Parallel programming

S.I

// programming

Françoise Baude

Université Côte d'Azur

Polytech Nice Sophia

baude@unice.fr

web site: https://lms.univ-cotedazur.fr/course/view.php?id=17901

(self registration)

March 2023

Chapter : OpenMP

Programming in OpenMP

- OpenMP model enables to exploit all variants of control parallelism
- It assumes a global address space (shared variables among tasks)

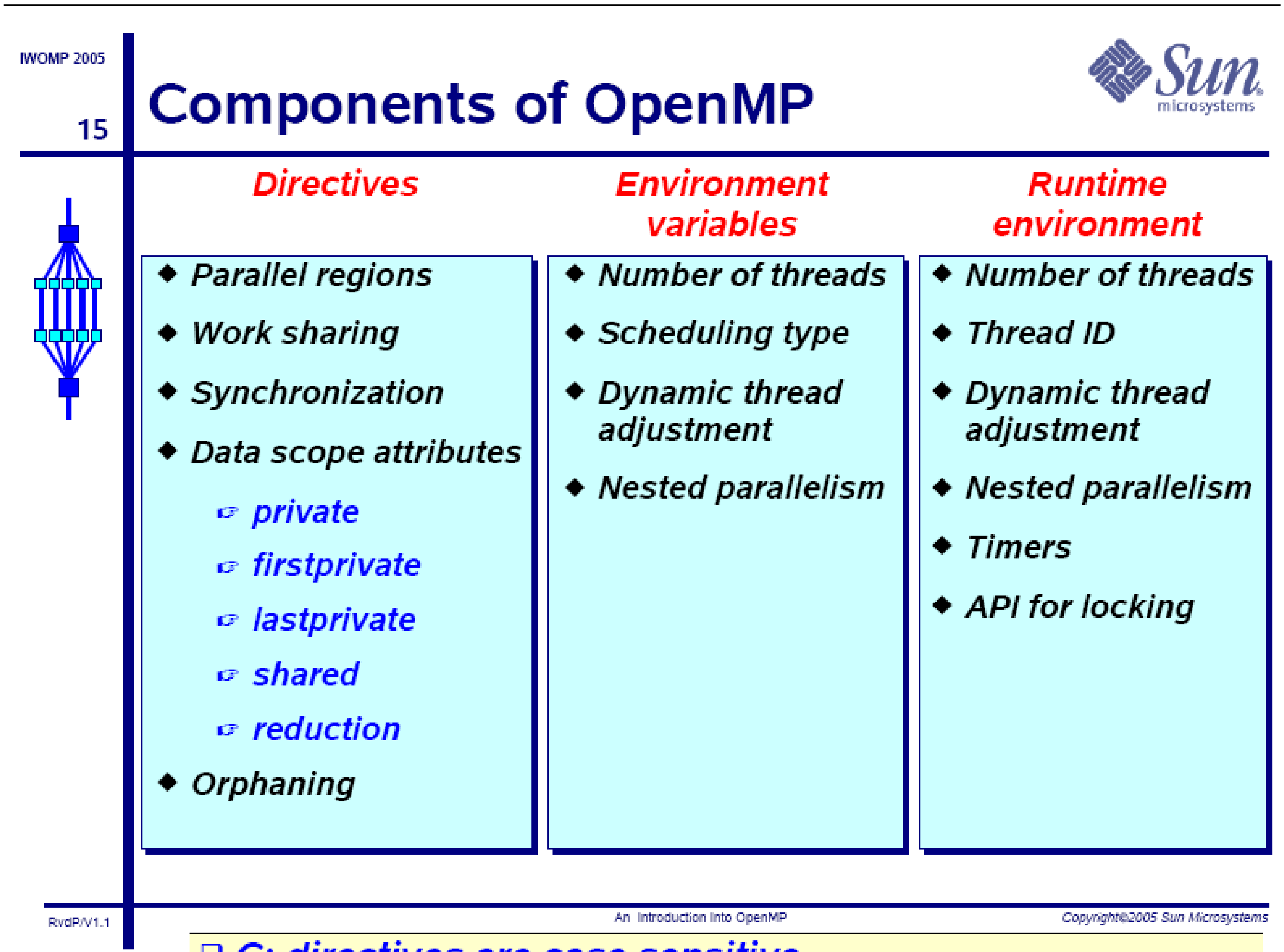
When to consider using OpenMP?

- *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 dependency 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*

OpenMP: Shared Memory

Application Programming Interface

- Multiplatform shared memory multi-threads programming
- Compiler directives, library routines, and environnement variables
- For C/C++ and Fortran
- Tutorials available, eg www.openmp.org



IWOMP 2005

15

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*

RvdP/V1.1 An Introduction Into OpenMP *Copyright©2005 Sun Microsystems*

❑ *C: directives are case sensitive*

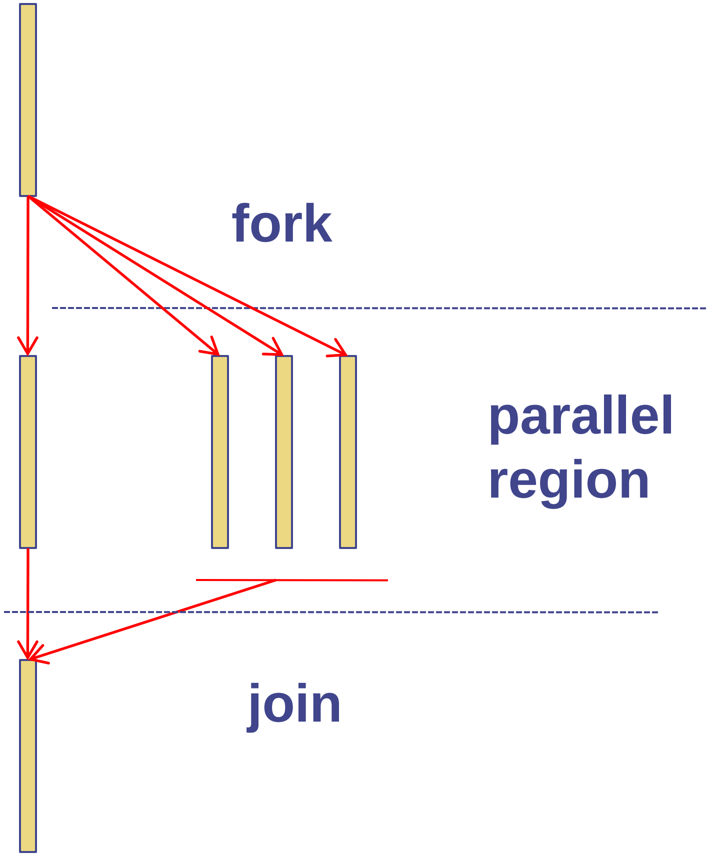
- *Syntax:* #pragma omp directive [clause [clause] ...]

❑ *Continuation: use \ in pragma*

❑ *Conditional compilation:* _OPENMP *macro is set*

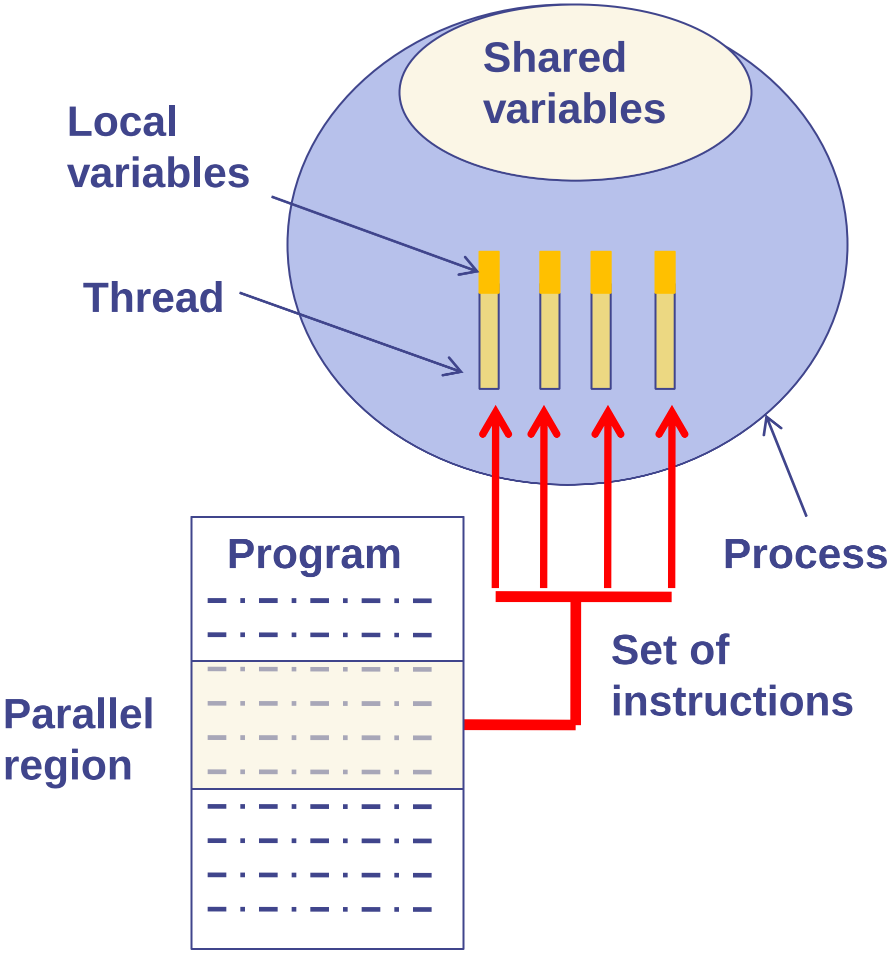
OpenMP

- The programmer has to introduce OpenMP directives within the code
- When program is executed, a parallel region will be created on the "fork and join" model



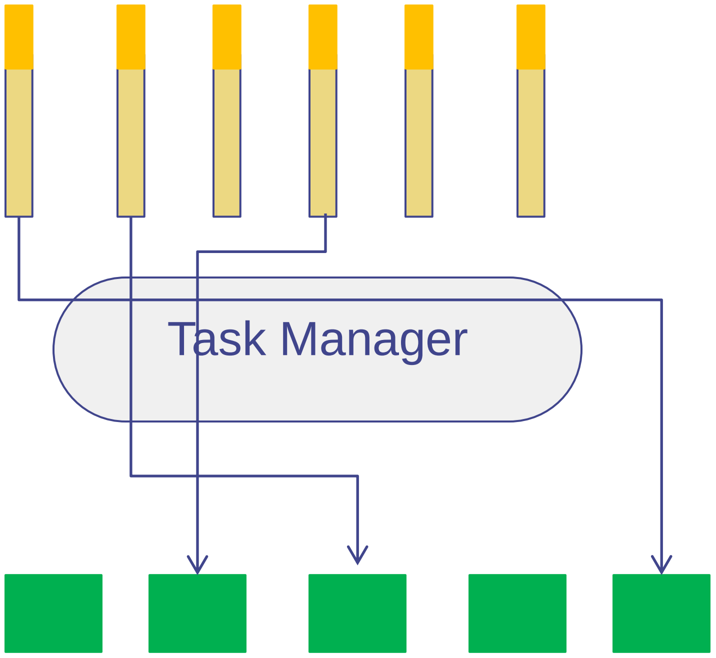
OpenMP: General Concepts

- An OpenMP program is executed by a unique process
- This process activates threads when entering a parallel region
- Each thread executes some tasks composed by several instructions
- Two kinds of variables:
  - Private
  - Shared



OpenMP 3+: General Concepts

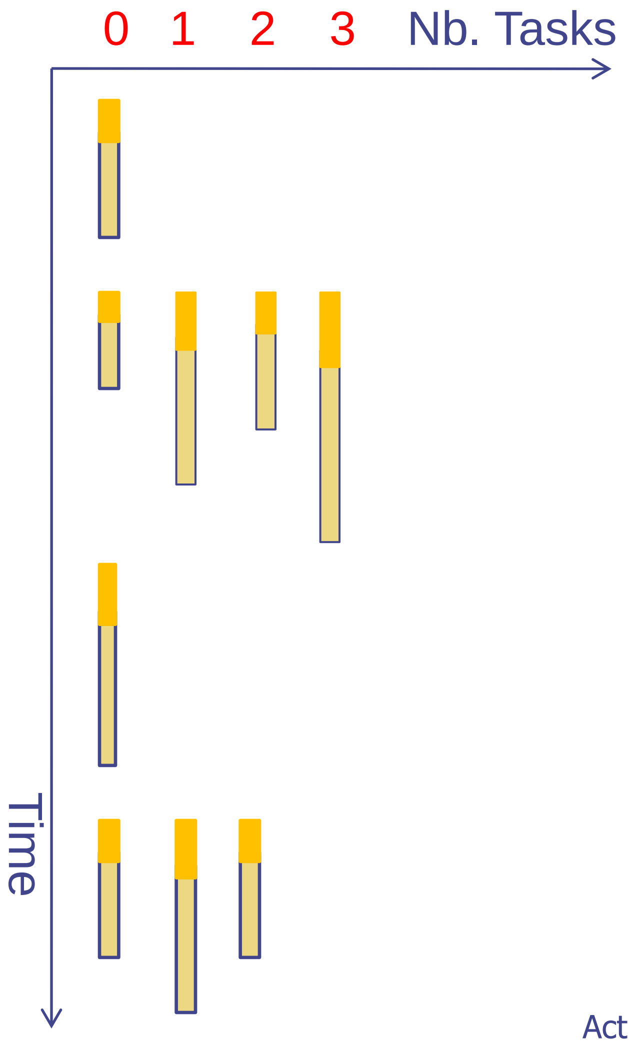
- A task is affected to a thread by the runtime system
- OpenMP 2.5: task and thread concepts are the same (ie, no explicit tasking model)



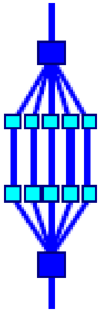
Threads on processors

OpenMP: General Concepts

- An OpenMP program is an alternation of sequential and parallel regions
- A sequence region is always executed by the master thread
- A parallel region can be executed by several threads at the same time



Terminology



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

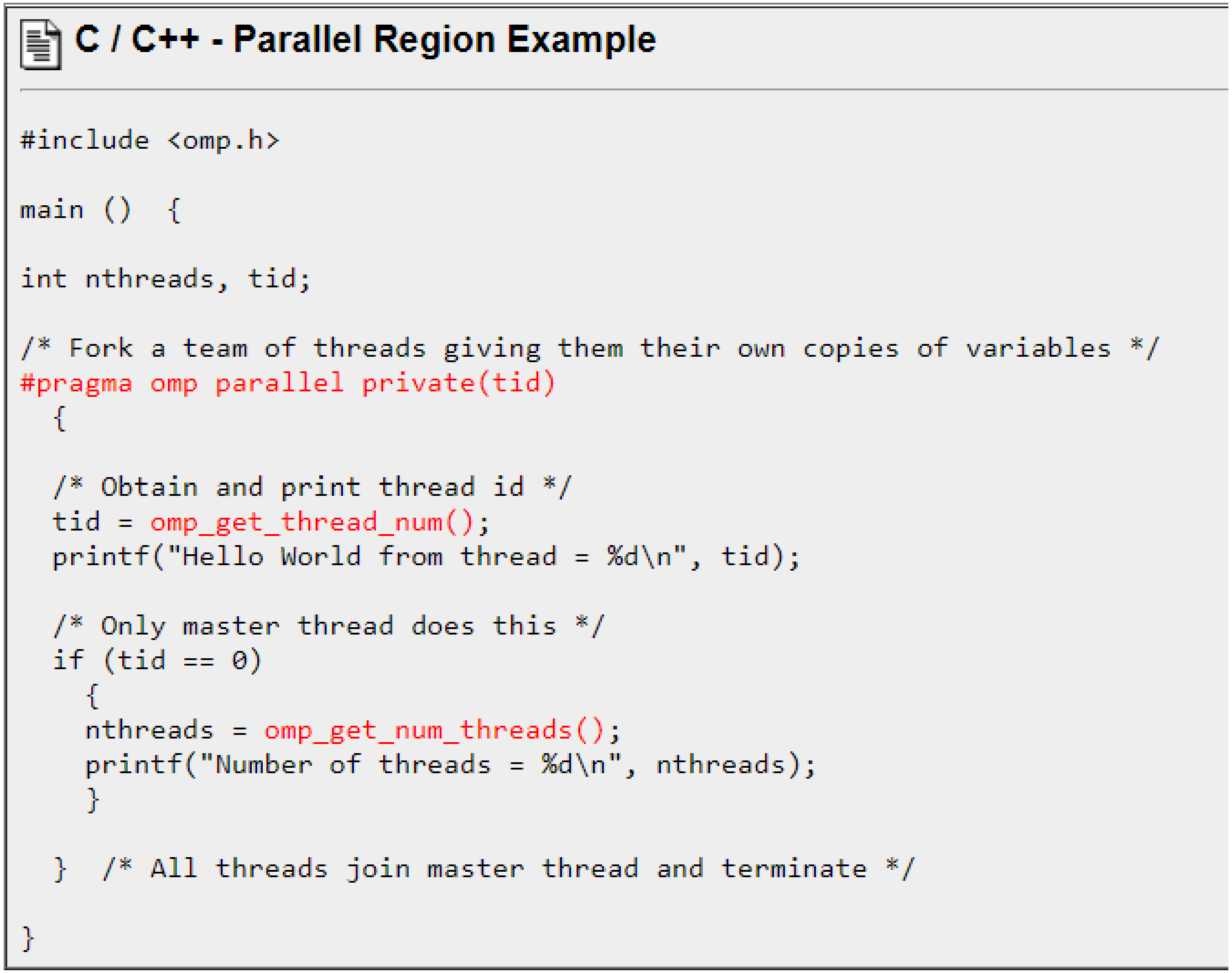
OpenMP Basics: Parallel region

- inside a parallel region:
  - by default, variables are shared
  - all concurrent tasks execute the same code
- there is a default synchronization barrier at the end of a parallel region

```
Program parallel
     use OMP_LIB
     implicit none
     real  ::a
     logical ::p

     a=9999.  ; p= false.
     !$OMP PARALLEL
        !$ p = OMP_IN_PARALLEL()
        print *, “A value is :”,a &
                    “; p value is: ”,p
     !$OMP END PARALLEL
end program parallel
```

```
> export OMP_NUM_THREADS=3; a. out;
> A value is 9999.  ; p value is: T
> A value is 9999.  ; p value is: T
> A value is 9999.  ; p value is: T
```

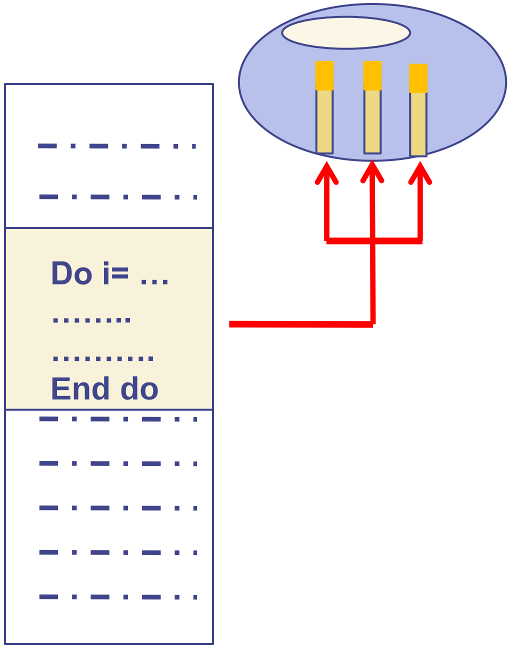
Parallel region without work sharing



C / C++ - Parallel Region Example

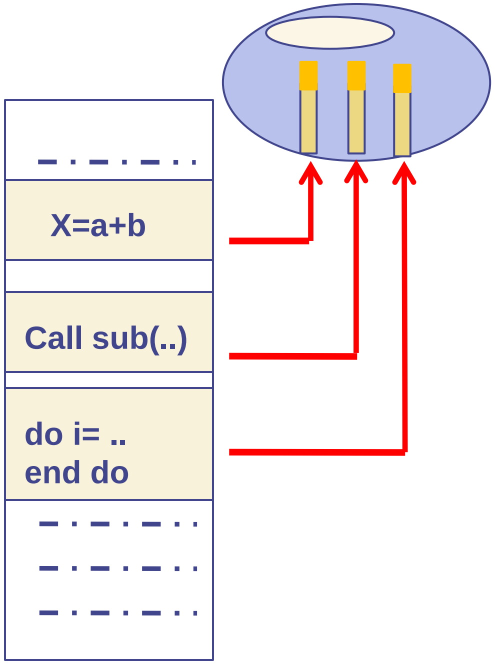
```
#include <omp.h>

main ()  {

int nthreads, tid;

/* Fork a team of threads giving them their own copies of variables */
#pragma omp parallel private(tid)
  {

  /* Obtain and print thread id */
  tid = omp_get_thread_num();
  printf("Hello World from thread = %d\n", tid);

  /* Only master thread does this */
  if (tid == 0)
    {
    nthreads = omp_get_num_threads();
    printf("Number of threads = %d\n", nthreads);
    }

  }  /* All threads join master thread and terminate */

}
```

OpenMP: Work-sharing Concepts



Do i= …
……..
……….
End do

Looplevel parallelism

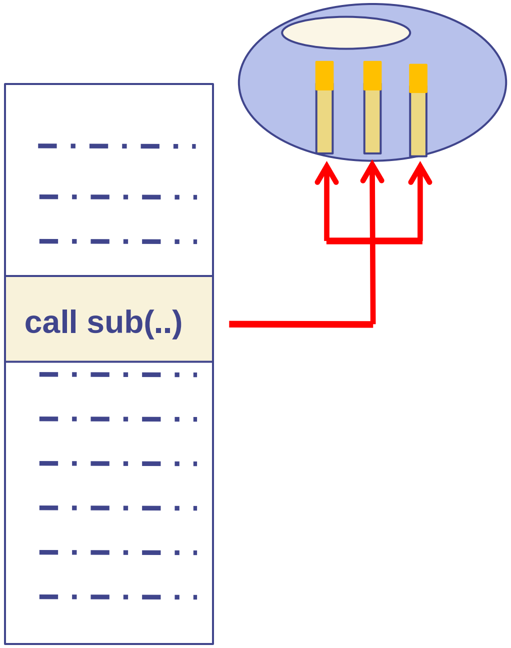


X=a+b

Call sub(..)

do i= ..
end do

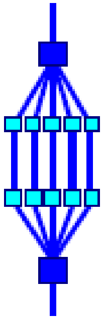
Parallel sections



call sub(..)

Parallel procedure (orphaning)

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

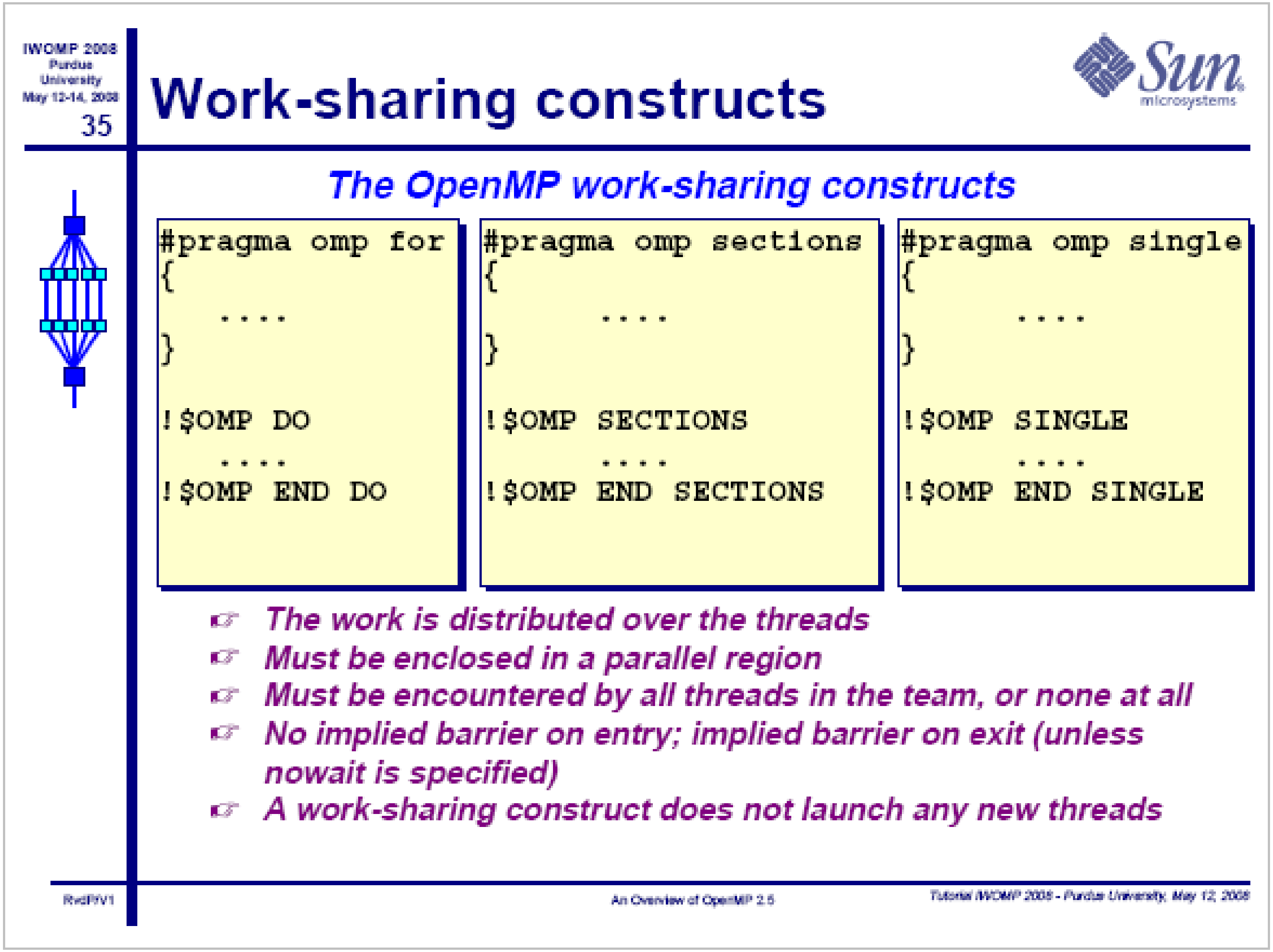
parallel loop
(work will be distributed)

parallel loop
(work will be distributed)

synchronization

Statement is executed by all threads

parallel region



Work-sharing constructs

The OpenMP work-sharing constructs

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

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

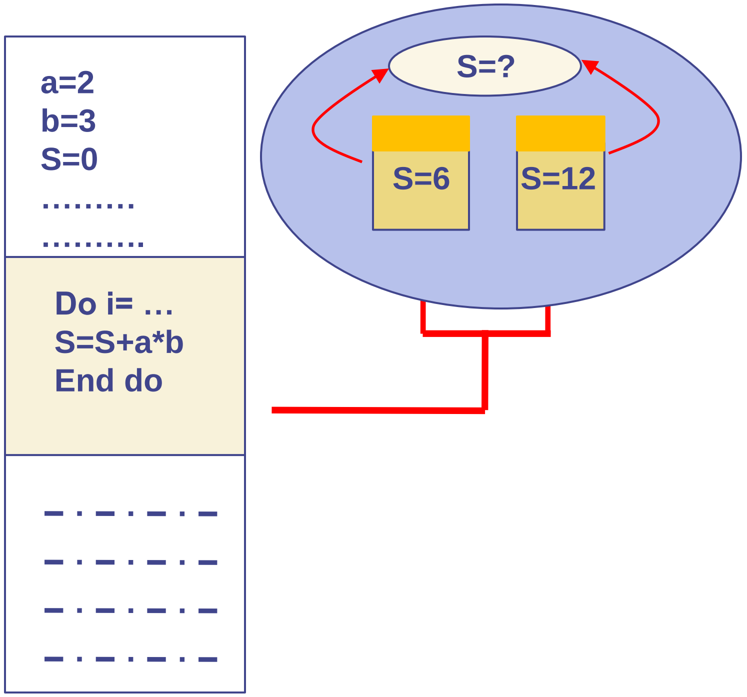
```
#pragma omp sections
{
        ....
}

!$OMP SECTIONS
        ....
!$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*

OpenMP: Synchronization Concepts

- ▶ Concurrency problems
- ▶ The need of task synchronization

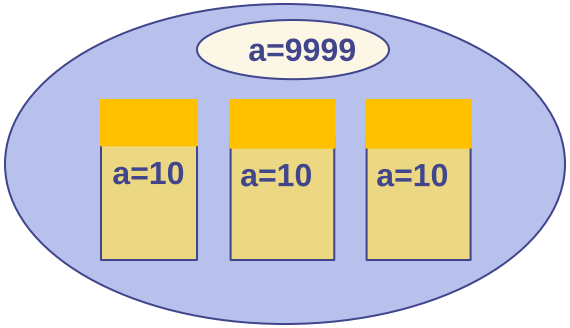


```
a=2
b=3
S=0
.........
..........
Do i= ...
S=S+a*b
End do
```

OpenMP Basics: Parallel region

- By using the DEFAULT clause one can change the default status of a variable within a parallel region
- If a variable has a private status (PRIVATE) an instance of it (with an undefined value) will exist in the stack of each task.

```
Program parallel
     use OMP_LIB
     implicit none
     real  ::a
     a=9999.
     !$OMP PARALLEL DEFAULT(PRIVATE)
        a=a+10.
        print *, "A value is : ",a
     !$OMP END PARALLEL
end program parallel
```

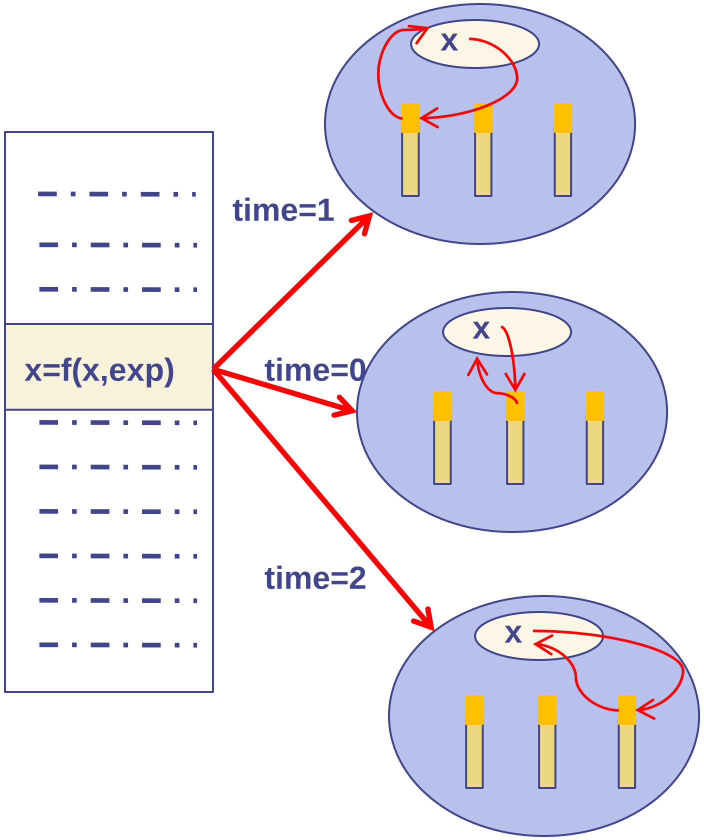


```
> export OMP_NUM_THREADS=3; a. out;
> A value is :  10
> A value is :  10
> A value is :  10
```

OpenMP: Synchronizations

Automatic update

- The **ATOMIC** directive guarantees that a shared variable is read and modified by a single task at a given moment



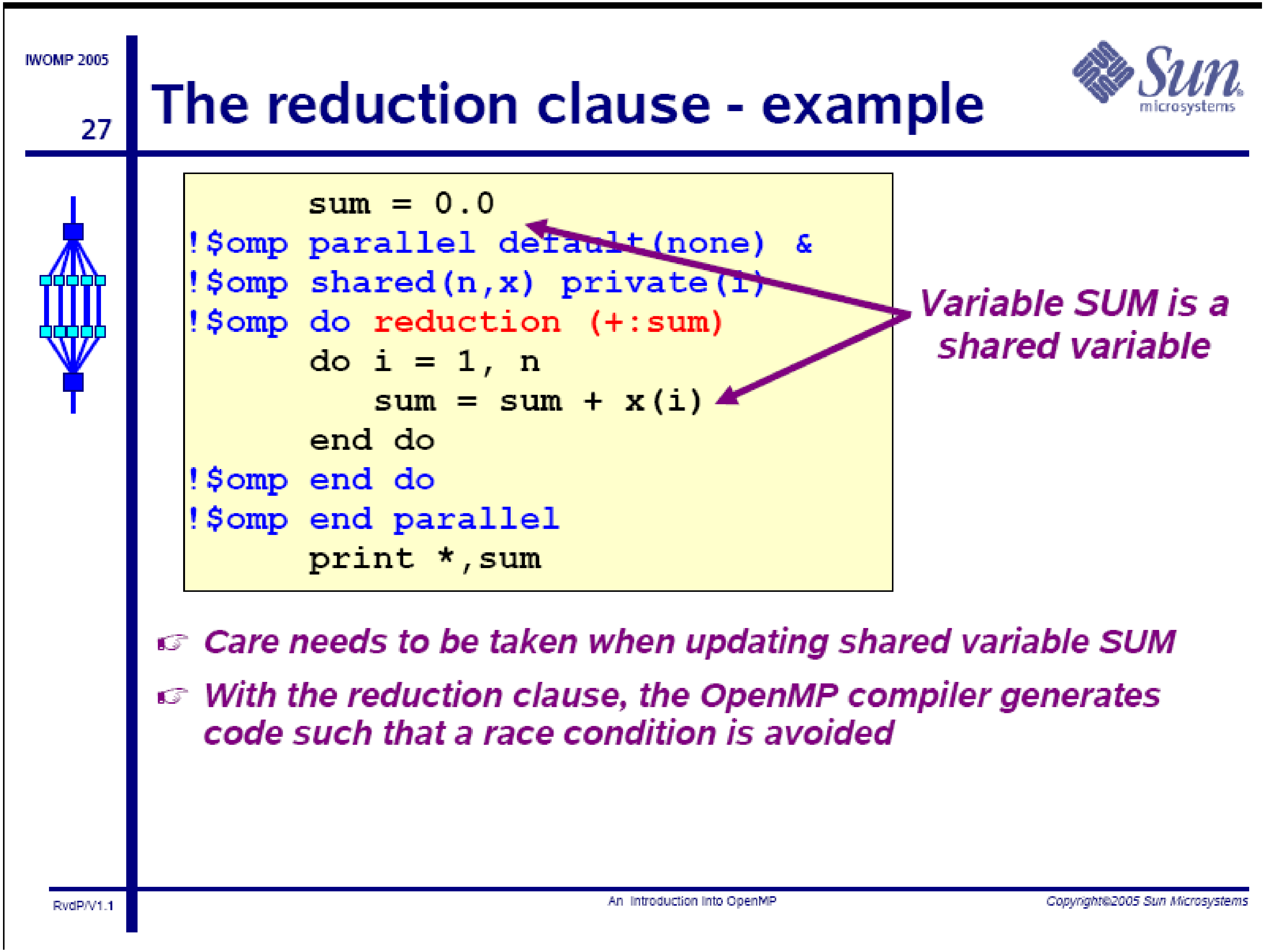
OpenMP : Synchronizations

Critical Region

- defined with the **CRITICAL** directive
- generalization of **ATOMIC** directive
- tasks in the critical region are executed in an un-deterministic order but one by one.

```
program parallel
      implicit none
      integer :: s, p
      s=0 ; p=1
      !$OMP PARALLEL
            !$OMP CRITICAL
                  s = s + 1
                  p = p * 2
            !$OMP END CRITICAL
      !$OMP END PARALLEL
      print *, "s=", s, " ; p=", p
end program parallel
```

```
> xlf_r ... -qsmp=omp prog.f90
> export OMP_NUM_THREADS =4 ; a.out

> s= 4 ; p= 16
```



The reduction clause - example

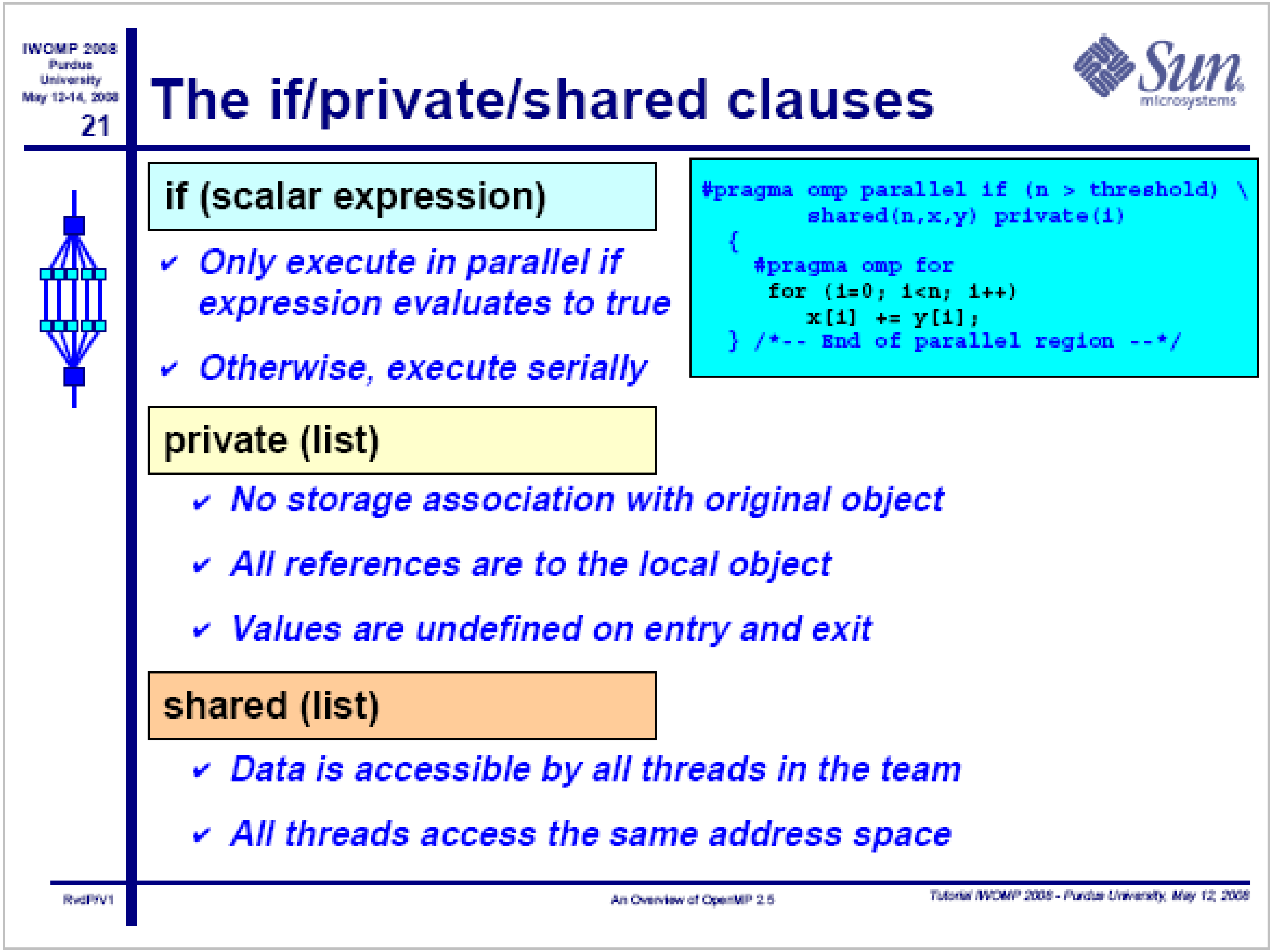
```
      sum = 0.0
!$omp parallel default(none) &
!$omp shared(n,x) private(i)
!$omp do reduction (+:sum)
      do i = 1, n
         sum = sum + x(i)
      end do
!$omp end do
!$omp end parallel
      print *,sum
```

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*

More Advanced topics

- About visibility of variables, global and private
- About the work sharing construct *sections*
- About the way to guide work sharing of for loops on threads (load distribution, aka scheduling)
- About nested parallelism
- One complete and interesting tuto: https://www.capsl.udel.edu/courses/cpeg421/2012/slides/openmp_tutorial_04_06_2012.pdf



The if/private/shared clauses

Sun microsystems

if (scalar expression)

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

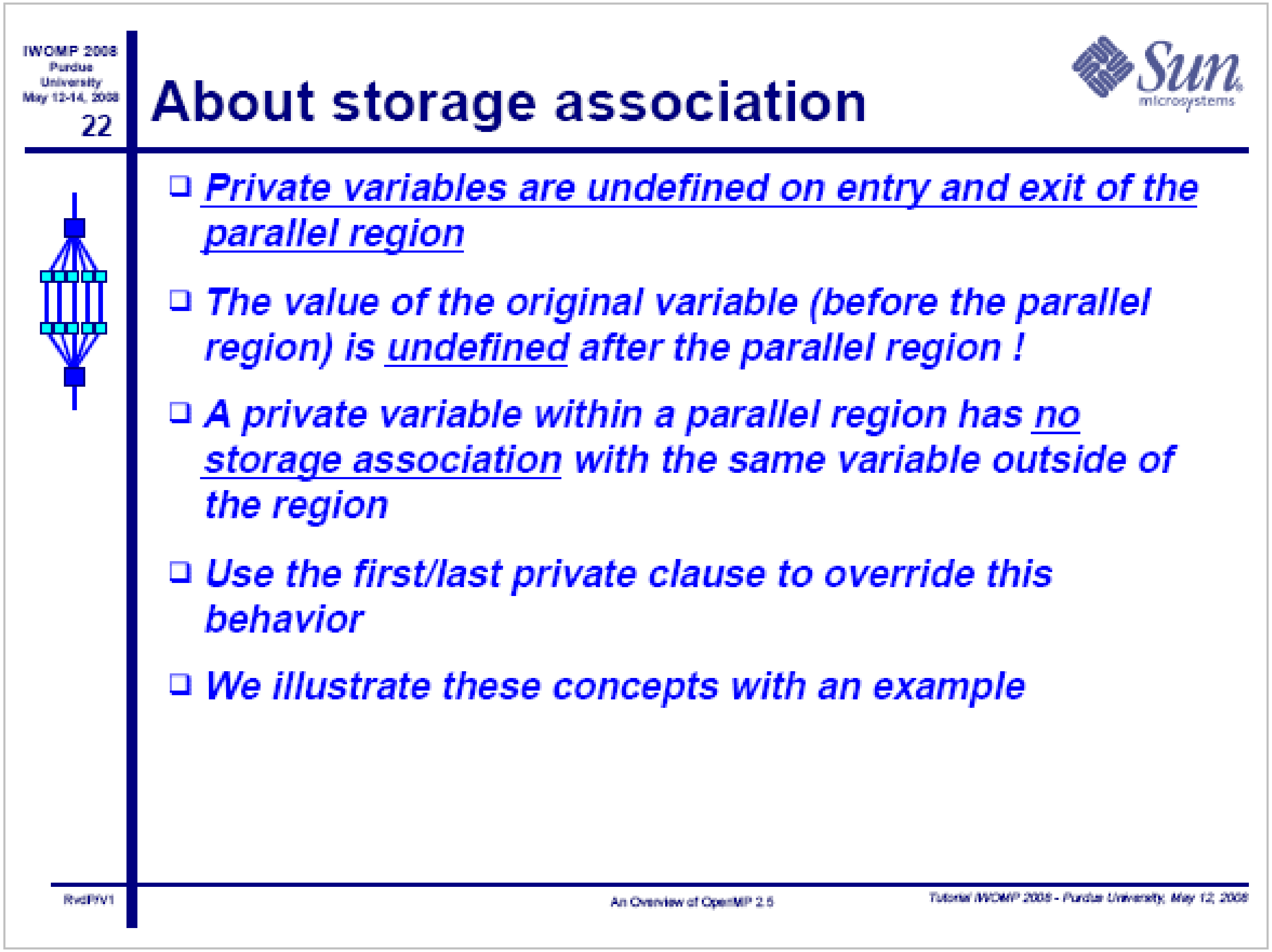
```
#pragma omp parallel if (n > threshold) \
        shared(n,x,y) private(i)
  {
    #pragma omp for
     for (i=0; i<n; i++)
        x[i] += y[i];
  } /*-- End of parallel region --*/
```

private (list)

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

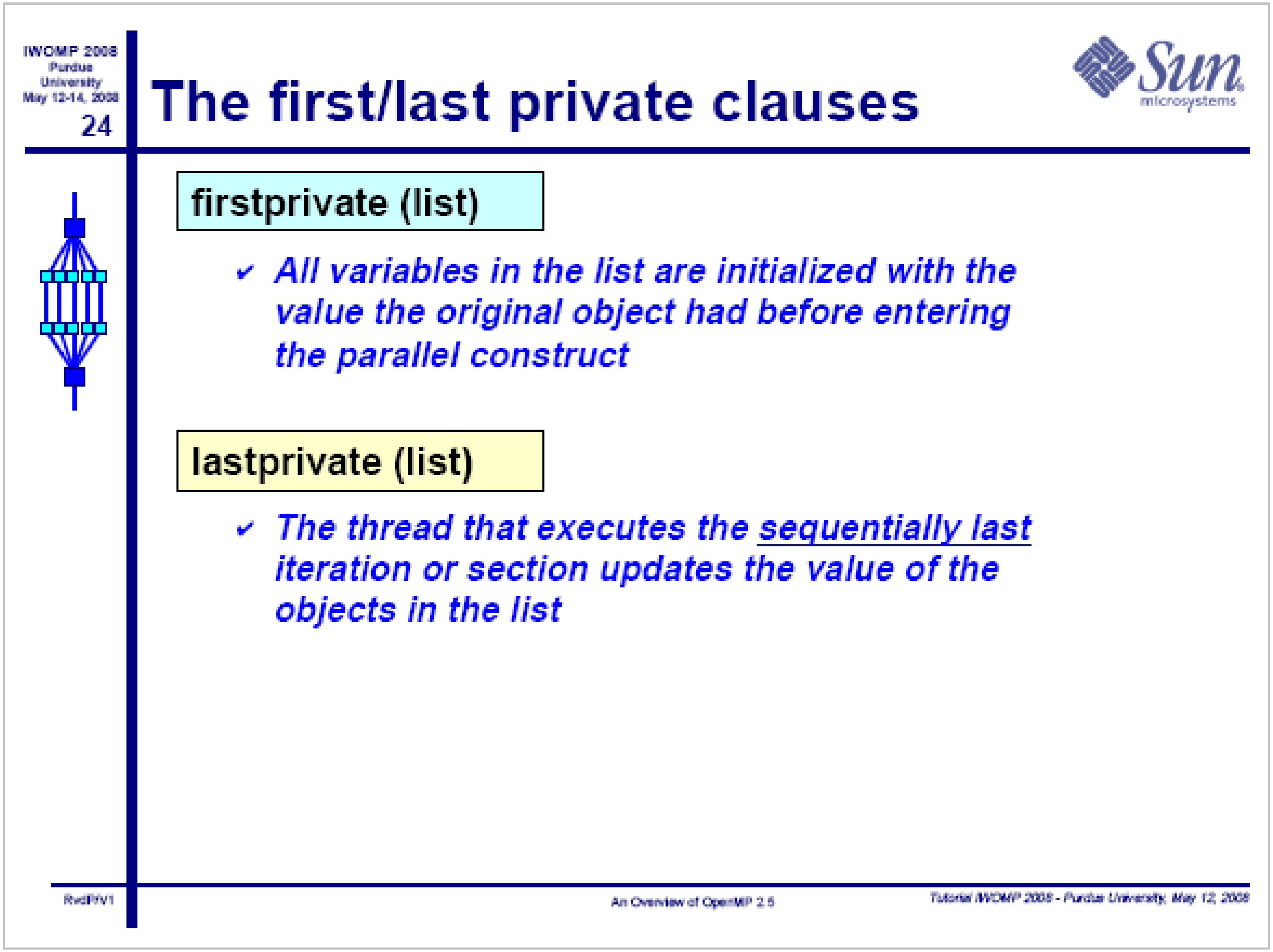
shared (list)

- ✔ *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*



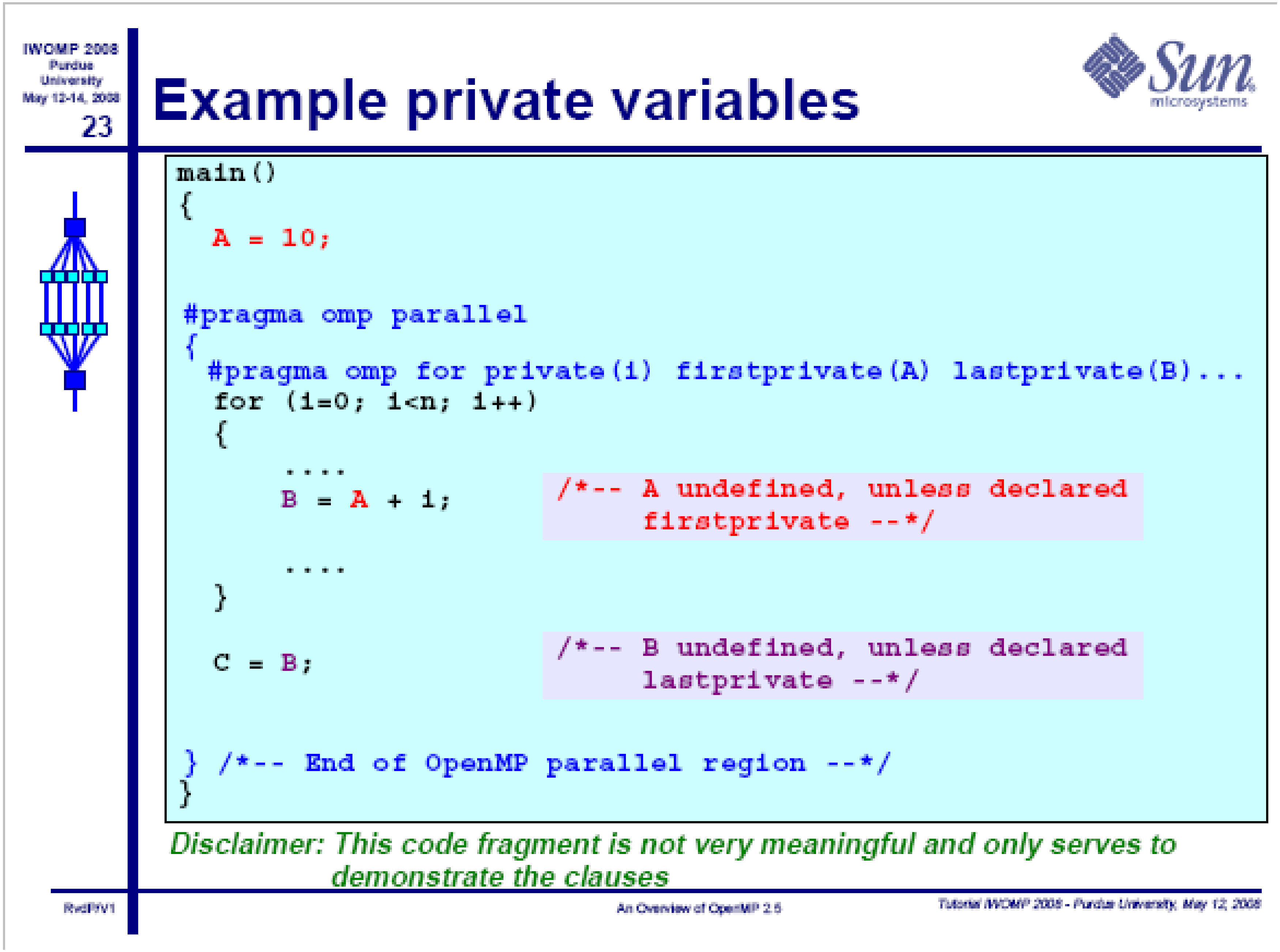
The first/last private clauses

firstprivate (list)

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

lastprivate (list)

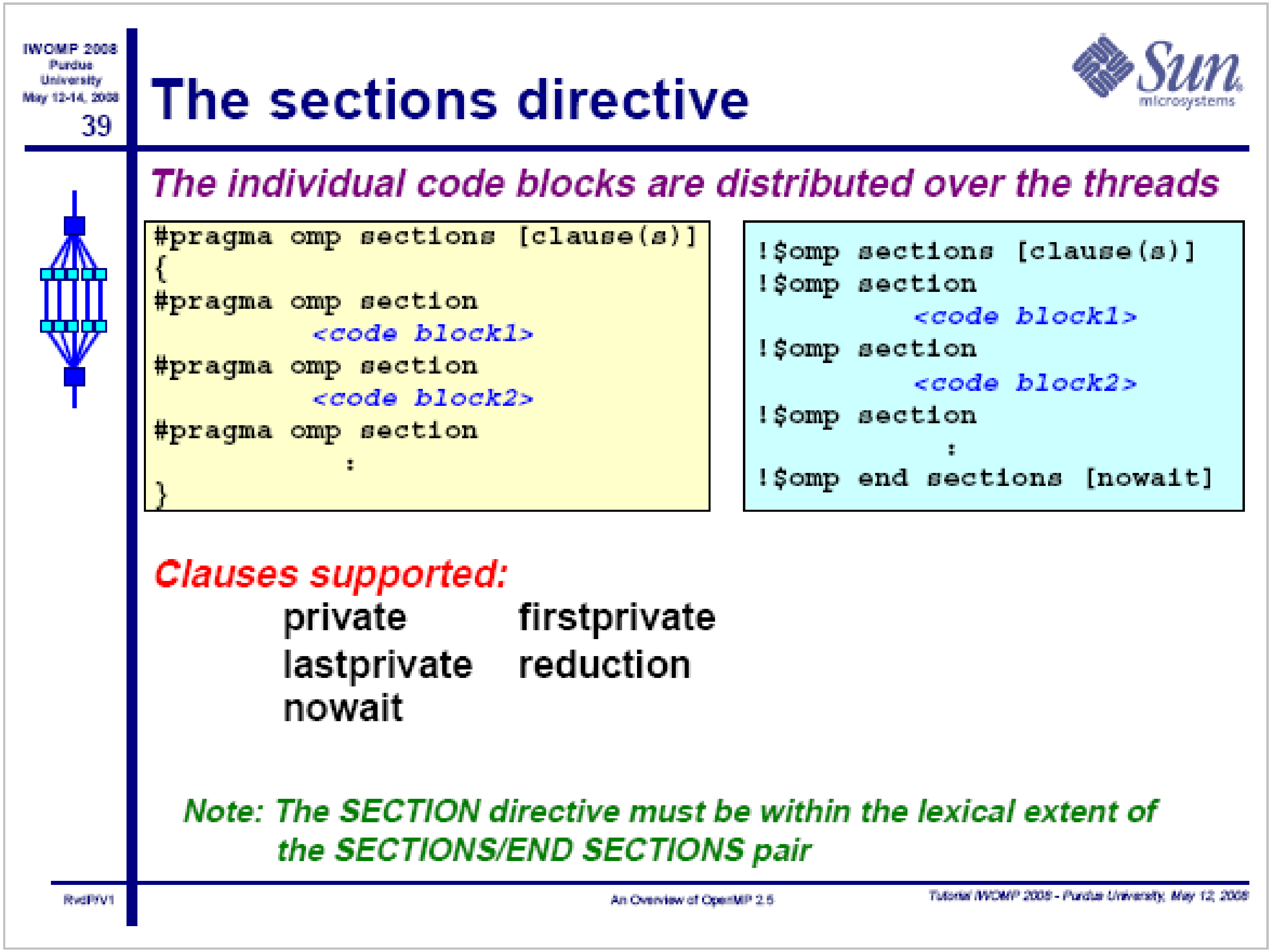
- ✔ *The thread that executes the sequentially last iteration or section updates the value of the objects in the list*



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 sections directive

The individual code blocks are distributed over the threads

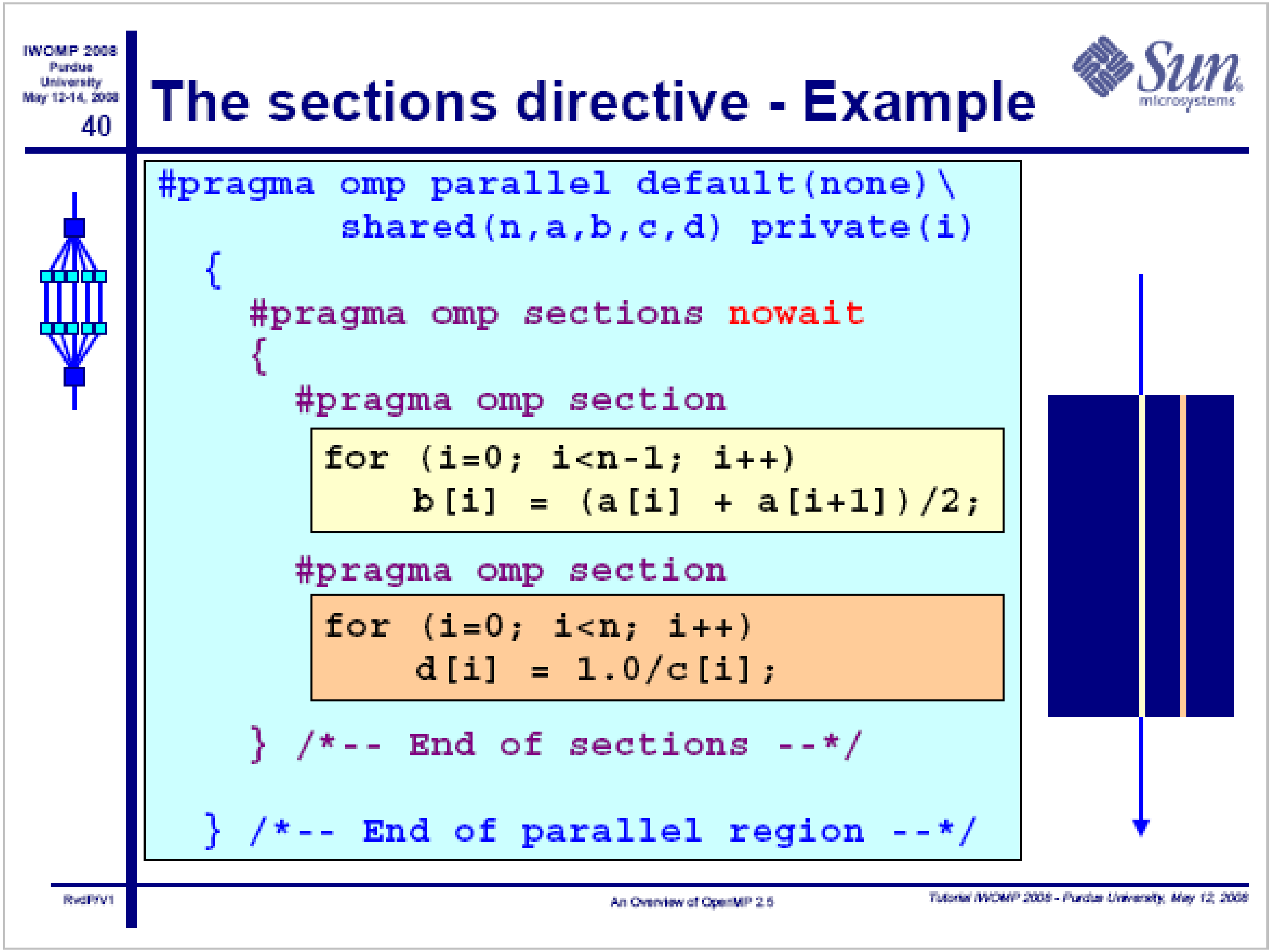
```
#pragma omp sections [clause(s)]
{
#pragma omp section
          <code block1>
#pragma omp section
          <code block2>
#pragma omp section
             :
}
```

```
!$omp sections [clause(s)]
!$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



IWOMP 2008
Purdue
University
May 12-14, 2008
40

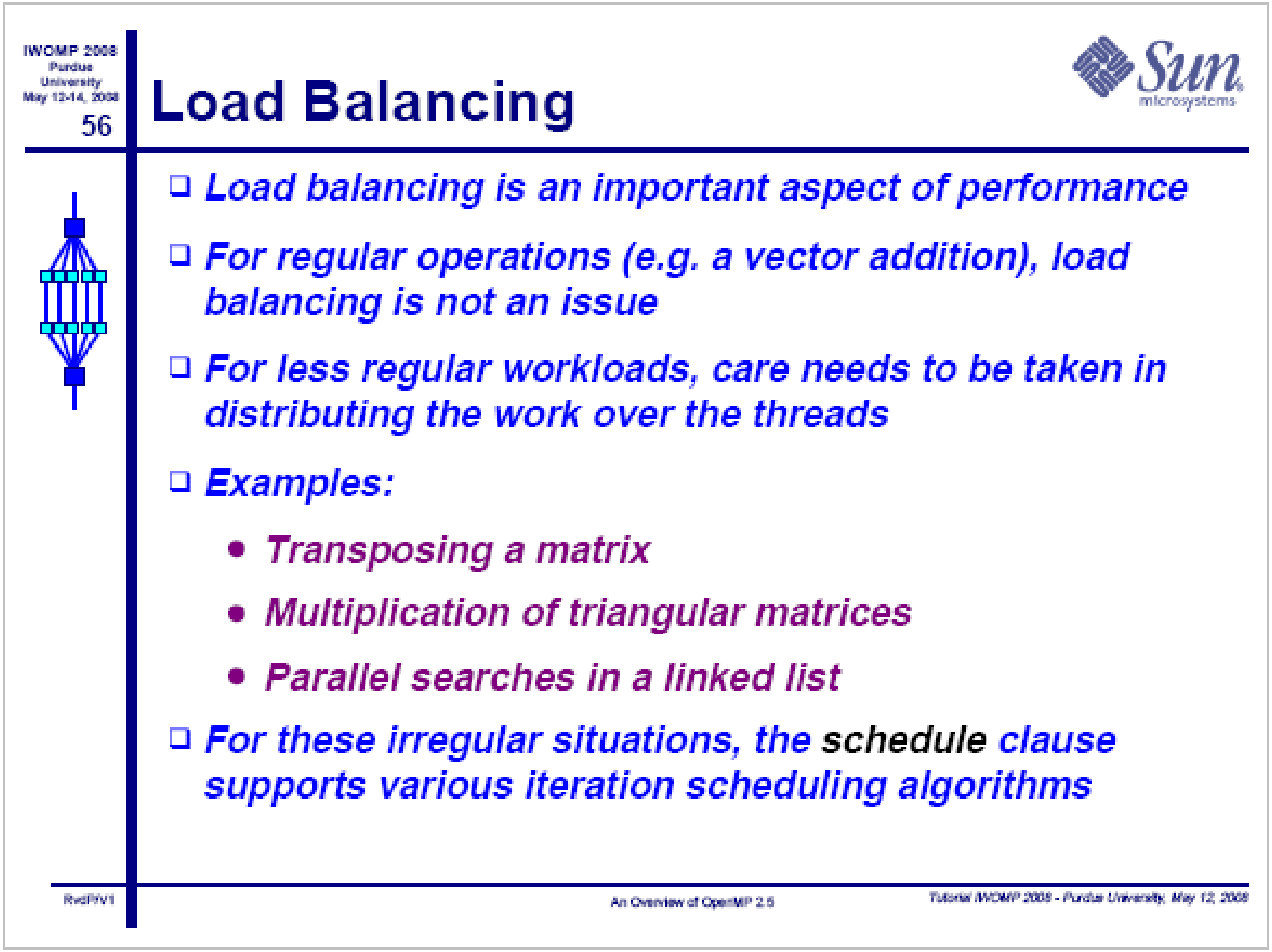
The sections directive - Example

```
#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 --*/
```

RvdP/V1

An Overview of OpenMP 2.5

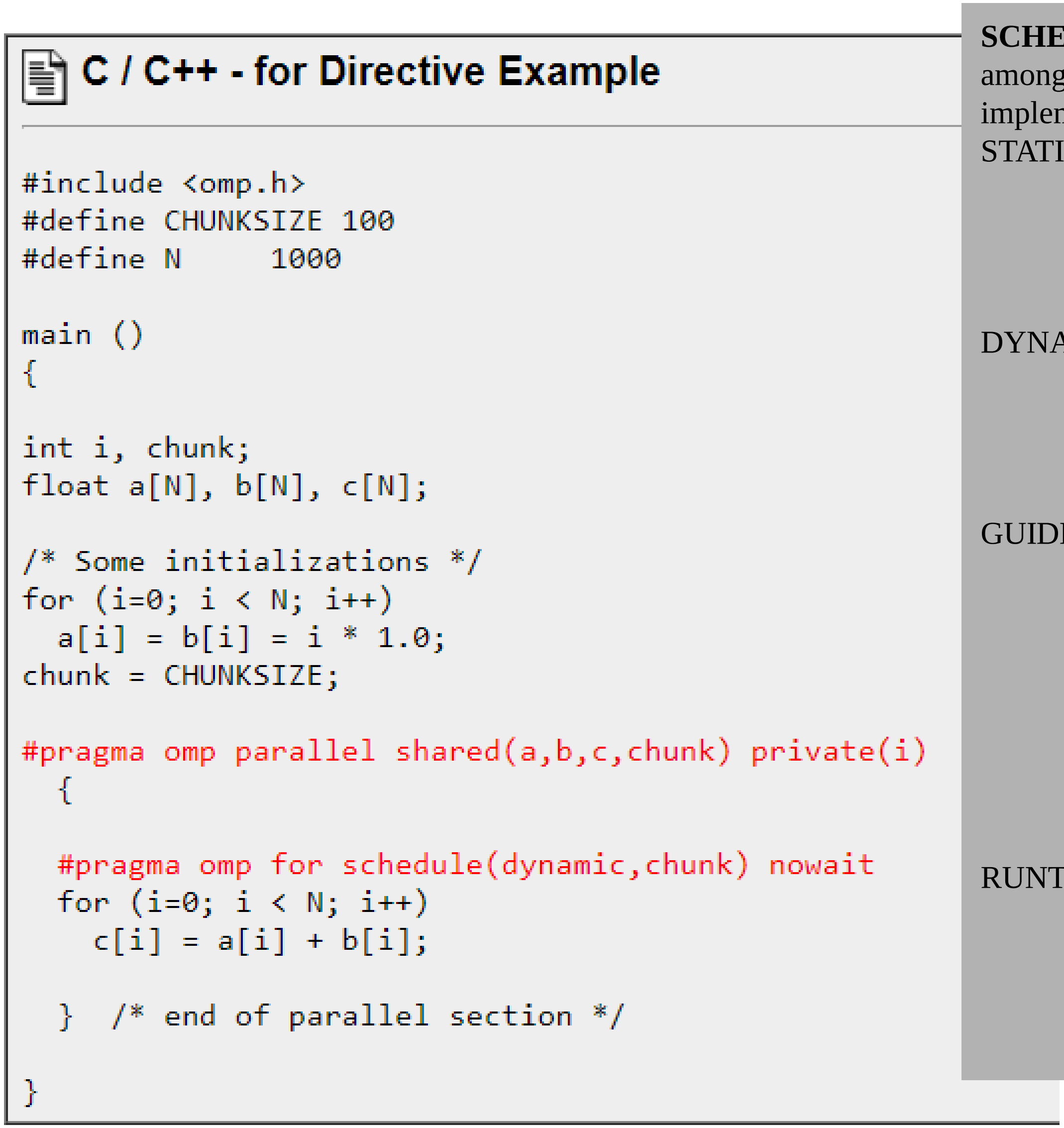
Tutorial IWOMP 2008 - Purdue University, May 12, 2008



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*

For-loop work sharing along chunk size



C / C++ - for Directive Example

```
#include <omp.h>
#define CHUNKSIZE 100
#define N     1000

main ()
{

int i, chunk;
float a[N], b[N], c[N];

/* Some initializations */
for (i=0; i < N; i++)
  a[i] = b[i] = i * 1.0;
chunk = CHUNKSIZE;

#pragma omp parallel shared(a,b,c,chunk) private(i)
  {

  #pragma omp for schedule(dynamic,chunk) nowait
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];

  }  /* end of parallel section */

}
```

SCHEDULE: Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

STATIC

Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC

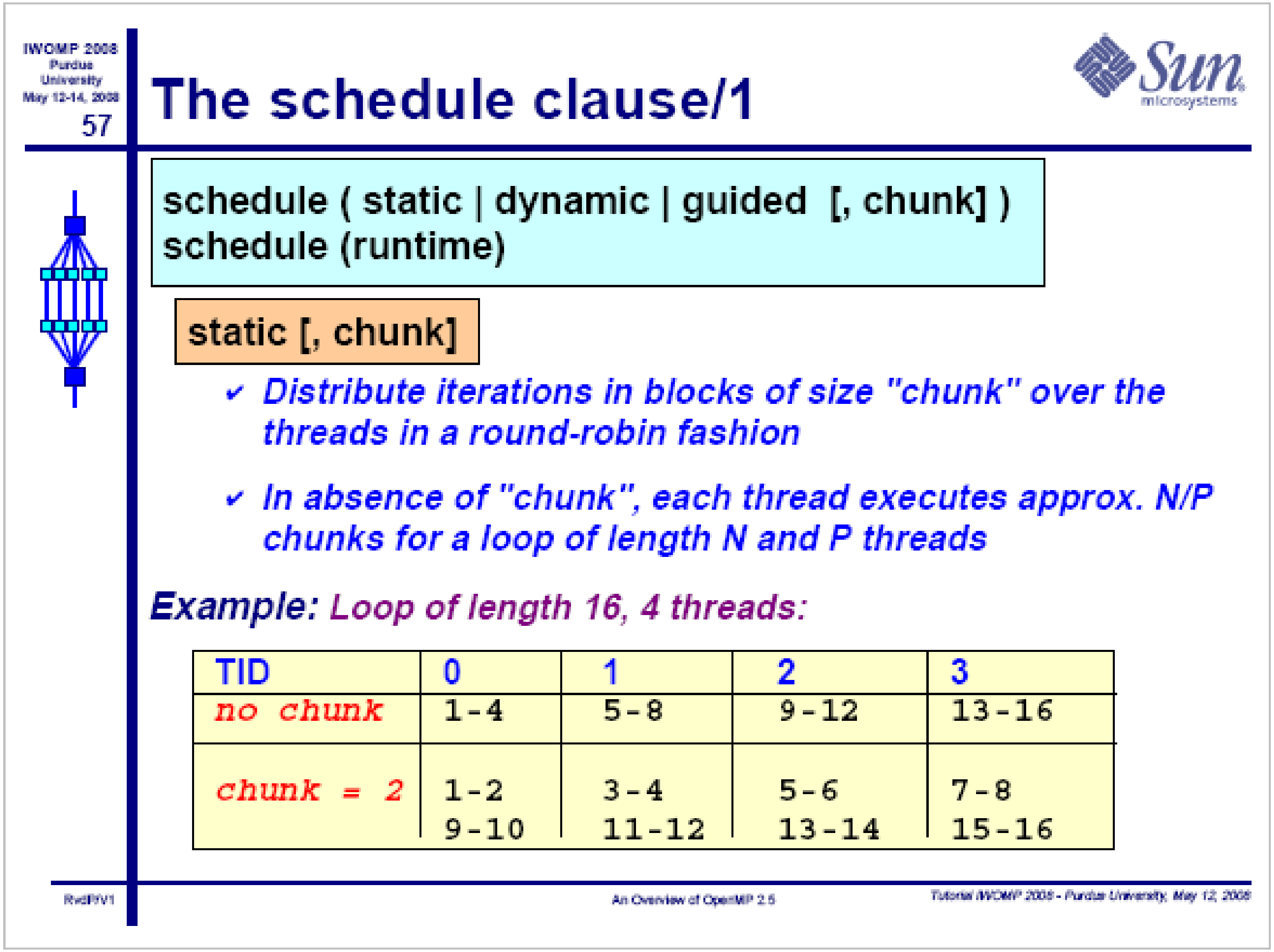
Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED

For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.

RUNTIME

The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause.



The schedule clause/1

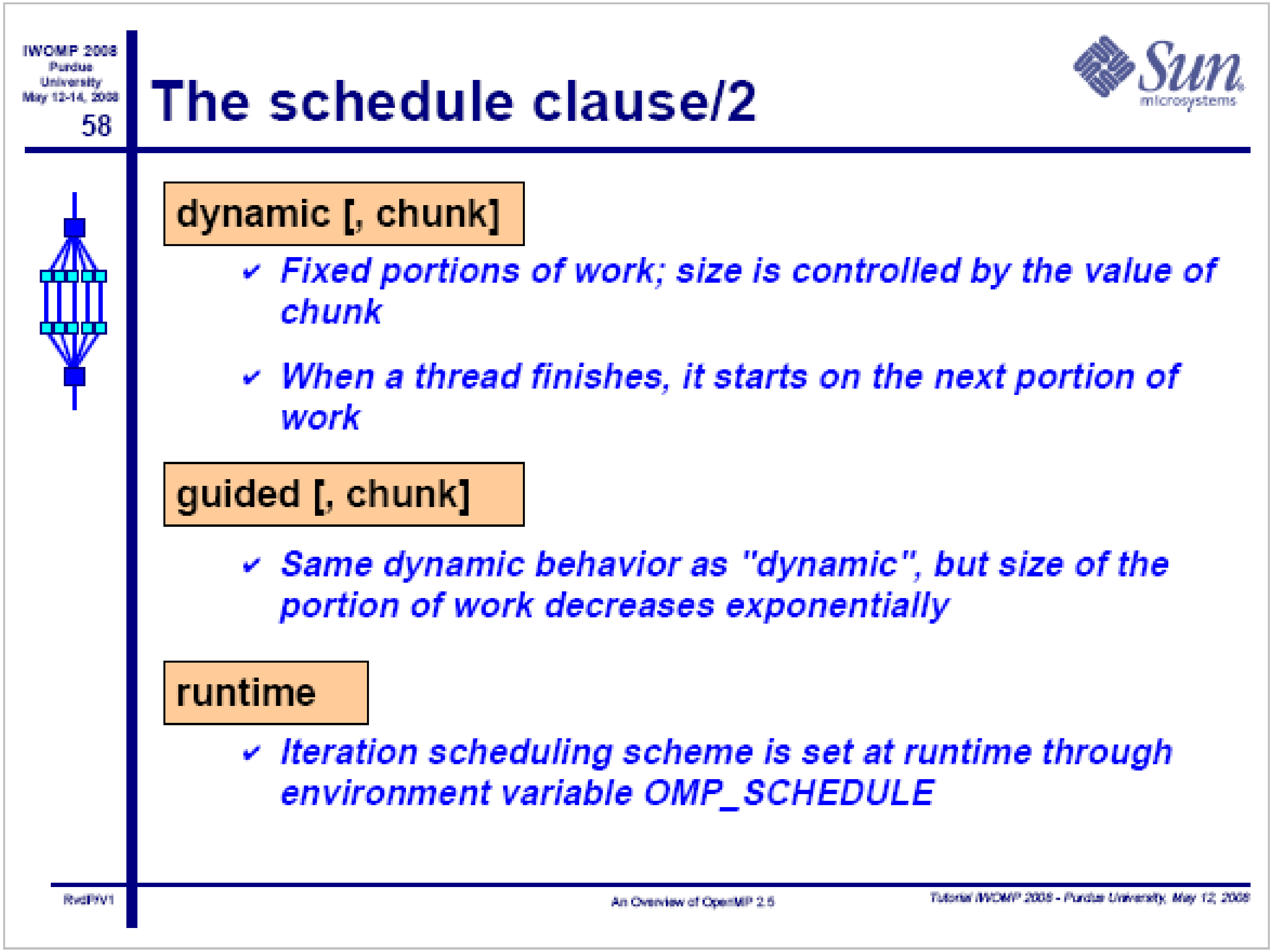
schedule (static | dynamic | guided [, chunk])
schedule (runtime)

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]

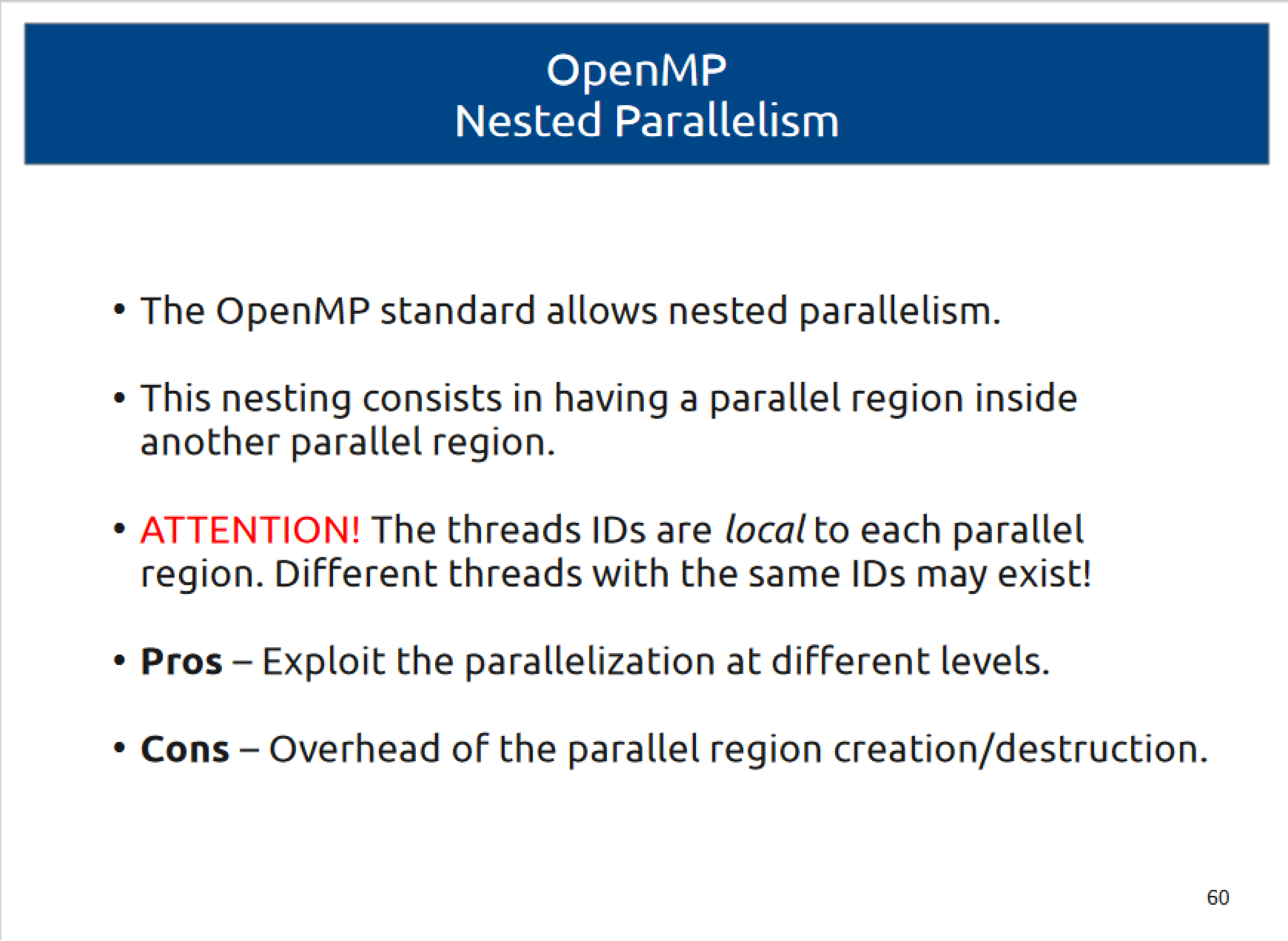
- ✔ *Fixed portions of work; size is controlled by the value of chunk*
- ✔ *When a thread finishes, it starts on the next portion of work*

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*



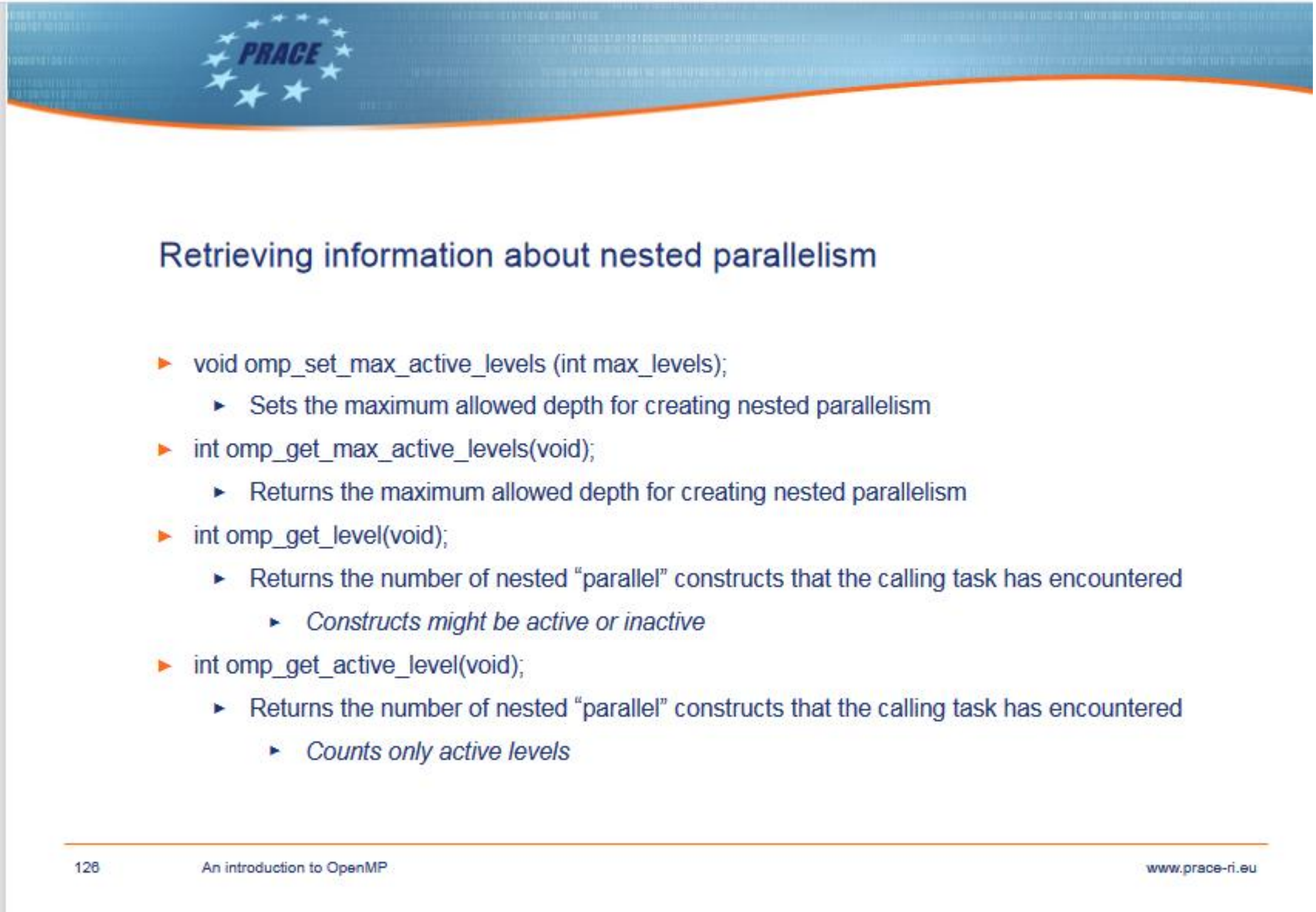
OpenMP
Nested Parallelism

- The OpenMP standard allows nested parallelism.
- This nesting consists in having a parallel region inside another parallel region.
- ATTENTION! The threads IDs are *local* to each parallel region. Different threads with the same IDs may exist!
- **Pros** – Exploit the parallelization at different levels.
- **Cons** – Overhead of the parallel region creation/destruction.

OpenMP
Nested Parallelism

```
#pragma omp parallel
{
  #pragma omp for
  for(i=0; i<n ; ++i) {
   ...
  }

  #pragma omp parallel
  {
    work(...);
  }
}
```

```
void work(...) {
  /* declarations */
  #pragma omp for
  for (j=0; j<m; ++j)
  {
   ...
  }
}
```



Retrieving information about nested parallelism

- void omp_set_max_active_levels (int max_levels);
  - Sets the maximum allowed depth for creating nested parallelism
- int omp_get_max_active_levels(void);
  - Returns the maximum allowed depth for creating nested parallelism
- int omp_get_level(void);
  - Returns the number of nested “parallel” constructs that the calling task has encountered
    - *Constructs might be active or inactive*
- int omp_get_active_level(void);
  - Returns the number of nested “parallel” constructs that the calling task has encountered
    - *Counts only active levels*