

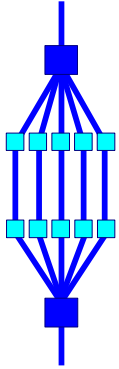
# *An Overview of OpenMP*

**Ruud van der Pas**

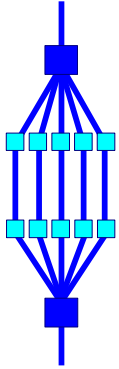
**Senior Staff Engineer  
Technical Developer Tools  
Sun Microsystems, Menlo Park, CA, USA**

***Nanyang Technological University  
Singapore  
Wednesday January 14, 2009***

# Outline



- *A Guided Tour of OpenMP*
- *Case Study*
- *Wrap-Up*



# OpenMP™

<http://www.openmp.org>



<http://www.compunity.org>

<http://www.openmp.org>



THE OPENMP API SPECIFICATION FOR PARALLEL PROGRAMMING



### 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](mailto:webmaster@openmp.org)

### Search OpenMP.org

Google Custom Search

### Archives

- o June 2008
- o May 2008
- o April 2008

### Admin

- o [Log in](#)

Copyright © 2008 OpenMP Architecture Review Board. All rights reserved.

## 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](#)

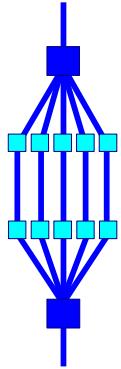
### Use It

» [OpenMP Compilers](#)

### Learn It



# Shameless Plug - "Using OpenMP"



## *"Using OpenMP"*

*Portable Shared Memory  
Parallel Programming*

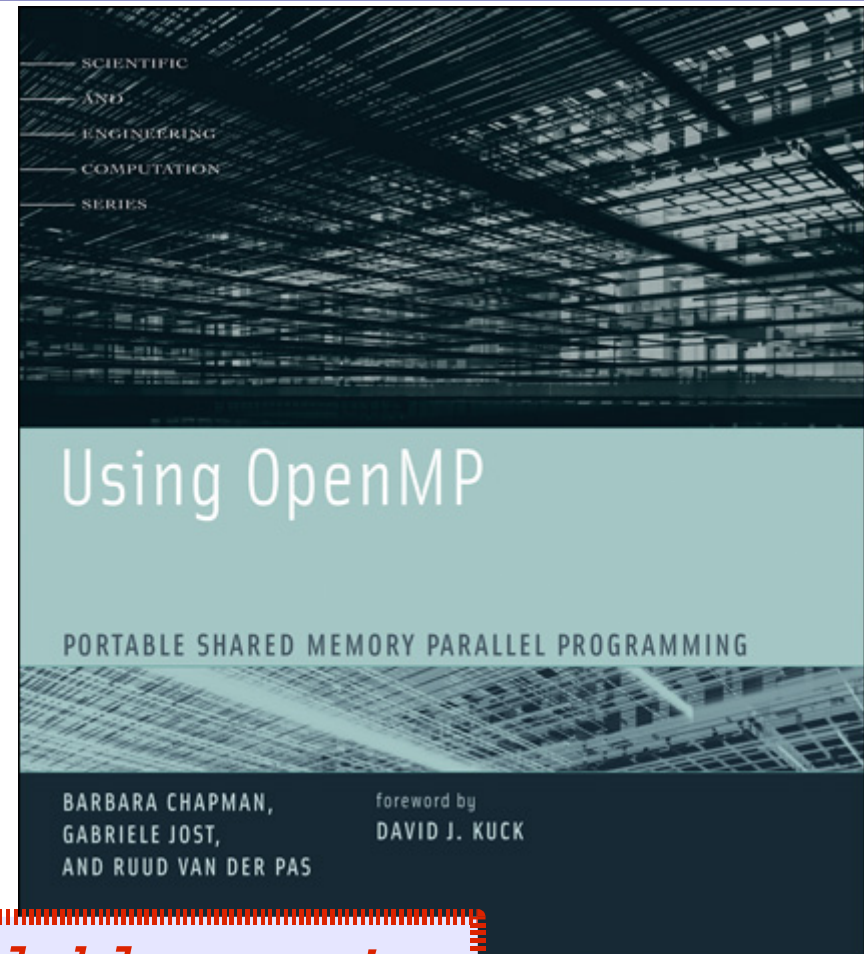
*Chapman, Jost, van der Pas*

**MIT Press, October 2007**

**ISBN-10: 0-262-53302-2**

**ISBN-13: 978-0-262-53302-7**

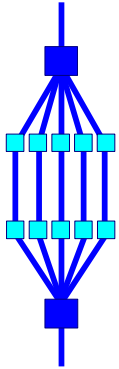
**List price: 35 \$US**



*All examples available soon!*

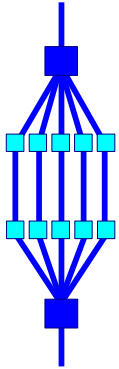
*(also plan to start a forum  
on [www.openmp.org](http://www.openmp.org))*

# What is OpenMP?



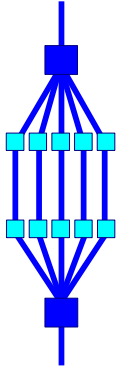
- *De-facto standard API for writing shared memory parallel applications in C, C++, and Fortran*
- *Consists of:*
  - *Compiler directives*
  - *Run time routines*
  - *Environment variables*
- *Specification maintained by the OpenMP Architecture Review Board (<http://www.openmp.org>)*
- *Version 3.0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

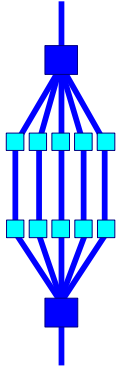
# Advantages of OpenMP



- ❑ *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***

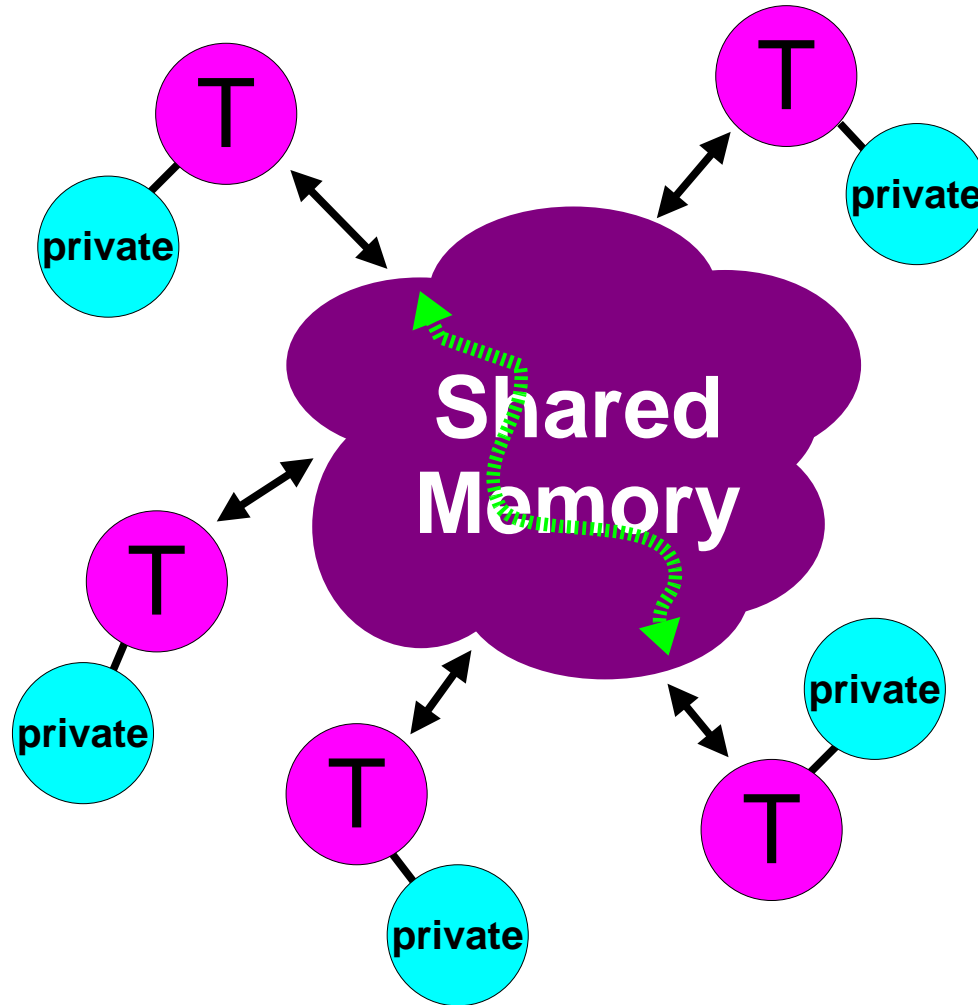
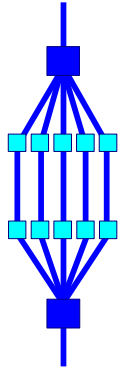
***Memory and threading model map naturally***

***Lightweight***

***Mature***

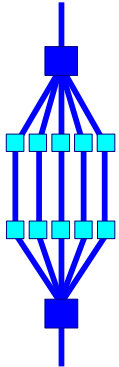
***Widely available and used***

# The OpenMP Memory Model



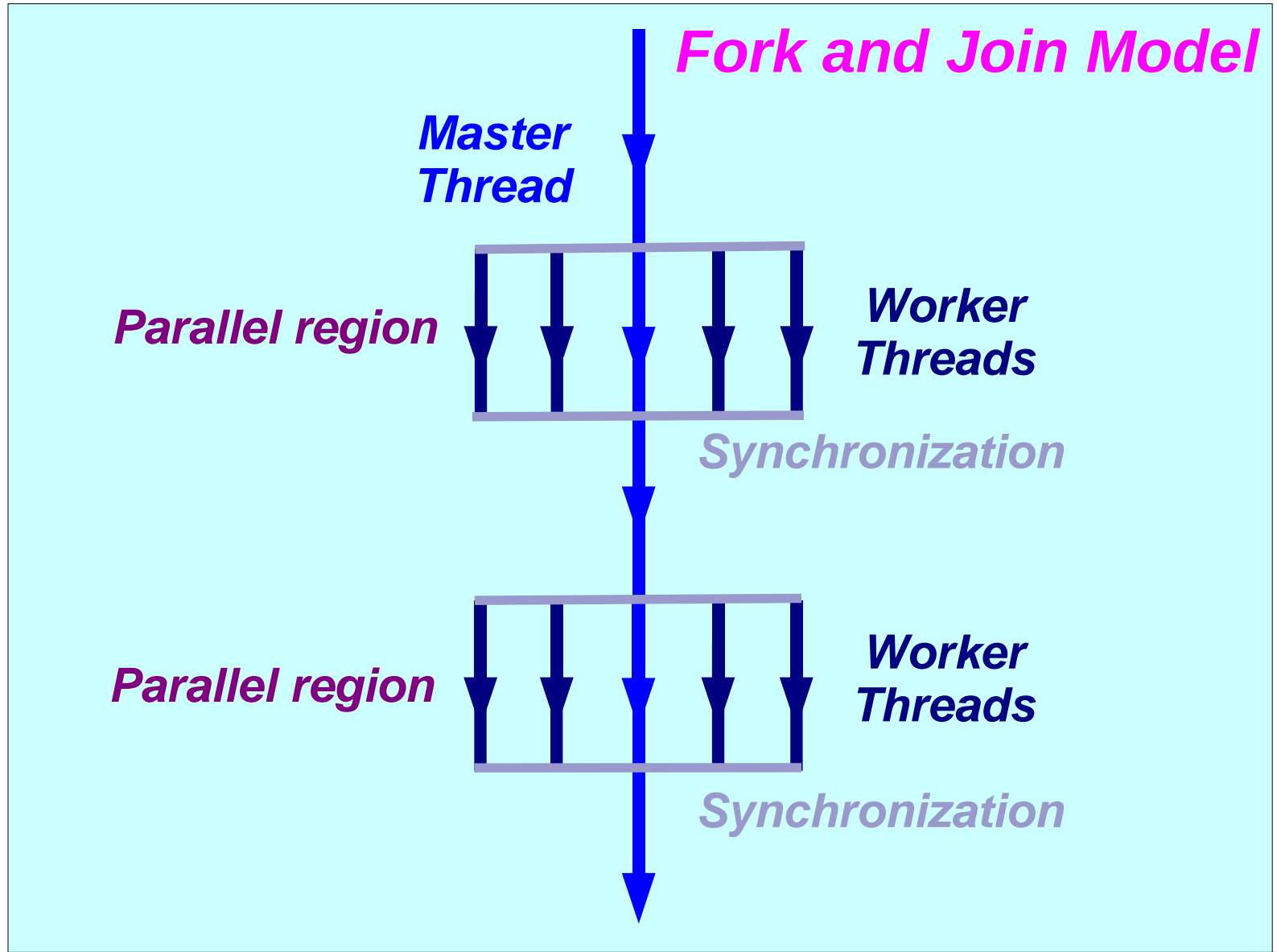
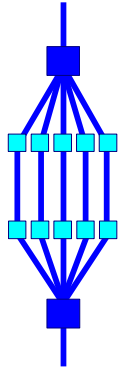
- ✓ All threads have access to the same, globally shared, memory
- ✓ Data can be shared or private
- ✓ Shared data is accessible by all threads
- ✓ Private data can only be accessed by the thread that owns it
- ✓ Data transfer is transparent to the programmer
- ✓ Synchronization takes place, but it is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

# Data-Sharing Attributes

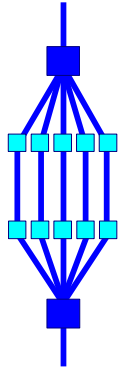


- *In an OpenMP program, data needs to be “labelled”*
- *Essentially there are two basic types:*
  - *Shared*
    - ✓ *There is only instance of the data*
    - ✓ *All threads can read and write the data simultaneously, unless protected through a specific OpenMP construct*
    - ✓ *All changes made are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



# A first OpenMP example



## For-loop with independent iterations

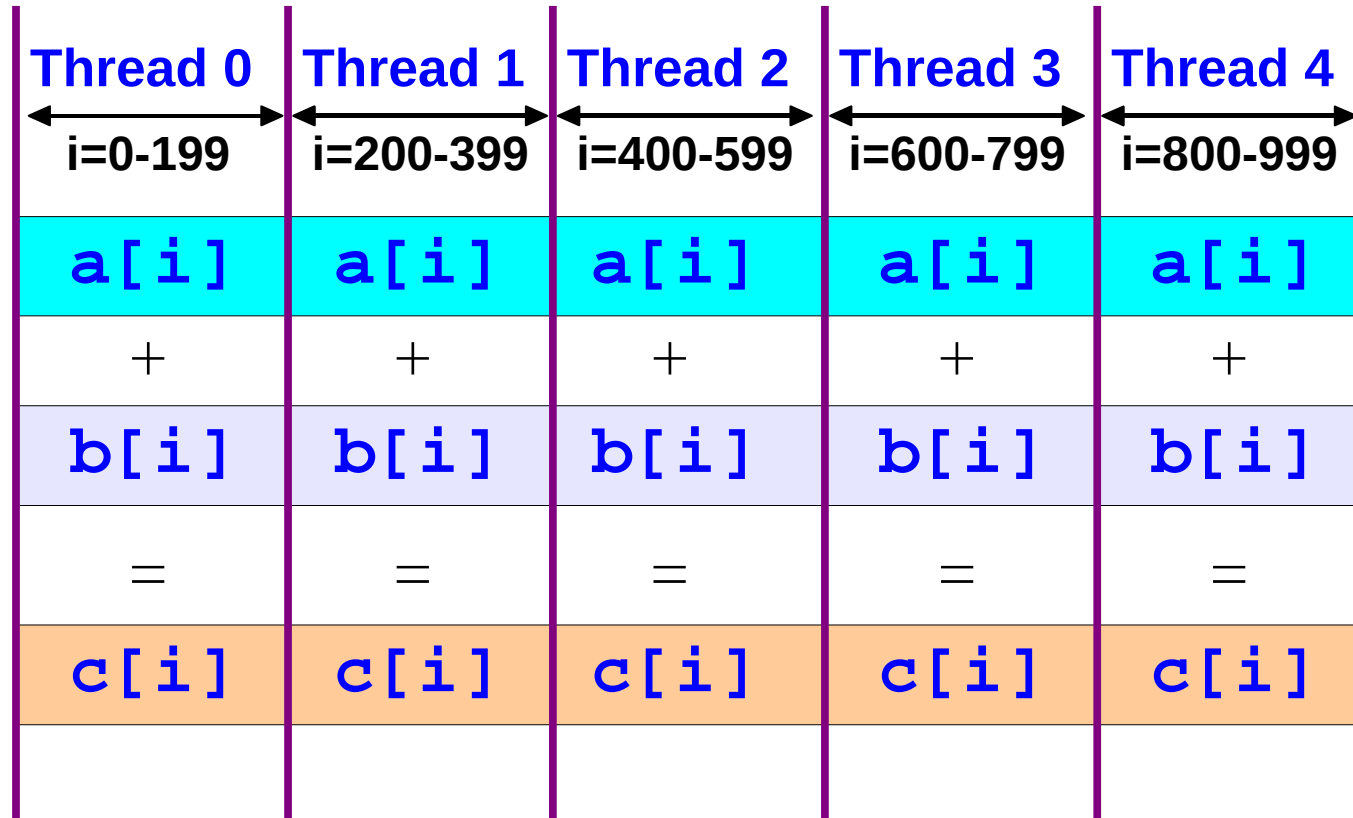
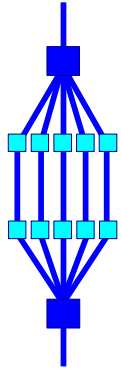
```
for (int i=0; i<n; i++)  
    c[i] = a[i] + b[i];
```

## For-loop parallelized using an OpenMP pragma

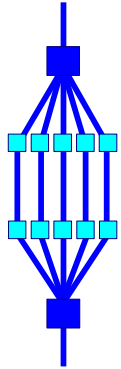
```
#pragma omp parallel for  
for (int i=0; i<n; i++)  
    c[i] = a[i] + b[i];
```

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

# Example parallel execution



# Components of OpenMP 2.5



## Directives

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

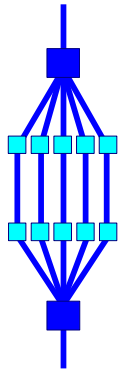
## Runtime environment

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

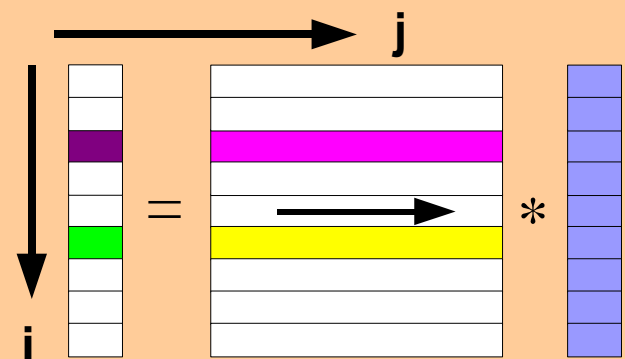
## Environment variables

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

# 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

TID = 1

```
for (i=0,1,2,3,4)
```

```

i = 0
sum =  $\sum$  b[i=0][j]*c[j]
a[0] = sum
```

```

i = 1
sum =  $\sum$  b[i=1][j]*c[j]
a[1] = sum
```

```
for (i=5,6,7,8,9)
```

```

i = 5
sum =  $\sum$  b[i=5][j]*c[j]
a[5] = sum
```

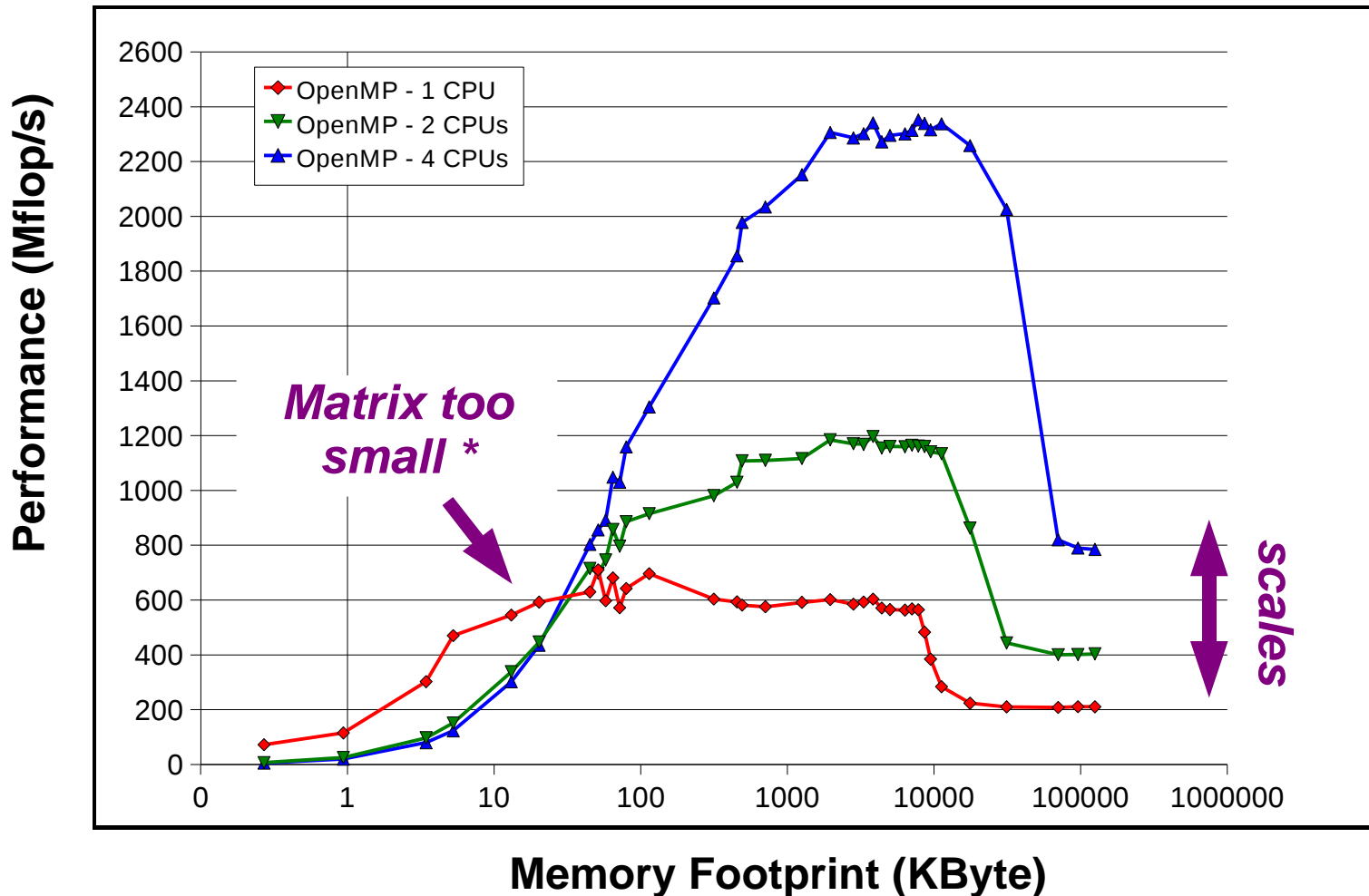
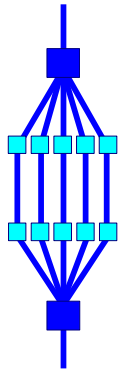
```

i = 6
sum =  $\sum$  b[i=6][j]*c[j]
a[6] = sum
```

... etc ...

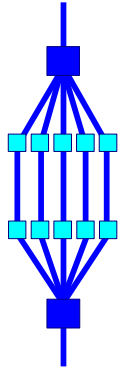


# OpenMP performance



*\*) 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

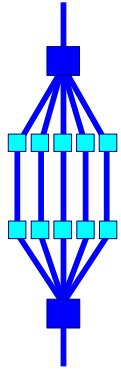
parallel loop  
(work is distributed)

parallel loop  
(work is distributed)

synchronization

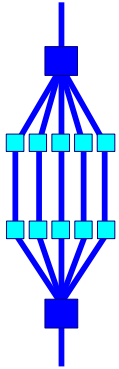
Statement is executed by all threads

parallel region



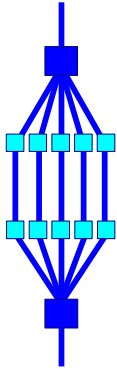
# *OpenMP In Some More Detail*

# 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***

# 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 --*/
```

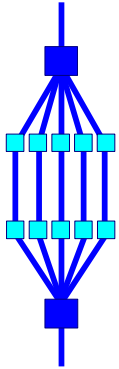
## private (list)

- ✓ *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*

# Barrier/1



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

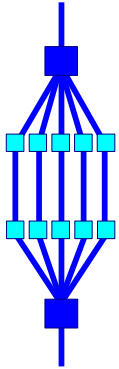
```
for (i=0; i < N; i++)  
    a[i] = b[i] + c[i];
```

```
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 !*

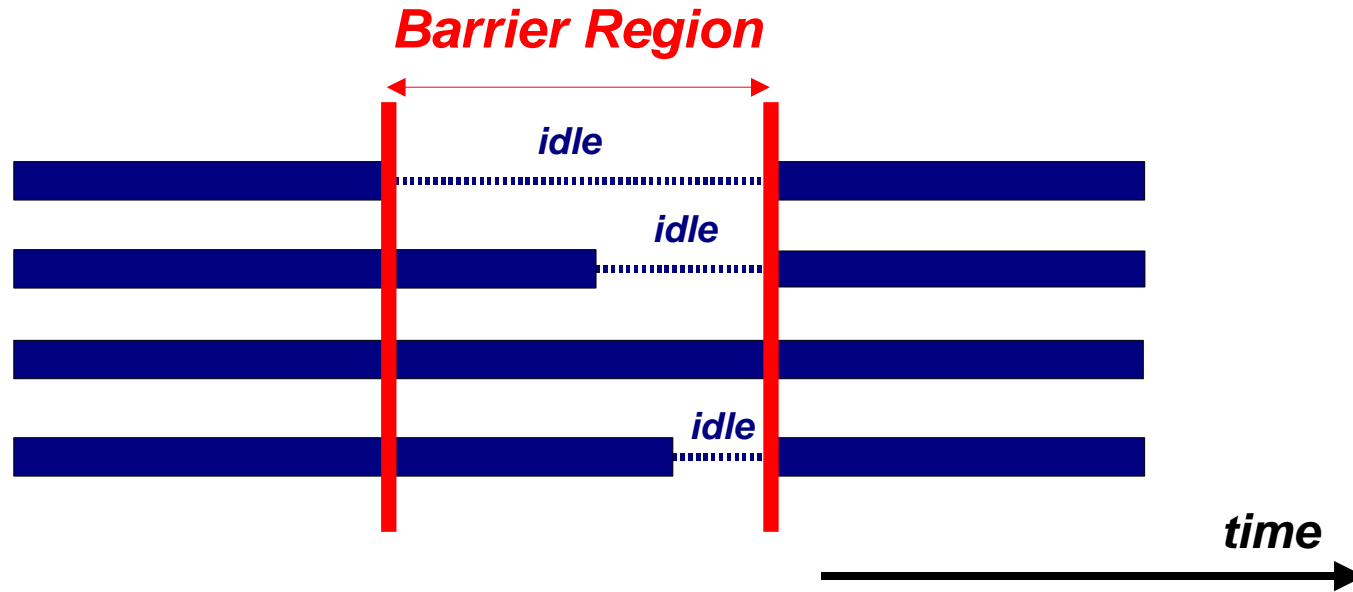
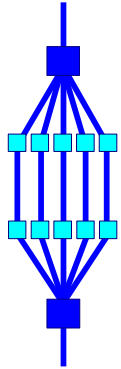
*barrier*

```
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



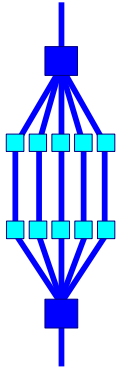
## Barrier syntax in OpenMP:

```
#pragma omp barrier
```

```
!$omp barrier
```



# The nowait clause

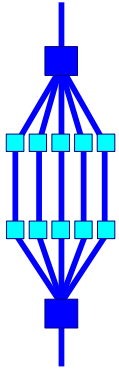


- ❑ *To minimize synchronization, some OpenMP directives/pragmas support the optional **nowait** clause*
- ❑ *If present, threads do not synchronize/wait at the end of that particular construct*
- ❑ *In Fortran the **nowait** clause is appended at the closing part of the construct*
- ❑ *In C, it is one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

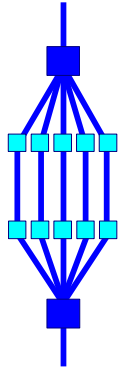
```
{
```

*"this is executed in parallel"*

```
} (implied barrier)
```

# Work-sharing constructs

## The OpenMP work-sharing constructs



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

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

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

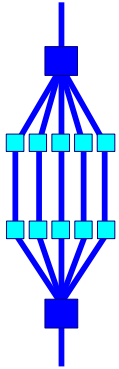
```
!$OMP SECTIONS  
    . . . . .  
!$OMP END SECTIONS
```

```
#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>
```

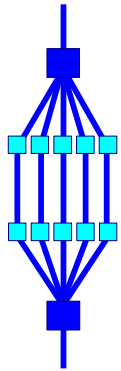
```
!$omp do [clause[[,] clause] ...]  
    <original do-loop>  
!$omp end do [nowait]
```

## *Clauses supported:*

private	firstprivate
lastprivate	reduction
<i>ordered*</i>	<i>schedule</i> ← <i>covered later</i>
nowait	

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

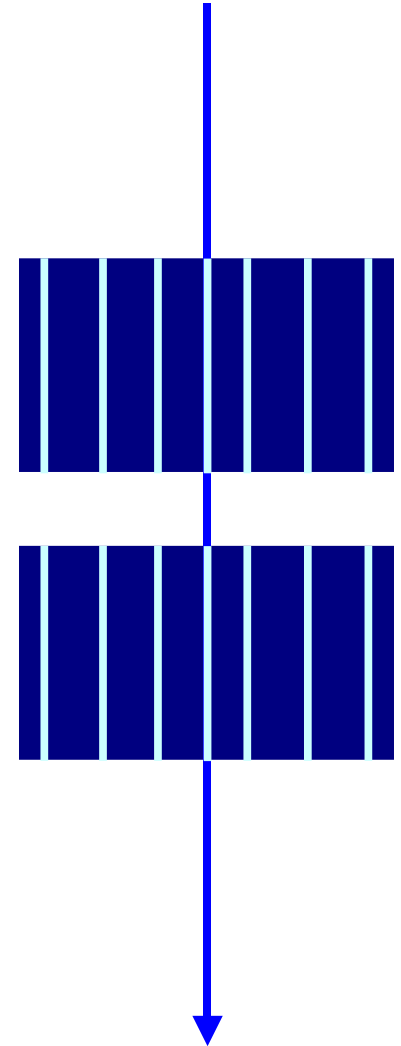
# The omp for directive - Example



```
#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;

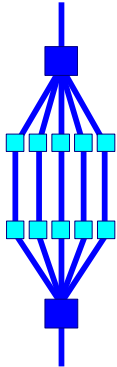
    #pragma omp for nowait
    for (i=0; i<n; i++)
        d[i] = 1.0/c[i];

} /* -- 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  
    :  
}
```

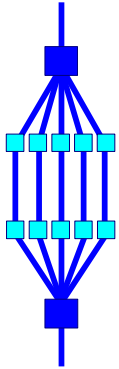
```
!$omp sections [clause(s)]  
!$omp section  
    <code block1>  
!$omp section  
    <code block2>  
!$omp section  
    :  
!$omp end sections [nowait]
```

## **Clauses supported:**

private            firstprivate  
lastprivate        reduction  
nowait

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

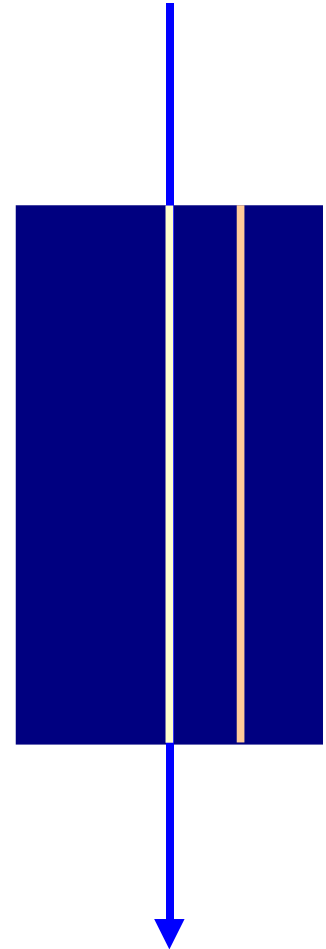
# The sections directive - Example



```
#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;

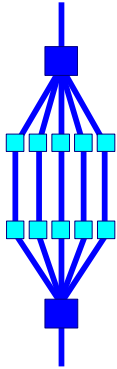
        #pragma omp section
        for (i=0; i<n; i++)
            d[i] = 1.0/c[i];

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

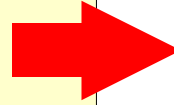




# Combined work-sharing constructs



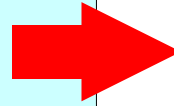
```
#pragma omp parallel
#pragma omp for
  for (...)
```



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

*Single PARALLEL loop*

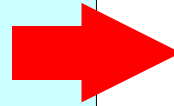
```
!$omp parallel
!$omp do
  ...
!$omp end do
!$omp end parallel
```



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

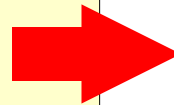
*Single WORKSHARE loop*

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



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

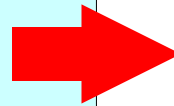
```
#pragma omp parallel
#pragma omp sections
{ ... }
```



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

*Single PARALLEL sections*

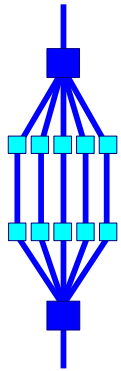
```
!$omp parallel
!$omp sections
  ...
!$omp end sections
!$omp end parallel
```



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

# Single processor region/1

*This construct is ideally suited for I/O or initializations*



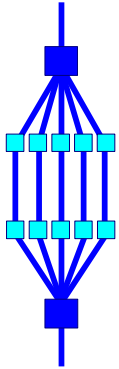
*Original Code*

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

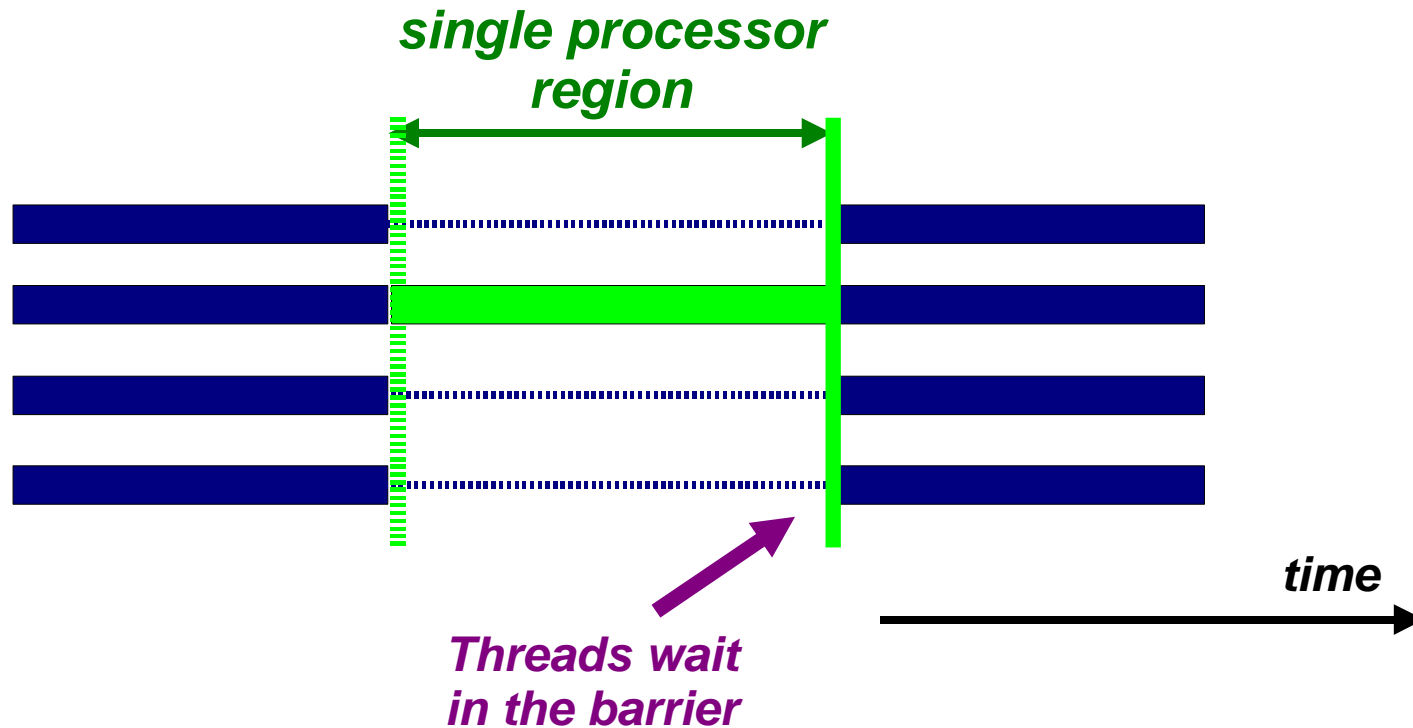
```
"declare A to be shared"  
  
#pragma omp parallel  
{  
.....  
.....  
one volunteer requested  
.....  
"read a[0..N-1]";  
.....  
thanks, we're done  
.....  
}  
  
Parallel Version
```

May have to insert a barrier here

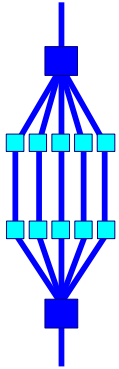
# Single processor region/2



- *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]
```

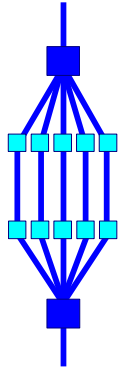
*Only the master thread executes the code block:*

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

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

*There is no implied  
barrier on entry or  
exit !*

# Critical Region/1



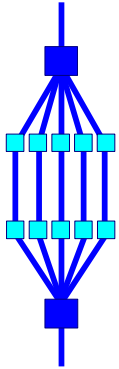
*If sum is a shared variable, this loop can not run in parallel*

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

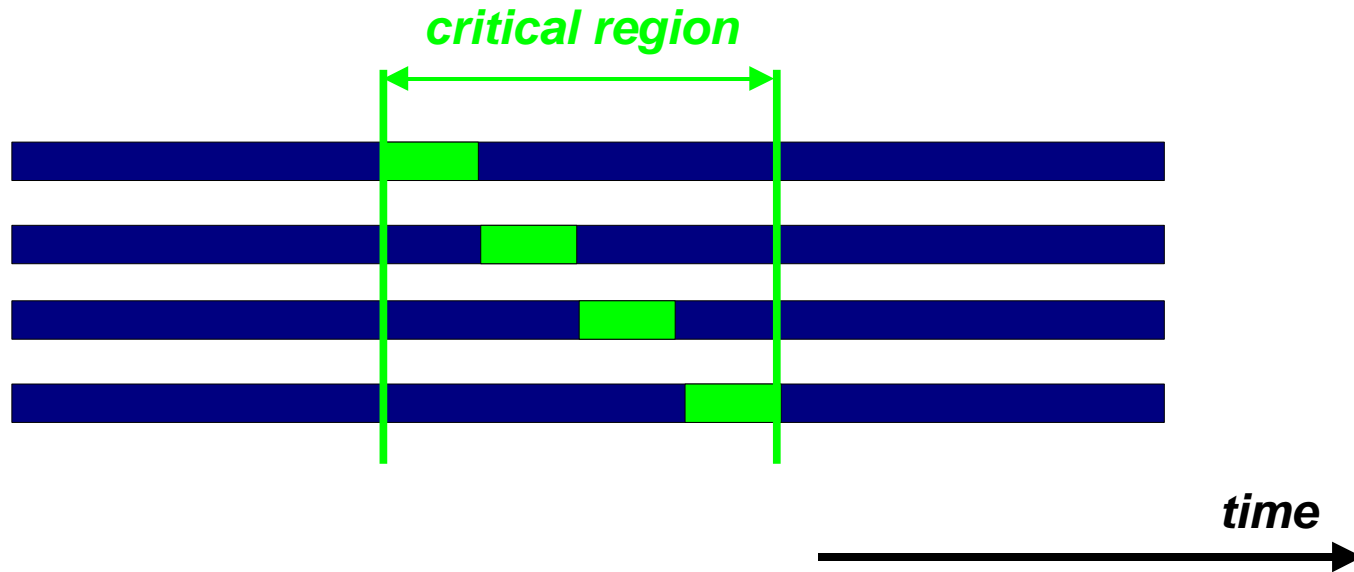
*We can use a critical region for this:*

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

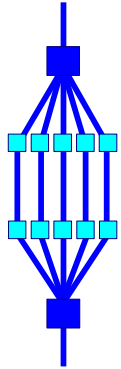
# Critical Region/2



- *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 !*

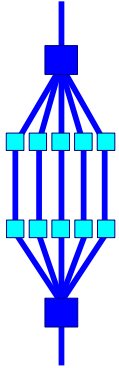
*Atomic: only the loads and store are atomic ....*

```
#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];
```



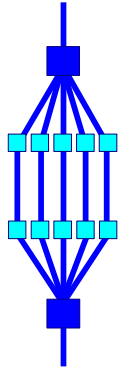
## *Why The Excitement About OpenMP 3.0 ?*

***Support for TASKS !***

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



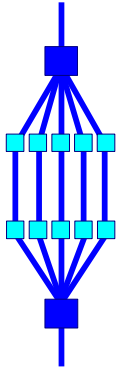
# Example - A Linked List



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

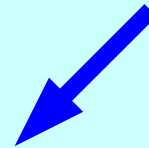
***Hard to do before OpenMP 3.0:  
First count number of iterations, then  
convert while loop to for loop***

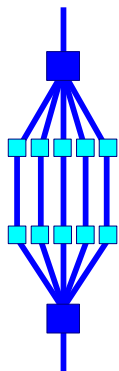
# Example - A Linked List With Tasking



```
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
```

OpenMP Task is specified  
here  
(executed in parallel)

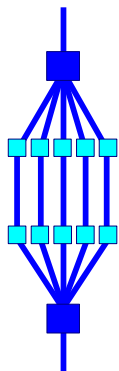




# *Case Study*

## *A Neural Network*

# Neural Network application\*



## Performance Analyzer Output

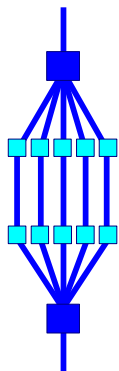
Excl. User CPU sec.	User CPU %	Incl. User CPU sec.	Excl. Wall sec.	Name
120.710	100.0	120.710	128.310	<Total>
116.960	96.9	116.960	122.610	calc_r_loop_on_neighbours
0.900	0.7	118.630	0.920	calc_r
0.590	0.5	1.380	0.590	_doprnt
0.410	0.3	1.030	0.430	init_visual_input_on_V1
0.280	0.2	0.280	1.900	_write
0.200	0.2	0.200	0.200	round_coord_cyclic
0.130	0.1	0.130	0.140	__arint_set_n
0.130	0.1	0.550	0.140	__k_double_to_decimal
0.090	0.1	1.180	0.090	fprintf

## Callers-callees fragment:

Attr. User CPU sec.	Excl. User CPU sec.	Incl. User CPU sec.	Name
116.960	0.900	118.630	calc_r
116.960	116.960	116.960	*calc_r_loop_on_neighbours

*\*) Program was said not to scale on a Sun SMP system....*

# Source line information



```
struct cell{
    double x; double y; double r; double I;
};
```

.....

```
struct cell V1[NPOSITIONS_Y][NPOSITIONS_X];
double      h[NPOSITIONS][NPOSITIONS];
```

.....

Excl. sec.	User CPU %	Excl. Wall sec.
------------	------------	-----------------

```
1040. void
```

```
1041. calc_r_loop_on_neighbours
      (int y1, int x1)
```

0.080	0.1	0.080
-------	-----	-------

```
1042. {
1043. struct interaction_structure *next_p;
1044.
```

0.130	0.1	0.130
-------	-----	-------

```
1045. for (next_p = JJ[y1][x1].next;
```

0.460	0.4	0.470
-------	-----	-------

```
1046.     next_p != NULL;
```

```
1047.     next_p = next_p->next) {
```

## 116.290	96.3	121.930
------------	------	---------

```
1048.     h[y1][x1] += next_p->strength *
          V1[next_p->y][next_p->x].r;
```

*96% of the time spent in  
this single statement*

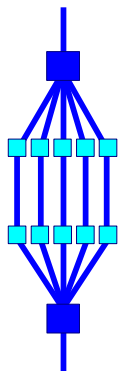
```
1049.
```

```
1052. }
```

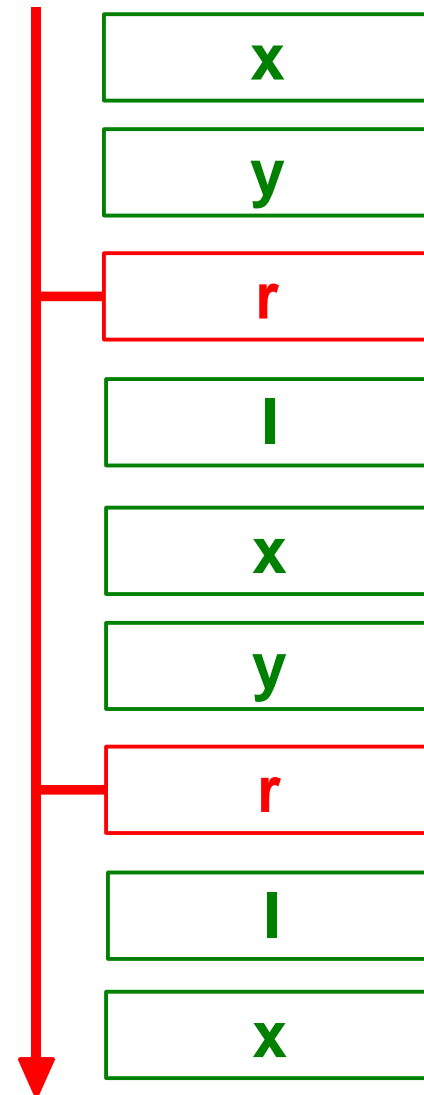
```
1053. }
```

*What is the  
problem ?*

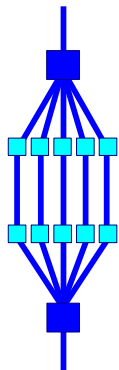
# Data structure problem



- ❑ *We only use 1/4 of a cache line !*
- ❑ *For sufficiently large problems this will:*
  - *Generate additional memory traffic*
    - ✓ *Higher interconnect pressure*
  - *Waste data cache capacity*
    - ✓ *Reduces temporal locality*
- ❑ *The above negatively affects both serial and parallel performance*
- ❑ *Fix: split the structure into two parts*
  - *One contains the "r" values only*
  - *The other one contains the {x,y,l} sets*



# Fragment of modified code



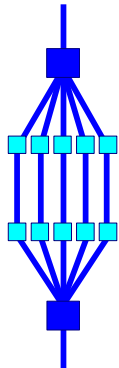
```
double V1_R[NPOSITIONS_Y][NPOSITIONS_X];

void
calc_r_loop_on_neighbours(int y1, int x1)
{
    struct interaction_structure *next_p;

    double sum = h[y1][x1];

    for (next_p = JJ[y1][x1].next;
         next_p != NULL;
         next_p = next_p->next) {
        sum += next_p->strength * V1_R[next_p->y][next_p->x];
    }
    h[y1][x1] = sum;
}
```

# Parallelization with OpenMP



```
void calc_r(int t)
{
#include <omp.h>

#pragma omp parallel for default(none) \
        private(y1,x1) shared(h,V1,g,T,beta_inv,beta)
```

```
    for (y1 = 0; y1 < NPOSITIONS_Y; y1++) {
        for (x1 = 0; x1 < NPOSITIONS_X; x1++) {

            calc_r_loop_on_neighbours(y1,x1);
            h[y1][x1] += V1[y1][x1].I;

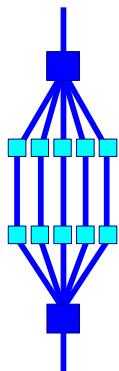
            <statements deleted>
        }
    }
```

*Can be executed  
in parallel*

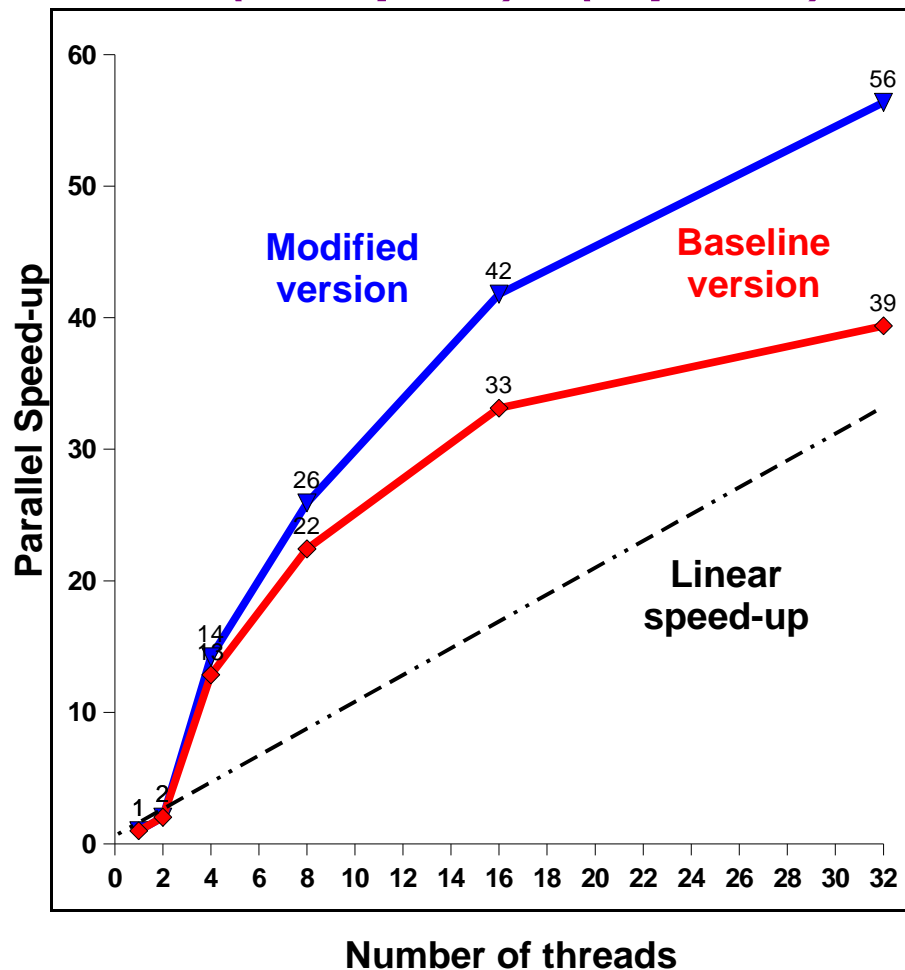
```
/*-- End of OpenMP parallel for --*/
```



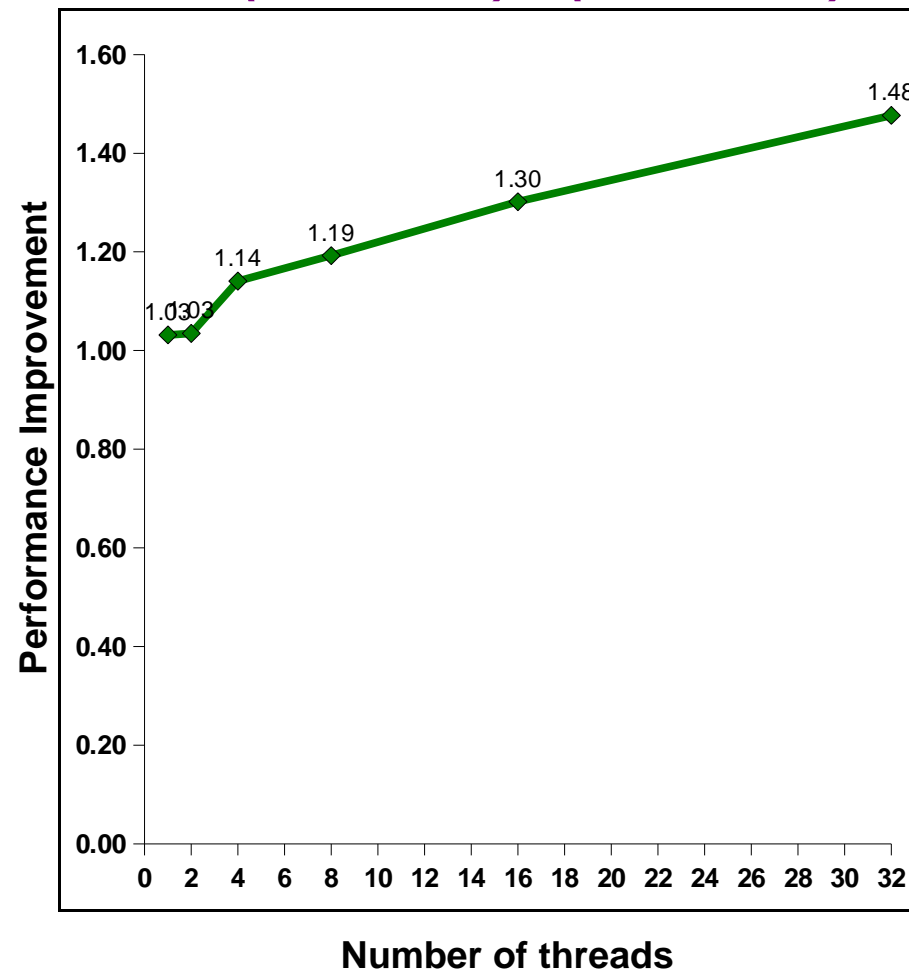
# Scalability results



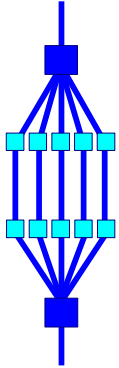
$T(\text{One proc})/T(P \text{ procs})$



$T(\text{baseline})/T(\text{modified})$



**Note:**  
 Single processor run time is 5001 seconds for the baseline version (4847 for the modified version)



***That's It***

***Thank You and ..... Stay Tuned !***

***Ruud van der Pas***  
***ruud.vanderpas@sun.com***