

Lecture 3

OpenMP

Outline

- Introduction to HPC computing
- OpenMP
 - Overview
 - Getting started
 - OpenMP constructs
 - Performance issues
 - Pitfalls
 - Case studies
- MPI

Documentation

- The material in these slides should be sufficient, but if in doubt you can always consult the official OpenMP document:

<http://www.openmp.org/mp-documents/OpenMP3.1.pdf>

OpenMP: What is it?

- An Application Program Interface (API) that may be used to explicitly direct **multi-threaded, shared memory parallelism**
- Using **compiler directives, library routines and environment variables** to automatically generate threaded (or multi-process) code that can run in a concurrent or parallel environment.
- **Portable:**
 - The API is specified for C/C++ and Fortran
 - Multiple platforms have been implemented including most Unix platforms and Windows
- **Standardized:** Jointly defined and endorsed by a group of major computer hardware and software vendors
- **What does OpenMP stand for?**

Open specifications for **Multi Processing** via collaborative work between interested parties from the hardware and software industry, government and academia.

OpenMP Is Not:

- Meant for distributed memory parallel systems (by itself)
 - Intel's Cluster OpenMP – an exception, but discontinued
- Necessarily implemented identically by all vendors
- Guaranteed to make the most efficient use of shared memory (currently there are no data locality constructs)

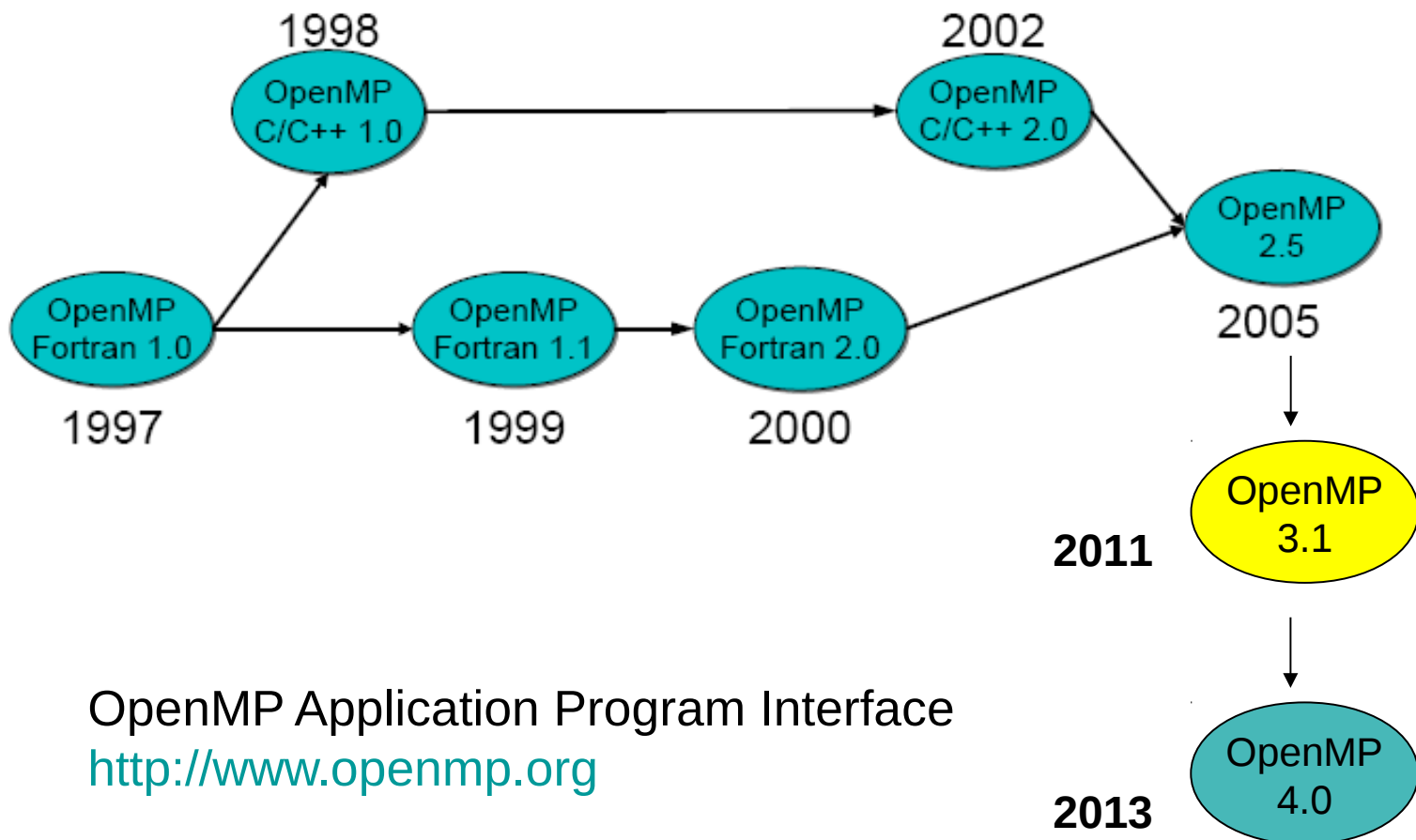
OpenMP: supporters

- Hardware vendors:
 - Intel, HP, SGI, IBM, AMD, ...
- Software vendors
 - compilers: PGI, Pathscale, Intel, GNU, ...
 - debuggers: DDT, totalview, etc
- Application vendors:
 - Gaussian, etc

OpenMP: Ancient History

- In the early 90's, vendors of shared-memory machines supplied similar, directive-based, Fortran programming extensions:
 - The user would augment a serial Fortran program with directives specifying which loops were to be parallelized
 - The compiler would be responsible for automatically parallelizing such loops across the SMP processors
- Implementations were all functionally similar, but were diverging (as usual)
- First attempt at a standard was the draft for ANSI X3H5 in 1994. It was never adopted, largely due to waning interest as distributed memory machines became popular.
- The OpenMP standard specification started in the spring of 1997, taking over where ANSI X3H5 had left off, as newer shared memory machine architectures started to become prevalent.

OpenMP Release History



OpenMP Application Program Interface
<http://www.openmp.org>

OpenMP: Goals

- **Standardization:**
Provide a standard among a variety of shared memory architectures/platforms
- **Lean and Mean:**
Establish a simple and limited set of directives for programming shared memory machines. Significant parallelism can be implemented by using just 3 or 4 directives.
- **Ease of Use:**
Provide capability to incrementally parallelize a serial program, unlike message-passing libraries which typically require an all or nothing approach
Provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:**
Supports Fortran (77, 90, and 95), C, and C++

OpenMP: Programming Model

- **Shared Memory, Thread Based Parallelism:**

OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.

- **Explicit Parallelism:**

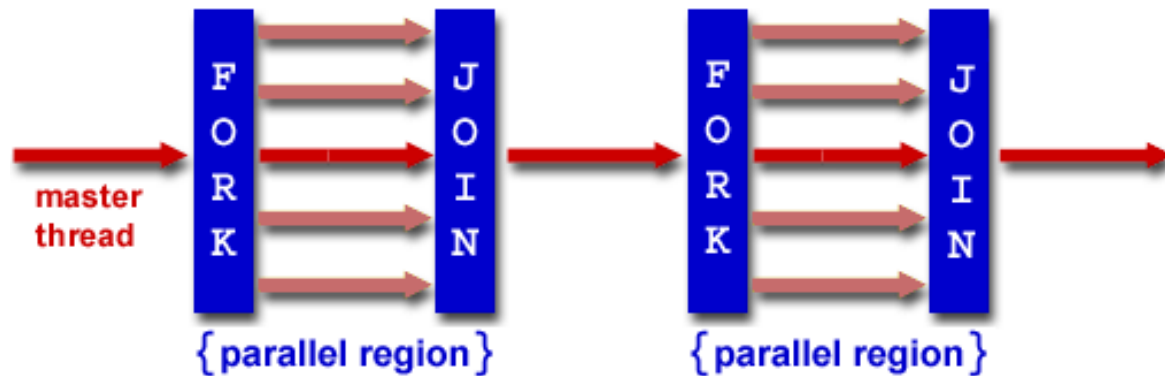
OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.

Notes:

- Intel compiler does provide the automatic feature via -parallel flag. No source code can be viewed, code will run but may have much worse performance.

OpenMP: Fork-Join Model

- OpenMP uses the fork-join model of parallel execution:



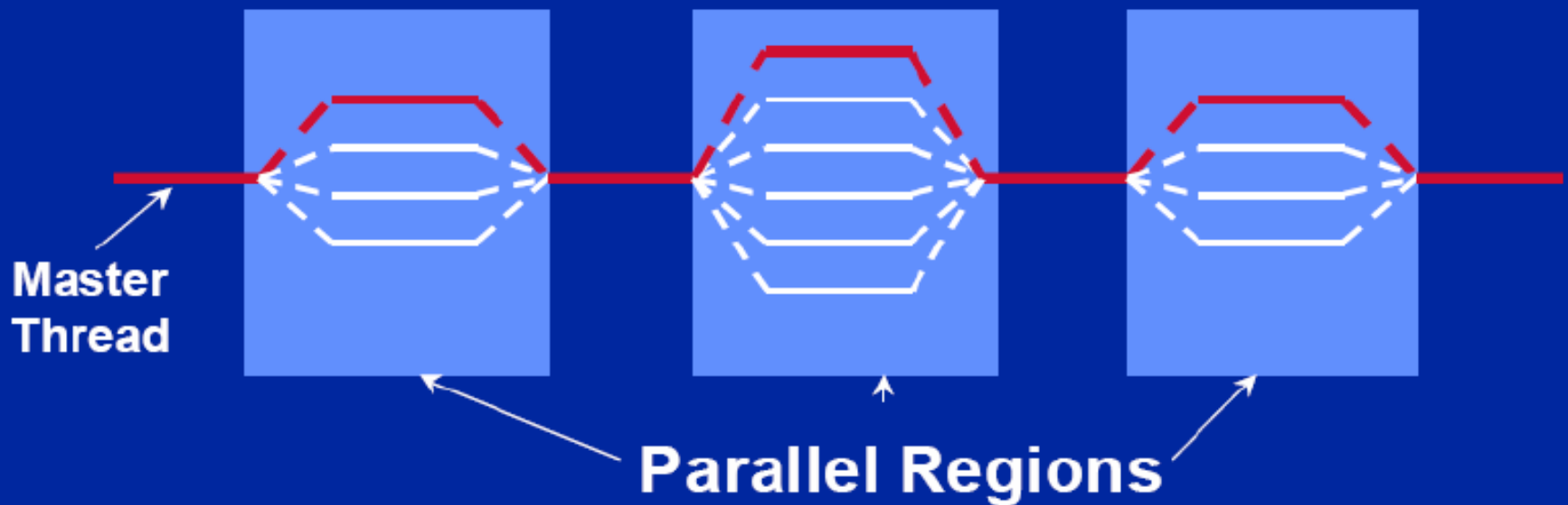
FORK: the master thread creates a *team* of parallel threads. The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the various team threads

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

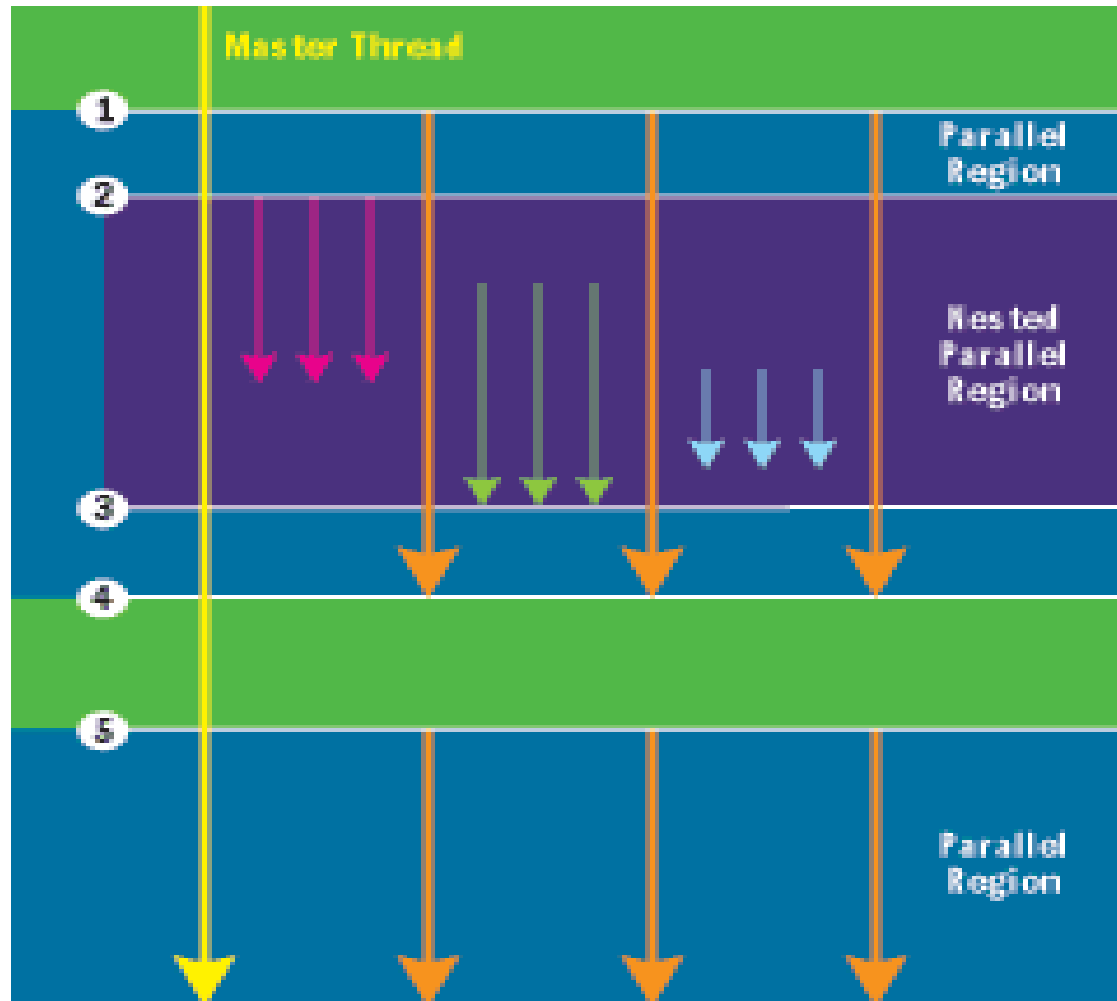
OpenMP dynamic execution model

Fork-Join Parallelism:

- ◆ **Master thread** spawns a **team of threads** as needed.
- ◆ Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.



OpenMP nested execution model



OpenMP: Programming Model (continue)

- **I/O:**
 - OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
 - If every thread carries out I/O to a different file, the issues are not as significant.
 - It is entirely up to the programmer to insure that I/O is conducted correctly within the context of a multi-threaded program.
- **FLUSH Often?:**
 - OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words).
 - When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.
 - More on this later...

OpenMP: Getting Started

OpenMP syntax: C/C++

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	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 default(shared) private(beta,pi)

General Rules:

- Case sensitive
- Directives follow conventions of the C/C++ standards for compiler directives
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block. (A structured block of code is a collection of one or more executable statements with a single point of entry at the top and a single point of exit at the bottom.)
- Long directive lines can be "continued" on succeeding lines by escaping the newline character with a backslash ("\") at the end of a directive line.

C / C++ - General Code Structure

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

```
main () {
```

```
    int var1, var2, var3;
```

```
    Serial code
```

```
    ...
```

```
    Beginning of parallel section. Fork a team of threads.
```

```
    Specify variable scoping
```

```
#pragma omp parallel private(var1, var2) shared(var3)
```

```
{
```

```
    Parallel section executed by all threads
```

```
    ...
```

```
    All threads join master thread and disband
```

```
}
```

```
    Resume serial code
```

```
    ...
```

```
}
```

OpenMP syntax: Fortran

Format: (case insensitive)

sentinel

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**

directive-name

A valid OpenMP directive. Must appear after the sentinel and before any clauses.

[clause ...]

Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.

Example: !\$OMP PARALLEL DEFAULT(SHARED) PRIVATE(BETA,PI)

Fixed Form Source (F77):

- **!\$OMP**, **C\$OMP**, ***\$OMP** are accepted sentinels and must start in column 1
- All Fortran fixed form rules for line length, white space, continuation and comment columns apply for the entire directive line
- Initial directive lines must have a space/zero in column 6.
- Continuation lines must have a non-space/zero in column 6.

Free Form Source (F90, F95):

- **!\$OMP** is the only accepted sentinel. Can appear in any column, but must be preceded by white space only.
- All Fortran free form rules for line length, white space, continuation and comment columns apply for the entire directive line
- Initial directive lines must have a space after the sentinel.
- Continuation lines must have an ampersand (&) as the last non-blank character in a line. The following line must begin with a sentinel and then the continuation directives.

General Rules for Fortran:

- Comments can not appear on the same line as a directive
- Only one directive-name may be specified per directive
- Fortran compilers which are OpenMP enabled generally include a command line option which instructs the compiler to activate and interpret all OpenMP directives.
- Several Fortran OpenMP directives come in pairs and have the form shown below. The "end" directive is optional for some directives (e.g. DO directive) but advised for readability.

!\$OMP *directive*

[*structured block of code*]

!\$OMP end *directive*

Fortran (77)- General Code Structure

PROGRAM HELLO

INTEGER VAR1, VAR2, VAR3

Serial code . . .

Beginning of parallel section. Fork a team of threads.

Specify variable scoping

!\$OMP PARALLEL PRIVATE(VAR1, VAR2) SHARED(VAR3)

Parallel section executed by all threads

. . .

All threads join master thread and disband

!\$OMP END PARALLEL

Resume serial code

. . .

END

SHARCNET

- Supercomputing consortium, one of 7 in Canada, covers a big chunk of Ontario
- 18 member institutions (universities and colleges)
- Provides access to ~20,000 cpu cores, GPUs etc. to >3000 researchers and grad. students
- Access is free and fair.
- To get an account, see the web portal:

www.sharcnet.ca

OpenMP: compilers

Compilers:

Intel (icc, ifort): -openmp : our default compiler

Pathscale (pathcc, pathf90), -openmp

PGI (pgcc, pgf77, pgf90), -mp

Open64 (opencc, openf90), -mp

GNU (gcc, g++, gfortran), -fopenmp

On **requin** only:

SHARCNET compile scripts: cc, CC (or c++), f77, f90

Add **-openmp** to the compile script:

f90 **-openmp** -o hello_openmp hello_openmp.f

OpenMP: simplest example

```
program hello
  write(*,*) "Hello, world!"
end program
```

```
[syam@saw-login1:~] f90 -o hello-seq hello-seq.f90
[syam@saw-login1:~] ./hello-seq
Hello, world!
```

```
program hello
  !$omp parallel
    write(*,*) "Hello, world!"
  !$omp end parallel
end program
```

```
[syam@saw-login1:~] f90 -o hello-par1-seq hello-par1.f90
[syam@saw-login1:~] ./hello-par1-seq
Hello, world!
```

Compiler ignores openmp directive!

</home/syam/ces745/openmp/Fortran/helloworld>

OpenMP: simplest example

```
program hello
  !$omp parallel
    write(*,*) "Hello, world!"
  !$omp end parallel
end program
```

```
[syam@saw-login1:~] f90 -openmp -o hello-par1 hello-par1.f90
```

```
[syam@saw-login1:~] ./hello-par1
```

```
Hello, world!
```

```
Hello, world!
```

```
.....
```

Default number of threads on saw login node is 8 (24 on orca).

OpenMP: simplest example

```
program hello
  write(*,*) "before"
  !$omp parallel
    write(*,*) "Hello, parallel world!"
  !$omp end parallel
  write(*,*) "after"
end program
```

```
[syam@saw-login1:~] f90 -openmp -o hello-par2 hello-par2.f90
```

```
[syam@saw-login1:~] ./hello-par2
```

before

Hello, parallel world!

Hello, parallel world!

.....

after

OpenMP: simplest example

[syam@saw-login1:~] `sqsub -q threaded -n 4 -r 1.0h -o hello-par2.log ./hello-par2`

WARNING: no memory requirement defined; assuming 2GB

submitted as jobid 378196

[syam@saw-login1:~] `sqjobs`

jobid queue state ncpus nodes time command

378196 test Q 4 - 8s ./hello-par3

2688 CPUs total, 2474 busy; 435 jobs running; 1 suspended, 1890 queued.

325 nodes allocated; 11 drain/offline, 336 total.

Job <3910> is submitted to queue <threaded>.

Before

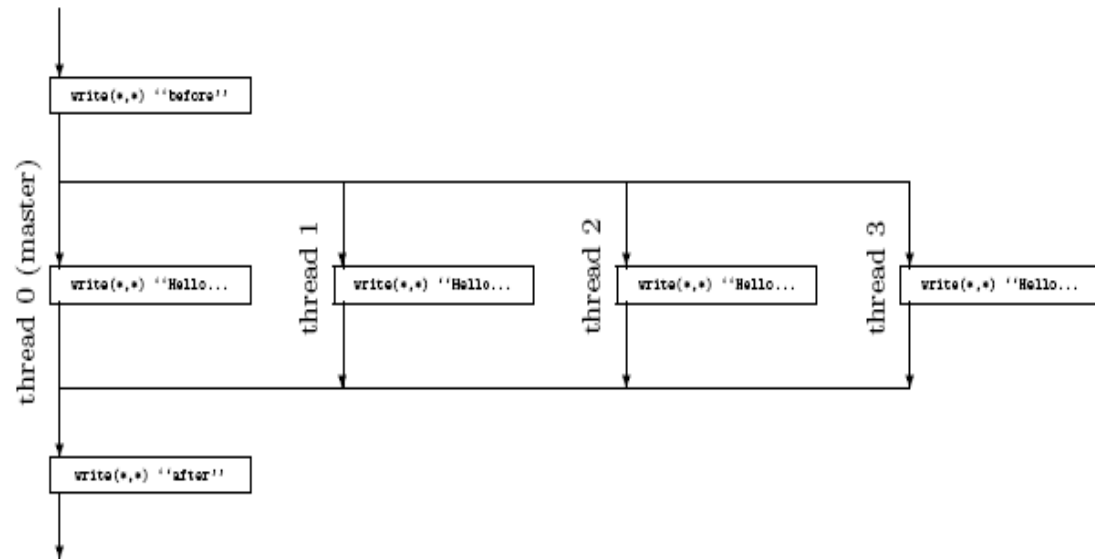
Hello, from thread

Hello, from thread

Hello, from thread

Hello, from thread

after



OpenMP: simplest example

```
program hello
```

```
use omp_lib ←———— Needed when using library routines
```

```
write(*,*) "before"
```

```
!$omp parallel
```

```
    write(*,*) "Hello, from thread ", omp_get_thread_num()
```

```
!$omp end parallel
```

```
write(*,*) "after"
```

```
end program
```

Run-Time Library Routine



```
before
```

```
Hello, from thread 1
```

```
Hello, from thread 0
```

```
Hello, from thread 2
```

```
Hello, from thread 3
```

```
after
```

Example to use OpenMP API to retrieve a thread's id (hello-par3.f90)

OpenMP example-1: hello world in C

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

int main (int argc, char *argv[]) {
    int id, nthreads;
    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        printf("Hello World from thread %d\n", id);
        #pragma omp barrier
        if ( id == 0 ) {
            nthreads = omp_get_num_threads();
            printf("There are %d threads\n",nthreads);
        }
    }
    return 0;
}
```

~syam/ces745/openmp/C/helloworld/helloworld.c – with barrier;
~syam/ces745/openmp/C/helloworld/helloworld2.c – without;

OpenMP example-1: hello world in F77

```
PROGRAM HELLO
INTEGER ID, NTHRDS
INTEGER OMP_GET_THREAD_NUM, OMP_GET_NUM_THREADS
!$OMP PARALLEL PRIVATE(ID)
  ID = OMP_GET_THREAD_NUM()
  PRINT *, 'HELLO WORLD FROM THREAD', ID
!$OMP BARRIER
  IF ( ID .EQ. 0 ) THEN
    NTHRDS = OMP_GET_NUM_THREADS()
    PRINT *, 'THERE ARE', NTHRDS, 'THREADS'
  END IF
!$OMP END PARALLEL
END
```

OpenMP example-1: hello world in F90

```
program hello90
  use omp_lib
  integer :: id, nthreads
  !$omp parallel private(id)
  id = omp_get_thread_num()
  write (*,*) 'Hello World from thread', id
  !$omp barrier
  if ( id .eq. 0 ) then
    nthreads = omp_get_num_threads()
    write (*,*) 'There are', nthreads, 'threads'
  end if
  !$omp end parallel
end program
```


Compile and Run Result

- **Compile**

```
f77 -openmp -o helloworld helloworld.f
```

- **Submit job**

```
sqsub -q threaded -n 4 -r 1.0h -o helloworld.log ./helloworld
```

- **Run Results** (use 4 cpus)

```
HELLO WORLD FROM THREAD 2
```

```
HELLO WORLD FROM THREAD 0
```

```
HELLO WORLD FROM THREAD 3
```

```
HELLO WORLD FROM THREAD 1
```

```
THERE ARE 4 THREADS
```

Exercise

For interactive jobs on a SMP machine, you need to set the maximum number of threads to use during execution.

For example:

```
export OMP_NUM_THREADS=4
```

- Hands on

MPI vs. OpenMP: Hello World

```
#include <stdio.h>
#include "mpi.h"

int main(int argc, char *argv[])
{
    int rank, size;

    MPI_Init(&argc, &argv);    /* starts MPI */
    MPI_Comm_rank(MPI_COMM_WORLD,
    &rank); /* get current process id */
    MPI_Comm_size(MPI_COMM_WORLD,
    &size); /* get number of processes */

    printf("Hello, world! from process %d of
    %d\n", rank, size);

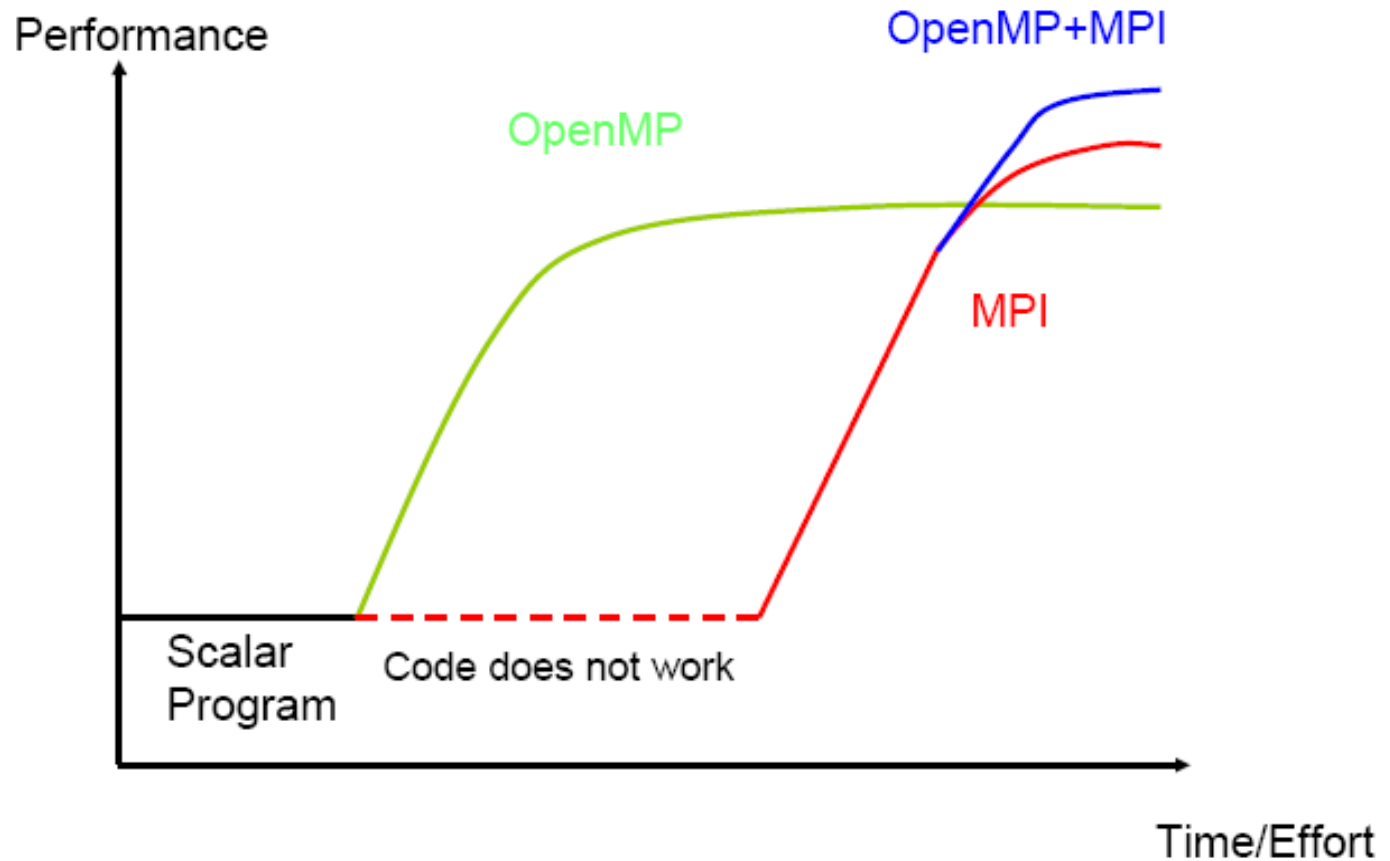
    MPI_Finalize();    /* end of mpi*/

    return(0);
}
```

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

int main (int argc, char *argv[]) {
    int id, nthreads;
    #pragma omp parallel private(id)
    {
        id = omp_get_thread_num();
        printf("Hello World from thread %d\n", id);
        #pragma omp barrier
        if ( id == 0 ) {
            nthreads = omp_get_num_threads();
            printf("There are %d
            threads\n",nthreads);
        }
    }
    return 0;
}
```

Motivation: Why should I use OpenMP?



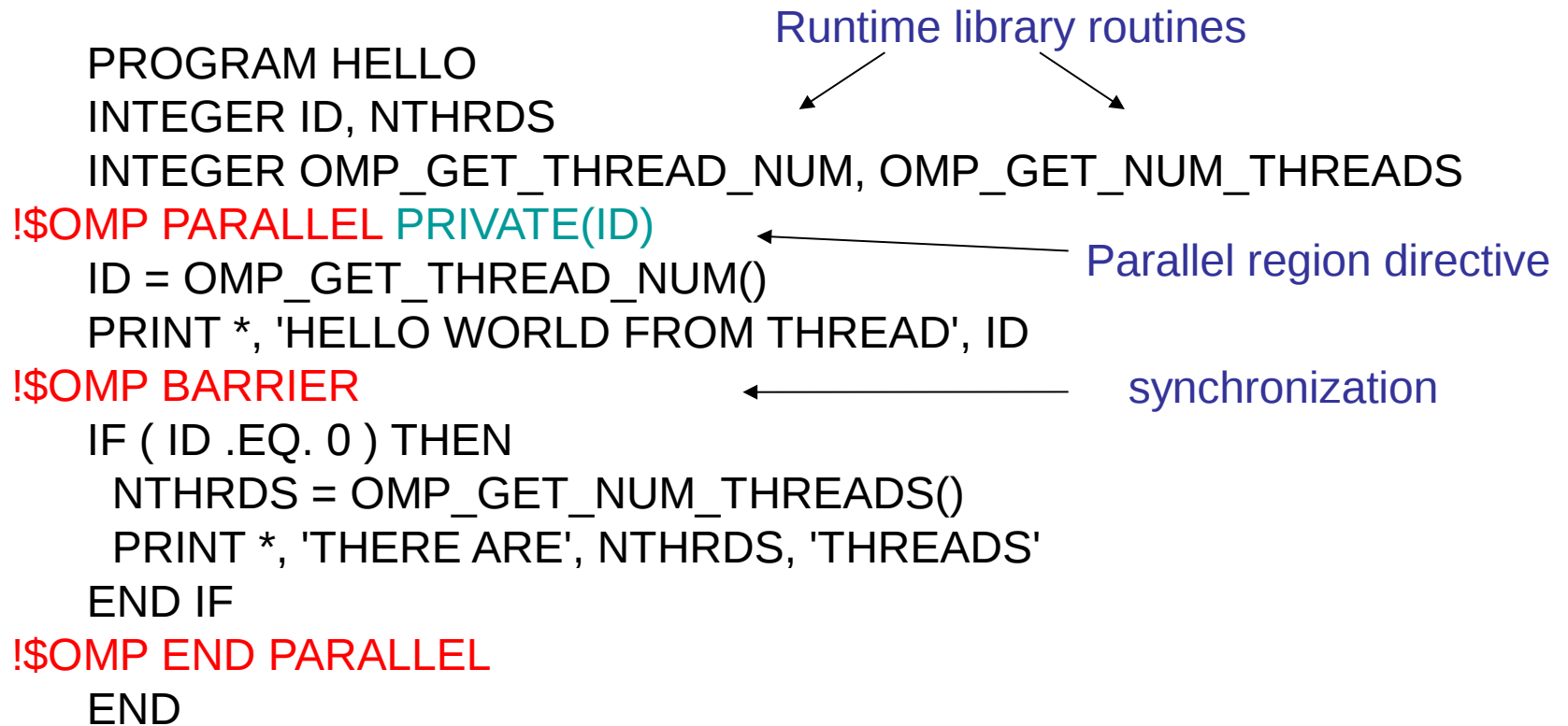
Re-examine OpenMP code:

```
PROGRAM HELLO
INTEGER ID, NTHRDS
INTEGER OMP_GET_THREAD_NUM, OMP_GET_NUM_THREADS
!$OMP PARALLEL PRIVATE(ID)
  ID = OMP_GET_THREAD_NUM()
  PRINT *, 'HELLO WORLD FROM THREAD', ID
!$OMP BARRIER
  IF ( ID .EQ. 0 ) THEN
    NTHRDS = OMP_GET_NUM_THREADS()
    PRINT *, 'THERE ARE', NTHRDS, 'THREADS'
  END IF
!$OMP END PARALLEL
END
```

Runtime library routines

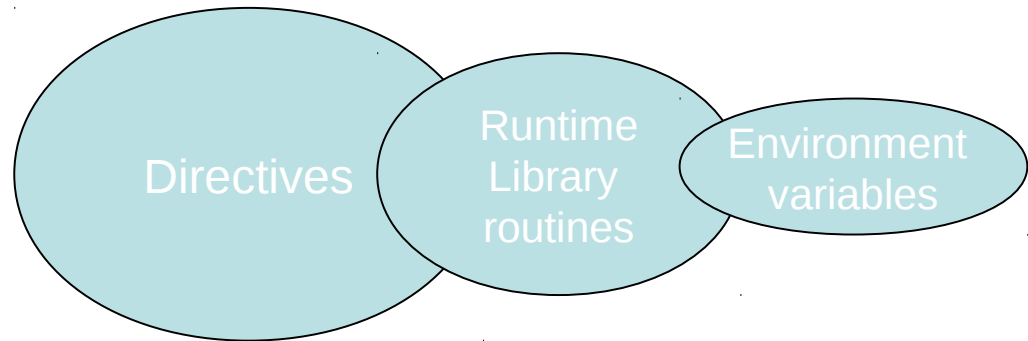
Parallel region directive

synchronization



Data types: private vs. shared

OpenMP Components



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*

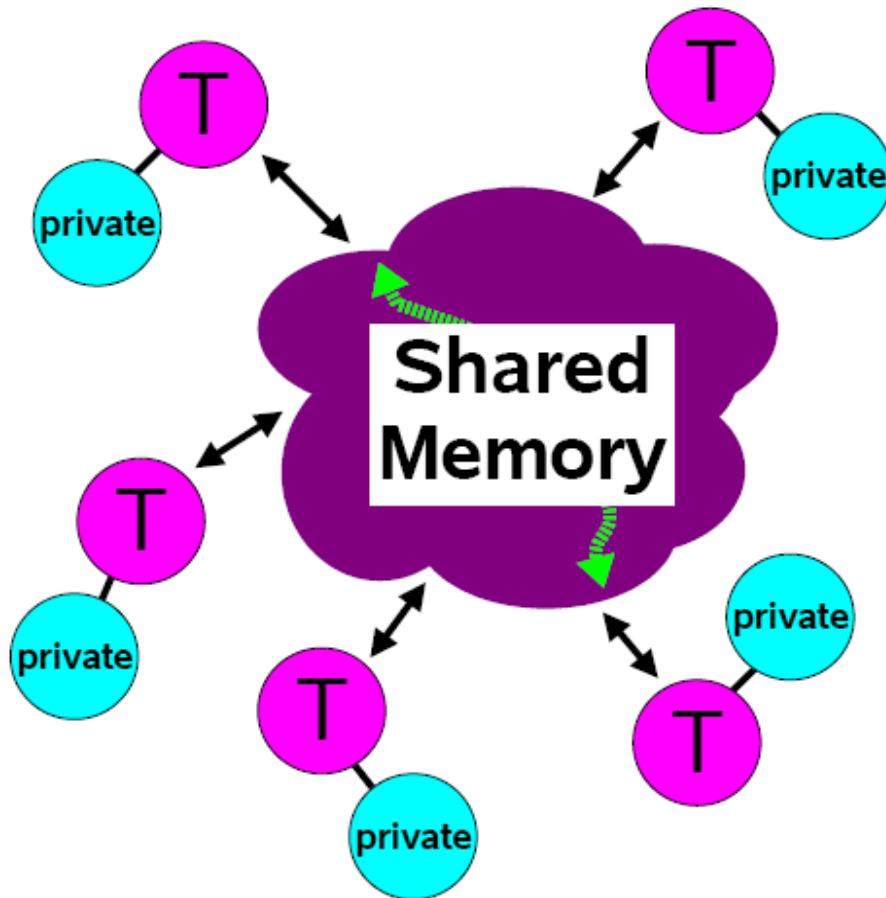
Runtime environment

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



- **Parallel programming: 3 aspects**
 - Specifying parallel execution
 - Communicating between multiple procs/threads
 - Synchronization
- **OpenMP approaches:**
 - Directive-based control structures – expressing parallelism
 - Data environment constructs – communicating
 - Synchronization constructs – synchronization

Shared Memory Model



Programming Model

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

Data (variable) types

◆ *In a shared memory parallel program variables have a "label" attached to them:*

☞ *Labelled "Private" ➡ Visible to one thread only*

✓ *Change made in local data, is not seen by others*

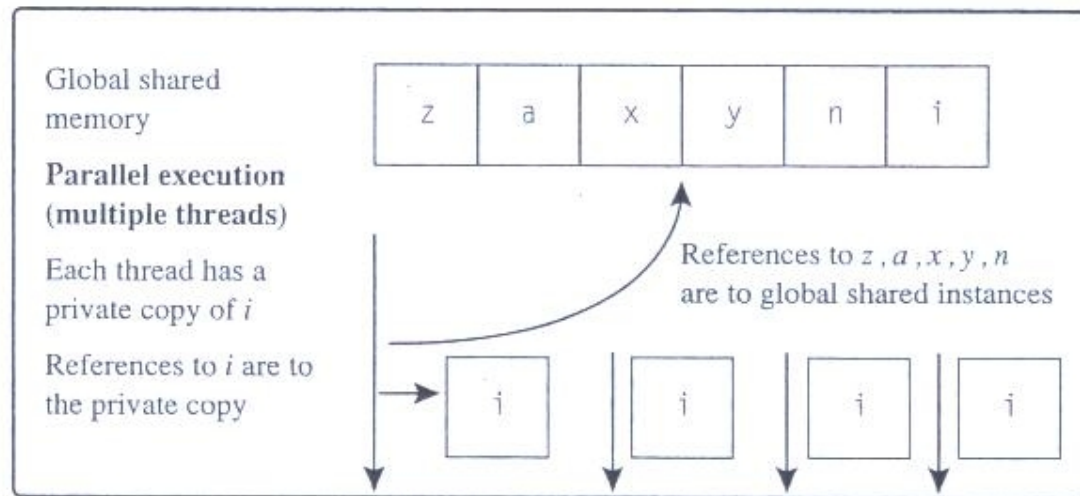
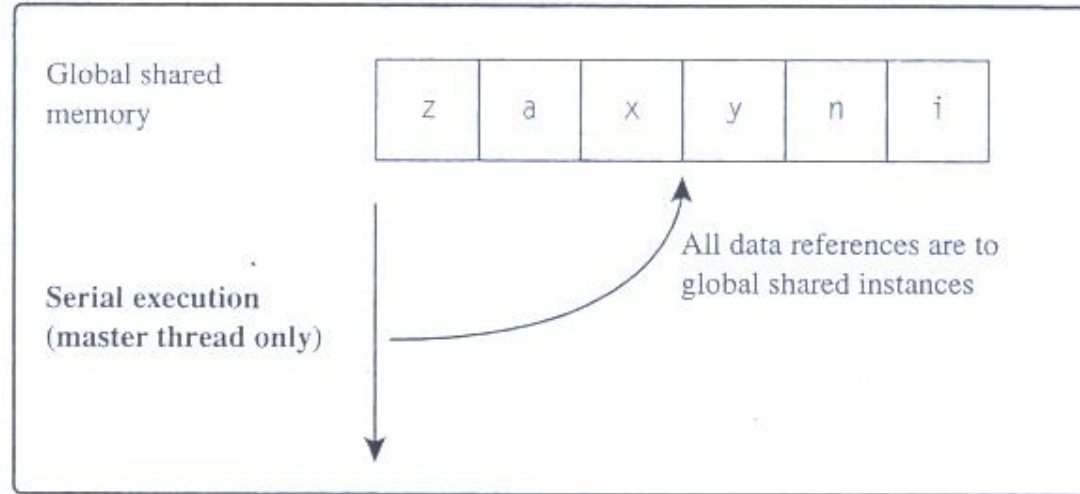
✓ *Example - Local variables in a function that is executed in parallel*

☞ *Labelled "Shared" ➡ Visible to all threads*

✓ *Change made in global data, is seen by all others*

✓ *Example - Global data*

Memory Model: example



The behavior of private variables in an OpenMP program.

OpenMP:

How is OpenMP typically used?

- 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

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

Parallel Program

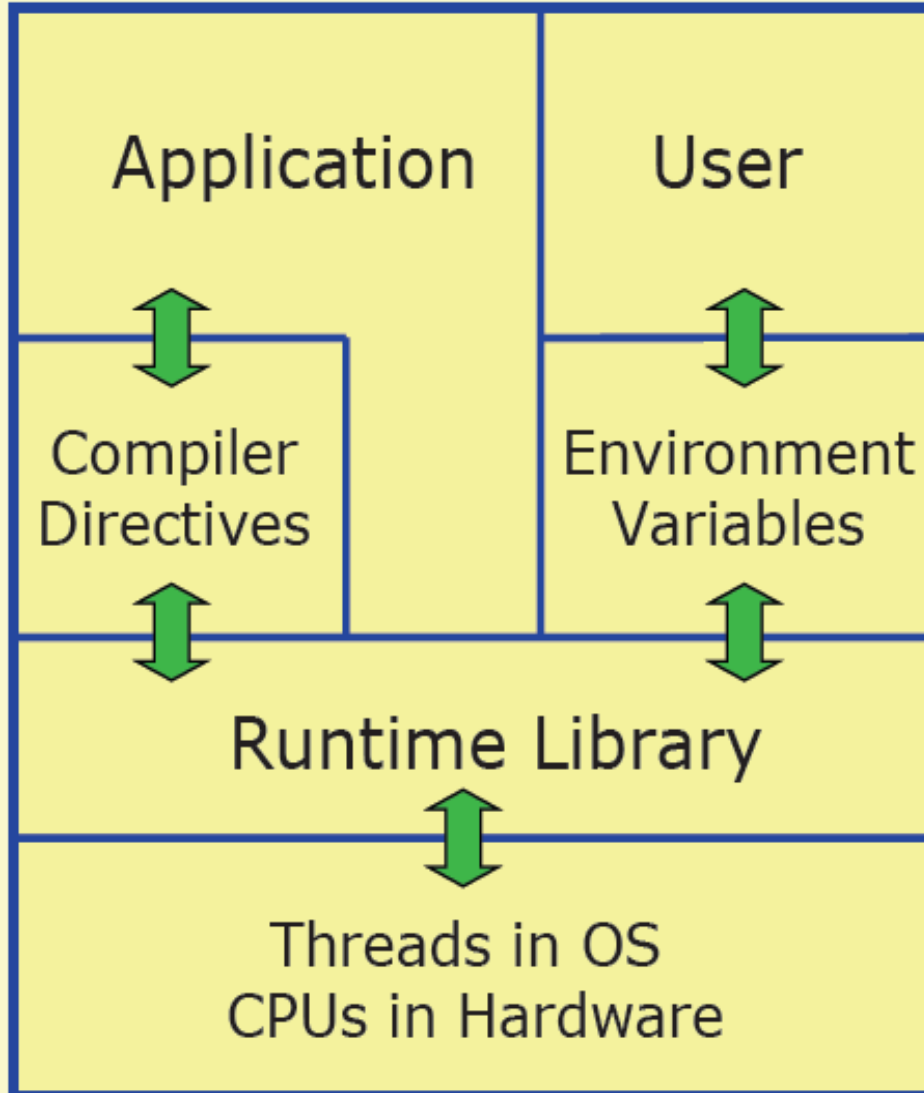


OpenMP:

How do threads interact?

- OpenMP is a 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.
- Synchronization is expensive so:
 - Change how data is stored to minimize the need for synchronization.

OpenMP Summary: OS and User perspective



- **OS View:**
 - parallel work done by **threads**
- **Programmer's View:**
 - **Directives** (comment lines)
 - Library Routines
- **User's View**
 - Environment Variables (Resources, Scheduling)

OpenMP Constructs

Basic Directive Formats

Fortran: directives come in pairs, The "end" directive is optional in some directives (DO,...) but advised for readability

!\$OMP directive *[clause, ...]*

[structured block of code]

!\$OMP end directive

C/C++: case sensitive

#pragma omp directive *[clause,...] newline*

[structured block of code]

OpenMP:

Structured blocks

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

```
C$OMP PARALLEL
10   wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id))) goto 10
C$OMP END PARALLEL
      print *,id
```

A structured block

```
C$OMP PARALLEL
10   wrk(id) = garbage(id)
30   res(id)=wrk(id)**2
      if(conv(res(id)))goto 20
      go to 10
C$OMP END PARALLEL
      if(not_DONE) goto 30
20   print *, id
```

Not A structured block

OpenMP: Contents

- **OpenMP's constructs fall into 5 categories:**
 - ◆ **Parallel Regions**
 - ◆ **Worksharing**
 - ◆ **Data Environment**
 - ◆ **Synchronization**
 - ◆ **Runtime functions/environment variables**
- **OpenMP is basically the same between Fortran and C/C++**

OpenMP: Parallel Regions

- You create threads in OpenMP with the “omp parallel” pragma.
- For example, To create a 4 thread Parallel region:

Each thread
redundantly
executes
the code
within the
structured
block

```
double A[1000];  
omp_set_num_threads(4);  
#pragma omp parallel  
{  
    int ID = omp_get_thread_num();  
    pooh(ID,A);  
}
```

- Each thread calls `pooh(ID)` for `ID = 0 to 3`

PARALLEL Region Construct: Summary

- A parallel region is a block of code that will be executed by multiple threads. This is the fundamental OpenMP parallel construct.
- A parallel region must be a structured block
- It may contain any of the following clauses:

Fortran

```
!$OMP PARALLEL [clause ...]  
    IF (scalar_logical_expression)  
    PRIVATE (list)  
    SHARED (list)  
    DEFAULT (PRIVATE | SHARED | NONE)  
    FIRSTPRIVATE (list)  
    REDUCTION (operator: list)  
    COPYIN (list)  
    structured_block  
!$OMP END PARALLEL
```

C/C++

```
#pragma omp parallel [clause ...] newline  
    if (scalar_expression)  
    private (list)  
    shared (list)  
    default (shared | none)  
    firstprivate (list)  
    reduction (operator: list)  
    copyin (list)  
    structured_block
```

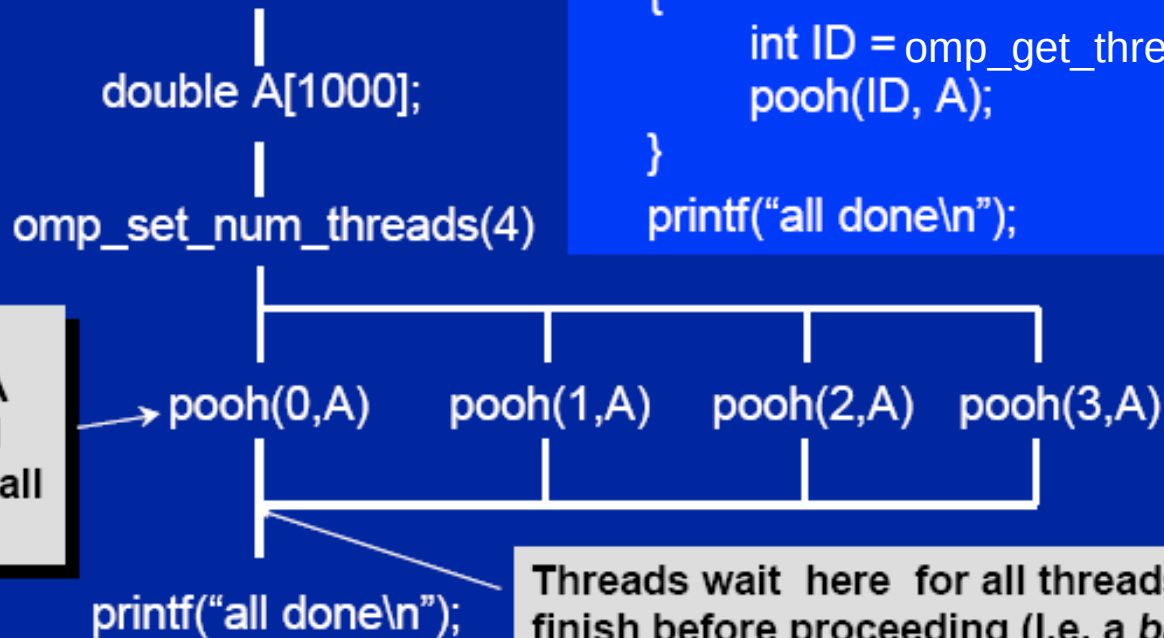
PARALLEL Region Construct: Notes

- When a thread reaches a PARALLEL directive, it creates a team of threads and becomes the master of the team. The master is a member of that team and has thread number 0 within that team.
- Starting from the beginning of this parallel region, the code is duplicated and all threads will execute that code.
- There is an implied barrier at the end of a parallel section. Only the master thread continues execution past this point.
- If any thread terminates within a parallel region, all threads in the team will terminate, and the work done up until that point is undefined.

OpenMP: Parallel Regions

- Each thread executes the same code redundantly.

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



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (i.e. a *barrier*)

Fortran - Parallel Region Example

```
PROGRAM HELLO  
  INTEGER NTHREADS, TID, OMP_GET_NUM_THREADS,  
+ OMP_GET_THREAD_NUM
```

C Fork a team of threads giving them their own copies of variables
!\$OMP PARALLEL PRIVATE(TID)

C Obtain and print thread id
TID = **OMP_GET_THREAD_NUM()**
PRINT *, 'Hello World from thread = ', TID

C Only master thread does this
IF (TID .EQ. 0) THEN
 NTHREADS = **OMP_GET_NUM_THREADS()**
 PRINT *, 'Number of threads = ', NTHREADS
END IF

C All threads join master thread and disband
!\$OMP END PARALLEL

END

- Every thread executes all code enclosed in the parallel section
- OpenMP library routines are used to obtain thread identifiers and total number of threads



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 */
```

```
}
```

- Clauses involved:
private

PARALLEL Region Construct: How Many Threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
 - Use of the **omp_set_num_threads()** library function
 - Setting of the **OMP_NUM_THREADS** environment variable
 - Implementation default - usually the number of CPU cores on a node, though it could be dynamic.
- Threads are numbered from 0 (master thread) to N-1

PARALLEL Region Construct: Dynamic Threads

- Use the **omp_get_dynamic()** library function to determine if dynamic threads are enabled.
- If supported, the two methods available for enabling dynamic threads are:
 - The **omp_set_dynamic()** library routine
 - Setting of the **OMP_DYNAMIC** environment variable to TRUE

PARALLEL Region Construct: Nested Parallel Regions

- Use the **omp_get_nested()** library function to determine if nested threads are enabled.
- The two methods available for enabling nested parallel regions (if supported) are:
 1. The **omp_set_nested()** library routine
 2. Setting of the **OMP_NESTED** environment variable to TRUE
- If not supported, a parallel region nested within another parallel region results in the creation of a new team, consisting of one thread, by default.

PARALLEL Region Construct: Clauses and Restrictions

- **IF** clause: If present, it must evaluate to .TRUE. (Fortran) or non-zero (C/C++) in order for a team of threads to be created. Otherwise, the region is executed serially by the master thread.
- It is illegal to branch into or out of a parallel region
- Only a single IF clause is permitted

```
!$omp parallel do if (n .ge. 800)
```

```
    do i = 1, n
```

```
        z(i) = a*x(i) + y
```

```
    enddo
```

if takes a Boolean expression as an argument. If 'True', the loop is run parallel, if 'False', the loop is executed serially, to avoid overhead

Example: Matrix-Vector Multiplication

$$A[n,n] \times B[n] = C[n]$$

```
for (i=0; i < SIZE; i++)  
{  
    for (j=0; j < SIZE; j++)  
        c[i] += (A[i][j] * b[j]);  
}
```

Can we simply add one parallel directive?

```
#pragma omp parallel  
for (i=0; i < SIZE; i++)  
{  
    for (j=0; j < SIZE; j++)  
        c[i] += (A[i][j] * b[j]);  
}
```

Matrix-Vector Multiplication: parallel region

```
/* Create a team of threads and scope variables */
#pragma omp parallel shared(A,b,c,total) private(tid,i,j,istart,iend)
{
    tid = omp_get_thread_num();
    nid = omp_get_num_threads();

    istart = tid*SIZE/nid;
    iend = (tid+1)*SIZE/nid;

    for (i=istart; i < iend; i++)
    {
        for (j=0; j < SIZE; j++)
            c[i] += (A[i][j] * b[j]);

        /* Update and display of running total must be serialized */
        #pragma omp critical
        {
            total = total + c[i];
            printf(" thread %d did row %d\t c[%d]=%.2f\t",tid,i,i,c[i]);
            printf("Running total= %.2f\n",total);
        }

    } /* end of parallel i loop */

} /* end of parallel construct */
```



Matrix-Vector Multiplication: parallel region Result

```
[syam@wha780 matrix]$ ./matrix-vector-parregion
```

Starting values of matrix A and vector b:

```
A[0]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[0]= 1.0
A[1]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[1]= 2.0
A[2]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[2]= 3.0
A[3]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[3]= 4.0
A[4]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[4]= 5.0
A[5]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[5]= 6.0
A[6]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[6]= 7.0
A[7]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[7]= 8.0
A[8]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[8]= 9.0
A[9]= 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0 10.0  b[9]= 10.0
```

Results by thread/row:

```
thread 0 did row 0  c[0]=55.00  Running total= 55.00
thread 1 did row 5  c[5]=330.00  Running total= 385.00
thread 1 did row 6  c[6]=385.00  Running total= 770.00
thread 1 did row 7  c[7]=440.00  Running total= 1210.00
thread 1 did row 8  c[8]=495.00  Running total= 1705.00
thread 1 did row 9  c[9]=550.00  Running total= 2255.00
thread 0 did row 1  c[1]=110.00  Running total= 2365.00
thread 0 did row 2  c[2]=165.00  Running total= 2530.00
thread 0 did row 3  c[3]=220.00  Running total= 2750.00
thread 0 did row 4  c[4]=275.00  Running total= 3025.00
```

Matrix-vector total - sum of all c[] = 3025.00

OpenMP: Some subtle details (don't worry about these at first)

- **Dynamic mode**

- The number of threads used in a parallel region can vary from one parallel region to another.
- Setting the number of threads only sets the maximum number of threads - you could get less.

- **Static mode (the default mode):**

- The number of threads is fixed and controlled by the programmer.

- **OpenMP lets you nest parallel regions, but...**

- A compiler can choose to *serialize* the nested parallel region (i.e. use a team with only one thread).