Programming with OpenMP*

Agenda

- Getting Started with OpenMP
  - What Is OpenMP?
  - Programming Model
  - Memory Model
- Using OpenMP
- What's New in OpenMP

What Is OpenMP?

OpenMP

- Standard Application Programming Interface (API) to write ***shared memory parallel applications*** in C, C++, and Fortran
  - *Compiler Directives*
  - *Run time routines*
  - *Environment variables*
- Specification maintained by the OpenMP Architecture Review Board (http://www.openmp.org)
- Version 4.0 released summer 2013
  - In this presentation we will assume conformance to standard version 3.1
  - See separate presentation for OpenMP 4.0 update

http://www.openmp.org – standard available for all downloads

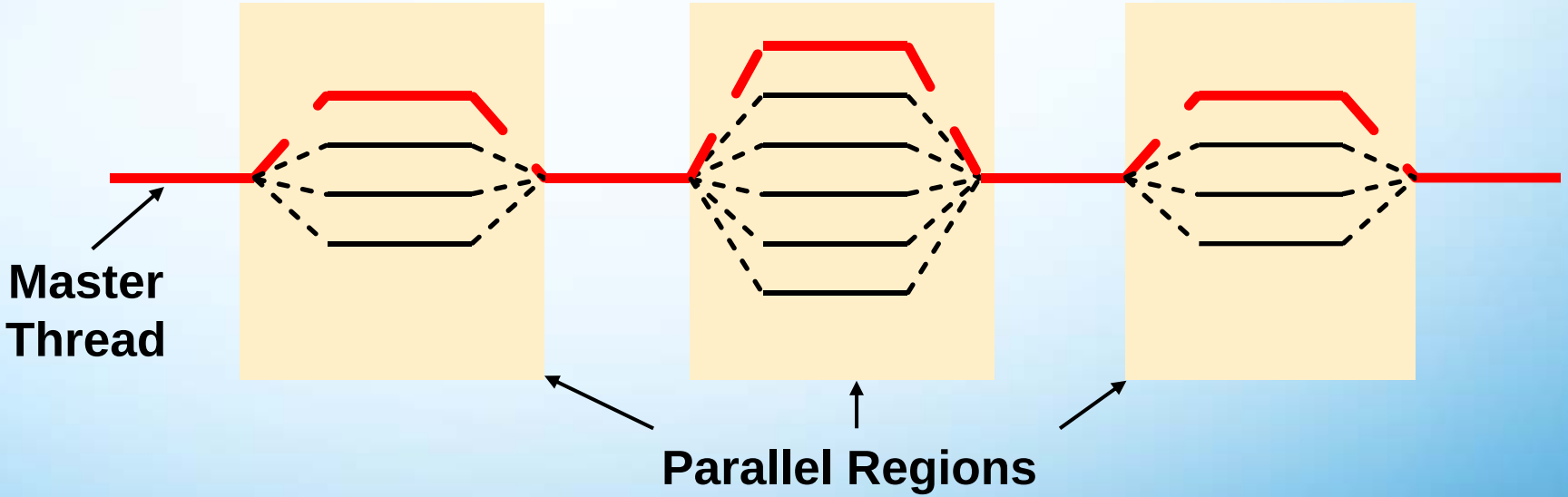
Advantages of OpenMP

- Good performance and scalability
- Available for C/C++ and Fortran
- Portability
  - Supported by a large number of compilers
- Requires little programming effort
- Allows the program to be parallelized incrementally
- Accepted programming model in particular at HPC domain
- Tools support – both for performance and correctness

Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed
- OpenMP Team := Master + Workers
- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program



A Few Details to Get Started

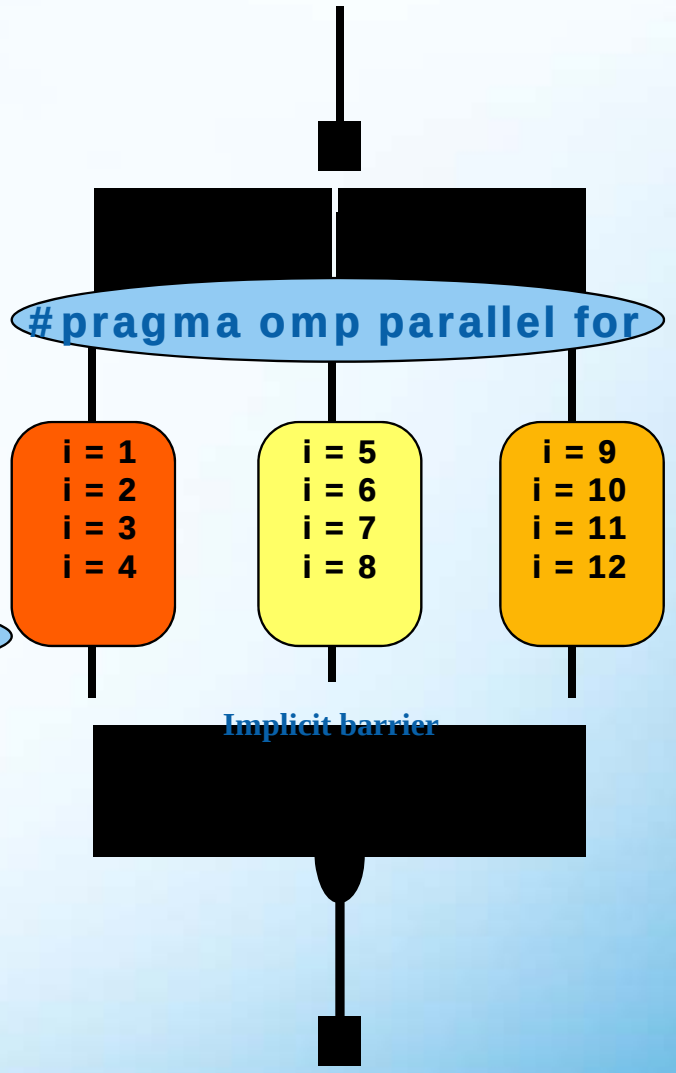
- Compiler option /Qopenmp in Windows or –openmp in Linux
- Most of the constructs in OpenMP are compiler directives or pragmas
- Header file or Fortran module

C/C+: `#include “omp.h”`

Fortran: `use omp_lib`

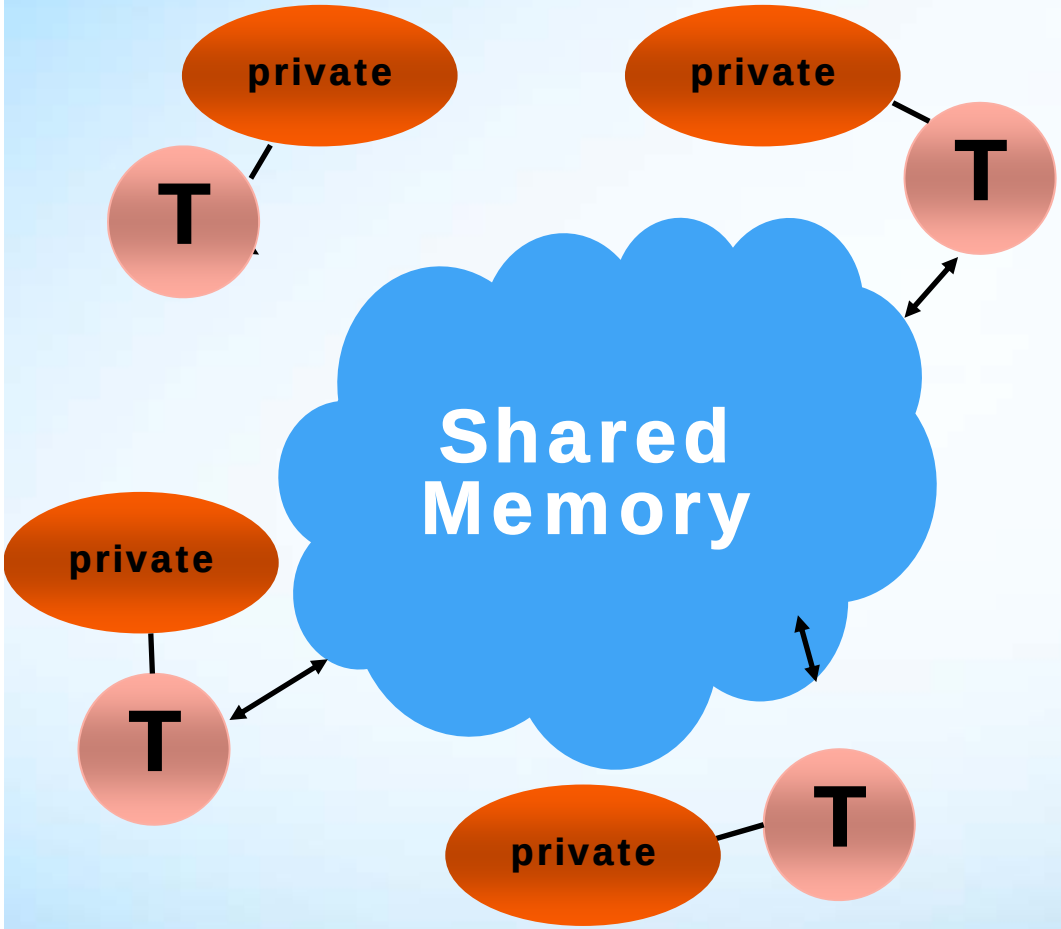
First omp Example

```
// assume N=12
#pragma omp parallel for
for(i = 1, i < N+1, i++)
      c[i] = a[i] + b[i];
```



- Threads are assigned an independent set of iterations
- Threads must wait at the end of "work-sharing" construct

The OpenMP Memory Model



- All threads have access to the same, *globally shared*, memory
- Data can be
  - shared
    - accessible by all threads
  - private
    - 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

Agenda

- Getting Started with OpenMP
- Using OpenMP
  - Directive format
  - Data-sharing Attributes
  - Parallel Region
  - Worksharing
  - Global Data and Synchronization
  - Runtime Routines
  - Environment Variables
  - Tasking
- What's New in OpenMP 3.1

Components of OpenMP

Directives	Runtime environment	Environment variables
•Parallel region •Worksharing constructs •Tasking •Synchronization •Data-sharing attributes	•Number of threads •Thread ID •Dynamic thread adjustment •Nested parallelism •Schedule •Active levels •Thread limit •Nesting level •Ancestor thread •Team size •Wallclock timer •Locking	•Number of threads •Scheduling type •Dynamic thread adjustment •Nested parallelism •Stacksize •Idle threads •Active levels •Thread limit

Directive format

C/C++

- C directives are case sensitive
  - Syntax: #pragma omp directive [clause [clause] …]
- Continuation: use \ in pragma
- Conditional compilation: **_OPENMP** macro is set

Fortran

- 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

Data-sharing Attributes

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

Private and shared clauses

private (list)

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

shared (list)

- all threads in the team
- All threads access the same address space

About storage association

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

firstprivate and lastprivate clauses

firstprivate (list)

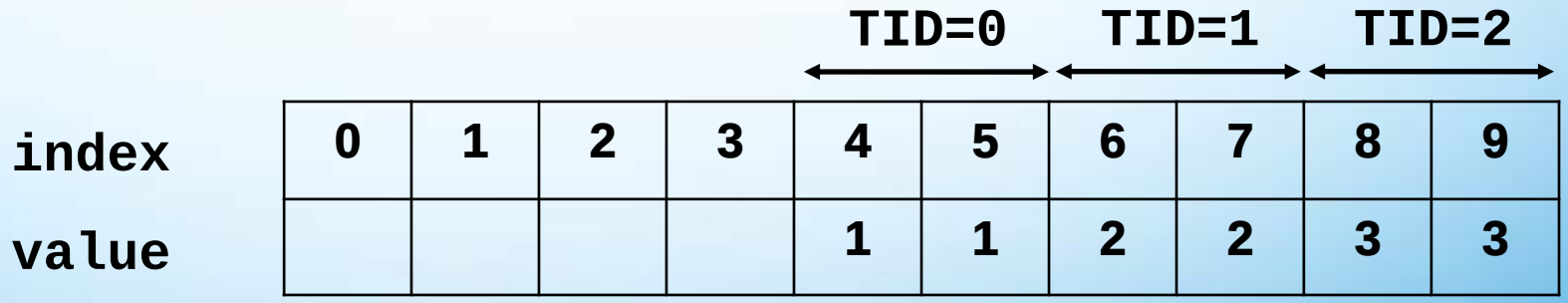
- All variables in the list are initialized with the value the original object had before entering the parallel construct
- C++ objects are copy-constructed

lastprivate (list)

- utes the sequentially last iteration or section updates the value of the objects in the list
- C++ objects are updated as if by assignment

Example firstprivate

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



Example lastprivate

```
#pragma omp parallel for default(none) lastprivate(b)
for (int i=0; i<n; i++)
{ .......
  b = i + 1;
.......
} // End of parallel region
c = 2 * b; // value of c is 2*n
```

default clause

C/C++	default (none | shared)
Fortran	default (none | shared | private | threadprivate)

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

firstprivate

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

Defining Parallelism in OpenMP

- OpenMP Team := Master + Workers
- A Parallel Region is a block of code executed by all threads simultaneously
  - The master thread always has thread ID 0
  - 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

Parallel Region

C/C++

```
#pragma omp parallel [clause[[,] clause] ...]
{
   "this code is executed in parallel"
} // End of parallel section (note: implied barrier)
```

Fortran

```
!$omp parallel [clause[[,] clause] ...]
   "this code is executed in parallel"
!$omp end parallel
!note: implied barrier
```

Parallel Region & Structured Blocks

- Most OpenMP constructs apply to structured blocks
  - Structured block: a block with one point of entry at the top and one point of exit at the bottom
  - The only "branches" allowed are STOP statements in Fortran and exit() in C/C++

```
#pragma omp parallel
{
 int id = omp_get_thread_num();

more: res[id] = do_big_job (id);

 if (conv (res[id])) goto more;
}
printf ("All done\n");
```

A structured block

```
if (go_now()) goto more;
#pragma omp parallel
{
 int id = omp_get_thread_num();
more:  res[id] = do_big_job(id);
 if (conv (res[id])) goto done;
 goto more;
}
done: if (!really_done()) goto more;
```

Not a structured block

Parallel Region - Example

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
      printf("Hello World\n");
      return(0);
}
```

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
      #pragma omp parallel
      {
            printf("Hello World\n");
      } // End of parallel region
      return(0);
}
```

Parallel Region - Example

```
$ icc -openmp test.c
$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

if clause

if (scalar expression)

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

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

Fortran - Allocatable Arrays

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

```
integer, allocatable,dimension (:) :: A
integer i
allocate (A(n))
!$omp parallel private (A)
      do i = 1, n
         A(i) = i
      end do
...
!$omp end parallel
```

Worksharing Constructs

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

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

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

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

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

```
!$OMP SINGLE
....
!$OMP END SINGLE
```

- The work is distributed over the threads
- Must be enclosed in a parallel region
- Must be encountered by all threads in the team, or none at all
- No implied barrier on entry; implied barrier on exit (unless *nowait* is specified)

Fortan Workshare construct

- Fortran has a fourth worksharing construct
- Example

```
!$OMP WORKSHARE
    <array syntax>
!$OMP END WORKSHARE [NOWAIT]
```

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

omp for/do directive

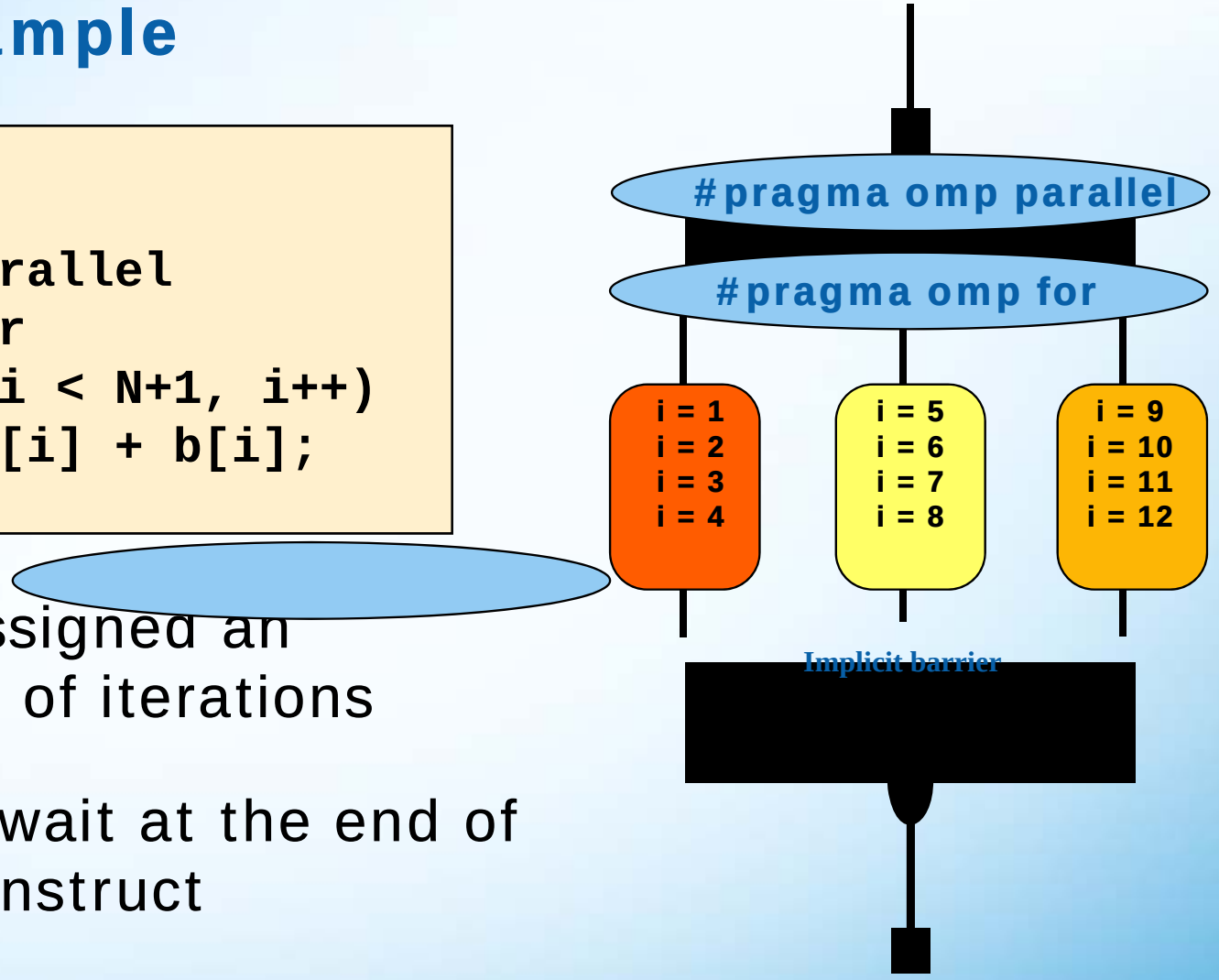
- The iterations of the loop are distributed over the threads

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

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

omp for example

```
// assume N=12
#pragma omp parallel
#pragma omp for
   for(i = 1, i < N+1, i++)
      c[i] = a[i] + b[i];
```



- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

Combining constructs

```
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
      res[i] = huge();
    }
}
```

```
#pragma omp parallel for
    for (i=0;i< MAX; i++) {
         res[i] = huge();
    }
```

These two code segments are equivalent

Loop Collapse

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

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

schedule clause

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

static [, chunk]

- Distribute iterations in blocks of size "chunk" over the threads in a round-robin fashion
- In absence of "chunk", each thread executes approx. N/P chunks for a loop of length N and P threads
- Under certain conditions, the assignment of iterations to threads is the same across multiple loops in the same parallel region

schedule clause example

Static schedule, loop of length 16, 4 threads:

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

schedule clause

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]

avior as "dynamic", but size of the portion of work decreases exponentially

- The compiler (or runtime system) decides what is best to use;

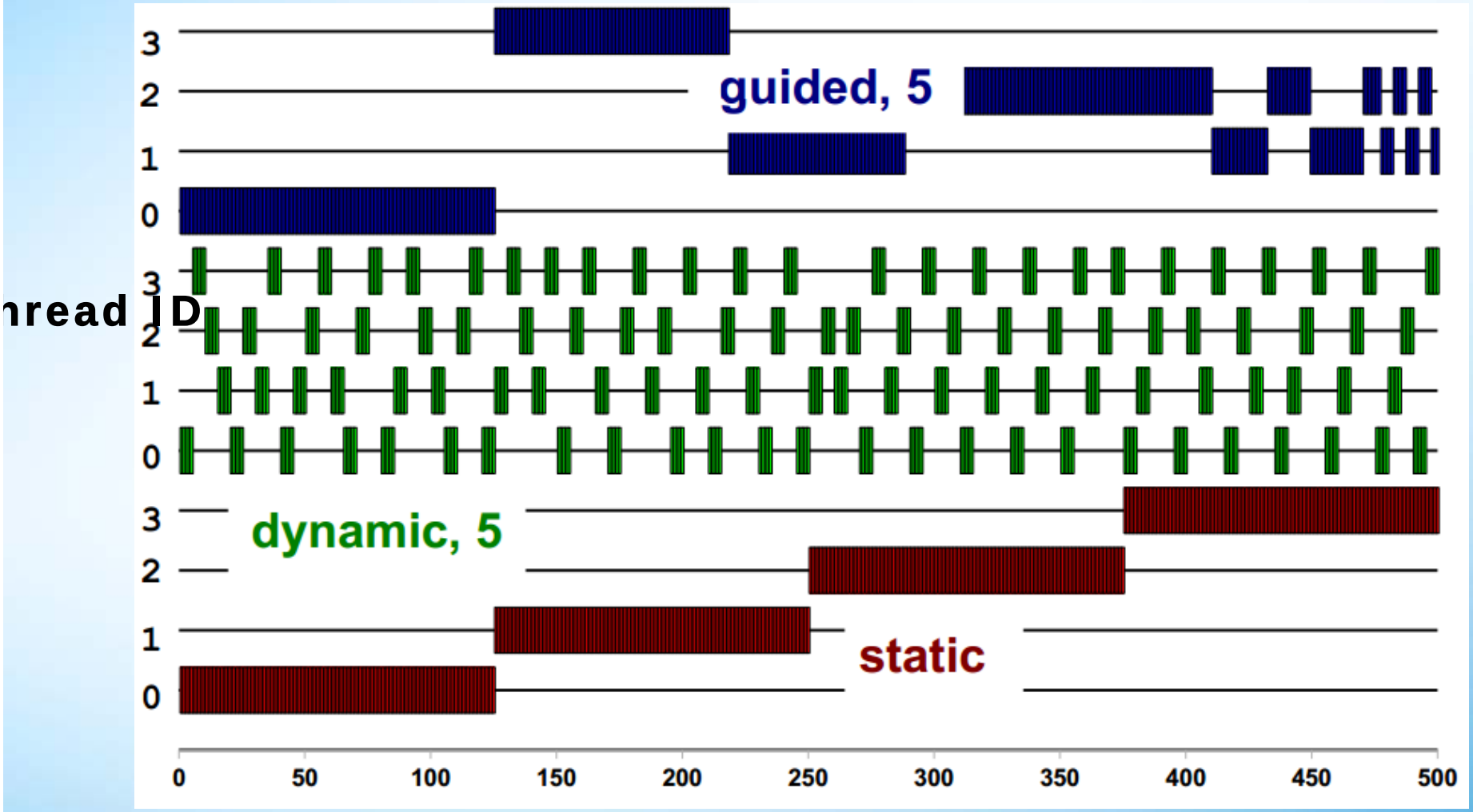
auto

ementation dependent

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

runtime

Experiment - 500 iterations, 4 threads



Parallel sections

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

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

Individual section blocks are executed in parallel

Parallel sections 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 --*/
```

Single Directive

Only one thread in the team executes the code enclosed

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

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

Single processor region

Original Code

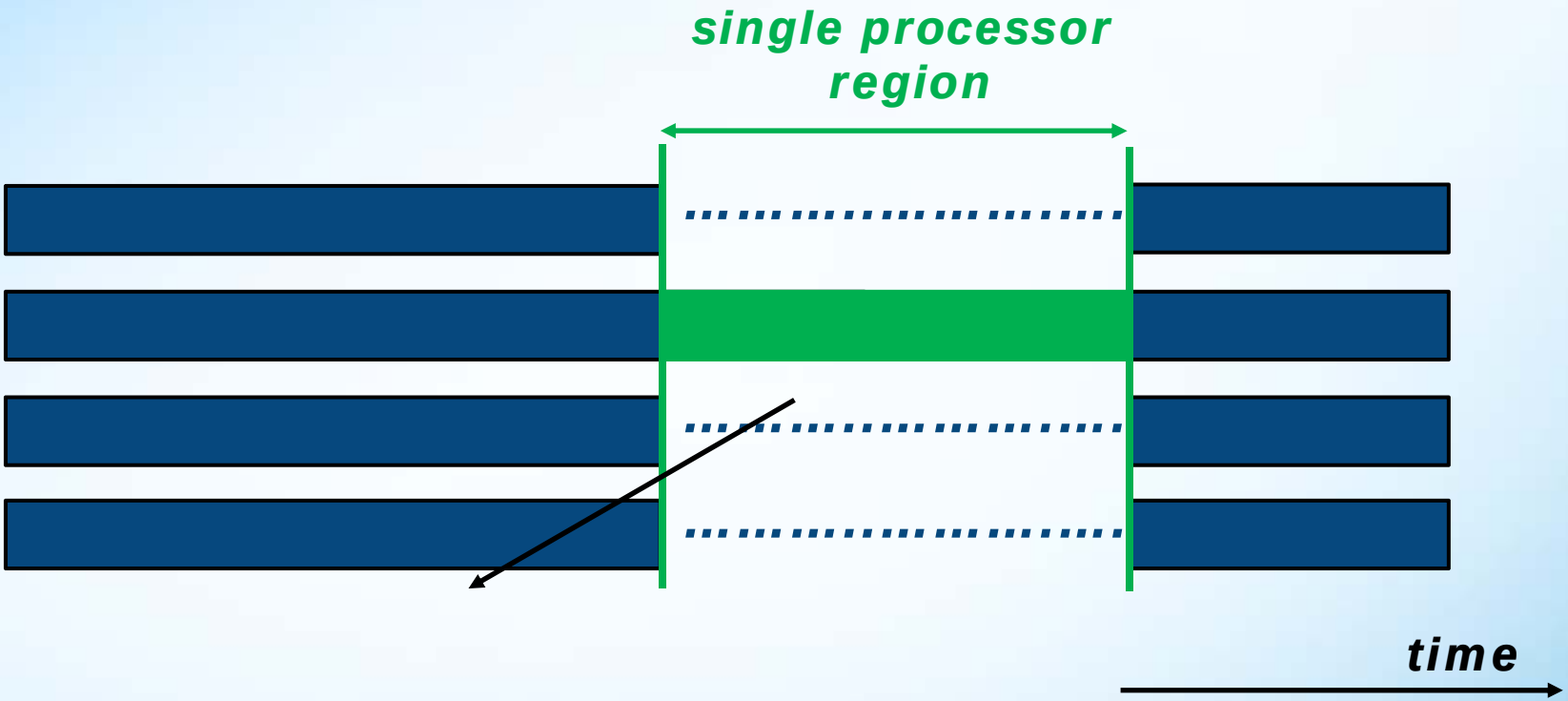
```
.....
"read A[0..N-1]";
.....
```

- Only one thread executes the single region
- This construct is ideally suited for I/O or initializations

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

Parallel Version

Single processor region



Other threads wait if there is a barrier here

Orphaning

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

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

orphaned work-sharing directive

- The OpenMP specification does not restrict worksharing and synchronization directives (omp for, omp single, critical, barrier, etc.) to be within the lexical extent of a parallel region. These directives can be orphaned
- That is, they can appear outside the lexical extent of a parallel region

More on orphaning

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

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

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

Master Directive

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!

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
          ......
```

file global.h

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

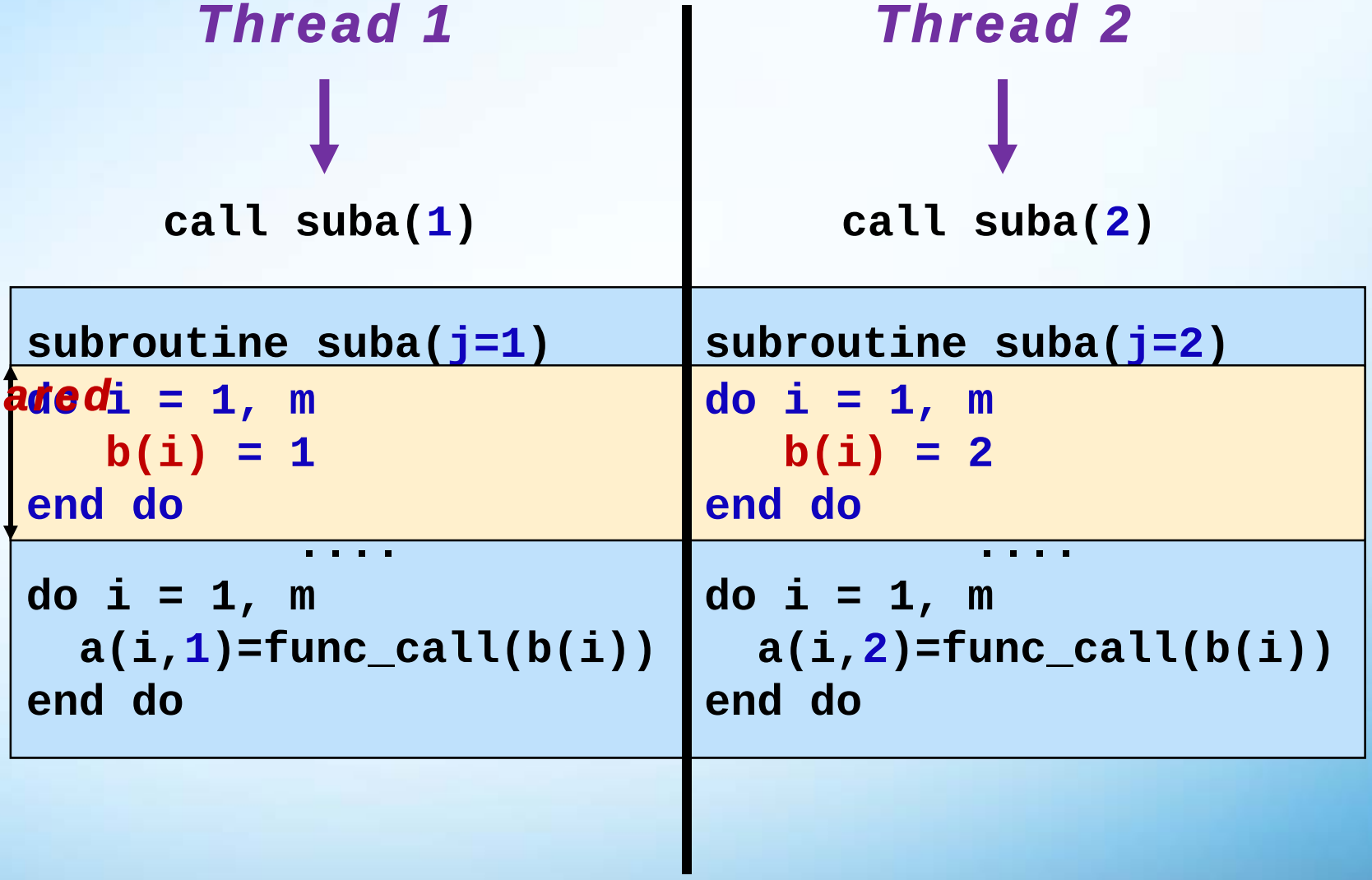
```
subroutine suba(j)
   .....
include "global.h"
   .....

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

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

Data Race

Race Condition

- A *race condition* is nondeterministic behavior caused by the times at which two or more threads access a shared variable and at least one of them modifies the variable
- Difficult to reproduce and debug, since the end result is nondeterministic
  - highly dependent on the relative timing between interfering threads

Global data - A Data Race



Global data – Solution

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

file global_ok.h

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

```
subroutine suba(j)
  .....
include "global_ok.h"
  .....
TID = omp_get_thread_num()+1
do i = 1, m
   b(i, TID) = j
end do

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

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

About global data

- Global data is shared and requires special care
- A problem may arise in case multiple threads access the same memory section simultaneously:
  - Read-only data is no problem
  - Updates have to be checked for race conditions
- It is your responsibility to deal with this situation
- In general one can do the following:
  - Split the global data into a part that is accessed in serial parts only and a part that is accessed in parallel
  - Manually create thread private copies of the latter
  - Use the thread ID to access these private copies
- Alternative: Use OpenMP's ***threadprivate*** directive

threadprivate directive

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

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

- Thread private copies of the designated global variables and common blocks are created
- Several restrictions and rules apply when doing this:
  - The number of threads has to remain the same for all the parallel regions (i.e. no dynamic threads)
  - Initial data is undefined, unless copyin is used

Global data – 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
        ......
```

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

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

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

! create thread private common blocks

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

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

C++ and threadprivate

copyin (list)

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

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

Reduction Clause

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

- Reduction variable(s) must be shared variables
- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the operator
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through operator into a single value and combined with the value in the original SHARED variable
- The reduction can be hidden in a function call

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

C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

Operand	Initial Value
+	0
*	1
-	0
^	0

Operand	Initial Value
&	~0
|	0
&&	1
||	0

Barrier

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)

Barrier

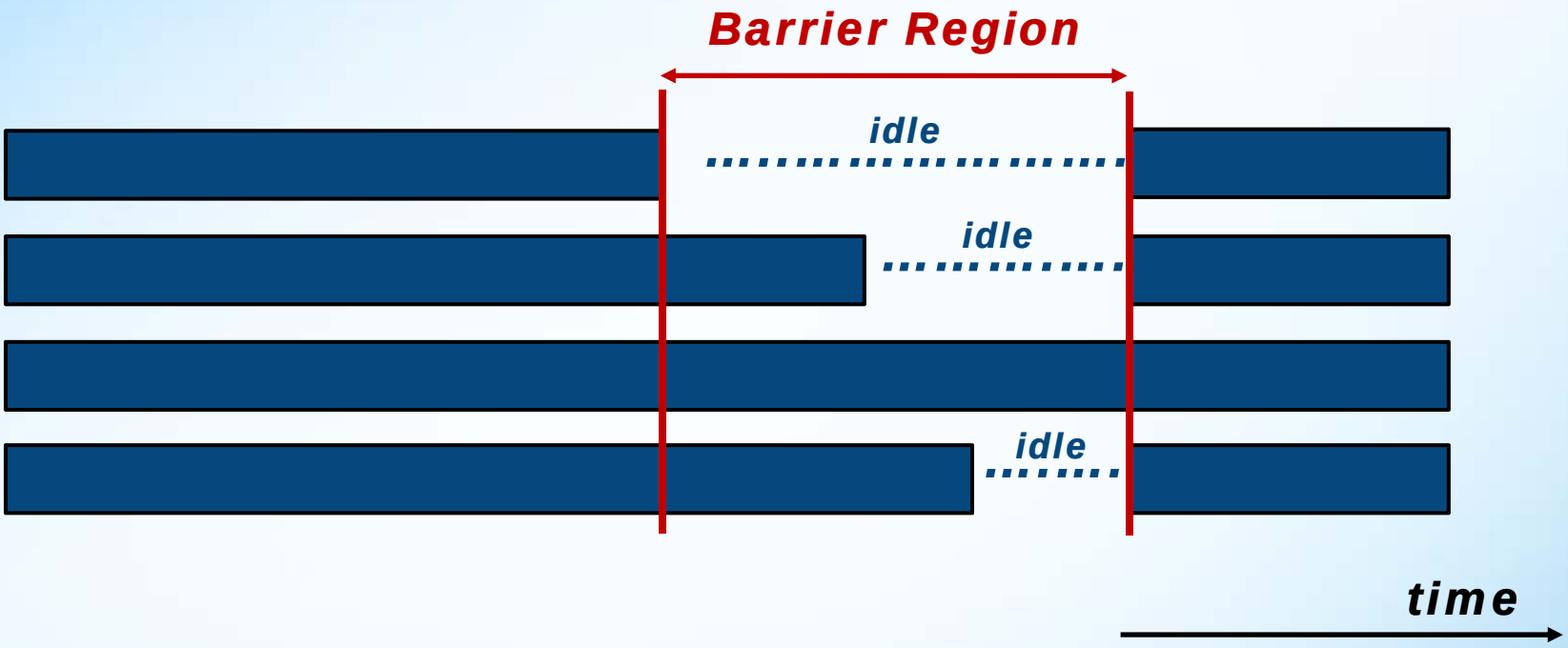
We need to have updated all of a[ ] first, before using a[ ]

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

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

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

Barrier



Barrier syntax in OpenMP:

C/C++	**#pragma omp barrier**
Fortran	**!$omp barrier**

When to use barriers?

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

Implicit Barriers

- Several OpenMP* constructs have implicit barriers
  - Parallel – necessary barrier – cannot be removed
  - for
  - single
- Unnecessary barriers hurt performance and can be removed with the nowait clause
  - The nowait clause is applicable to:
    - For clause
    - Single clause

nowait clause

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

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

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

Critical Region

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

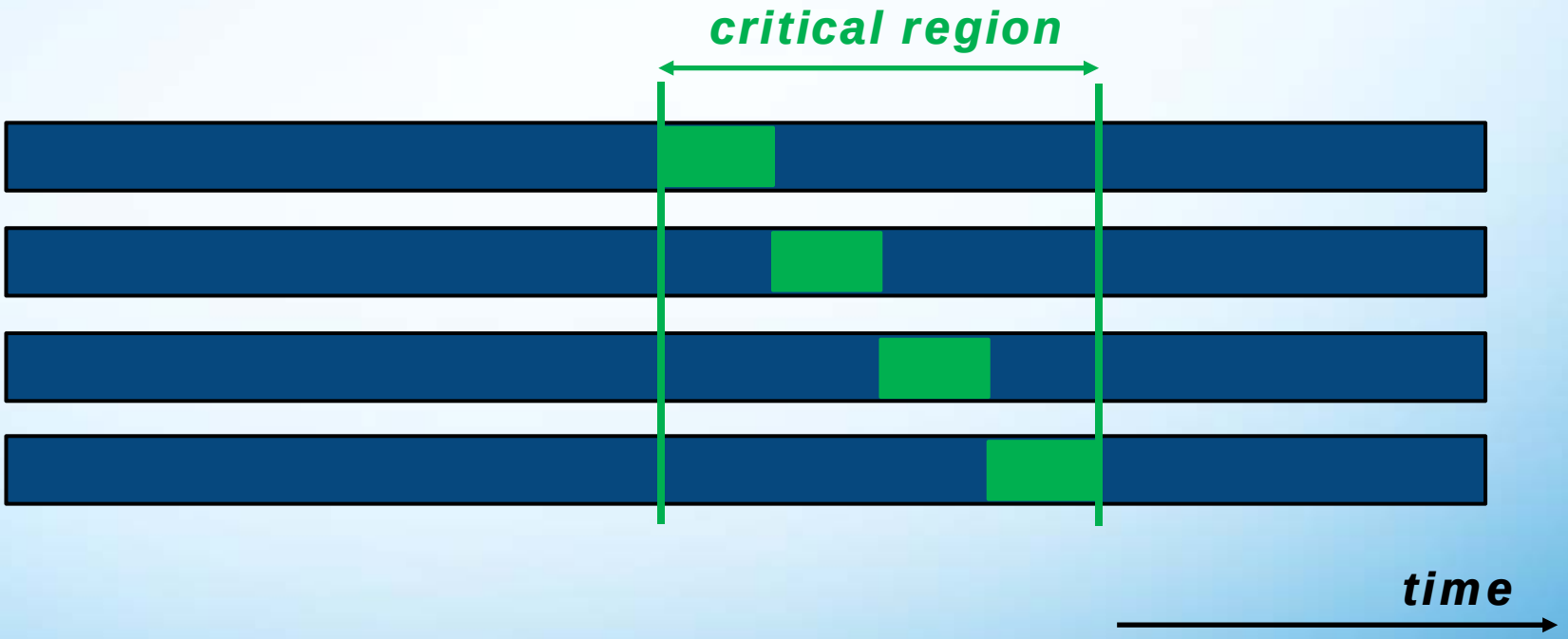
```
for (i=0; i < n; i++){
      .....
   sum += a[i];
      .....
}
```

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

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

Critical Region

- 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>}
```

There is no implied barrier on entry or exit!

- Atomic:

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

mic ....

```
#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];
```

OpenMP Runtime Functions

- Usually not needed for OpenMP codes
  - Can lead to code not being serially consistent
  - Does have specific uses (debugging)
  - Must include a header file

```
#include <omp.h>
```

Name	Functionality
omp_set_num_threads	Set number of threads
omp_get_thread_num	Get thread ID
omp_get_num_threads	Number of threads in team
omp_get_max_threads	Max num of threads for parallel region
omp_set_schedule	Set schedule (if “runtime” is used)
omp_get_schedule	Returns the schedule in use

and many other functions

Library Routines

- To fix the number of threads used in a program
  - Set the number of threads
  - Then save the number returned

```
#include <omp.h>

void main ()
{
  int num_threads;
  omp_set_num_threads (omp_num_procs ());

  #pragma omp parallel
  {
    int id = omp_get_thread_num ();

    #pragma omp single
    num_threads = omp_get_num_threads ();

    do_lots_of_stuff (id);
  }
}
```

Request as many threads as you have processors.

Protect this operation because memory stores are not atomic

OpenMP locking routines

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

Locking Example

```
      program Locks
            ....
      call omp_init_lock (LCK)
!$omp parallel shared(LCK)
       Do While ( omp_test_lock (LCK) .EQV. .FALSE. )
          Call Do_Something_Else()
       End Do
      call Do_Work()

      call omp_unset_lock (LCK)

!$omp end parallel
      Call omp_destroy_lock (LCK)

      Stop
      End
```

Initialize lock variable

Check availability of lock
(also sets the lock)

Release lock again

Remove lock association

OpenMP Environment Variables

OpenMP environment variable	Default
OMP_NUM_THREADS	Number of processors visible to the OS
OMP_SCHEDULE “schedule,[chunk]”	STATIC, no chunk size specified
OMP_DYNAMIC { TRUE | FALSE }	FALSE
OMP_NESTED { TRUE | FALSE }	FALSE
OMP_STACKSIZE size [B|K|M|G]	IA-32: 2M Intel® 64: 4M
OMP_WAIT_POLICY [ACTIVE | PASSIVE]	PASSIVE
OMP_MAX_ACTIVE_LEVELS	1
OMP_THREAD_LIMIT	No enforced limit
OMP_PROC_BIND { TRUE | FALSE }	FALSE

Tasking

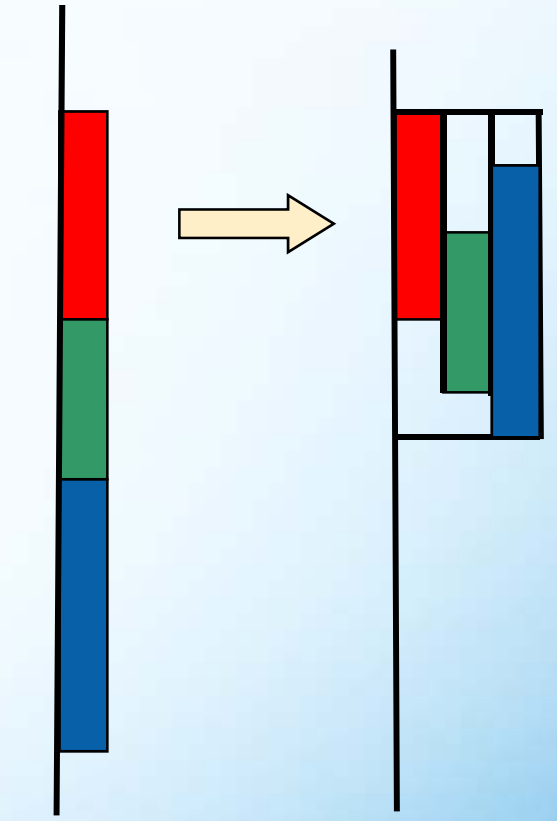
- Introduced in OpenMP 3.0
- When any thread encounters a task construct, a new explicit task is generated
  - Tasks can be nested
- Execution of explicitly generated tasks is assigned to one of the threads in the current team
- Completion of the task can be guaranteed using a task synchronization construct
- Allows parallelization of irregular problems
  - unbounded loops
  - recursive algorithms
  - producer/consumer

What are tasks?

```
#pragma omp task
```

```
!$omp task
```

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
  - Tasks may be deferred
- Tasks may be executed immediately
- The runtime system decides which of the above
  - Tasks are composed of:
    - code to execute
    - data environment
    - internal control variables (ICV)



Simple Task Example

A pool of 8 threads is created here

```
#pragma omp parallel
// assume 8 threads
{
  #pragma omp single private(p)
  {
    …
    while (p) {
    #pragma omp task
      {
        processwork(p);
      }
      p = p->next;
    }
  }
}
```

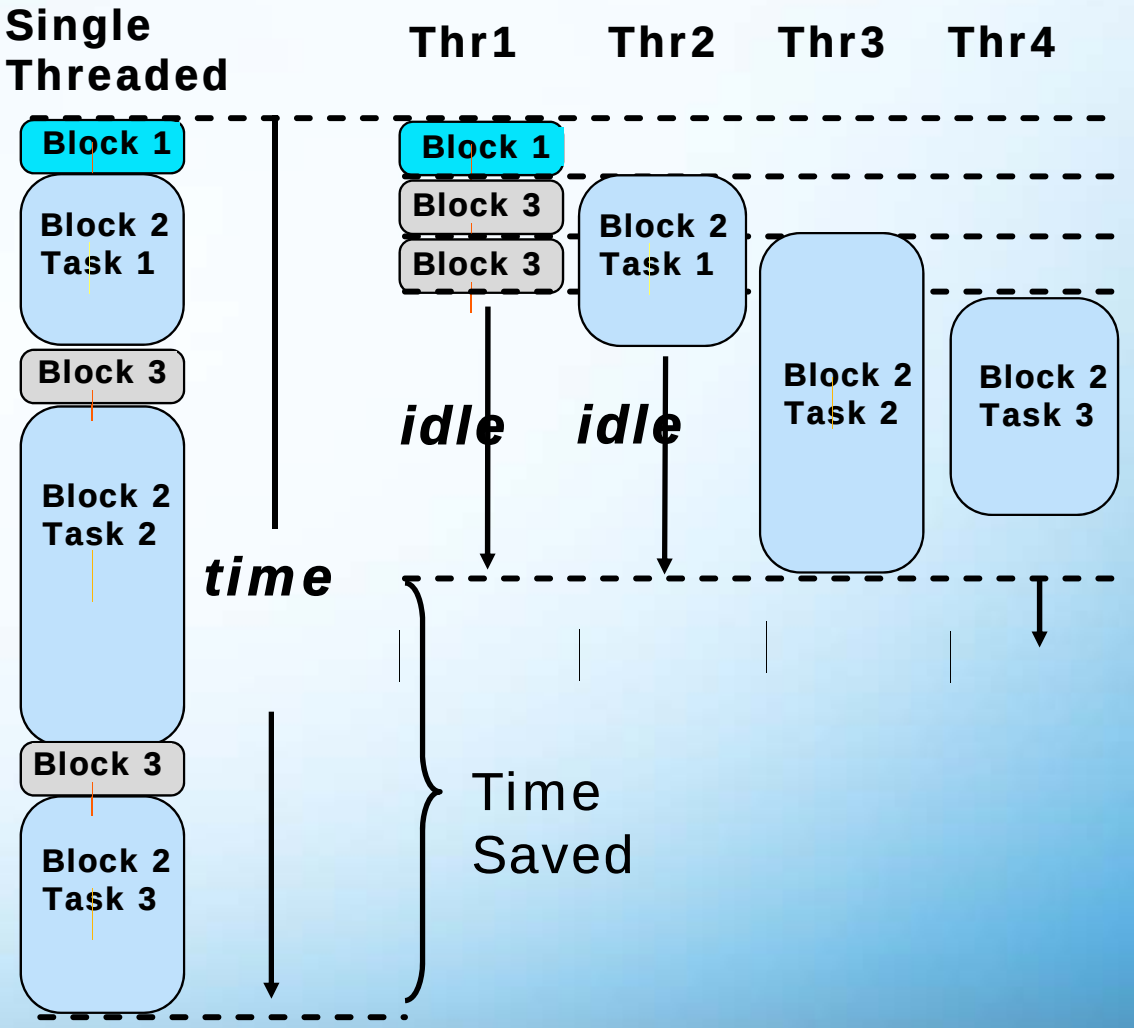
One thread executes a while loop

single “while loop” thread creates a task for each instance of processwork()

Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {// block 1
    node * p = head;
    while (p) {  //block 2
    #pragma omp task
      process(p);
    p = p->next;  //block 3
    }
  }
}
```



Task Completion

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

```
#pragma omp taskwait
```

```
!$omp flush taskwait
```

Task Completion Example

```
#pragma omp parallel
{
  #pragma omp task
  foo();
  #pragma omp barrier
  #pragma omp single
  {
    #pragma omp task
    bar();
  }
}
```

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here

Example

```
$ icc test.c
$ ./a.out
A race car
$
```

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

int main(int argc, char *argv[]) {
  printf("A ");
  printf("race ");
  printf("car ");

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

Example

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

int main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    printf("A ");
    printf("race ");
    printf("car ");
  }// End of parallel region
  printf("\n");
  return(0);
}
```

```
$ icc -openmp test.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car A race car
```

Note that this program could for example also print
“A A race race car car ” or
“A race A car race car”, or
“A race A race car car”, ...

Example

```
$ icc -openmp test.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
```

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

int main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      printf("race ");
      printf("car ");
    }
  }// End of parallel region
  printf("\n");
  return(0);
}
```

But now only 1 thread executes!

Example

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

int main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      {printf("race ");}
      #pragma omp task
      {printf("car ");}
    }
  }// End of parallel region
  printf("\n");
  return(0);
}
```

```
$ icc -openmp test.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A race car
$ ./a.out
A race car
$ ./a.out
A car race
$
```

Tasks can be executed in arbitrary order

Example

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

int main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      {printf("race ");}
      #pragma omp task
      {printf("car ");}
      printf(“is fun to watch “);
    }
  }// End of parallel region
  printf("\n");
  return(0);
}
```

```
$ icc -openmp test.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A is fun to watch race car
$ ./a.out
A is fun to watch race car
$ ./a.out
A is fun to watch car race
$
```

Tasks are executed at a task execution point

Example

```
#include <stdlib.h>
#include <stdio.h>
int main(int argc, char *argv[]) {
  #pragma omp parallel
  {
    #pragma omp single
    {
      printf("A ");
      #pragma omp task
      {printf("race ");}
      #pragma omp task
      {printf("car ");}
      #pragma omp taskwait
      printf(“is fun to watch “);
    }
  }// End of parallel region
  printf("\n");
  return(0);
}
```

```
$ icc -openmp test.c
$ export OMP_NUM_THREADS=2
$ ./a.out
A car race is fun to watch
$ ./a.out
A car race is fun to watch
$ ./a.out
A race car is fun to watch
$
```

Tasks are executed first now

Example – Fibonacci Numbers

The Fibonacci Numbers are defined as follows:

```
F(0) = 1
F(1) = 1
F(n) = F(n-1) + F(n-2) (n=2,3,4,.....)
```

Sequence:

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

Recursive Algorithm

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

  fnm1 = comp_fib_numbers(n-1);
  fnm2 = comp_fib_numbers(n-2);
  fn   = fnm1 + fnm2;

  return(fn);
}
```

Parallel Recursive Algorithm

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

  if ( n == 0 || n == 1 )
    return(n);

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

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

  #pragma omp taskwait
  fn   = fnm1 + fnm2;

  return(fn);
}
```

Clauses on the task directive

```
#pragma omp task [clause[[,]clause] ...]
```

if(scalar-expression)

If false, create an undeferred task, encountering thread must suspend the encountering task region, resume execution of the current task region until the task is completed

Any task can resume after suspension

untied

Specifies that the generated task will be a *final task*

default(shared | none) **private(list)** **firstprivate(list)** **shared(list)**

final(scalar-expression)

rred task or an included task, the implementation may generate a merged task

mergeable

Tied & Untied Tasks

- Tied Tasks:
  - A tied task gets a thread assigned to it at its first execution and the same thread services the task for its lifetime
  - A thread executing a tied task, can be suspended, and sent off to execute some other task, but eventually, the same thread will return to resume execution of its original tied task
  - Tasks are tied unless explicitly declared untied

- Untied Tasks:
  - An united task has no long term association with any given thread. Any thread not otherwise occupied is free to execute an untied task. The thread assigned to execute an untied task may only change at a "task scheduling point".
  - An untied task is created by appending “untied” to the task clause
  - Example: #pragma omp task untied

Task switching

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

Task switching example

The thread executing the “for loop” , AKA the generating task, generates many tasks in a short time so…
The SINGLE generating task will have to suspend for a while when “task pool” fills up

- Task switching is invoked to start draining the “pool”
- When “pool” is sufficiently drained – then the single task can being generating more tasks again

```
int exp;  //exp either T or F;
#pragma omp single
{
    for (i=0; i<ONEZILLION; i++)
     #pragma omp task
         process(item[i]);
    }
}
```

Agenda

- Getting Started with OpenMP
  - What Is OpenMP?
  - Programming Model
  - Memory Model
- Using OpenMP
- What's New in OpenMP

New OpenMP 3.1 Features

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

Additions to tasking

- The “mergeable” clauses
  - When a mergeabe clause is present on a task construct, and the generated task is undeferred or included, the implementation may generate a merged task instead
    - A merged task is a task whose data environment, inclusive of ICVs, is the same as that of its generating task region
- The “final” clause
  - If true, the generated task will be a final task
  - All tasks generated encountered during execution of a final task will generate included tasks
    - An included task is a task for which execution is sequentially included in the generating task region; that is, it is undeferred and executed immediately by the encountering thread

Additions to tasking

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

```
#pragma omp taskyield
```

```
!$omp taskyield
```

atomic enhancements

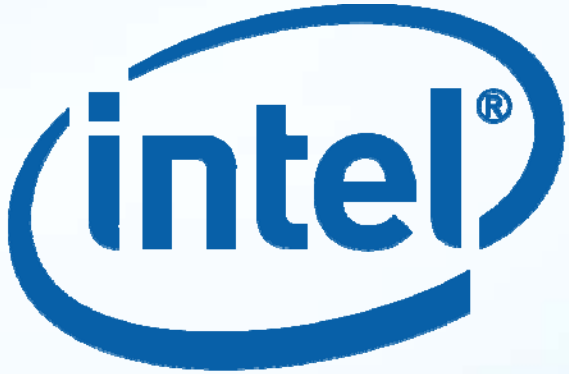
- New “update” clause
- New “read”, “write” and “capture” forms
- Disallow closely nested parallel regions within atomic”
  - Clarification of existing restriction

Summary OpenMP

- OpenMP provides for a small, but yet powerful, programming model
- It can be used on a shared memory system of any size
  - This includes a single socket multicore system
- Compilers with OpenMP support are widely available
- The tasking concept opens up opportunities to parallelize a wider range of applications
- OpenMP continues to evolve!
  - The new 4.0 specifications are again a step forward

References

- OpenMP 3.1
  - http://www.openmp.org/mp-documents/OpenMP3.1.pdf
- Intel® C++/Fortran Compiler XE 13.0 User and Reference Guides
- Chapman, Jost, van der Pas “Using OpenMP”, MIT Press, 2008
  - ISBN-10: 0-262-53302-2
  - ISBN-13: 978-0-262-53302-7



Software

Optimization Notice

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

Legal Disclaimer

INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.

Performance tests and ratings are measured using specific computer systems and/or components and reflect the approximate performance of Intel products as measured by those tests. Any difference in system hardware or software design or configuration may affect actual performance. Buyers should consult other sources of information to evaluate the performance of systems or components they are considering purchasing. For more information on performance tests and on the performance of Intel products, reference www.intel.com/software/products.

BunnyPeople, Celeron, Celeron Inside, Centrino, Centrino Atom, Centrino Atom Inside, Centrino Inside, Centrino logo, Cilk, Core Inside, FlashFile, i960, InstantIP, Intel, the Intel logo, Intel386, Intel486, IntelDX2, IntelDX4, IntelSX2, Intel Atom, Intel Atom Inside, Intel Core, Intel Inside, Intel Inside logo, Intel. Leap ahead., Intel. Leap ahead. logo, Intel NetBurst, Intel NetMerge, Intel NetStructure, Intel SingleDriver, Intel SpeedStep, Intel StrataFlash, Intel Viiv, Intel vPro, Intel XScale, Itanium, Itanium Inside, MCS, MMX, Oplus, OverDrive, PDCharm, Pentium, Pentium Inside, skoool, Sound Mark, The Journey Inside, Viiv Inside, vPro Inside, VTune, Xeon, and Xeon Inside are trademarks of Intel Corporation in the U.S. and other countries.
*Other names and brands may be claimed as the property of others.

http://intel.com/software/products