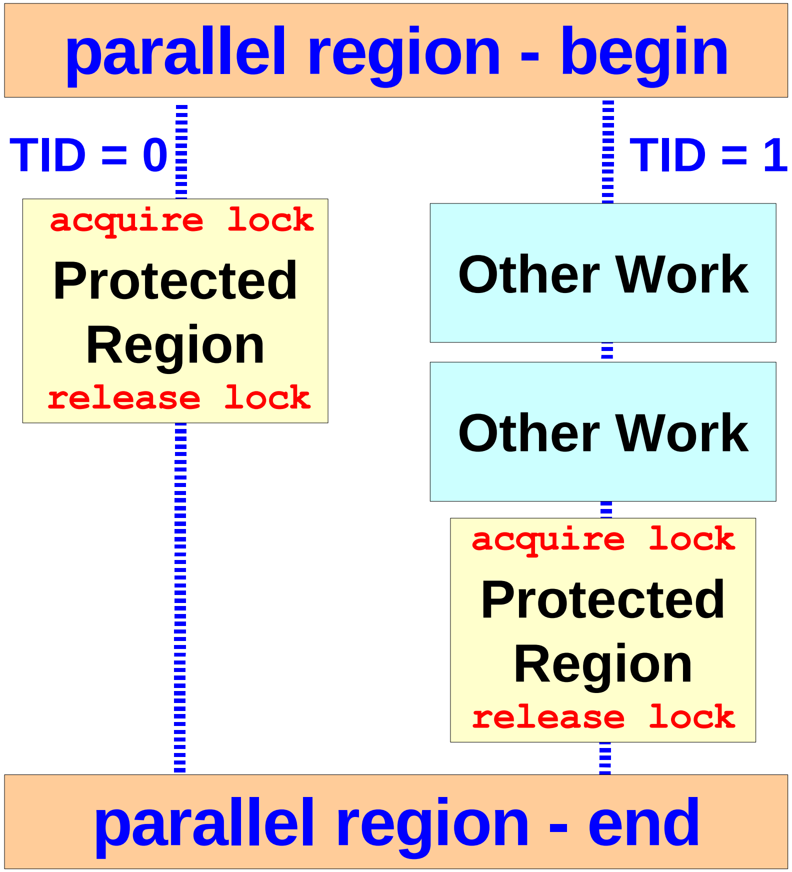
- ❑ *Locks provide greater flexibility over critical sections and atomic updates:*
  - *Possible to implement asynchronous behavior*
  - *Not block structured*
- ❑ *The so-called lock variable, is a special variable:*
  - *Fortran: type INTEGER and of a KIND large enough to hold an address*
  - *C/C++: type omp_lock_t and omp_nest_lock_t for nested locks*
- ❑ *Lock variables should be manipulated through the API only*
- ❑ *It is illegal, <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



parallel region - begin

TID = 0

acquire lock
Protected Region
release lock

TID = 1

Other Work

Other Work

acquire lock
Protected Region
release lock

parallel region - end

- *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(SUM,LCK) private(TID)

       TID = omp_get_thread_num()

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

       Call Do_Work(SUM,TID)

       Call omp_unset_lock (LCK)

!$omp end parallel

      Call omp_destroy_lock (LCK)

      Stop
      End
```

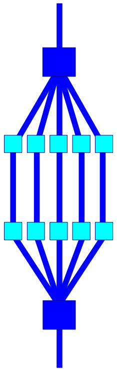
Initialize lock variable

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 and Global Data

Global data - 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
         ......
      stop
```

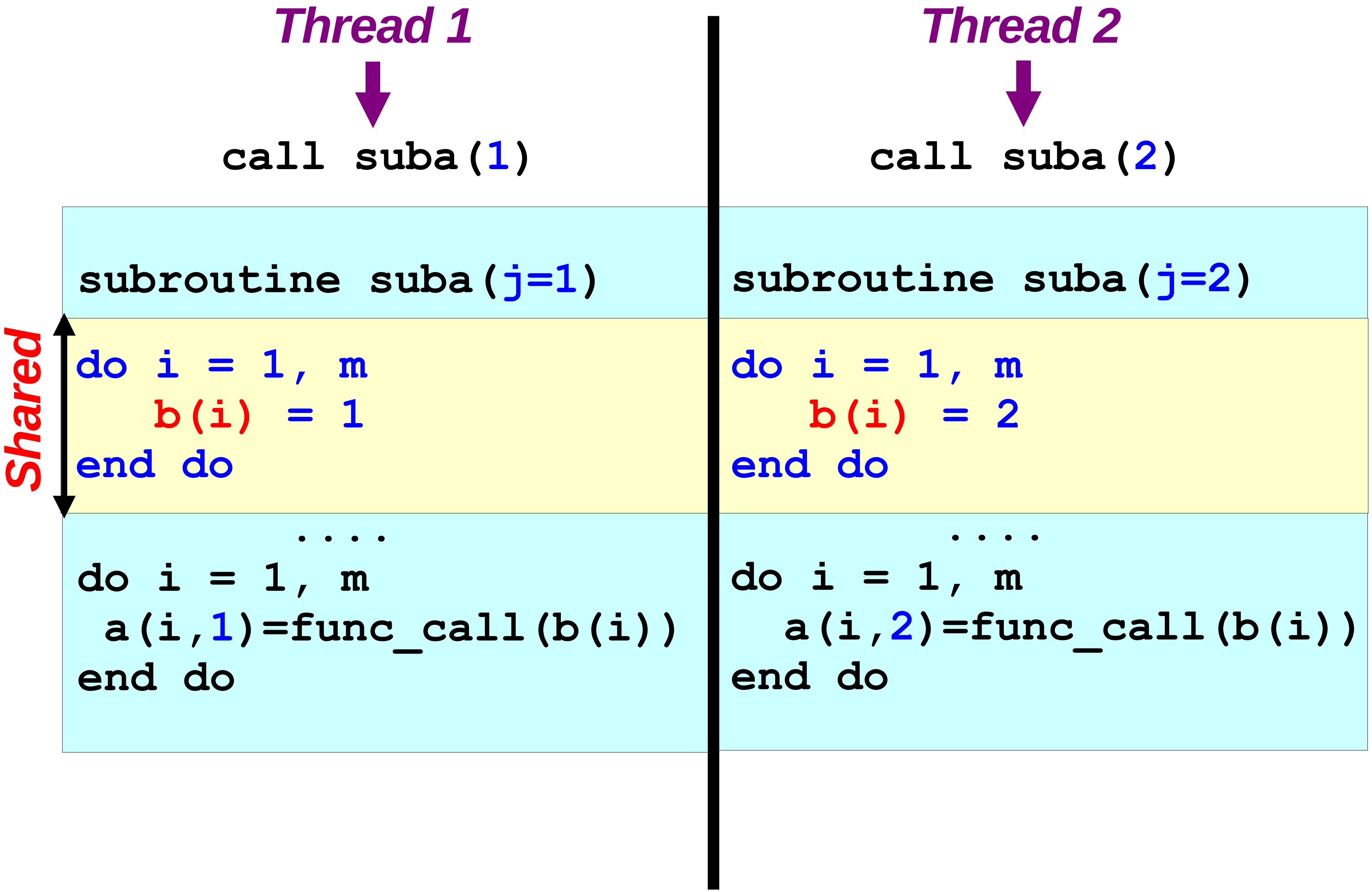
file global.h

```
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
```

Race condition !

Global data - race condition



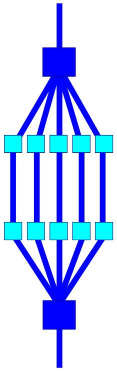
Thread 1

```
call suba(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

```
call suba(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
```

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
        ......
      stop
      end
```

file global_ok.h

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

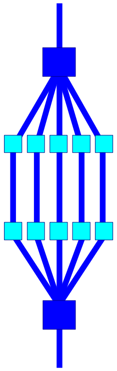
```
subroutine suba(j)
  .....
include "global_ok.h"
  .....

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
```

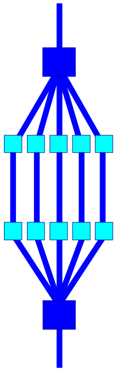
file global_ok2.h

```
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 )
```

Wrap-Up

Summary

- *OpenMP provides for a compact, but yet powerful programming model for shared memory programming*
- *OpenMP supports C, C++ and Fortran*
- *OpenMP programs are portable to a wide range of systems*
- *OpenMP allows for incremental parallelization*
- *An OpenMP program can be written such that the sequential version is still “built-in”*

Thank You !

Ruud van der Pas
ruud.vanderpas@sun.com