C-DAC Four Days Technology Workshop

ON

Hybrid Computing – Coprocessors/Accelerators
Power-Aware Computing – Performance of Applications Kernels

hyPACK-2013
(Mode-1:Multi-Core)

Lecture Topic:

Multi-Core Processors : Shared Memory Prog: OpenMP Part-I

Venue : CMSD, UoHYD ; Date : October 15-18, 2013

Explicit Parallelism :Shared Memory Programming: The OpenMP Standard

Lecture outline

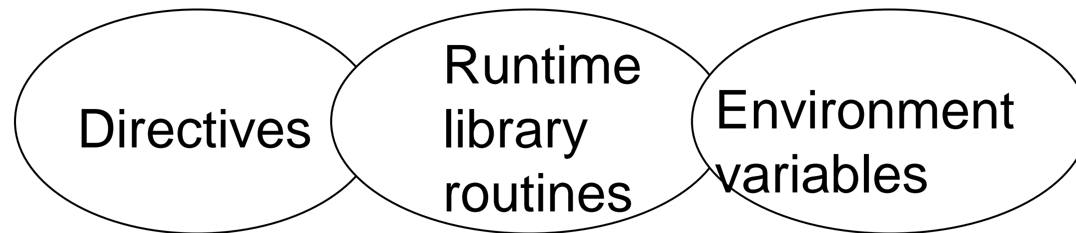
- ❖Introduction to OpenMP
- ❖OpenMP Programming Model
- ❖OpenMP Constructs
  - ➢ Directives
  - ➢Runtime Libraries
  - ➢ Environment variables

Source : Reference : [4], [6], [14],[17], [22], [28]

OpenMP: Introduction

OpenMP is an API for writing Multithreaded Applications

- It is a specification for



- Portable : Makes it easy to crate multi-threaded programs in C,C++ and Fortran.
- Standardizes the SMP practice

Source : Reference : [4], [6], [14],[17], [22], [28]

OpenMP: Introduction

Why OpenMP ?

- ❖ Relatively easy to do parallelization for small parts of an application at a time.
- ❖ Impact on code quantity (e.g., amount of additional code required) and code quality (e.g., readability of parallel code)
- ❖ Feasibility of scaling an application to a large number of processes.
- ❖ Readability of the parallel code is high
- ❖ Availability of application development and debugging environment
- ❖ Standard and portable API

Commonly Encountered Questions While Threading Application ?

- Where to thread ?
- How long would it take to thread?
- How much re-design / efforts is required?
- Is it worth threading the selected region ?
- What should the expected speedup be?
- Will the performance meet expectations?
- Will it scale if the more number of processor added?
- Which threading model is it?

OpenMP: Introduction

History

- ❖ First standard ANSI X3H5 in 1994
- ❖ OpenMP Standard SPECs started in 1997
- ❖ OpenMP Architecture review board
  - ➢ Compaq, HP, IBM, Sun Micro System, Intel Corp, Kuck & Associate Inc (KAI), SGI, US Dept. of Energy, ASCI program.

OpenMP: Introduction

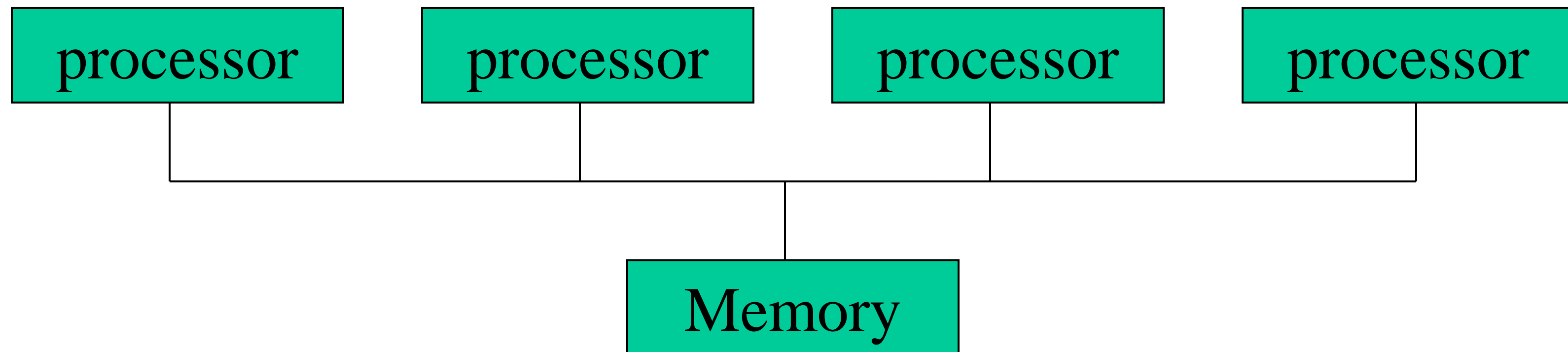
Supporters

- Hardware vendors
  - Intel, HP, SGI, IBM, SUN, Cray, AMD
- Software tools vendors
  - pathscale, Intel   PGI, SGI, Sun, Absoft

OpenMP: Programming Model

Shared Memory Model

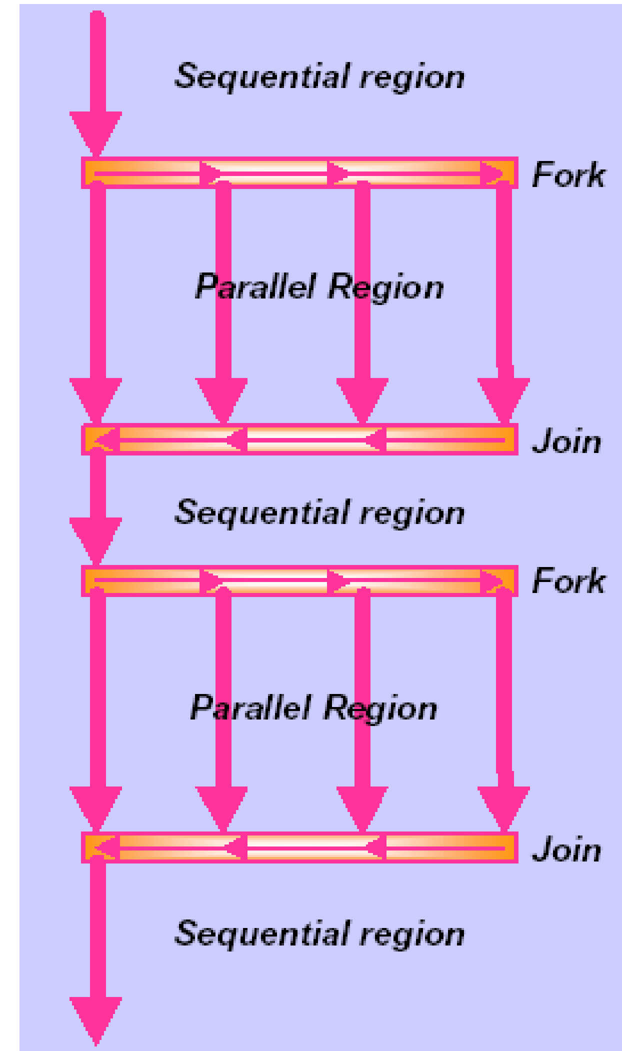
- Processes synchronize and communicate with each other through shared variables
- Supports *incremental parallelization.*



OpenMP: Programming Model

❖ **Fork – Join parallelism**

- OpenMP uses fork and join model for parallel execution
- OpenMP programs begin with single process: **master thread**.
- FORK : Master thread creates a team of parallel threads
- JOIN: When the team threads complete the statements in parallel region, they synchronize and terminate leaving master thread.
- Parallelism is added incrementally



OpenMP: Programming Model

❖ **Threads based parallelization**

➢ Open MP is based on the existence of multiple threads in the shared memory programming paradigm

❖ **Explicit parallelization**

➢ It is an explicit programming model, and offers full control over parallelization to the programmer

❖ **Compiler directive based**

➢ All of OpenMP parallelization is supported through the use of compiler directives

❖ **Nested parallelism support**

➢ The API support placement of parallel construct inside other parallel construct

Source : Reference : [4], [6], [14],[17], [22], [28]

- **Dynamic threads**
  - The API provides dynamic altering of number of threads (Depends on the implementation)

How do threads interact?

- **OpenMP is shared memory model.**
  - Threads communicate by sharing variables
- **Unintended sharing of data can lead to race conditions:**
  - Race condition : when the program's outcome changes as the threads are scheduled differently
  - To control race conditions: Use synchronization to protect data conflicts

OpenMP : Fortran Directives Format

Format

sentinel	directive-name	[clause ...]
All Fortran OpenMP directives must begin with a sentinel. The accepted sentinels depend upon the type of Fortran source. Possible sentinels are: **!$OMP** **C$OMP** ***$OMP**	A valid OpenMP directive. Must appear after the sentinel and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

Example

```
!$OMP PARALLEL SHARED(ALPHA) PRIVATE(BETA)
```

Source : Reference : [4], [6], [14],[17], [22], [28]

OpenMP : C/C++ Directives Format

Format

sentinel	directive-name	[clause ...]	newline
Required for all OpenMP C/C++ directives #pragma omp	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Proceeds the structured block which is enclosed by this directive.

Example

```
#pragma omp parallel shared(alpha), private(beta)
```

OpenMP : C/C++ Directives Format

Format : Compiler Directive

❖ C/C++

```
sentinel  directive-name clause
```

Ex.

```
#pragma omp parallel shared(alpha), private(beta)
```

❖ Fortran

```
sentinel  directive-name clause
```

Ex.

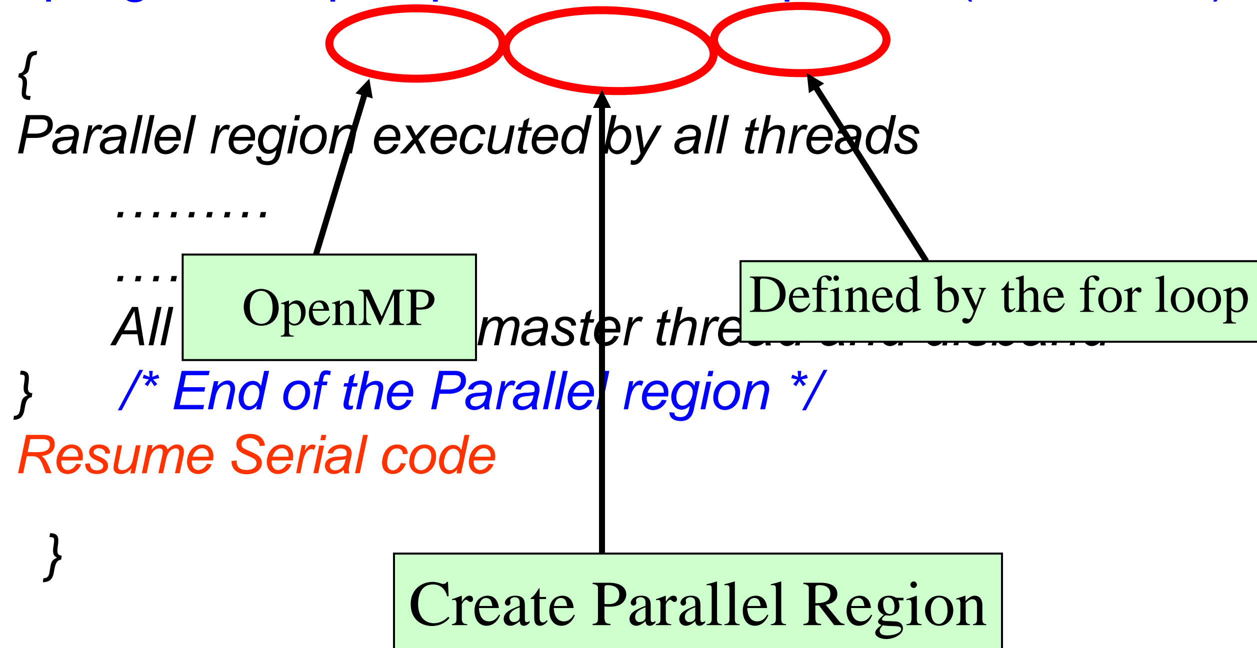
```
!$OMP directive [clause, …]
!C$OMP directive [clause, …]
!*$OMP directive [clause, …]
```

OpenMP : C/C++ General Code Structure

```
# include <omp.h>
main()  {
int var1, var2, var3;
serial code
……
Beginning of parallel region, fork a team of threads.
Specify variable scoping
#pragma omp  parallel private (var1, var2), shared(var3)
{
          Parallel region executed by all threads
          ………
          ……….
          All threads join master thread and disband
}
Resume Serial code
}
```

OpenMP : C/C++ General Code Structure

```
# include <omp.h>
main()  {

int var1, var2, var3; Create thread here for this parallel region        serial
code                                                                     ……
Beginning of parallel region, fork a team of threads.
#pragma  omp    parallel    for   private (var1, var2)
```




```
{
Parallel region executed by all threads
      ………
      ….
      All        master thread and disband
}     /* End of the Parallel region */
Resume Serial code

  }
```

OpenMP : FORTRAN General Code Structure

```
        PROGRAM HELLO
        INTEGER VAR1, VAR2, VAR3
         Serial code
         .......
         .......
         Beginning of parallel region, fork a team of threads.
         Specify variable scoping
!$OMP PARALLEL PRIVATE (VAR1, VAR2), SHARED(VAR3)
         Parallel region executed by all threads
         .........
         ..........
         All threads join master thread and disband
!$OMP END PARALLEL
         .......
         Resume Serial code
         END
```

OpenMP : How is OpenMP typically used? (C/C++)

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop between multiple threads

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

Sequential Program

```
#include "omp.h"
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
          do_huge_comp(Res[i]);
   }
}
```

Parallel Program

OpenMP : How is OpenMP typically used? (Fortran)

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop between multiple threads

```
program example
double precision Res(1000)

do I=1,1000
   call huge_comp(Res(I))
end do
end
```

Sequential Program

→

```
      program example
      double precision Res(1000)
C$OMP PARALLEL DO
      do I=1,1000
         call huge_comp(Res(I))
      end do
      end
```

Parallel Program

OpenMP : Constructs

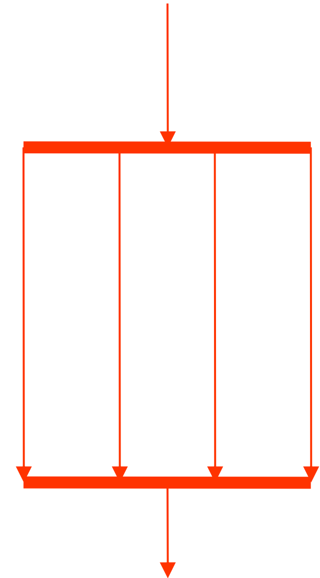
Main categories of OpenMP's constructs:

- **Directives**
  - Parallel Regions
  - Work-sharing
  - Data Environment
  - Synchronization
- **Runtime library functions**
  - Execution Environment Functions
  - Lock functions
  - Timing routines
- **Environment variables**

OpenMP : 'PARALLEL' Region Construct

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



All thread perform identical task

OpenMP : PARALLEL Region Construct

Example

Each thread redundantly executes the code within the structured block

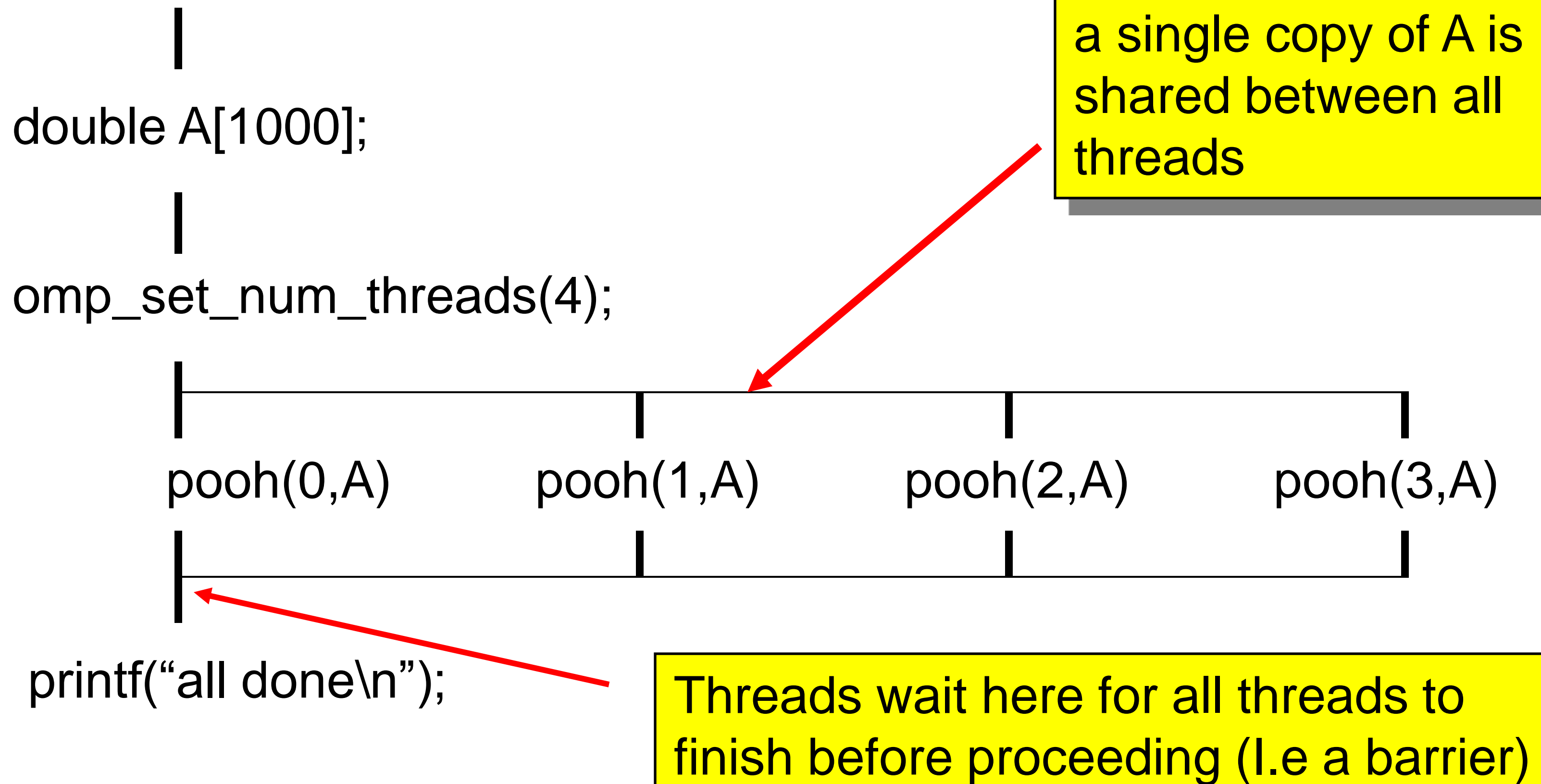
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_thread_num();
        pooh(ID,A);
}
Printf("all done\n");
```

Runtime function to request a certain number of threads

Runtime function returning a thread ID

OpenMP : PARALLEL Region Construct

Each thread executes the same code reduntantly.



OpenMP : Work-sharing Construct

It distributes the execution of the associated statement among the members of the team that encounter it

- Work sharing construct do not launch new threads
- There is no barrier upon entry to work-sharing construct.
- There is an implied barrier at the end of a worksharing construct

Restrictions

- Must be enclosed in the parallel region for parallel execution
- Must be encountered by all the members of the team or none of them

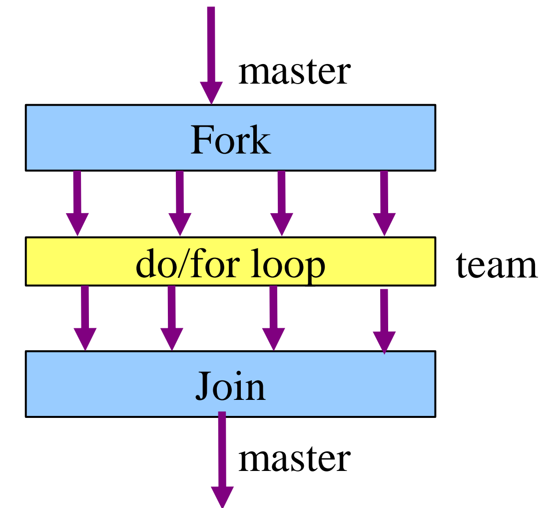
OpenMP : Work-sharing Construct

OpenMP defines the following work-sharing constructs.

- ❖ **for** directive
- ❖ **sections** directive
- ❖ **single** directive

OpenMP : Work-sharing construct 'for'

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.



Split-up this loop between multiple threads

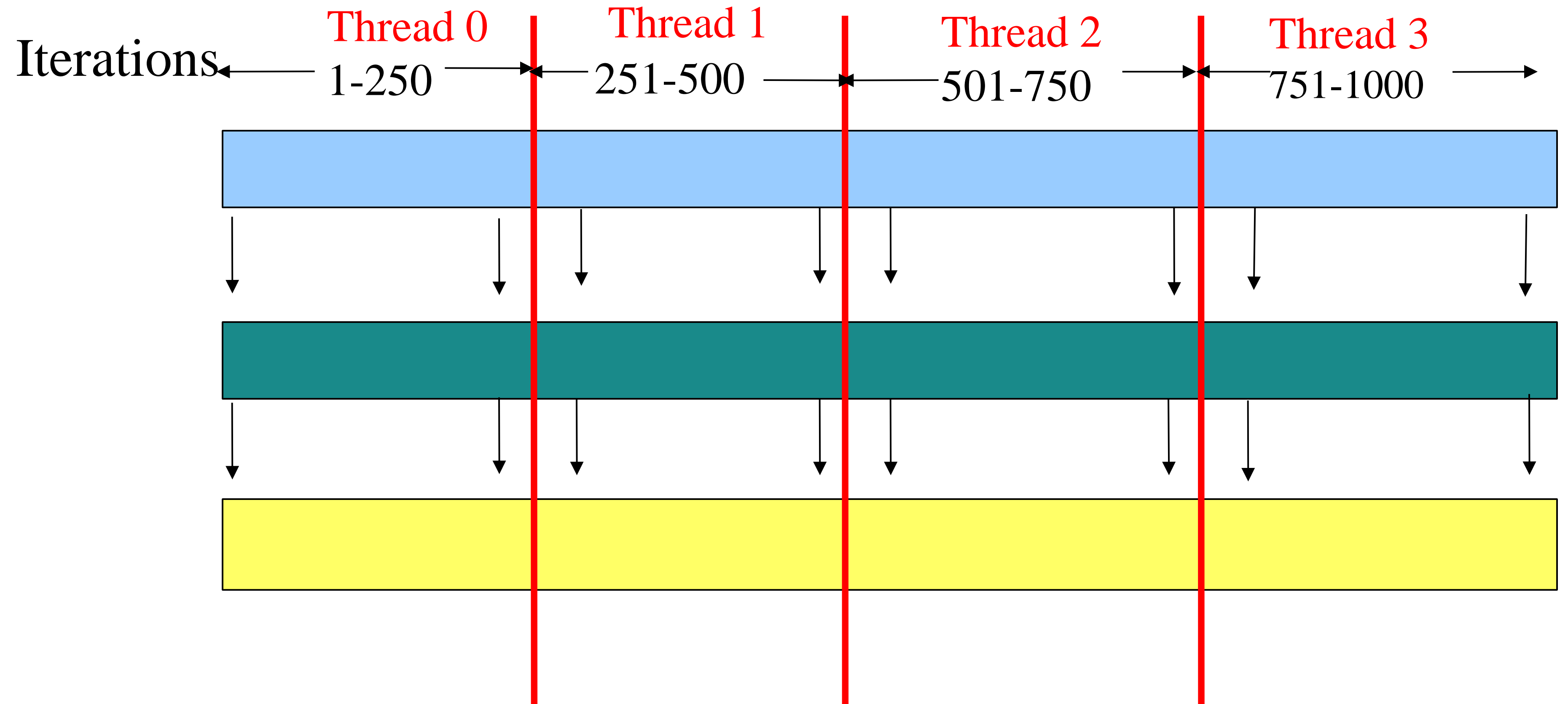
```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
do_huge_comp(Res[i]);
   }
}
```

Sequential Program

```
#include "omp.h"
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
do_huge_comp(Res[i]);
   }
}
```

Parallel Program

OpenMP : Work-sharing construct 'for'



OpenMP : Work-sharing Construct

for directive

for directive identifies the iterative work-sharing construct.

#***pragma omp for*** *[clause[[,]clause]…] new-line*

for-loop

Clause is one of the following:

private(*variable list*)

firstprivate (*variable list*)

lastprivate (*variable list*)

reduction (*variable list*)

ordered , nowait

OpenMP : Work-sharing Construct

for directive

The "for" Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
 #pragma omp for
    for (I=0;I<N;I++) {
        NEAT_STUFF(I);
    }
```

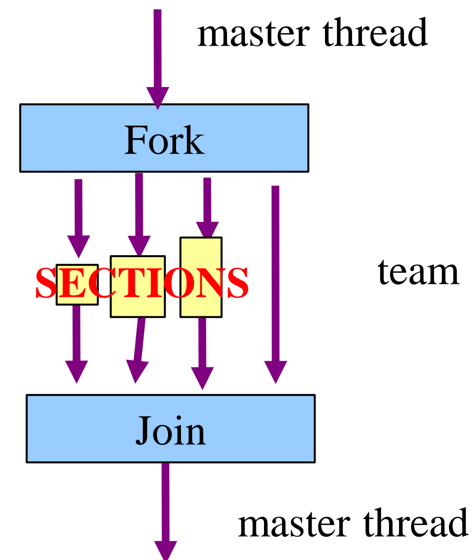
By default, there is a barrier at the end of the "omp for".

OpenMP : Work-sharing Construct

sections directive

sections directive gives different structured blocks to each thread.

```
#pragma omp parallel
#pragma omp sections {
#pragma omp section
    x_calculation(); // thread 1 work
#pragma omp section
    y_calculation();  // thread 2 work
……
}
```



OpenMP : Work-sharing Construct

single directive

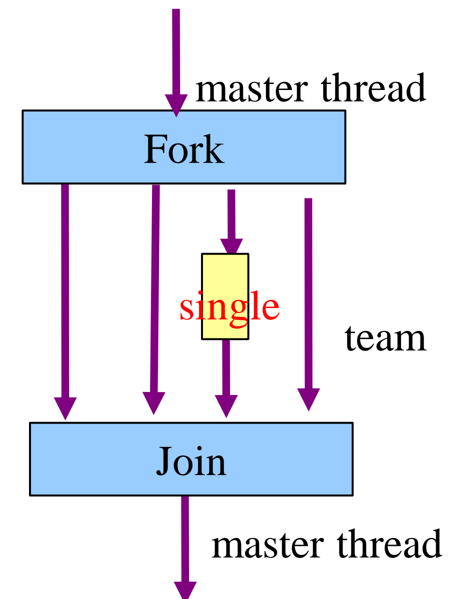
This identifies that the associated structured block is to be executed by only one thread in the team (It can be any thread including master thread).

#pragma omp single *[clause[[,] clause] ...] new-line*

structured-block

Example : !$omp single
call read_array(in, len)

!$omp end single



OpenMP : Data Environment

Default Storage attributes

- ❖Shared Memory programming model:
  - ➢Most variables are shared by default
- ❖Global variables and SHARED among threads
  - ➢Fortran : COMMON blocks, SAVE variables, MODULE variables.
  - ➢C: File scope variables, static
- ❖But not everything is shared…
  - ➢Stack variables in sub-programs called from parallel regions are PRIVATE
  - ➢Automatic variables within a statement block are PRIVATE.

OpenMP : Data Environment

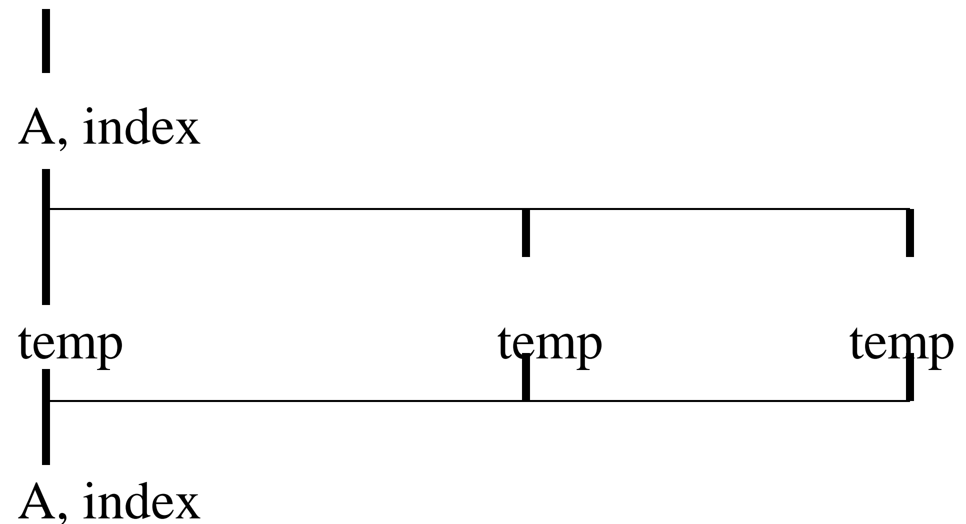
Example : Storage attributes

```
      program sort
      common  /input/ A(10)
      integer index(10)
      call input
C$OMP PARALLEL
      call work(index)
C$OMP END PARALLEL
      print *, index(1)
```

```
      subroutine work
      common  /input/ A(10)
      real temp(10)
```

A, index are shared by all threads

temp is local to each thread



OpenMP : Data Environment

Changing the storage attributes

One can selectively change storage attributes constructs using the following clauses*

SHARED declares variables to be shared among all threads in the team

PRIVATE declares variables to be private to each thread.

FIRSTPRIVATE performs initialization of private variables

LASTPRIVATE performs finalization of private variables

REDUCTION performs a reduction on the variables subject to given operator.

The default status can be modified with:

DEFAULT (PRIVATE | SHARED)

OpenMP : Data Environment

private clause

Creates a local copy of variable for each thread.

- The value is **un-initialized**
- Private copy is **not** storage associated with the original

```
program sample
      IS = 0
!$OMP PARALLEL DO PRIVATE(IS)
      DO J=1,1000
         IS = IS + J
      ENDDO
      print *, IS
```

IS was not initialized

IS is undefined at this point

OpenMP : Data Environment

- First private initializes each thread's copy of a private variable to the value of the master copy.
- Last private writes back to the master's copy the value of private copy that executed the Sequentially last iteration.

```
program closer
        IS = 0
C$OMP   PARALLEL DO
FIRSTPRIVATE(IS) LASTPRIVATE(IS)
            DO J=1,1000
           IS = IS + J
      ENDDO
 print *, IS
```

Each thread gets its own `IS` with an initial value of 0

`IS` is defined as its value at the last iteration (i.e. for J=1000)

Example : Firstprivate & lastprivate

```
int x;

x = 0;

#pragma omp parallel for firstprivate(x)
for (i = 0; i < 10000; i++) {

x = x + i;
}

printf( “x is %d\n”, x );
```

Initialise x to zero

Each thread gets its own is with an initial value of 0

Print out value of x

Oops! The value x is undefined!
Need lastprivate(x) to copy value back out to master

OpenMP : Data Environment

Example : Default Clause

```
        itotal = 1000
#pragma omp parallel private(np, each)
{
        np = omp_get_num_threads()
        each = itotal/np
}
```

These two codes are equivalent

```
        itotal = 1000
#pragma omp parallel Default(private) shared(itotal)
{
        np = omp_get_num_threads()
        each = itotal/np
}
```

Default clause

This clause is used for changing the default status of the variables.

- default (private)
  - *Each* variable in *static* extent of the parallel region is made **private** as if specified in a private clause
- default (shared)
  - *Each* variable in *static* extent of the parallel region is made **shared** as if specified in a shared clause
- default (none)
  - *no* default for variables in static extent.

OpenMP : Data Environment

Example : Default Clause

```
        itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
        np = omp_get_num_threads()
        each = itotal/np
C$OMP END PARALLEL
```

These two codes are equivalent

```
        itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE)
SHARED(itotal)
        np = omp_get_num_threads()
        each = itotal/np
C$OMP END PARALLEL
```

Reduction

- Another clause that effects the way variables are shared
  - reduction(op:list)
- The variables in "list" must be shared in the enclosing parallel region.
- Inside a parallel or a work sharing construct:
  - A local copy of each list variable is made and initialized depending on the "op" (e.g 0 for "+")
  - Pair wise "op" is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct

OpenMP Clauses

Example : Reduction

```
#include <omp.h>
#define NUM_THREADS 2
void main() {
        int I;
        double ZZ, func(), res = 0.0;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+;res) private(ZZ)
        for(I=0;I<1000;I++)}
                ZZ=func(I);
                res = res + ZZ;
        }
}
```

OpenMP: Synchronization

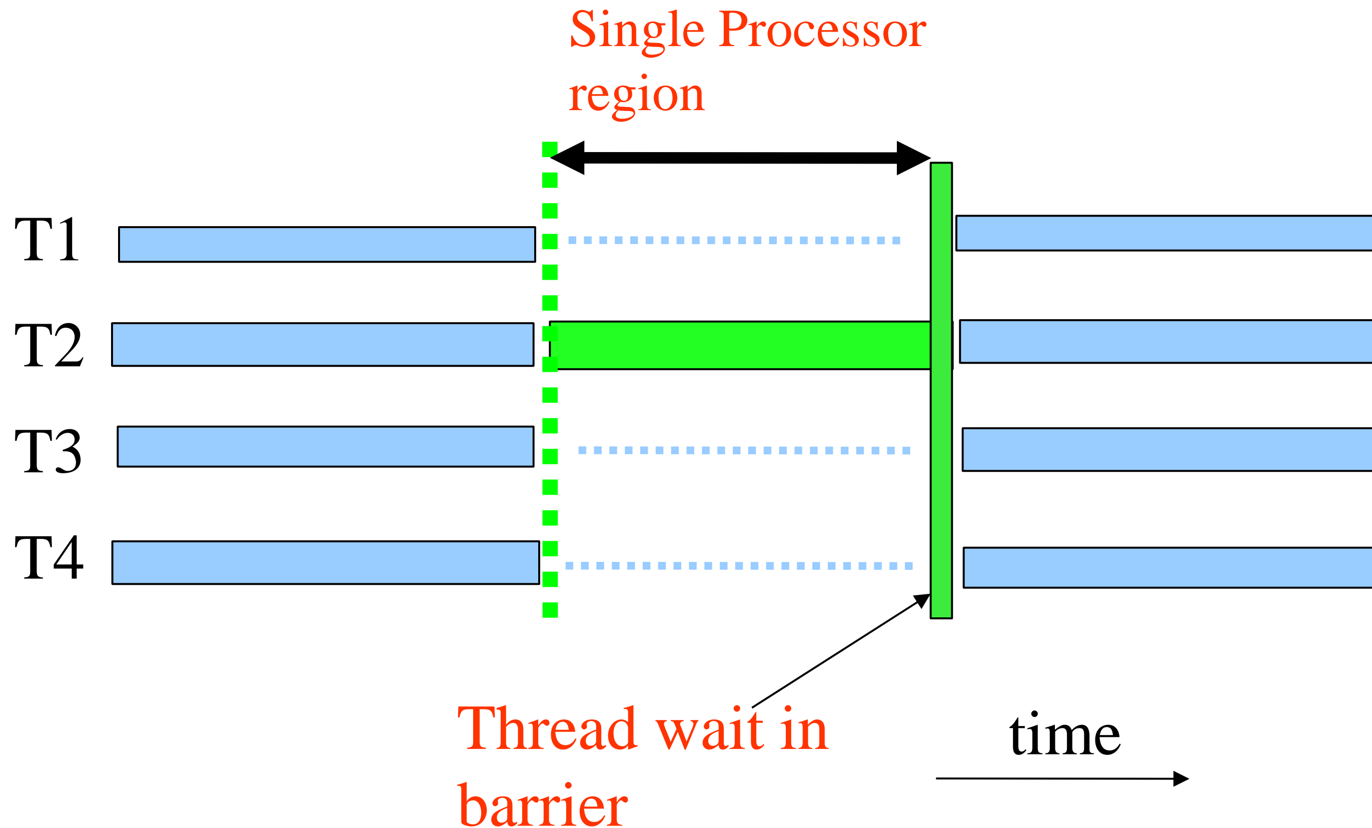
- OpenMP has the following constructs support synchronization
  - Atomic
  - Critical section
  - Barrier
  - Flush
  - Ordered

OpenMP: Synchronization constructs

- Some of the OpenMP synchronization constructs
  - Single
  - Master
  - Atomic
  - Critical section
  - Barrier
  - Ordered

SINGLE and MASTER Construct

Usually, there is a barrier at the end of the region



SINGLE and MASTER Construct

Only one thread in a team executes the code enclosed

```
#pragma omp single [clause[[,] clause] ...]
{
      <code-block>
}
```

Only the master thread executes the code block

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

There is no implied barrier on the entry & exit

Synchronization : Critical Section and Atomic

Only one thread at a time can enter a critical section

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

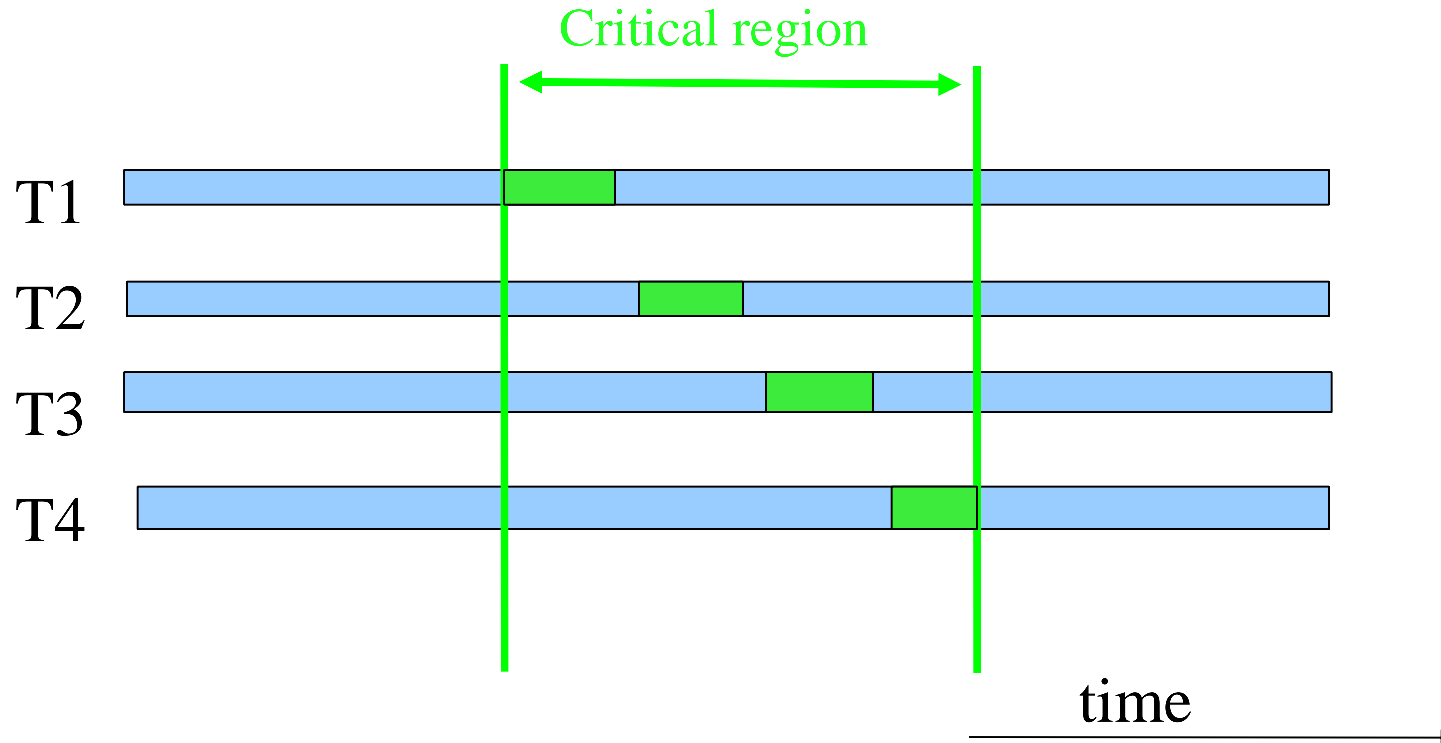
```
for (i=0; i < N; i++){
   .....
  sum += a[i];
   ..... }
```

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
}
```

Synchronization : Critical Section and Atomic

- Useful to avoid a race condition, or to perform I/O
- Be aware that your parallel computation may be serialized



Synchronization : Critical Section and Atomic

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

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

This is a lightweight, special form of a critical section

Synchronization : Critical Section and Atomic

Only one thread at a time can enter a critical section

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

Synchronization construct: 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 ?

Synchronization construct : 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];
```

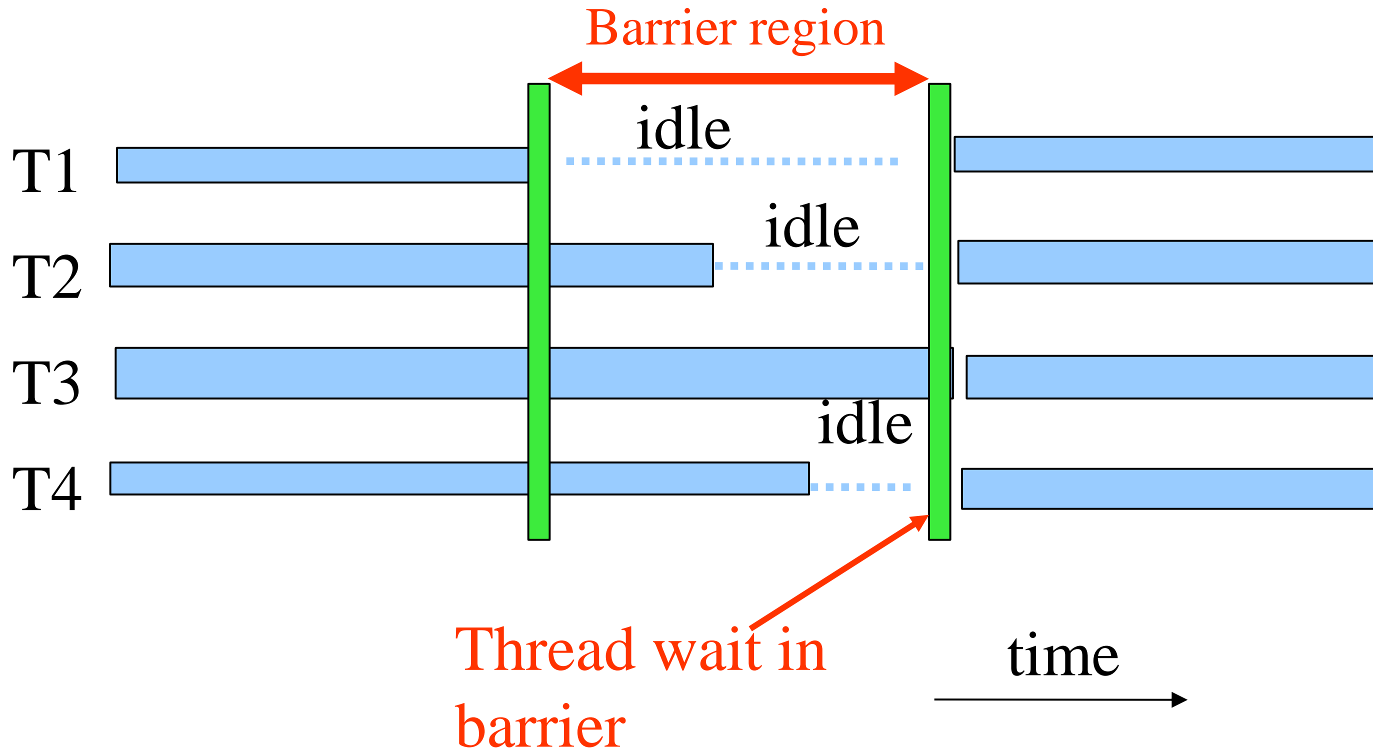
wait !

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

Synchronization Construct :Barrier



Each thread waits until all others have reached this point:

```
#pragma omp barrier
```

Critical Section

❖ Only one thread at a time can enter a critical section

```
cur_max = MINUS_INFINITY
 !$omp parallel do
     do I = 1 , n
 !omp critical
                 if (a(I) .gt. Cur_max) then
                      cur_max = a(I)
                 endif
 !omp  end critical
             endif
      enddo
```

OpenMP: Synchronization

```
#pragma omp parallel shared (A, B, C)
    private(d)
{
    id = omp_get_thread_num();
    A[id]=big_calc1(id);
#pragma omp barrier
#pragma omp for
    for(I=0;I<N;I++){ C[I]=big_calc3(I,A);}
#pragma omp for nowait
    for(I=0;I<N;I++){ B[I]=big_calc2(C,I);}
    A[id]=big_calc3(id);
}
```

Barrier : Each thread waits until all threads arrive

Implicit barrier at the end of a for work-sharing construct

No implicit barrier due to nowait

Implicit barrier at the end of a parallel region

OpenMP: Synchronization

Ordered

- The ordered construct enforces the sequential order for a block.

```
#pragma omp parallel private(tmp)
#pragma omp for ordered
   for (I=0;I<N;I++){
         tmp = NEAT_STUFF(I);
#pragma ordered
         res = consum(tmp);
   }
```

OpenMP: Synchronization

flush

- The FLUSH construct denotes a sequence point where a thread tries to create a consistent view of memory.
  - All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - Variables in registers or write buffers must be updated in memory.
- Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

OpenMP: Library Routines

❖**Lock routines**

omp_init_lock(), omp_set_lock(),

omp_destroy_lock(),

omp_unset_lock(),

omp_test_lock()

❖**Runtime environment routines:**

- Modify/Check the number of threads

omp_set_num_threads(),

omp_get_num_threads(),

omp_get_thread_num(),

omp_get_max_threads()

OpenMP: Library Routines

❖ **Runtime environment routines**

- How many processors in the system?
  - –omp_num_procs()
- Turn on/off nesting and dynamic mode

  omp_set_nested(),

  omp_get_nested(),

  omp_set_dynamic(),

  omp_get_dynamic()

❖Protect resources with locks

```
      omp_lock_tlck;
      omp_init_lock(&lck);
#pragma omp parallel private(tmp)
{
      id = omp_get_thread_num();
      tmp= do_lots_of_work(id);
      omp_set_lock(&lck);
      printf(“%d %d”, id, tmp);
      omp_unset_lock(&lck);
}
```

OpenMP: Library Routines

- **To fix the number of threads used in a program, first turn off dynamic mode and then set the number of threads.**

```
#include <omp.h>
Void main()
{
    omp_set_dynamic(0);
    omp_set_num_threads(4);
#pragma omp parallel {
    Int id=omp_get_thread_num();
    do_lots_of-stuff(id); }
}
```

❖ Environment Variables

Variables	Value	Description
OMP_NUM_THREADS	4	Specify the no. of threads
OMP_DYNAMIC	TRUE or FALSE	Enable/disable dynamic adj of threads
OMP_NESTED	TRUE or FALSE	Enable/disable nested parallelism

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
for (i=0; i<n; i++)
        z[i] = x[i] + y[i];

#pragma omp for
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

Compilation & Execution of OpenMP programs

```
#include<stdio.h>
#include<omp.h>
main() {
#pragma omp parallel
{
    printf( “hello world from thread %d of
              %d\n”, omp_get_thread_num(),
                 omp_get_num_threads() );
}
}
```

Compilation & Execution of OpenMP programs

Compilation :

$ cc –o <objectFileName> <programName> <omp-compiler-flag>

Ex.

$gcc –o omp-hello-world omp-hello-world.c -fopenmp

Setting the Number of Threads :

❖ Environment Variables :

$ export OMP_NUM_THREADS= <No. of threads>

❖ Environment variable can be overridden by the programmer :

omp_set_num_threads(int n)

Sample Output

From a Dual Socket Quad Core machine:

hello world from thread 0 of 8
hello world from thread 2 of 8
hello world from thread 3 of 8
hello world from thread 7 of 8
hello world from thread 6 of 8
hello world from thread 1 of 8
hello world from thread 4 of 8
hello world from thread 5 of 8

Commonly Encountered Questions While Threading Application ?

- ❖What should the expected speedup be?
- ❖Will the performance meet expectations?
- ❖ Will it scale if the more number of processor added?
- ❖ Which threading model is it?

Commonly Encountered Questions While Threading Application ?

Analysis

- ❖ Where to thread ?
  - ➢ thread the more time consuming section of code like loops
- ❖ How long would it take to thread?
  - ➢ Very minimum time just need to use some directives/library routine
- ❖ How much re-design / efforts is required?
  - ➢ Very less
- ❖ Is it worth threading the selected region ?
  - ➢ Appears to have minimal dependencies
  - ➢ Consuming over 90% of run time

OpenMP :Conclusions

- Features and advantages of OpenMP is discussed.
- OpenMP programming models are covered.
- Parallelization using OpenMP is explained.
- Various OpenMP Constructs are discussed with examples.

Source : Reference : [4], [6], [14],[17], [22], [28]

References

1. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Programming, Boston, MA : Addison-Wesley

2. Butenhof, David R **(1997),** Programming with POSIX Threads , Boston, MA : Addison Wesley Professional

3. Culler, David E., Jaswinder Pal Singh **(1999),** Parallel Computer Architecture - A Hardware/Software Approach , San Francsico, CA : Morgan Kaufmann

4. Grama Ananth, Anshul Gupts, George Karypis and Vipin Kumar **(2003),** Introduction to Parallel computing, Boston, MA : Addison-Wesley

5. Intel Corporation, **(2003),** Intel Hyper-Threading Technology, Technical User's Guide, Santa Clara CA : Intel Corporation Available at : http://www.intel.com

6. Shameem Akhter, Jason Roberts **(April 2006),** Multi-Core Programming - Increasing Performance through Software Multi-threading , Intel PRESS, Intel Corporation,

7. Bradford Nichols, Dick Buttlar and Jacqueline Proulx Farrell **(1996),** Pthread Programming O'Reilly and Associates, Newton, MA 02164,

8. James Reinders, Intel Threading Building Blocks – (**2007**) , O'REILLY series

9. Laurence T Yang & Minyi Guo (Editors), (**2006**) *High Performance Computing - Paradigm and Infrastructure* Wiley Series on Parallel and Distributed computing, Albert Y. Zomaya, Series Editor

10. Intel Threading Methodology ; Principles and Practices Version 2.0 copy right **(March 2003),** Intel Corporation

References

11. William Gropp, Ewing Lusk, Rajeev Thakur **(1999),** Using MPI-2, Advanced Features of the Message-Passing Interface, The MIT Press..
12. Pacheco S. Peter, **(1992),** Parallel Programming with MPI, , University of Sanfrancisco, Morgan Kaufman Publishers, Inc., Sanfrancisco, California
13. Kai Hwang, Zhiwei Xu, (**1998**), Scalable Parallel Computing (Technology Architecture Programming), McGraw Hill New York.
14. Michael J. Quinn (**2004**), Parallel Programming in C with MPI and OpenMP McGraw-Hill International Editions, Computer Science Series, McGraw-Hill, Inc. Newyork
15. Andrews, Grogory R. **(2000),** Foundations of Multithreaded, Parallel, and Distributed Progrmaming, Boston, MA : Addison-Wesley
16. SunSoft. Solaris multithreaded programming guide. SunSoft Press, Mountainview, CA, **(1996),** Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill,
17. Chandra, Rohit, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, and Ramesh Menon, **(2001)**,Parallel Programming in OpenMP San Fracncisco Moraan Kaufmann
18. S.Kieriman, D.Shah, and B.Smaalders **(1995),** Programming with Threads, SunSoft Press, Mountainview, CA. 1995
19. Mattson Tim, **(2002),** Nuts and Bolts of multi-threaded Programming Santa Clara, CA : Intel Corporation, Available at : http://www.intel.com
20. I. Foster **(1995,** Designing and Building Parallel Programs ; Concepts and tools for Parallel Software Engineering, Addison-Wesley (1995)
21. J.Dongarra, I.S. Duff, D. Sorensen, and H.V.Vorst **(1999),** Numerical Linear Algebra for High Performance Computers (Software, Environments, Tools) SIAM, 1999

References

22. OpenMP C and C++ Application Program Interface, Version 1.0". **(1998),** OpenMP Architecture Review Board. October 1998

23. D. A. Lewine. *Posix Programmer's Guide:* **(1991),** Writing Portable Unix Programs with the Posix. 1 Standard. O'Reilly & Associates, 1991

24. Emery D. Berger, Kathryn S McKinley, Robert D Blumofe, Paul R.Wilson, *Hoard : A Scalable Memory Allocator for Multi-threaded Applications* ; The Ninth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX). Cambridge, MA, November (**2000)**. Web site URL : http://www.hoard.org/

25. Marc Snir, Steve Otto, Steyen Huss-Lederman, David Walker and Jack Dongarra, (**1998**) *MPI-The Complete Reference: Volume 1, The MPI Core, second edition* [MCMPI-07].

26. William Gropp, Steven Huss-Lederman, Andrew Lumsdaine, Ewing Lusk, Bill Nitzberg, William Saphir, and Marc Snir (**1998**) *MPI-The Complete Reference: Volume 2, The MPI-2 Extensions*

27. A. Zomaya, editor. Parallel and Distributed Computing Handbook. McGraw-Hill, **(1996)**

28. OpenMP C and C++ Application Program Interface, Version 2.5 (**May 2005**)", From the OpenMP web site, URL **: http://www.openmp.org/**

29. Stokes, Jon 2002 Introduction to Multithreading, Super-threading and Hyper threading *Ars Technica*, October **(2002)**

30. Andrews Gregory R. 2000, Foundations of Multi-threaded, Parallel and Distributed Programming, Boston MA : Addison – Wesley (**2000)**

31. Deborah T. Marr , Frank Binns, David L. Hill, Glenn Hinton, David A Koufaty, J . Alan Miller, Michael Upton, "Hyperthreading, Technology Architecture and Microarchitecture", Intel (**2000-01)**

Thank You

Any questions ?