Parallel Programming with OpenMP

Jemmy Hu
SHARCNET
University of Waterloo

Contents

Parallel Computing – programming model

- **Distributed memory systems**
  - For processors to share data, the programmer must explicitly arrange for communication -**“Message Passing”**
  - Message passing libraries:
    - MPI (“Message Passing Interface”)
    - PVM (“Parallel Virtual Machine”)

- **Shared memory systems**
  - “Thread” based programming (pthread, …)
  - Compiler directives (OpenMP)
  - Can also do explicit message passing, of course

What is Shared Memory Parallelization?

- All processors can access all the memory in the parallel system (one address space).

- The time to access the memory may not be equal for all processors
  – not necessarily a flat memory

- Parallelizing on a SMP does not reduce CPU time
  – it reduces wallclock time

- Parallel execution is achieved by generating multiple threads which execute in parallel

- Number of threads (in principle) is independent of the number of processors

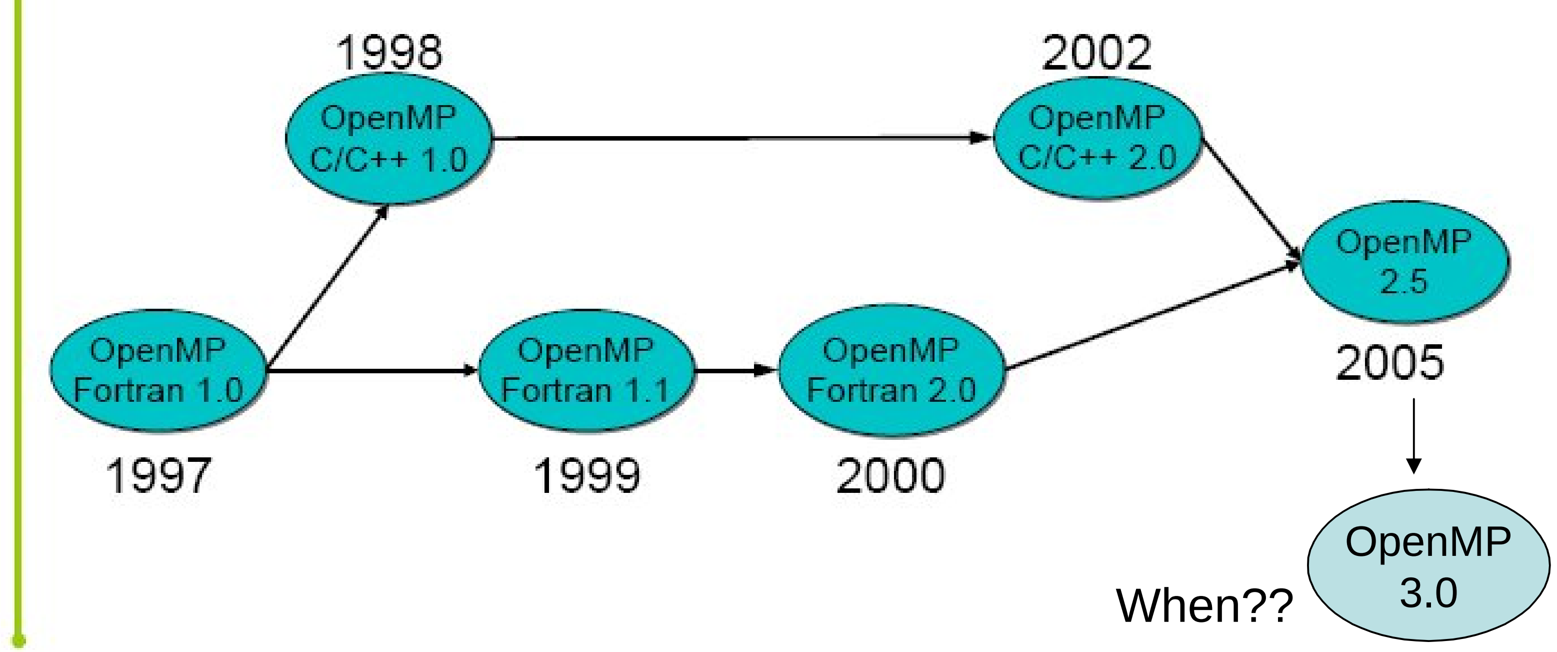
OpenMP Concepts

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 NT
- **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 Release History



OpenMP Application Program Interface: Version 2.5 May 2005
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

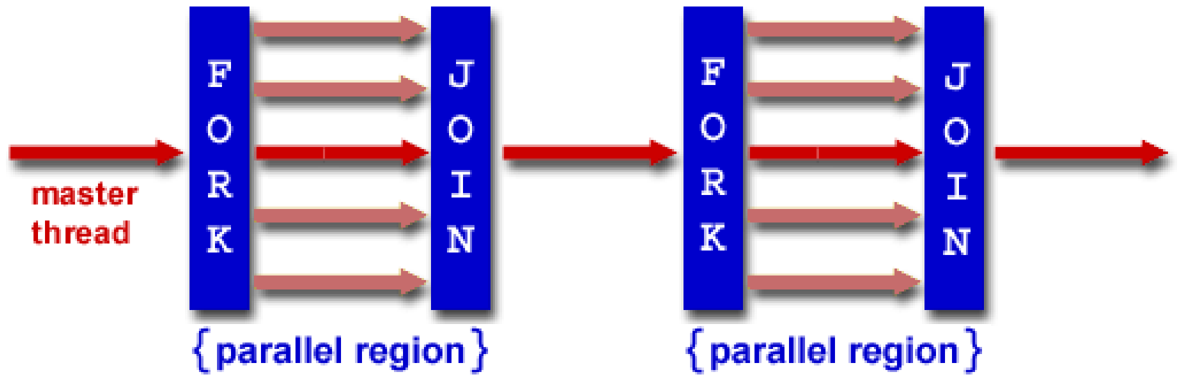
  Provide the capability to implement both coarse-grain and fine-grain parallelism
- **Portability:**

  Supports Fortran (77, 90, and 95), C, and C++

  Public forum for API and membership

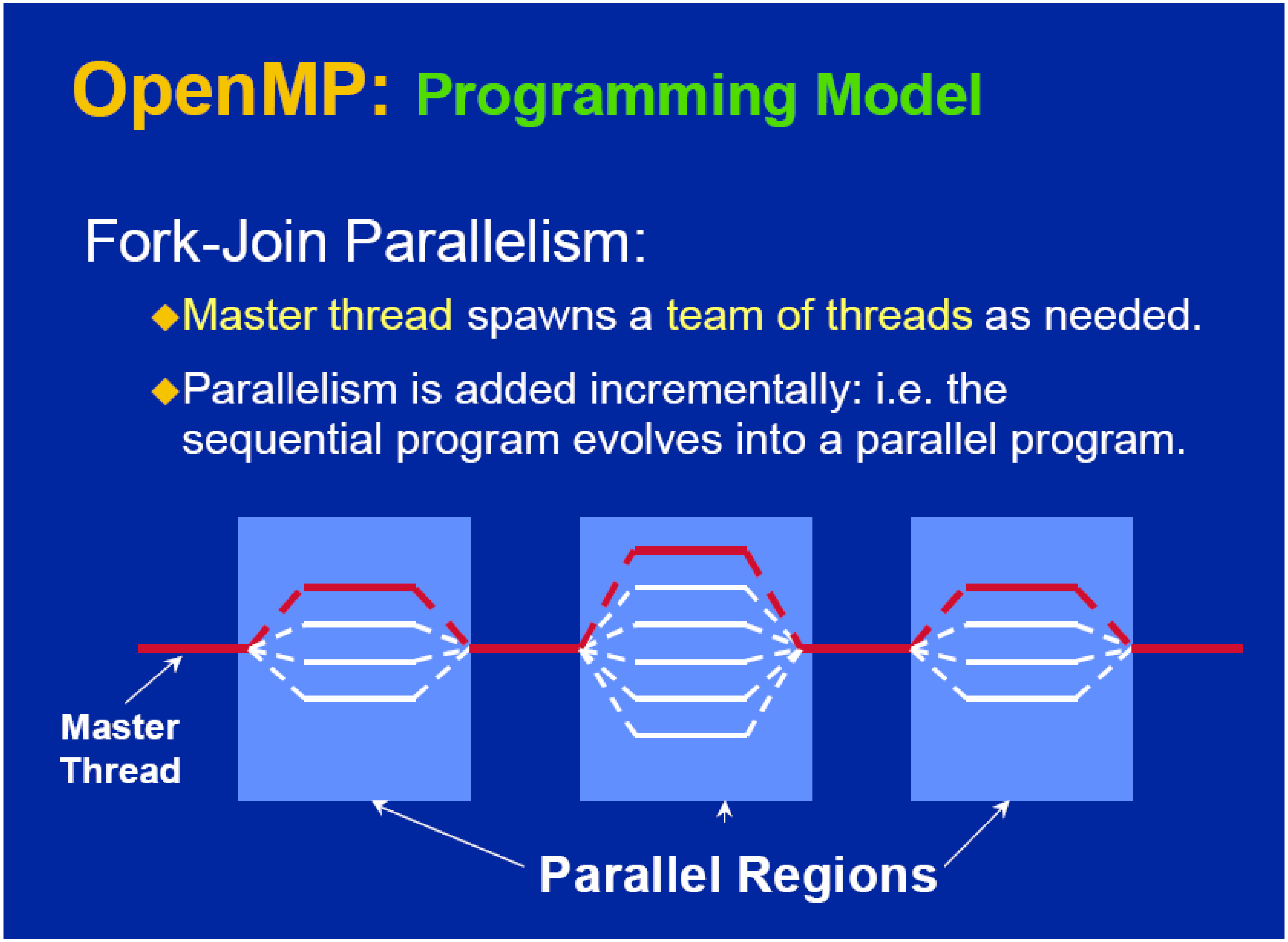
OpenMP: Fork-Join Model

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



FORK: the master thread then 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

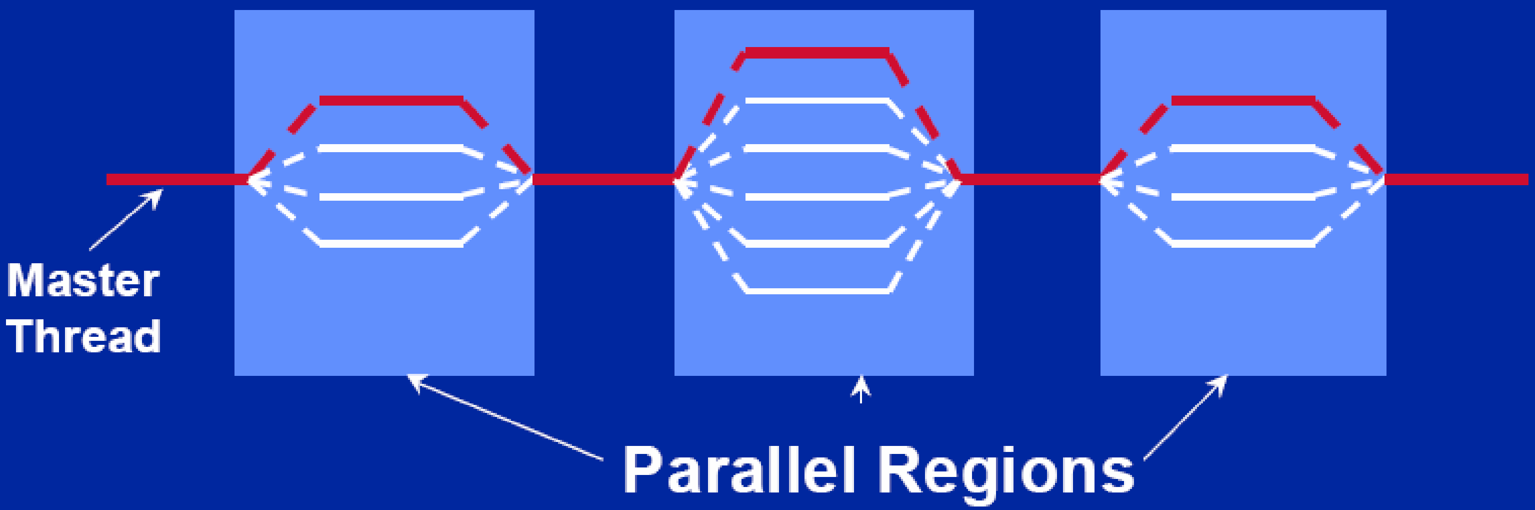
JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread



OpenMP: Programming Model

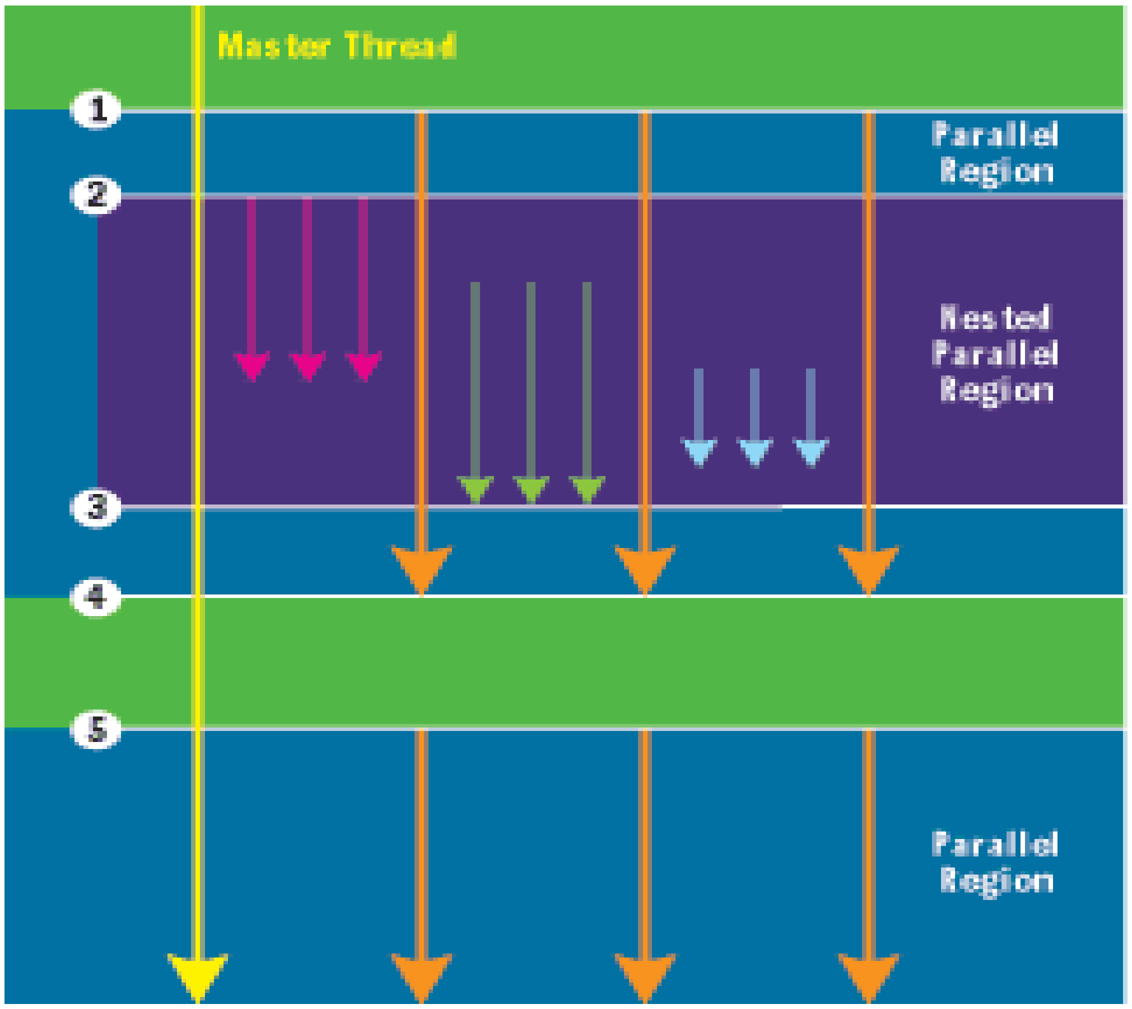
Fork-Join Parallelism:

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



Dynamic threading

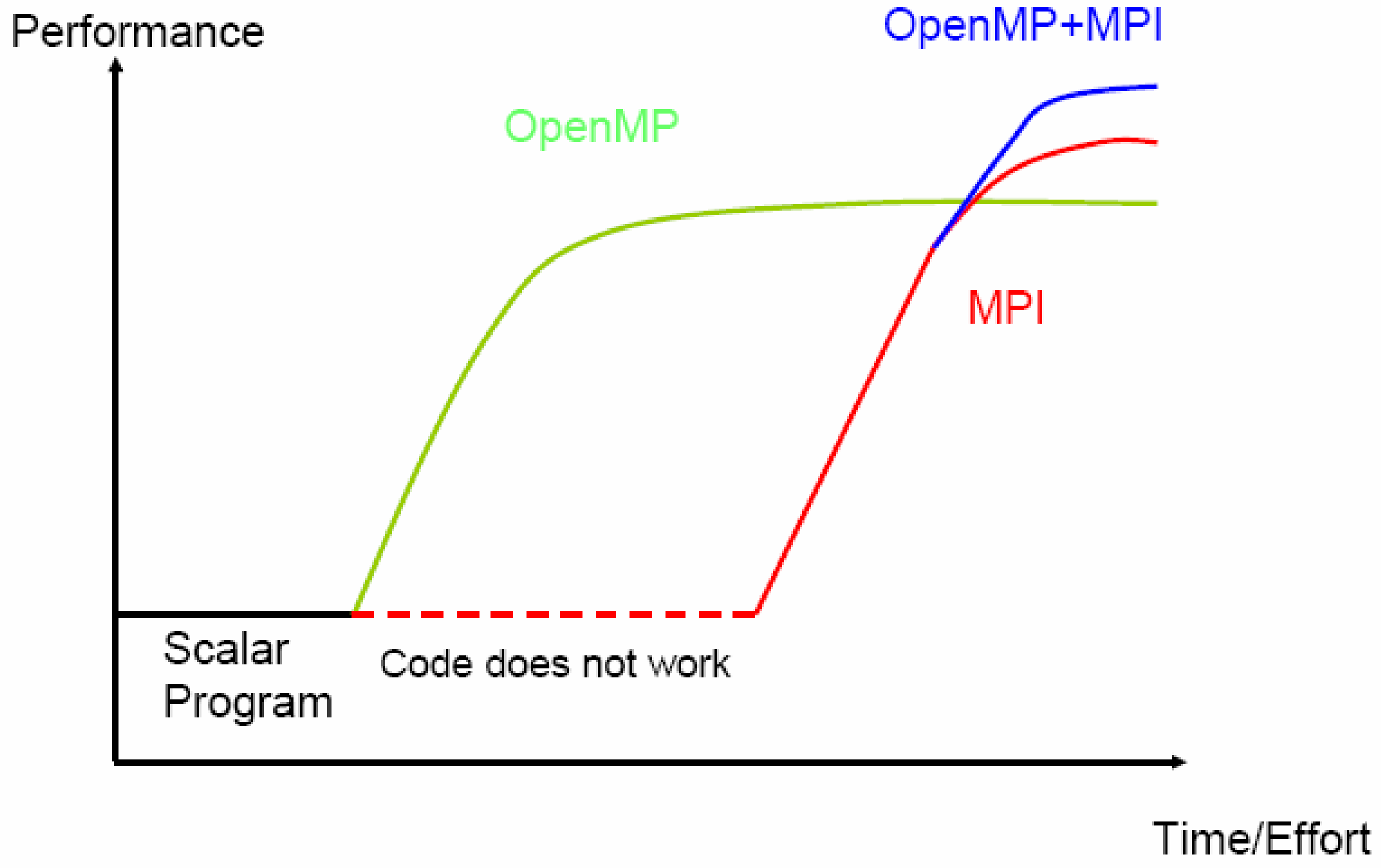
OpenMP excution model (nested parallel)



OpenMP Execution Model Description

- Fork-join model of parallel execution
- Begin execution as a single process (master thread)
- Start of a parallel construct:
  Master thread creates team of threads
- Completion of a parallel construct:
  - Threads in the team synchronize:
    - implicit barrier
- Only master thread continues execution

Motivation: Why should I use OpenMP?



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.
- 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	**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 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:

• 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 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
```

OpenMP: compiler

- **Compiler flags:**

Intel (icc, ifort)	-openmp
Pathscale (cc, c++, f77, f90)	-openmp
PGI (pgcc, pgf77, pgf90)	-mp

```
f90 –openmp –o hello_openmp hello_openmp.f
```

OpenMP: simplest example

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

```
[jemmyhu@wha780 helloworld]$ f90 -o hello-seq hello-seq.f90
[jemmyhu@wha780 helloworld]$ ./hello-seq
Hello, world!
[jemmyhu@wha780 helloworld]$
```

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

```
[jemmyhu@wha780 helloworld]$ f90 -o hello-par1-seq hello-par1.f90
[jemmyhu@wha780 helloworld]$ ./hello-par1-seq
Hello, world!
[jemmyhu@wha780 helloworld]$
```

Compiler ignore openmp directive; parallel region concept

OpenMP: simplest example

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

```
[jemmyhu@wha780 helloworld]$ f90 -openmp -o hello-par1 hello-par1.f90
[jemmyhu@wha780 helloworld]$ ./hello-par1
Hello, world!
Hello, world!
[jemmyhu@wha780 helloworld]$
```

Default threads on whale login node is 2, it may vary from system to system

OpenMP: simplest example

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

```
[jemmyhu@wha780 helloworld]$ f90 -openmp -o hello-par2 hello-par2.f90
[jemmyhu@wha780 helloworld]$ ./hello-par2
before
Hello, parallel world!
Hello, parallel world!
after
[jemmyhu@wha780 helloworld]$
```

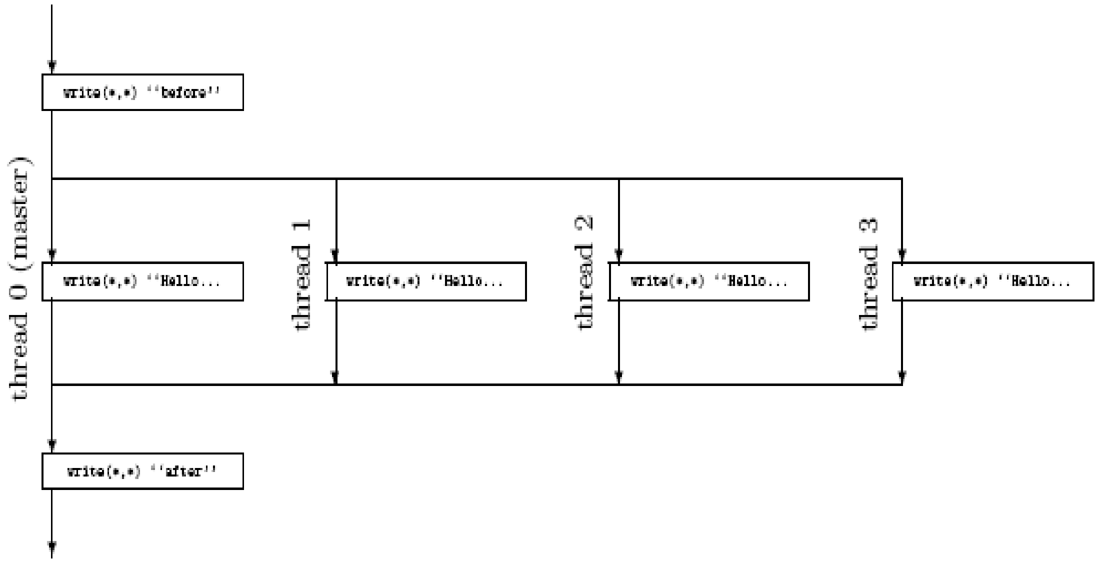
OpenMP: simplest example

```
[jemmyhu@meg34 helloworld]$ sqsub -q threaded -n 4 -o hello-par2.log ./hello-par2
Job <3910> is submitted to queue <threaded>.
[jemmyhu@meg34 helloworld]$ sqjobs
jobid   queue state ncpus nodes time command
----- -------- ----- ----- ----- ---- ------------
 3910 threaded    Q     4        5s ./hello-par2
128 CPUs total, 94 idle, 34 busy; 4 jobs running; 0 suspended, 1 queued.
[jemmyhu@meg34 helloworld]$
```

```
Before

Hello, parallel world!
Hello, parallel world!
Hello, parallel world!
Hello, parallel world!

after
```



OpenMP: simplest example

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

```
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

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;
}
```

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**

```
f90 –openmp  –o  hello_openmp_f  hello_world_openmp.f
```

- **Submit job**

```
sqsub –q threaded –n 4 –o hello_openmp.log ./hello_openmp_f
```

- **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
```

Re-examine OpenMP code:

Runtime library routines

```
      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
```

Parallel region directive

synchronization

Data types: private vs. shared

OpenMP: 3 categories

- **Parallel programming: 3 aspects**
  - specifying parallel execution
  - communicating between multiple procs/threads
  - Synchronization

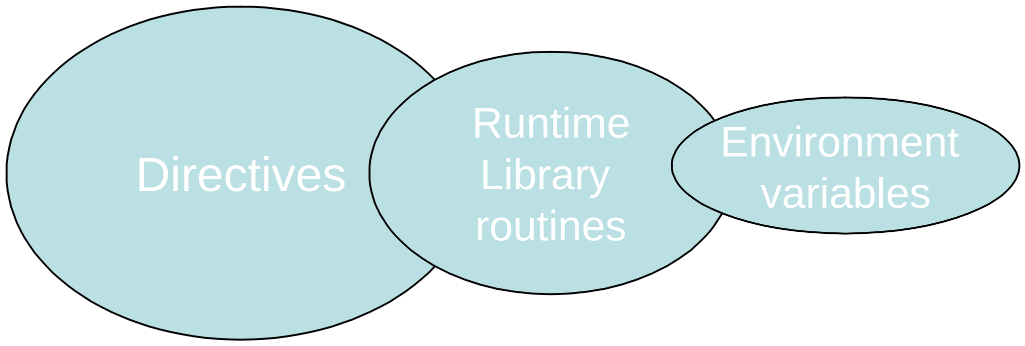
- **OpenMP approaches:**

  Directive-based control structures – expressing parallelism

  Data environment constructs – communicating

  Synchronization constructs – synchronization

Components of OpenMP



Directives	*Environment variables*	*Runtime environment*
◆ *Parallel regions*	◆ *Number of threads*	◆ *Number of threads*
◆ *Work sharing*	◆ *Scheduling type*	◆ *Thread ID*
◆ *Synchronization*	◆ *Dynamic thread adjustment*	◆ *Dynamic thread adjustment*
◆ *Data scope attributes*	◆ *Nested parallelism*	◆ *Nested parallelism*
☞ *private*		◆ *Timers*
☞ *firstprivate*		◆ *API for locking*
☞ *lastprivate*		
☞ *shared*		
☞ *reduction*		
◆ *Orphaning*		

OpenMP Directives

Basic Directive Formats

Fortran: directives come in pairs, The "end" directive is optional 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’s constructs fall into 5 categories:

- Parallel Regions
- Worksharing Constructs
- Data Environment
- Synchronization
- Runtime functions/environment variables

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

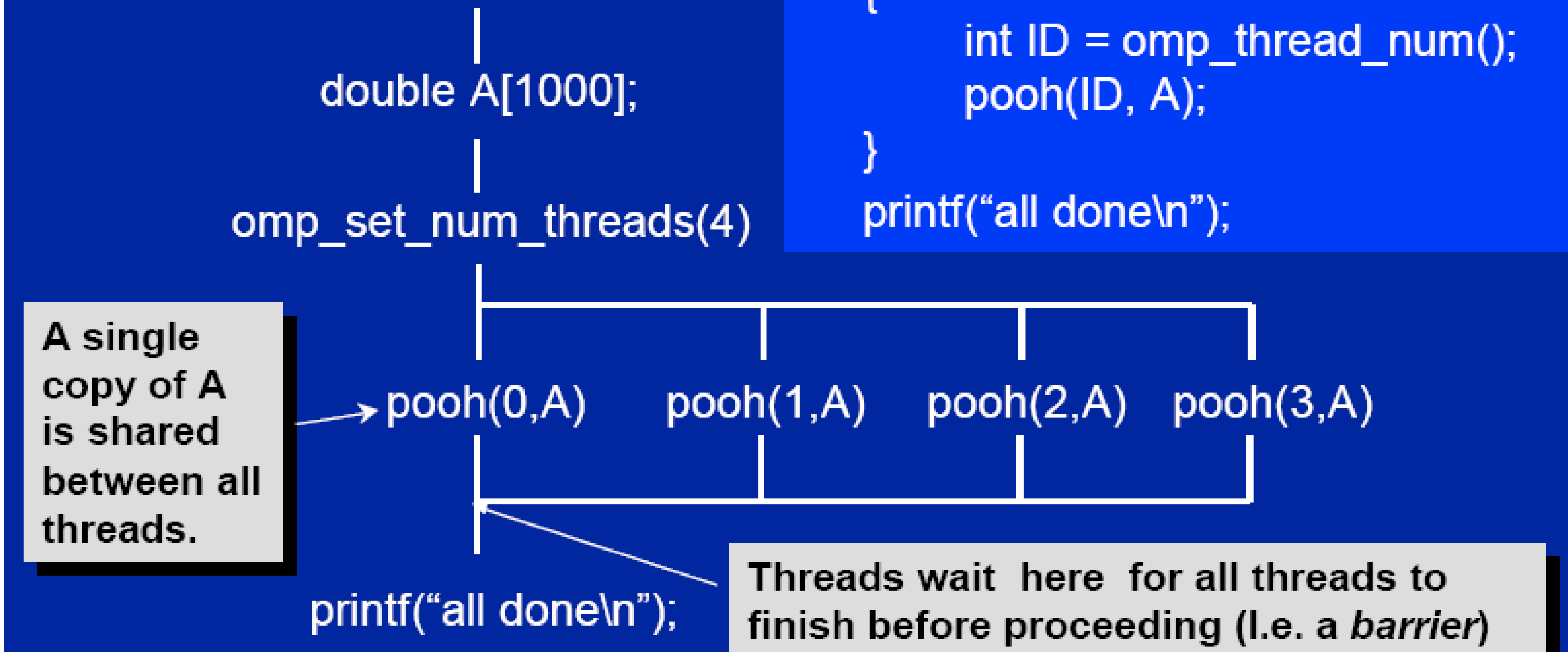
PARALLEL Region Construct: Notes

- 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_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 CPUs on a node, though it could be dynamic (see next bullet).

- Threads are numbered from 0 (master thread) to N-1

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.
- A parallel region must be a structured block that does not span multiple routines or code files
- 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 excuted 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[i]);
}
```

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[i]);
}
```

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[i]);

   /* 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 */
```

OpenMP: Work-sharing constructs:

- A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it.
- Work-sharing constructs do not launch new threads
- There is no implied barrier upon entry to a work-sharing construct, however there is an implied barrier at the end of a work sharing construct.

A motivating example

Sequential code

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

OpenMP parallel region

```
#pragma omp parallel
{
            int id, i, Nthrds, istart, iend;
            id = omp_get_thread_num();
            Nthrds = omp_get_num_threads();
            istart = id * N / Nthrds;
            iend = (id+1) * N / Nthrds;
            for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

```
#pragma omp parallel
#pragma omp for schedule(static)
            for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

DO/for Format

Fortran	**!$OMP DO** ***[clause ...]***
	    SCHEDULE ***(type [,chunk])***
	    ORDERED PRIVATE ***(list)***
	    FIRSTPRIVATE ***(list)***
	    LASTPRIVATE ***(list)***
	    SHARED ***(list)***
	    REDUCTION ***(operator | intrinsic : list)***
	  do_loop
	!$OMP END DO [NOWAIT]

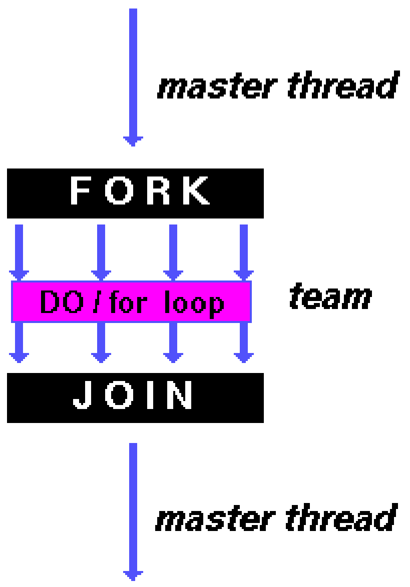
C/C++	**#pragma omp for** ***[clause ...] newline***
	    schedule ***(type [,chunk])***
	    ordered private ***(list)***
	    firstprivate ***(list)***
	    lastprivate ***(list)***
	    shared ***(list)***
	    reduction ***(operator: list)***
	    nowait
	for_loop

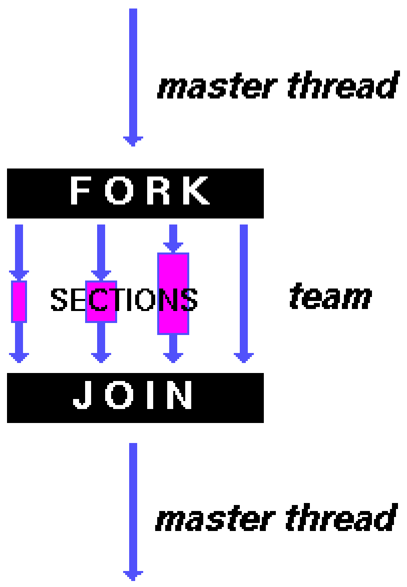
Types of Work-Sharing Constructs:

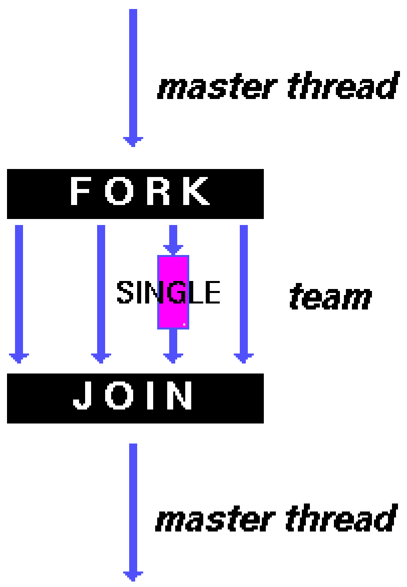
DO / for - shares iterations of a loop across the team. Represents a type of "data parallelism".

SECTIONS - breaks work into separate, discrete sections. Each section is executed by a thread. Can be used to implement a type of "functional parallelism".

SINGLE - serializes a section of code







OpenMP Work-sharing constructs: Notes

- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- Work-sharing constructs must be encountered by all members of a team or none at all
- Successive work-sharing constructs must be encountered in the same order by all members of a team

Work-sharing constructs: Loop construct

- The DO / for directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team. This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.

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

```
!$omp parallel
!$omp do
            do-loop
!$omp end do
!$omp parallel
```

Simple examples: serial do-loop code

```
program loop
  implicit none
  integer, parameter :: N = 60000000
  integer :: i
  real :: x(N)

  do i = 1, N
     x(i) = 1./real(i)
  end do

end program
```

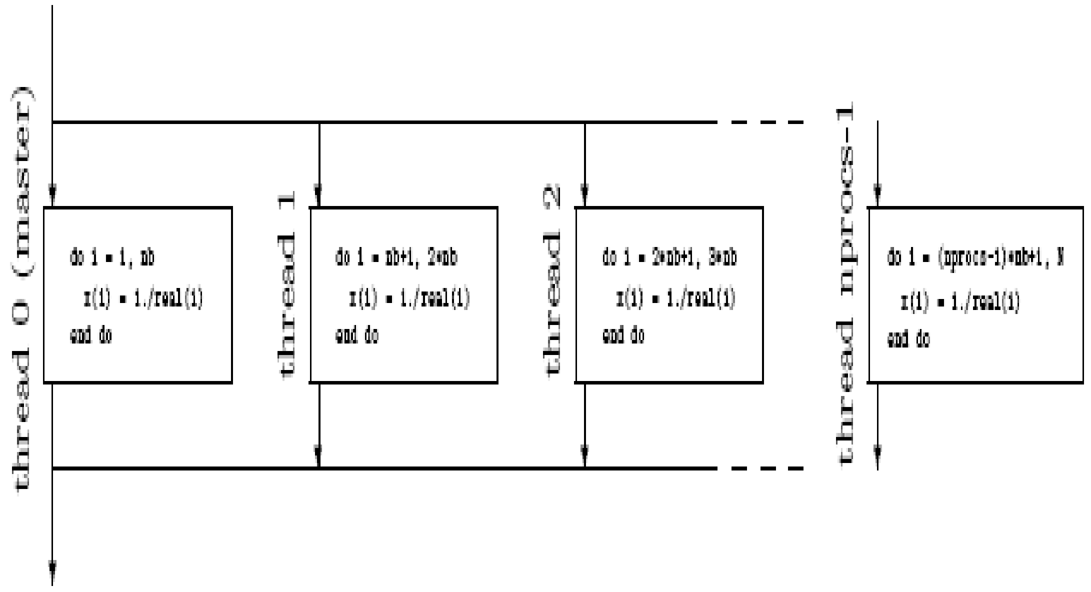
parallel do-loop

```
program loop
  implicit none
  integer, parameter :: N = 60000000
  integer :: i
  integer :: nprocs, myid, nb, istart, iend
  real :: x(N)

  !$omp parallel private(myid,istart,iend)
     nprocs = omp_get_num_threads()
     myid = omp_get_thread_num()
     nb = N/nprocs
     istart = myid*nb + 1
     if (myid /= nprocs-1) then
         iend = (myid + 1)*nb
     else
         iend = N
     end if
     do i = istart, iend
        x(i) = 1./real(i)
     end do
  !$omp end parallel

end program
```

one possible parallel version of the preceding code.
(distribute the loop to different threads by hard coding)



```
do i = 1, nb
  x(i) = 1./real(i)
end do
```

```
do i = nb+1, 2*nb
  x(i) = 1./real(i)
end do
```

```
do i = 2*nb+1, 3*nb
  x(i) = 1./real(i)
end do
```

```
do i = (nprocs-1)*nb+1, N
  x(i) = 1./real(i)
end do
```

Do directive

Instead of hard-coding, we can use OpenMP provides task sharing directives (section) to achieve the same goal.

```
program loop
  implicit none
  integer, parameter :: N = 60000000
  integer :: i
  real :: x(N)

  !$omp parallel
    !$omp do
      do i = 1, N
         x(i) = 1./real(i)
      end do
    !$omp end do
  !$omp end parallel

end program
```

Parallel do: Combined Directives

```
program loop
  implicit none
  integer, parameter :: N = 60000000
  integer :: i
  real :: x(N)

  !$omp parallel do
     do i = 1, N
        x(i) = 1./real(i)
     end do
  !$omp end parallel do

end program
```

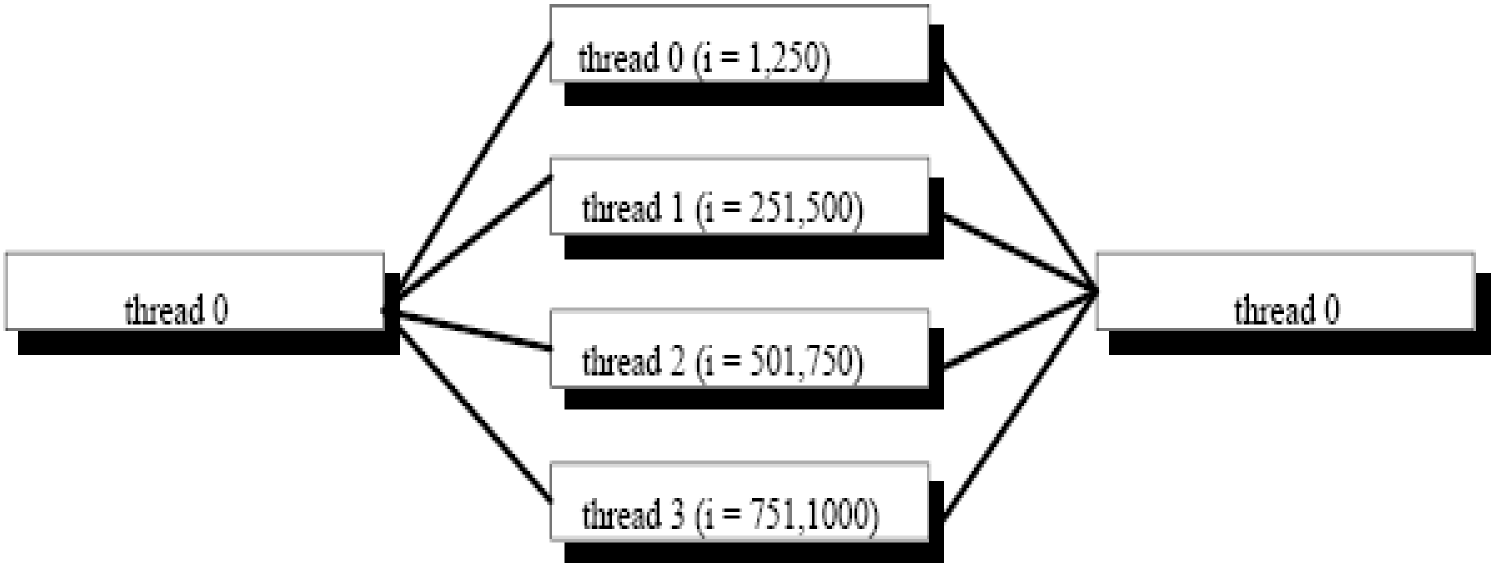
The schedule clause

- **The schedule clause effects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

schedule(static)

- **Iterations are divided evenly among threads**

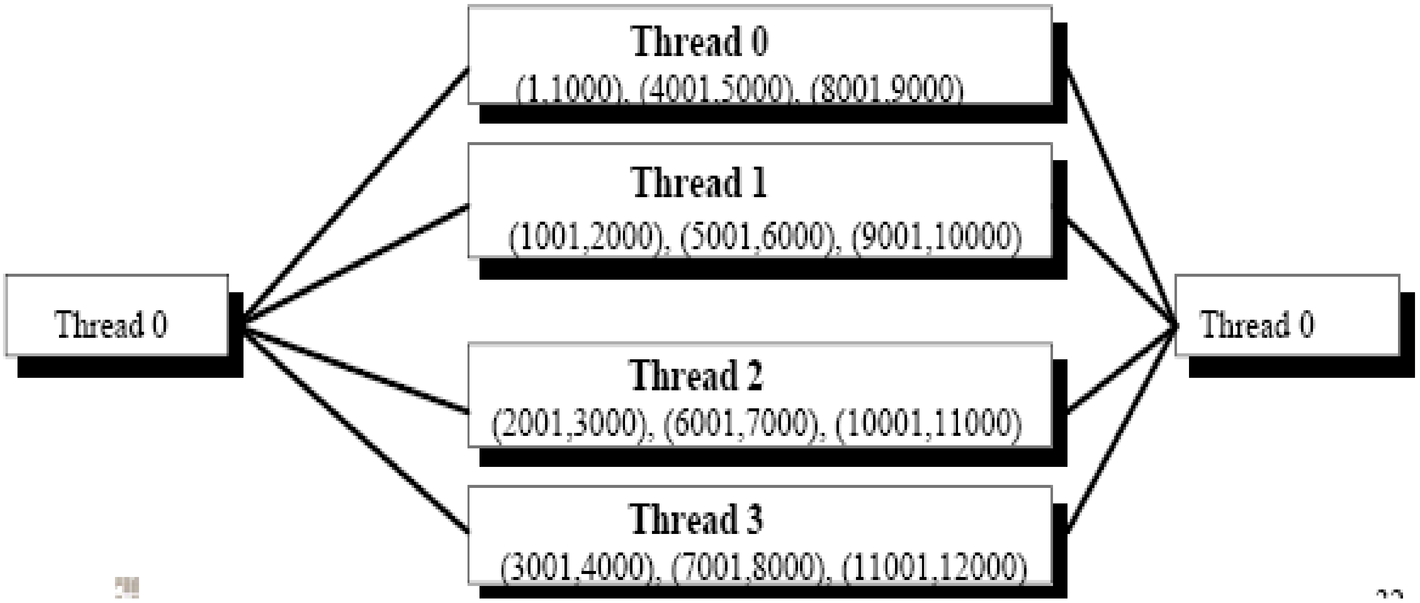
```
c$omp do shared(x) private(i)
c$omp& schedule(static)
        do i = 1, 1000
           x(i)=a
        enddo
```



schedule(static,chunk)

- **Divides the work load in to chunk sized parcels**
- **If there are N threads, each thread does every Nth chunk of work**

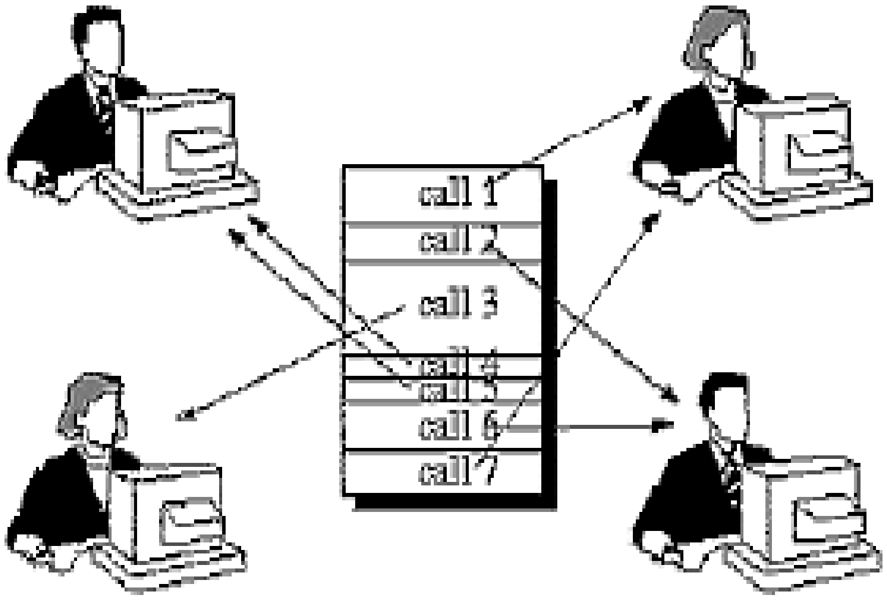
```
c$omp do shared(x)private(i)
c$omp& schedule(static,1000)
        do i = 1, 12000
            … work …
        enddo
```



schedule(dynamic,chunk)

- Divides the workload into chunk sized parcels.
- As a thread finishes one chunk, it grabs the next available chunk.
- Default value for chunk is 1.
- More overhead, but potentially better load balancing.

```
c$omp do shared(x) private(i)
c$omp& schedule(dynamic,1000)
          do i = 1, 10000
                … work …
          end do
```

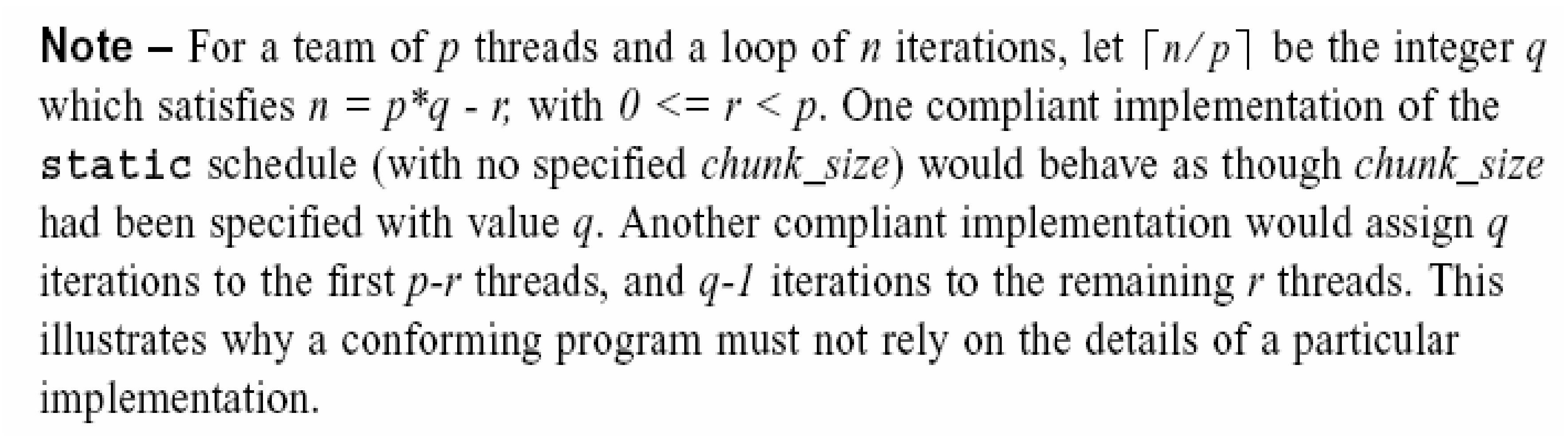


schedule(guided,chunk)

- Like dynamic scheduling, but the chunk size varies dynamically.
- Chunk sizes depend on the number of unassigned iterations.
- The chunk size decreases toward the specified value of chunk.
- Achieves good load balancing with relatively low overhead.
- Insures that no single thread will be stuck with a large number of leftovers while the others take a coffee break.

```
c$omp do shared(x) private(i)
c$omp& schedule(guided,55)
do i = 1, 12000
… work …
end do
```

More about Chunk_size



Note – For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q which satisfies $n = p*q - r$, with $0 <= r < p$. One compliant implementation of the `static` schedule (with no specified *chunk_size*) would behave as though *chunk_size* had been specified with value q. Another compliant implementation would assign q iterations to the first p-r threads, and q-1 iterations to the remaining r threads. This illustrates why a conforming program must not rely on the details of a particular implementation.

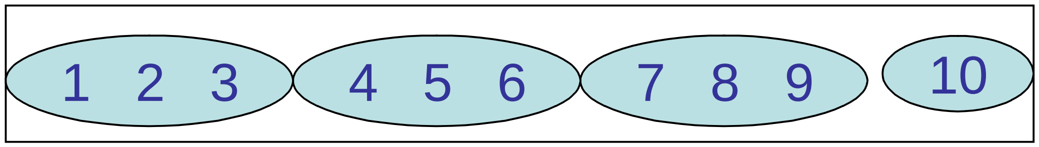
Examples (no k):

n=10 (iterations)
p=4 (threads)
q = cerling (10/4) =3

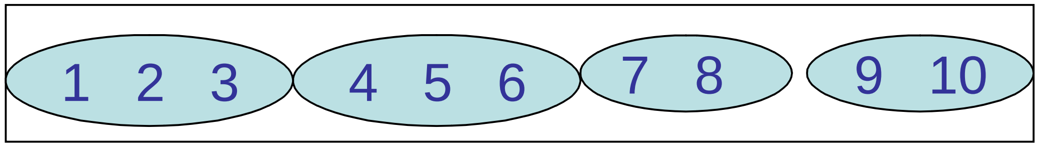
r = p*q – n =12 – 10 = 2

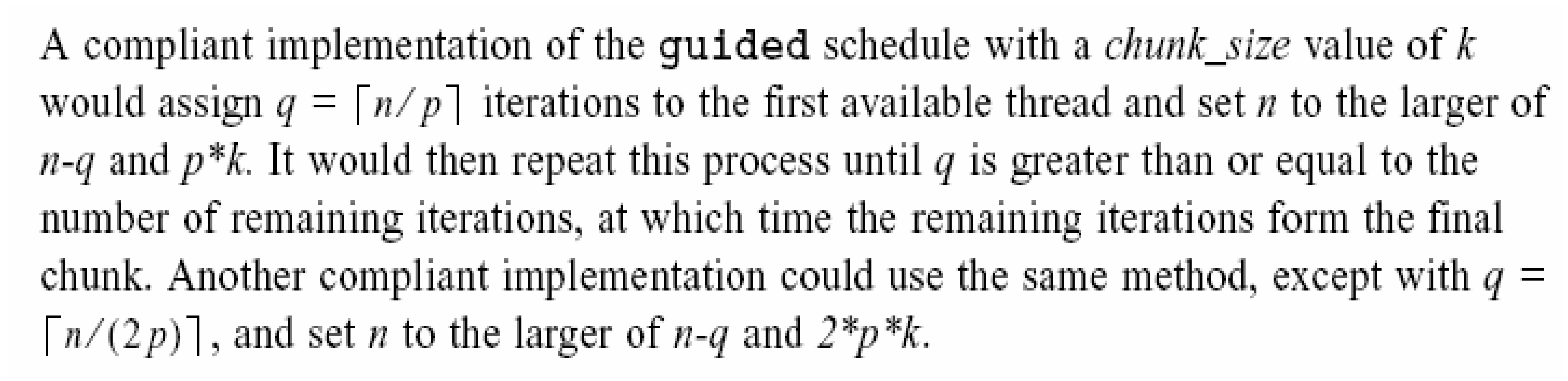
0 < 2 (r) < 4 (p)

Compliant-1: chunk_size k = 3



Compliant-2: chunk_size=3 for 2 (p-r) threads, and 2 (q-1) for 2 (r) threads





A compliant implementation of the `guided` schedule with a *chunk_size* value of k would assign $q = \lceil n/p \rceil$ iterations to the first available thread and set n to the larger of n-q and p*k. It would then repeat this process until q is greater than or equal to the number of remaining iterations, at which time the remaining iterations form the final chunk. Another compliant implementation could use the same method, except with $q = \lceil n/(2p) \rceil$, and set n to the larger of n-q and 2*p*k.

Examples (k=2):

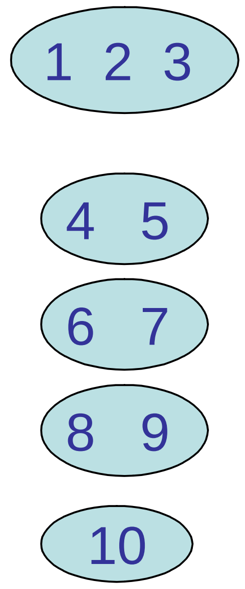
n=10 (iterations);   p=4 (threads)

q = cerling (10/4) =3   (3 to thread-1)

n = max (n-q, p*k) = max(10-3, 4*2) = 8
q2 = cerling (8/4) =2 (2 to thread-2)

q3 = cerling (8/4) =2 (2 to thread-3)
q4 = cerling (8/4) =2 (2 to thread-4)

Remaining 1 for whoever finished first



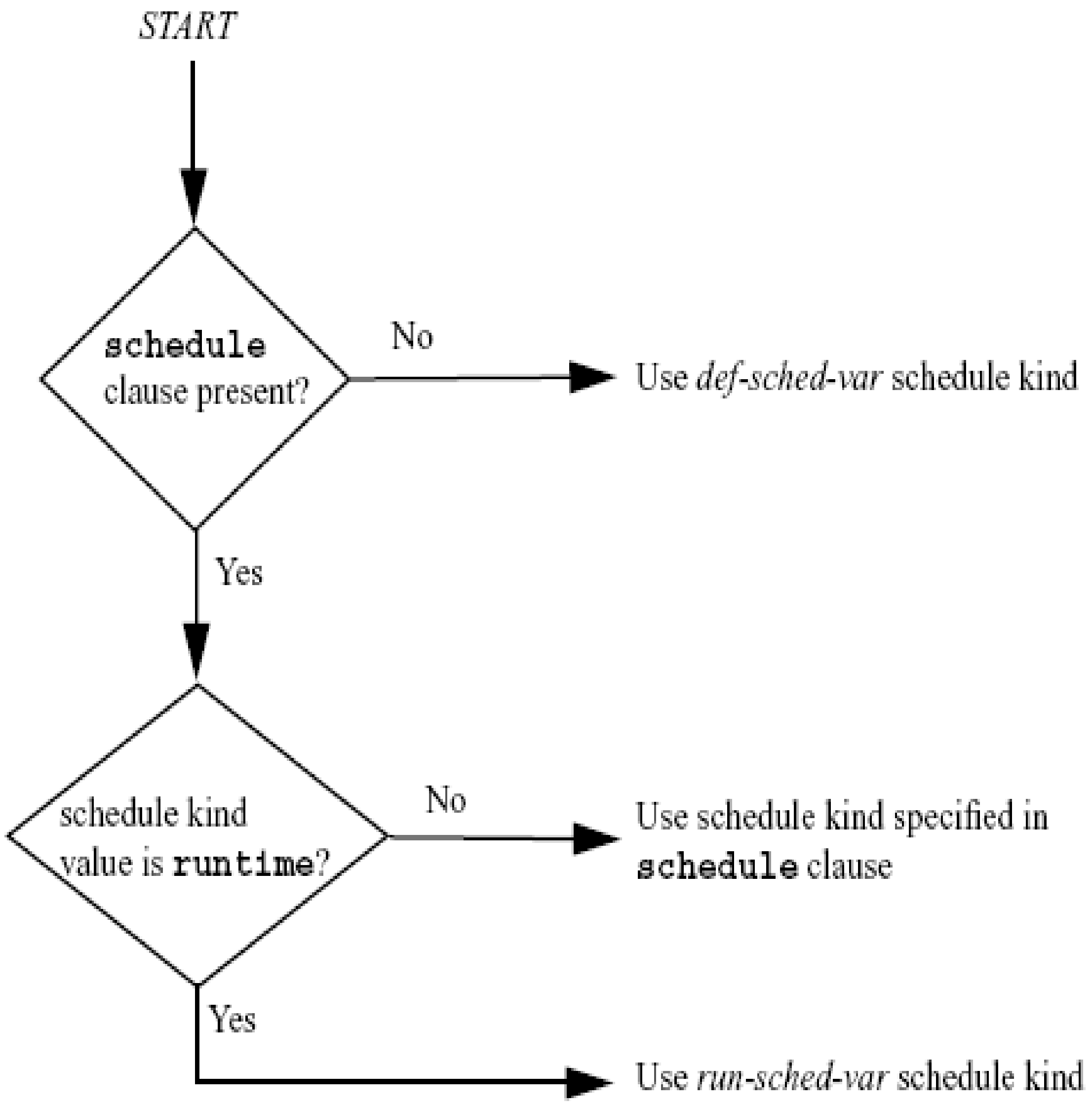
schedule(runtime)

- Scheduling method is determined at runtime.
- Depends on the value of environment variable **OMP_SCHEDULE**
- This environment variable is checked at runtime, and the method is set accordingly.
- Scheduling method is static by default.
- Chunk size set as (optional) second argument of string expression.
- Useful for experimenting with different scheduling methods without recompiling.

```
origin% setenv OMP_SCHEDULE static,1000
origin% setenv OMP_SCHEDULE dynamic
```

DO/for construct: Notes

- The DO loop can not be a DO WHILE loop, or a loop without loop control. Also, the loop iteration variable must be an integer and the loop control parameters must be the same for all threads.
- Program correctness must not depend upon which thread executes a particular iteration.
- It is illegal to branch out of a loop associated with a DO/for directive.
- The chunk size must be specified as a loop invarient integer expression, as there is no synchronization during its evaluation by different threads.
- ORDERED and SCHEDULE clauses may appear once each.


START
schedule clause present?
No
Use def-sched-var schedule kind
Yes
schedule kind value is runtime?
No
Use schedule kind specified in schedule clause
Yes
Use run-sched-var schedule kind

Determining the schedule for a work-sharing loop.

Example: Simple vector-add program

- Three Arrays: A, B, C
- Arrays A, B, C, and variable N will be shared by all threads.
- Variable I will be private to each thread; each thread will have its own unique copy.
- The iterations of the loop will be distributed dynamically in CHUNK sized pieces.
- Threads will not synchronize upon completing their individual pieces of work (NOWAIT).

Fortran - DO Directive Example

```
          PROGRAM VEC_ADD_DO

          INTEGER N, CHUNKSIZE, CHUNK, I
          PARAMETER (N=1000)
          PARAMETER (CHUNKSIZE=100)
          REAL A(N), B(N), C(N)

!         Some initializations
          DO I = 1, N
            A(I) = I * 1.0
            B(I) = A(I)
          ENDDO
          CHUNK = CHUNKSIZE

!$OMP PARALLEL SHARED(A,B,C,CHUNK) PRIVATE(I)
!$OMP DO SCHEDULE(DYNAMIC,CHUNK)
          DO I = 1, N
            C(I) = A(I) + B(I)
          ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL

          END
```

C / C++ - for Directive Example

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

main () {
int i, chunk;
float a[N], b[N], c[N];

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

#pragma omp parallel shared(a,b,c,chunk) private(i)
    {
    #pragma omp for schedule(dynamic,chunk) nowait
    for (i=0; i < N; i++)
        c[i] = a[i] + b[i];
    } /* end of parallel section */

}
```

Work-Sharing Constructs: SECTIONS Directive

Purpose:

1. The SECTIONS directive is a non-iterative work-sharing construct. It specifies that the enclosed section(s) of code are to be divided among the threads in the team.

2. Independent SECTION directives are nested within a SECTIONS directive. Each SECTION is executed once by a thread in the team. Different sections may be executed by different threads. It is possible that for a thread to execute more than one section if it is quick enough and the implementation permits such.

Format:

Fortran	**!$OMP SECTIONS** ***[clause ...]***
	        PRIVATE ***(list)***
	        FIRSTPRIVATE ***(list)***
	        LASTPRIVATE ***(list)***
	        REDUCTION ***(operator* | *intrinsic : list)***
	!$OMP SECTION
	        block
	!$OMP SECTION
	        block
	!$OMP END SECTIONS [NOWAIT]

C/C++	**#pragma omp sections** ***[clause ...] newline***
	        private ***(list)***
	        firstprivate ***(list)***
	        lastprivate ***(list)***
	        reduction ***(operator: list)***
	        nowait
	{
	#pragma omp section ***newline***
	     structured_block
	#pragma omp section ***newline***
	    Structured_block
	}

Clauses:

1. There is an implied barrier at the end of a SECTIONS directive, unless the NOWAIT/nowait clause is used.

2. Clauses are described in detail later, in the Data Scope Attribute section.

Restrictions:

1. It is illegal to branch into or out of section blocks.

2. SECTION directives must occur within the lexical extent of an enclosing SECTIONS directive

Questions:

What happens if the number of threads and the number of SECTIONs are different? More threads than SECTIONs? Less threads than SECTIONs?

Answer: If there are more threads than sections, some threads will not execute a section and some will. If there are more sections than threads, the implementation defines how the extra sections are executed.

Which thread executes which SECTION?

Answer: It is up to the implementation to decide which threads will excute a section and which threads will not, and it can vary from execution to execution

Examples: 3-loops

Serial code with three independent tasks, namely, three do loops. each operating on a dierent array using dierent loop counters and temporary scalar variables.

```
program compute
  implicit none
  integer, parameter :: NX = 10000000
  integer, parameter :: NY = 20000000
  integer, parameter :: NZ = 30000000
  real :: x(NX)
  real :: y(NY)
  real :: z(NZ)
  integer :: i, j, k
  real :: ri, rj, rk
  write(*,*) "start"
  do i = 1, NX
     ri = real(i)
    x(i) = atan(ri)/ri
  end do
  do j = 1, NY
     rj = real(j)
     y(j) = cos(rj)/rj
  end do
  do k = 1, NZ
     rk = real(k)
    z(k) = log10(rk)/rk
  end do
  write(*,*) "end"
end program
```

Examples: 3-loops

one possible parallel version of the preceding code.
(distribute the loop to different threads by hard coding)

```
program compute
……

  write(*,*) "start"
 !$omp parallel
  select case (omp_get_thread_num())
    case (0)
       do i = 1, NX
           ri = real(i)
          x(i) = atan(ri)/ri
       end do
    case (1)
       do j = 1, NY
           rj = real(j)
           y(j) = cos(rj)/rj
       end do
    case (2)
       do k = 1, NZ
           rk = real(k)
          z(k) = log10(rk)/rk
       end do
   end select
 !$omp end parallel
 write(*,*) "end"
end program
```

Examples: 3-loops

Instead of hard-coding, we can use OpenMP provides task sharing directives (section) to achieve the same goal.

```
program compute
……

write(*,*) "start"
!$omp parallel
    !$omp sections
        !$omp section
            do i = 1, NX
                ri = real(i)
                x(i) = atan(ri)/ri
            end do
        !$omp section
            do j = 1, NY
                rj = real(j)
                y(j) = cos(rj)/rj
            end do
        !$omp section
            do k = 1, NZ
                rk = real(k)
                z(k) = log10(rk)/rk
            end do
    !$omp end sections
!$omp end parallel
write(*,*) "end"
end program
```

Example: Vector-add

Fortran: vector-add

```
      PROGRAM VEC_ADD_SECTIONS
      INTEGER N, I
      PARAMETER (N=1000)
      REAL A(N), B(N), C(N)
!     Some initializations
      DO I = 1, N
         A(I) = I * 1.0
         B(I) = A(I)
      ENDDO
!$OMP PARALLEL SHARED(A,B,C),
              PRIVATE(I)
!$OMP SECTIONS
!$OMP SECTION
              DO I = 1, N/2
                 C(I) = A(I) + B(I)
              ENDDO
!$OMP SECTION
              DO I = 1+N/2, N
                 C(I) = A(I) + B(I)
              ENDDO
!$OMP END SECTIONS NOWAIT
!$OMP END PARALLEL
              END
```

- The first n/2 iterations of the DO loop will be distributed to the first thread, and the rest will be distributed to the second thread
- When each thread finishes its block of iterations, it proceeds with whatever code comes next (NOWAIT)

C/C++: vector-add

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

main () {
int i; float a[N], b[N], c[N];
/* Some initializations */
for (i=0; i < N; i++)
     a[i] = b[i] = i * 1.0;

#pragma omp parallel shared(a,b,c) private(i)
{
     #pragma omp sections nowait
     {
          #pragma omp section
          for (i=0; i < N/2; i++)
                c[i] = a[i] + b[i];
          #pragma omp section
           for (i=N/2; i < N; i++)
                c[i] = a[i] + b[i];
       } /* end of sections */
} /* end of parallel section */
}
```

Work-Sharing Constructs: SINGLE Directive

Purpose:

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team.
May be useful when dealing with sections of code that are not thread safe (such as I/O)

OpenMP Work Sharing Constructs - single

- Ensures that a code block is executed by only one thread in a parallel region.
- The thread that reaches the single directive first is the one that executes the single block.
- Equivalent to a sections directive with a single section - but a more descriptive syntax.
- All threads in the parallel region must encounter the single directive.
- Unless nowait is specified, all noninvolved threads wait at the end of the single block

```
c$omp parallel private(i) shared(a)
c$omp do
          do i = 1, n
          …work on a(i) …
          enddo
c$omp single
          … process result of do …
c$omp end single
c$omp do
          do i = 1, n
          … more work …
          enddo
c$omp end parallel
```

OpenMP Work Sharing Constructs - single

- Fortran syntax:

c$omp single [clause [clause…]]
structured block
c$omp end single [nowait]

where clause is one of

- private(*list*)
- firstprivate(*list*)

OpenMP Work Sharing Constructs - single

- C syntax:

  #pragma omp single [clause [clause…]]

  structured block

where clause is one of

- private(*list*)
- firstprivate(*list*)
- nowait

OpenMP Work Sharing Constructs - single

Clauses:

- Threads in the team that do not execute the SINGLE directive, wait at the end of the enclosed code block, unless a NOWAIT/nowait clause is specified.

Restrictions:

- It is illegal to branch into or out of a SINGLE block.

Examples

```
PROGRAM single_1

write(*,*) 'start'
!$OMP PARALLEL DEFAULT(NONE), private(i)

!$OMP DO
do i=1,5
write(*,*) i
enddo
!$OMP END DO

!$OMP SINGLE
write(*,*) 'begin single directive'
do i=1,5
write(*,*) 'hello',i
enddo
!$OMP END SINGLE

!$OMP END PARALLEL

write(*,*) 'end'

END
```

```
[jemmyhu@wha780 single]$ ./single-1
 start
 1
 4
 5
 2
 3
 begin single directive
 hello 1
 hello 2
 hello 3
 hello 4
 hello 5
 end
[jemmyhu@wha780 single]$
```

```
PROGRAM single_3
INTEGER NTHREADS, TID, TID2,
OMP_GET_NUM_THREADS, OMP_GET_THREAD_NUM

write(*,*) "Start"
!$OMP PARALLEL PRIVATE(TID, i)

!$OMP DO
do i=1,8
TID = OMP_GET_THREAD_NUM()
write(*,*) "thread: ", TID, 'i = ', i
enddo
!$OMP END DO

!$OMP SINGLE
write(*,*) "SINGLE - begin"
do i=1,8
TID2 = OMP_GET_THREAD_NUM()
PRINT *, 'This is from thread = ', TID2
write(*,*) 'hello',i
enddo
!$OMP END SINGLE

!$OMP END PARALLEL
write(*,*) "End "
END
```

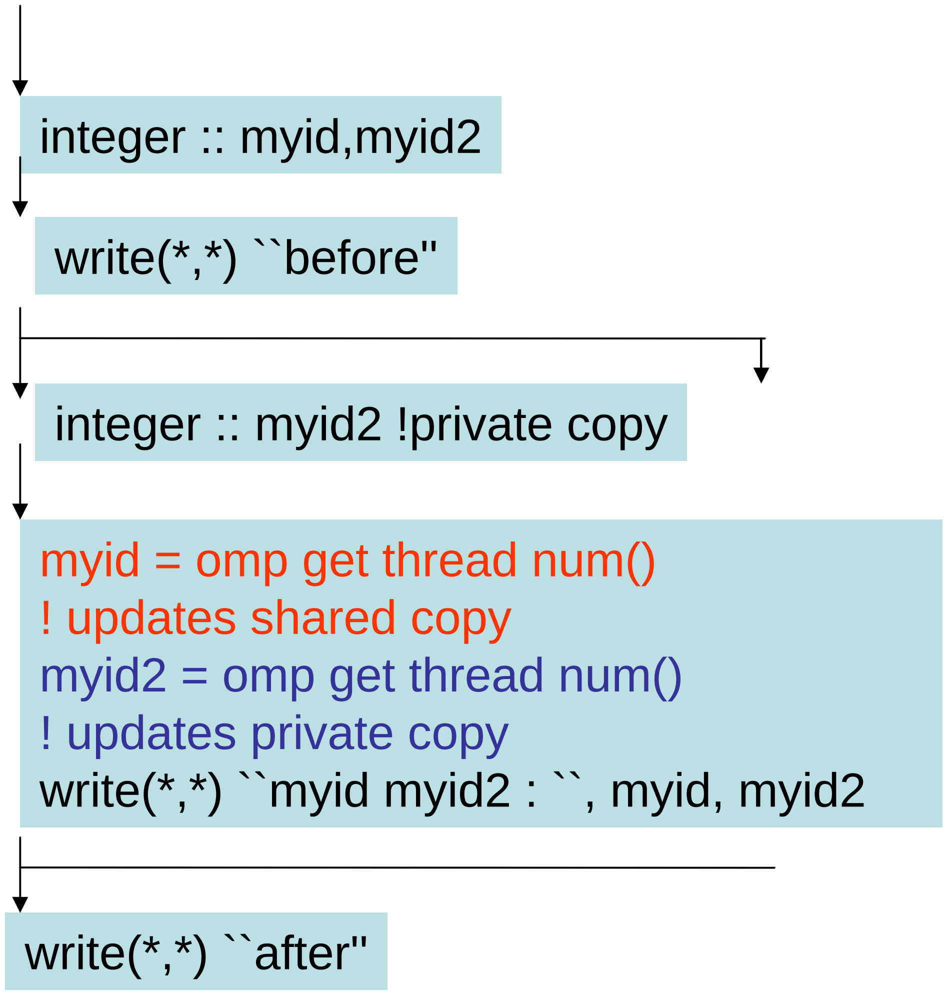
```
[jemmyhu@wha780 single]$
./single-3
 Start
 thread:  0 i =  1
 thread:  1 i =  5
 thread:  1 i =  6
 thread:  1 i =  7
 thread:  1 i =  8
 thread:  0 i =  2
 thread:  0 i =  3
 thread:  0 i =  4
 SINGLE - begin
 This is from thread =  0
 hello 1
 This is from thread =  0
 hello 2
 This is from thread =  0
 hello 3
 This is from thread =  0
 hello 4
 This is from thread =  0
 hello 5
 This is from thread =  0
 hello 6
 This is from thread =  0
 hello 7
 This is from thread =  0
 hello 8
 End
```

Data Scope Clauses

- **SHARED (list)**
- **PRIVATE (list)**
- **FIRSTPRIVATE (list)**
- **LASTPRIVATE (list)**
- **DEFAULT (list)**
- **THREADPRIVATE (list)**
- **COPYIN (list)**
- **REDUCTION (operator | intrinsic : list)**

Data Scope Example (shared vs private)

```
program scope
    implicit none
    integer :: myid, myid2
    write(*,*) "before"
    !$omp parallel private(myid2)
        myid = omp_get_thread_num()
        myid2 = omp_get_thread_num()
        write(*,*) "myid myid2 : ", myid, myid2
    !$omp end parallel
    write(*,*) "after"
end program
```



```
integer :: myid,myid2

write(*,*) ``before''

integer :: myid2 !private copy

myid = omp get thread num()
! updates shared copy
myid2 = omp get thread num()
! updates private copy
write(*,*) ``myid myid2 : ``, myid, myid2

write(*,*) ``after''
```

```
[jemmyhu@silky:~/CES706/openmp/Fortran/data-scope] ./scope-ifort
 before
 myid myid2 :          50           8
 myid myid2 :          32          18
 myid myid2 :          62          72
 myid myid2 :          79          17
 myid myid2 :         124          73
 myid myid2 :          35          88
 myid myid2 :          35          37
 ………..
 ………..

 myid myid2 :          35         114
 myid myid2 :          35          33
 myid myid2 :          35         105
 myid myid2 :          35         122
 myid myid2 :          35          68
 myid myid2 :          35          51
 myid myid2 :          35          81
 after
[jemmyhu@silky:~/CES706/openmp/Fortran/data-scope]
```

Changing default scoping rules: C vs Fortran

- Fortran

  default (shared | private | none)

  index variables are private

- C/C++

  default(shared | none)

  - no defualt (private): many standard C libraries are implemented using macros that reference global variables

  serial loop index variable is shared

  - C for construct is so general that it is difficult for the compiler to figure out which variables should be privatized.

Default (none): helps catch scoping errors

Default scoping rules in Fortran

```
subroutine caller(a, n)
Integer n, a(n), I, j, m
m = 3

!$omp parallel do
    do i = 1, N
        do j = 1, 5
            call callee(a(j), m, j)
        end do
    end do
end

subroutine callee(x, y, z)
common /com/ c
Integer x, y, z, c, ii, cnt
save cnt

cnt = cnt +1
do ii = 1, z
     x = y +z
end do
end
```

Variable	Scope	Is Use Safe?	Reason dor Scope
a	shared	yes	declared outside par construct
n	shared	yes	declared outside par construct
i	private	yes	parallel loop index variable
j	private	yes	Fortran seq. loop index var
m	shared	yes	declared outside par construct
x	shared	yes	actual para. is a, which is shared
y	shared	yes	actual para. is m, which is shared
z	private	yes	actual para. is j, which is private
c	shared	yes	in a common block
ii	private	yes	local stack var of called subrout
cnt	shared	no	local var of called subrout with save attribute

Default scoping rules in C

```
void caller(int a[ ], int n)
{
    int I, j, m=3;

#pragma omp parallel for
    for ( i = 0; i<n; i++){
        int k = m;
        for(j=1; j<=5; j++){
            callee(&a[i], &k, j);
        }
    }
extern int c;

void callee(int *x, int *y, int z)
{
  int ii;
  static int, cnt;

   cnt++;
   for(ii=0; ii<z; ii++){
     *x = *y +c;
}
```

Variable	Scope	Is Use Safe?	Reason dor Scope
a	shared	yes	declared outside par construct
n	shared	yes	declared outside par construct
i	private	yes	parallel loop index variable
j	shared	no	loop index var, but not in Fortran
m	shared	yes	declared outside par construct
k	private	yes	auto var declared inside par const
x	private	yes	Value parameter
*x	shared	yes	actual para. is a, which is shared
y	private	yes	Value parameter
*y	shared	yes	actual para. is k, which is shared
z	private	yes	Value parameter
c	shared	yes	declared as extern
ii	private	yes	local stack var of called subrout
cnt	shared	no	declared as static

reduction(operator|intrinsic:var1[,var2])

- Allows safe global calculation or comparison.
- A private copy of each listed variable is created and initialized depending on operator or intrinsic (e.g., 0 for +).
- Partial sums and local mins are determined by the threads in parallel.
- Partial sums are added together from one thread at a time to get gobal sum.
- Local mins are compared from one thread at a time to get gmin.

```
c$omp do shared(x) private(i)
c$omp& reduction(+:sum)
do i = 1, N
sum = sum + x(i)
end do

c$omp do shared(x) private(i)
c$omp& reduction(min:gmin)
do i = 1,N
gmin = min(gmin,x(i))
end do
```

reduction(operator|intrinsic:var1[,var2])

- Listed variables must be shared in the enclosing parallel context.

- In Fortran

– operator can be **+, *, -, .and., .or., .eqv., .neqv.**
– intrinsic can be **max, min, iand, ior, ieor**

- In C

– operator can be **+, *, -, &, ^, |, &&, ||**
– pointers and reference variables are not allowed in reductions!

```
[jemmyhu@nar316 reduction]$ ./para-reduction
 Before Par Region: I= 1  J= 1  K= 1

 Thread  0         I= 0  J= 0  K= 0
 Thread  1         I= 1  J= 1  K= 1

 Operator          +     *    MAX
 After Par Region:  I= 2  J= 0  K= 1
[jemmyhu@nar316 reduction]$
```

```
 PROGRAM REDUCTION
       USE omp_lib
       IMPLICIT NONE
       INTEGER tnumber
       INTEGER I,J,K
       I=1
       J=1
       K=1
       PRINT *, "Before Par Region: I=",I," J=", J," K=",K
       PRINT *, ""

!$OMP PARALLEL PRIVATE(tnumber) REDUCTION(+:I) REDUCTION(*:J)
REDUCTION(MAX:K)
       tnumber=OMP_GET_THREAD_NUM()
       I = tnumber
       J = tnumber
       K = tnumber
       PRINT *, "Thread ",tnumber, "        I=",I," J=", J," K=",K
!$OMP END PARALLEL

       PRINT *, ""
       print *, "Operator          +     *    MAX"
       PRINT *, "After Par Region:  I=",I," J=", J," K=",K

       END PROGRAM REDUCTION
```

Scope clauses that can appear on a parallel construct

- *shared* and *private* explicitly scope specific variables
- *firstprivate* and *lastprivate* perform initialization and finalization of privatized variables
- *default* changes the default rules used when variables are not explicitly scoped
- *reduction* explicitly identifies reduction variables

General Properties of Data Scope Clauses

- directive with the scope clause must be within the lexical extent of the declaration
- A variable in a data scoping clause cannot refer to a portion of an object, but must refer to the entire object (e.g., not an individual array element but the entire array)
- A directive may contain multiple shared and private scope clauses; however, an individual variable can appear on at most a single clause (e.g., a variable cannot be declared as both shared and private)
- data references to variables that occur within the lexical extent of the parallel loop are affected by the data scope clauses; however, references from subroutines invoked from within the loop are not affected

OpenMP: Synchronization

OpenMP has the following constructs to support synchronization:

- – atomic
- – critical section
- – barrier
- – flush
- – ordered
- – single
- – master

Synchronization categories

- **Mutual Exclusion Synchronization**
  critical
  atomic

- **Event Synchronization**
  barrier
  ordered
  master

- **Custom Synchronization**
  flush
  (lock – runtime library)

Named Critical Sections

A named critical section must synchronize with other critical sections of the same name but can execute concurrently with critical sections of a different name.

```
        cur_max = min_infinity
        cur_min = plus_infinity
!$omp parallel do
        do I = 1, n

           if (a(i).gt. cur_max) then
!$omp critical (MAXLOCK)
             if (a(i).gt. cur_max) then
               cur_max = a(i)
             endif
!$omp critical (MAXLOCK)
           endif
           if (a(i).lt. cur_min) then
!$omp critical (MINLOCK)
             if (a(i).lt. cur_max) then
               cur_min = a(i)
             endif
!$omp critical (MINLOCK)
           endif
        enddo
```

Barriers are used to synchronize the execution of multiple threads within a parallel region, not within a work-sharing construct.

Ensure that a piece of work has been completed before moving on to the next phase

```
!$omp parallel private(index)
        index = generate_next_index()
        do while (inex .ne. 0)
            call add_index (index)
            index = generate_next_index()
        enddo

        ! Wait for all the indices to be generated
!$omp barrier
        index = get_next_index()
        do while (inex .ne. 0)
            call process_index (index)
            index = get_next_index()
        enddo
!omp end parallel
```

Ordered Sections

- Impose an order across the iterations of a parallel loop
- Identify a portion of code within each loop iteration that must be executed in the original, sequential order of the loop iterations.
- Restrictions:

  If a parallel loop contains an ordered directive, then the parallel loop directive itself must contain the ordered clause

  An iteration of a parallel loop is allowed to encounter at most one ordered section

```
!$omp parallel do ordered
          do i = 1, n
               a(i) = … complex calculation here …

       ! Wait until the previous iteration has finished its section
!$omp ordered
          print *, a(i)
       ! Signal the completion of ordered from this iteration
!omp end ordered
        enddo
```

OpenMP: Library routines

- **Lock routines**
  - omp_init_lock(), omp_set_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()
  - **Turn on/off nesting and dynamic mode**
    - omp_set_nested(), omp_set_dynamic(), omp_get_nested(), omp_get_dynamic()
  - **Are we in a parallel region?**
    - omp_in_parallel()
  - **How many processors in the system?**
    - omp_num_procs()

Lock: low-level synchronization functions

- **Why use lock**

  1) The synchronization protocols required by a problem cannot be expressed with OpenMP's high-level synchronization constructs

  2) The parallel overhead incurred by OpenMP's high-level synchronization constructs is too large

The simple lock routines are as follows:

- **omp_init_lock** routine initializes a simple lock.
- **omp_destroy_lock** routine uninitializes a simple lock.
- **omp_set_lock** routine waits until a simple lock is available, and then sets it.
- **omp_unset_lock** routine unsets a simple lock.
- **omp_test_lock** routine tests a simple lock, and sets it if it is available.

Formats (omp.h)

C/C++	**Fortran**
data type omp_lock_t	*nvar* must be an integer variable of Fortran kind=omp_nest_lock_kind.
void omp_init_lock(omp_lock_t **lock*);	subroutine omp_init_lock(*svar*) integer (kind=omp_lock_kind) *svar*

OpenMP: Environment Variables

- Control how “omp for schedule(RUNTIME)” loop iterations are scheduled.
  - **OMP_SCHEDULE “schedule[, chunk_size]”**
- Set the default number of threads to use.
  - **OMP_NUM_THREADS *int_literal***
- Can the program use a different number of threads in each parallel region?
  - **OMP_DYNAMIC TRUE || FALSE**
- Will nested parallel regions create new teams of threads, or will they be serialized?
  - **OMP_NESTED TRUE || FALSE**

OpenMP: Performance Issues

Performance Matrices

- Speedup: refers to how much a parallel algorithm is faster than a corresponding sequential algorithm

$$S_p = \frac{T_1}{T_p}$$

- Size up:
- Scalability

	Data	CPUs	Time
Speedup		n ×	1/n ?
Size up	n ×		n?
Scalability	n ×	n ×	?

Key Factors that impact performance

- Coverage
- Granularity
- Load balancing
- Locality
- synchronization

Software/Programming issues

Highly tied with Hardware

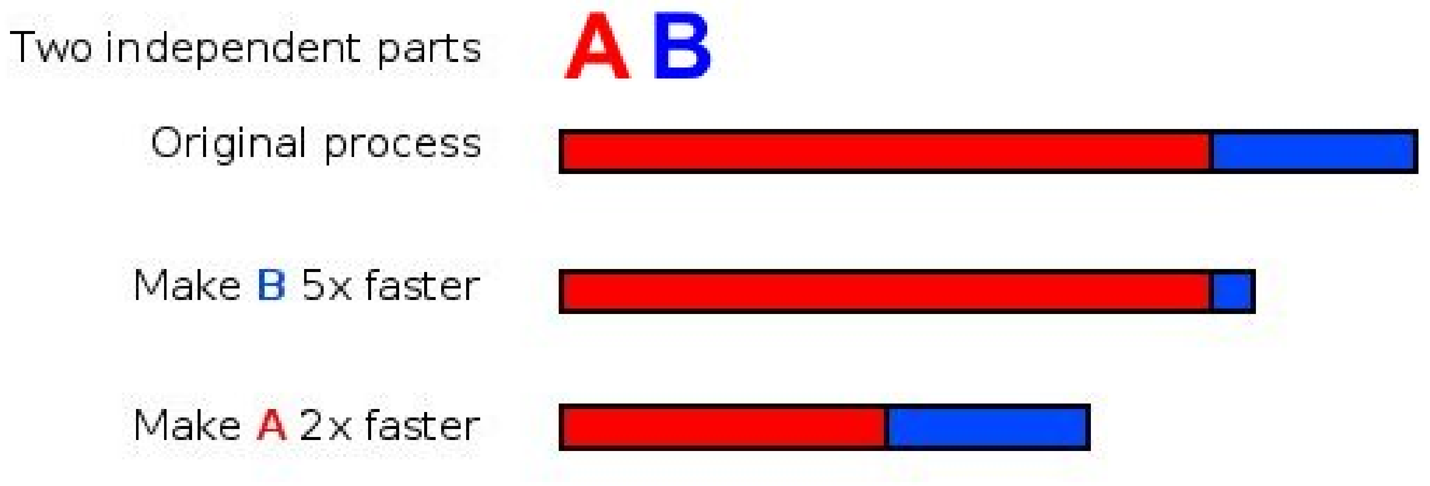
Coverage and Amdahl's law

More technically, the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion *P* of that computation where the improvement has a speedup of *Sp*. (For example, if an improvement can speed up 30% of the computation, *P* will be 0.3; if the improvement makes the portion affected twice as fast, *S* will be 2). Amdahl's law states that the overall speedup of applying the improvement will be

.

$$\frac{1}{(1-P)+\frac{P}{S}}$$

S = 1/[(1-0.3)+(0.3/2)] = 1.176

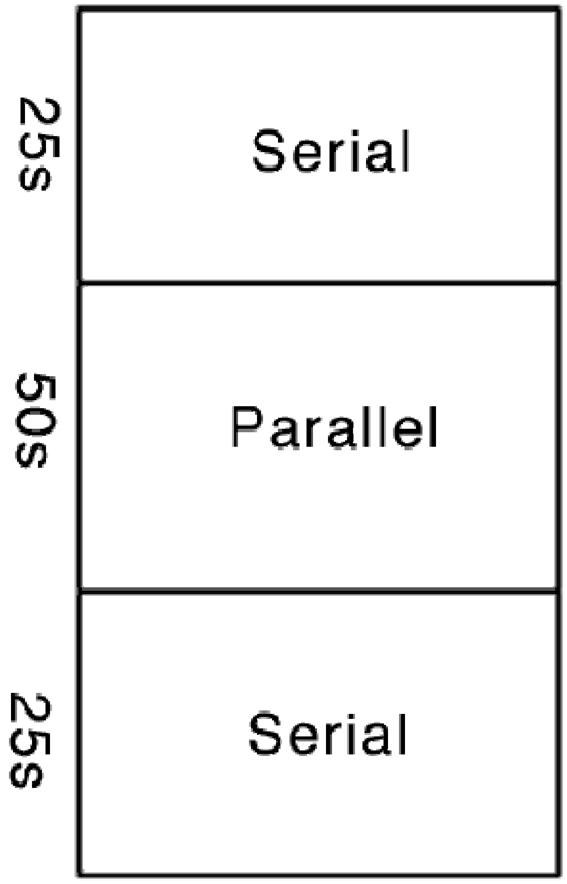


Assume that a task has two independent parts, A and B. B takes roughly 25% of the time of the whole computation. By working very hard, one may be able to make this part 5 times faster, but this only reduces the time for the whole computation by a little. In contrast, one may need to perform less work to make part A be twice as fast. This will make the computation much faster than by optimizing part B, even though B got a bigger speed-up, (5x versus 2x).

Amdahl's Law

How many processors can we really use?

Let's say we have a legacy code such that is it only feasible to convert half of the heavily used routines to parallel:

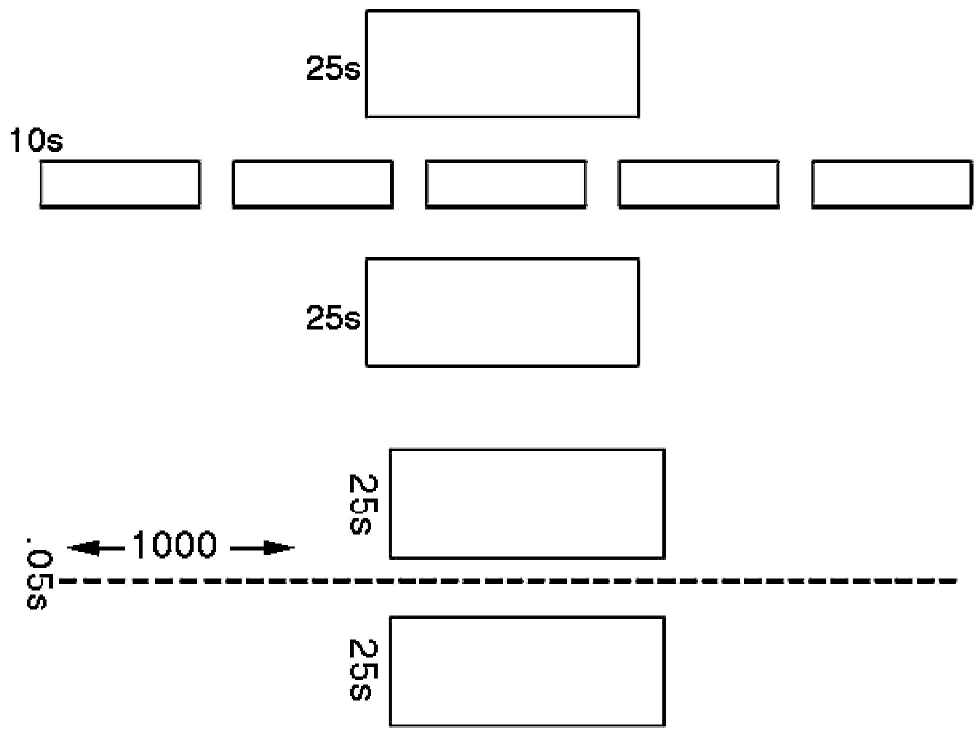


Amdahl's Law

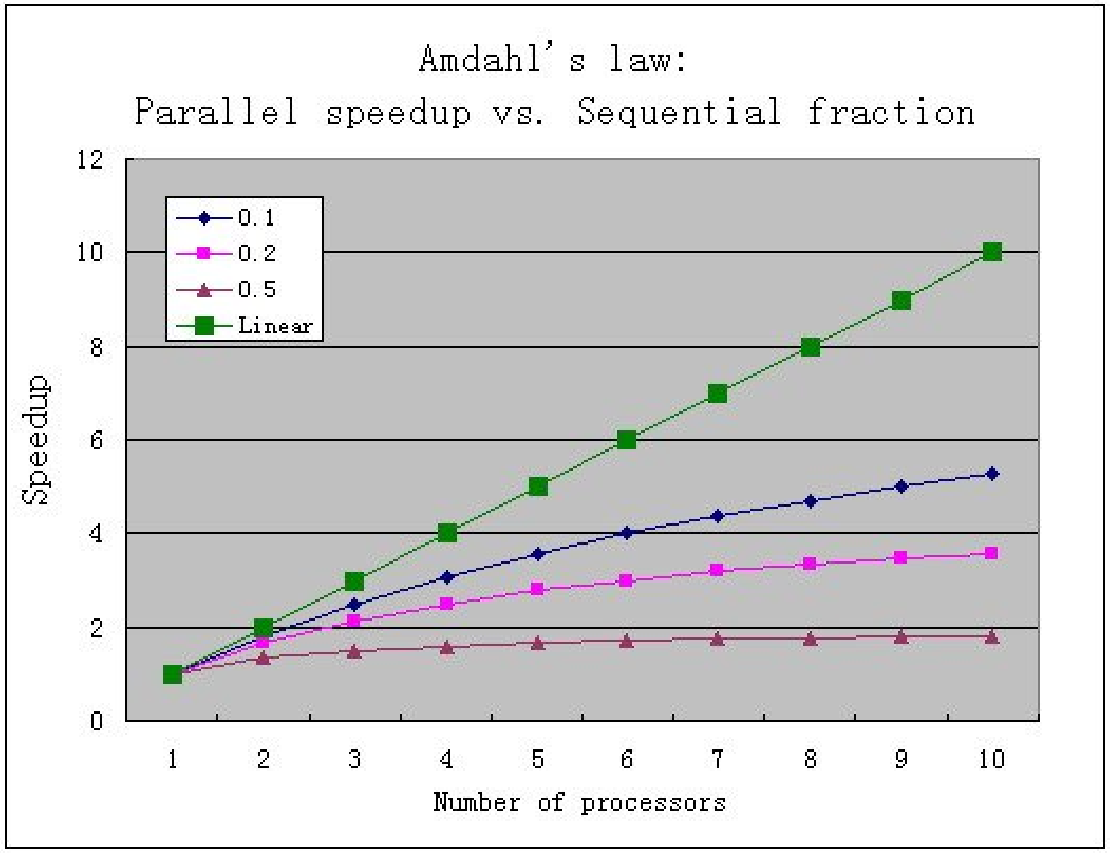
If we run this on a parallel machine with five processors:

Our code now takes about 60s. We have sped it up by about 40%. Let's say we use a thousand processors:

We have now sped our code by about a factor of two.



If only half portion of the program is sequential, the theoretical maximum speedup using parallel computing would be 2 as shown in the diagram no matter how many processors are used. *i.e. (1/(0.5+(1-0.5)/N)) when N is very big*



The speedup of a program using multiple processors in parallel computing is limited by the sequential fraction of the program.

Case 1: use 2 CPUs to get overall 1.8 times speedup

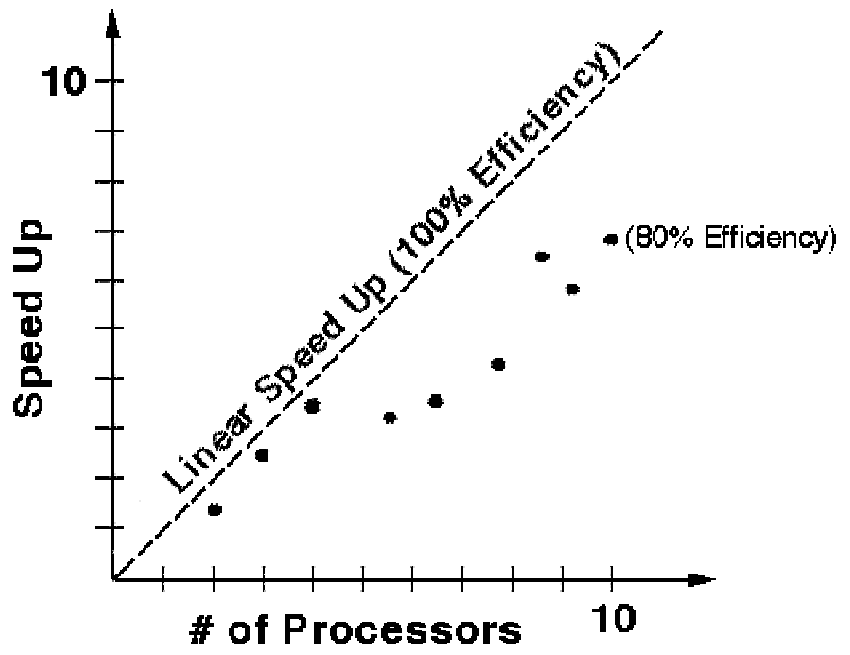
$1.8 = 1/[(1-p) + p/2]$ $p = 2 - 2/1.8 = .89$

Case 2: use 10 CPUs to get overall 9 times speedup

$9 = 1/[(1-p) + p/10]$ $9p = 10 - 10/9$ $p = .988$

Amdahl’s Law

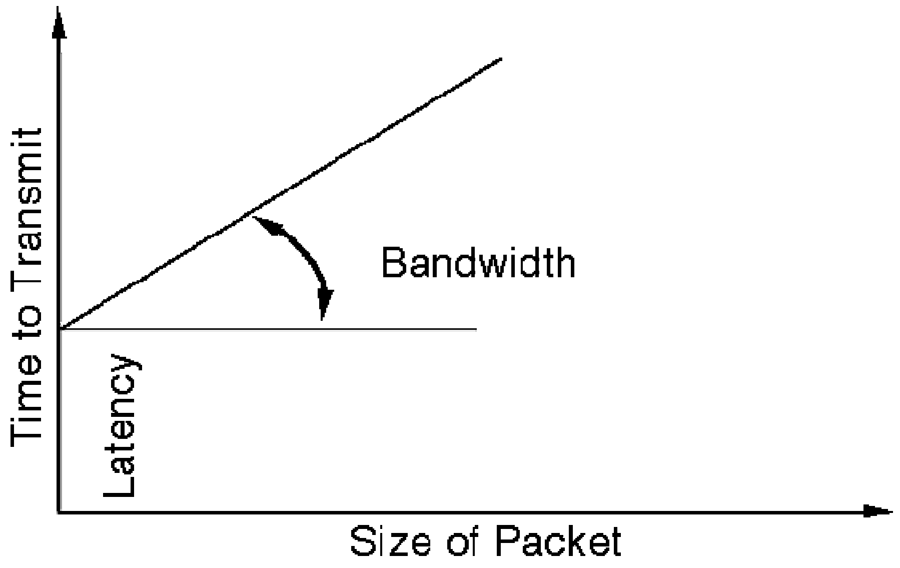
This seems pretty depressing, and it does point out one limitation of converting old codes one subroutine at a time. However, most new codes, and almost all parallel algorithms, can be written almost entirely in parallel (usually, the “start up” or initial input I/O code is the exception), resulting in significant practical speed ups. This can be quantified by how well a code scales which is often measured as efficiency.



Latency and Bandwidth

Even with the "perfect" network we have here, performance is determined by two more quantities that, together with the topologies we'll look at, pretty much define the network: latency and bandwidth. Latency can nicely be defined as the time required to send a message with 0 bytes of data. This number often reflects either the overhead of packing your data into packets, or the delays in making intervening hops across the network between two nodes that aren't next to each other.

Bandwidth is the rate at which very large packets of information can be sent. If there was no latency, this is the rate at which all data would be transferred. It often reflects the physical capability of the wires and electronics connecting nodes.



Granularity

- Invoke a parallel region or loop incurs a certain overhead for going parallel – create save threads and hand off work to the threads
- All threads execute a barrier at the end of parallel region or loop
- Overhead? parallel region vs. loop (from book, on SGI Origin 2000)

```
!omp parallel do
        do i = 1, 16
        enddo
!$omp end parallel do
```

Processors/Threads	Cycles
1	1800
2	2400
4	2900
8	4000
16	8000

```
!omp do
        do i = 1, 16
        enddo
!$omp end do
```

Processors/Threads	Cycles
1	2200
2	1700
4	1700
8	1800
16	2900

Granularity: continue

- In general, one should not parallelize a loop or region unless it takes significant more time to execute then the parallel overhead

- Loop-level parallelism vs. domain decomposition

!$omp do

scales much better (cheaper) than the

!$omp parallel do

using the coarse-grained approach will decrease the overhead significantly

Load Balance

Example: Sparse matrix

Data is not uniformly distributed, one thread will get more points than another.

Solution: Dynamic schedule

If load balancing is the most important issue to performance, perhaps we should use dynamic scheduling.

However, dynamic schedule is more cost than static:

1) more synchronization cost: each thread needs to go to the runtime library after each iteration and ask for another iteration to execute. Increase the chunk size can reduce the synchronization, but it may back to load-balance again.

2) data locality (distance in the cache, etc)

Load Balance: continue

Example: dense triangle matrix-scaling

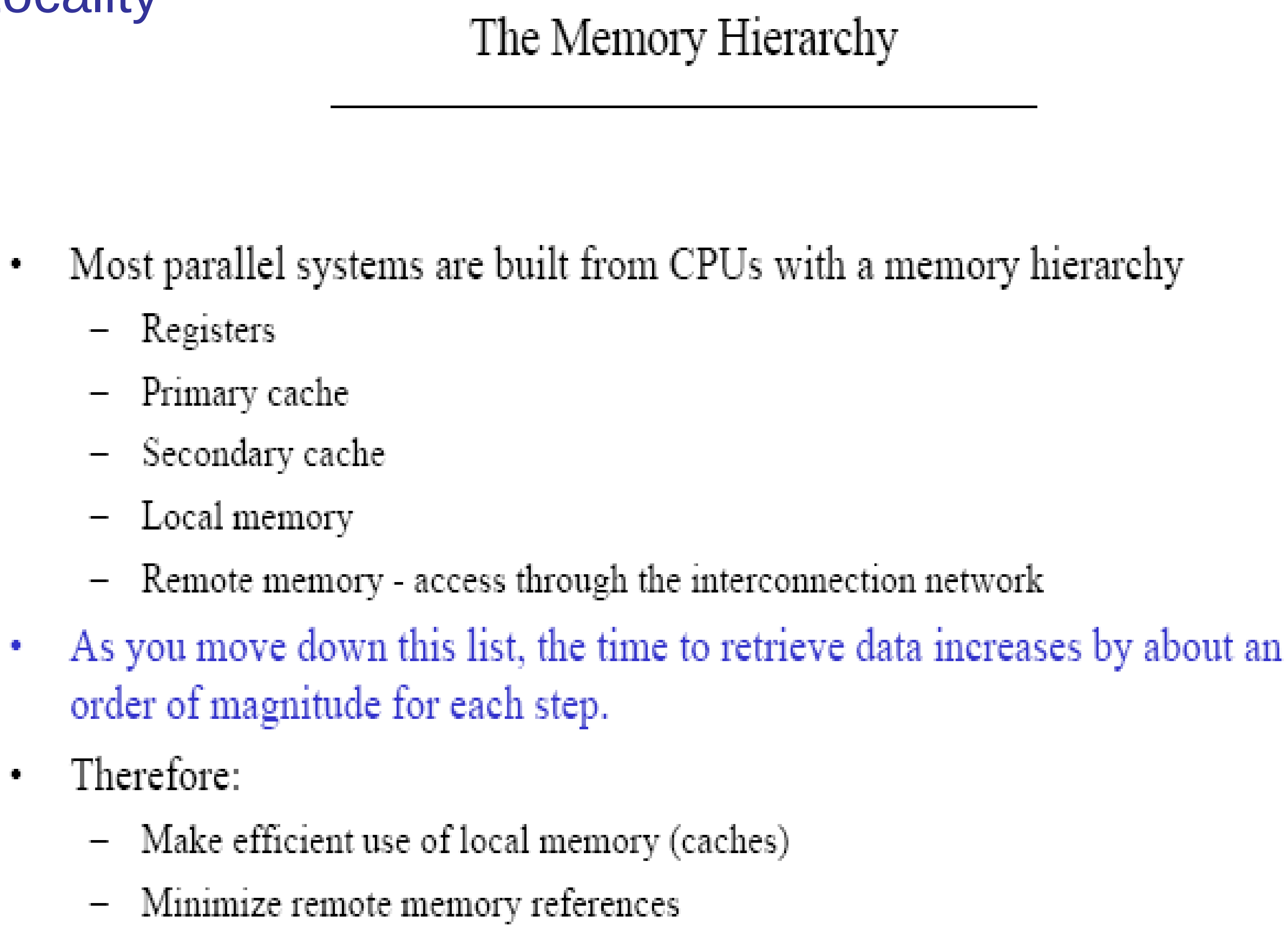
```
for (i=0; I < n; i++){
    for (j=I; j<n; j++){
        a[i][j] = c* a[i][j]
    }
}
```

Each iteration has a different amount of work, but the amount of work varies regularly

Each successive iteration has a linearly decreasing amount of work

Solution: static schedule with a relatively small chunk size

Locality



The Memory Hierarchy

- Most parallel systems are built from CPUs with a memory hierarchy
  - Registers
  - Primary cache
  - Secondary cache
  - Local memory
  - Remote memory - access through the interconnection network
- As you move down this list, the time to retrieve data increases by about an order of magnitude for each step.
- Therefore:
  - Make efficient use of local memory (caches)
  - Minimize remote memory references

Performance Tuning - Cache Locality

- The basic rule for efficient use of local memory (caches):

  Use a memory stride of one

- This means array elements are accessed in the same order they are stored in memory.
- Fortran: "Column-major" order
  - Want the leftmost index in a multi-dimensional array varying most rapidly in a loop
- C: "Row-major" order
  - Want rightmost index in a multi-dimensional array varying most rapidly in a loop
- Interchange nested loops if necessary (and possible!) to achieve the preferred order.

Column major arrays vs. row major arrays

A two dimentional array like A[3][3]:

A11 A12 A13
A21 A22 A23
A31 A32 A33

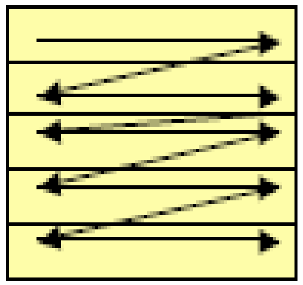
Main memory is just like a big 1D array with indices from 0x0 to 0Xffffff

This is **FORTRAN**'s column major order in memory:
A11 A21 A31 A12 A22 A32 A13 A23 A33

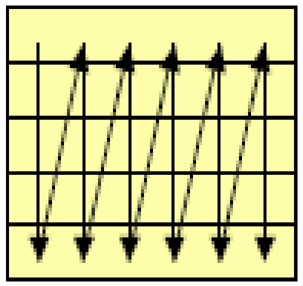
This is **C/C++**'s row major order in memory:
A11 A12 A13 A21 A22 A23 A31 A32 A33

Which of the following is faster in C?

```
for (i=0; i < 10000; i++)
  for (j=0; j < 10000; j++)
   sum += a[i][j];
```



```
for (j=0; j < 10000; j++)
  for (i=0; i < 10000; i++)
   sum += a[i][j];
```



Performance Tuning - Data Locality

- On NUMA ("non-uniform memory access") platforms, it may be important to know
  - Where threads are running
  - What data is in their local memories
  - The cost of remote memory references
- OpenMP itself provides no mechanisms for controlling
  - the binding of threads to particular processors
  - the placement of data in particular memories
- Designed with true (UMA) SMP in mind
  - For NUMA, the possibilities are many and highly machine-dependent
- Often there are system-specific mechanisms for addressing these problems
  - Additional directives for data placement
  - Ways to control where individual threads are running

OpenMP: pitfalls

- Race condition
- Data Dependences
- Deadlock

2 major SMP errors

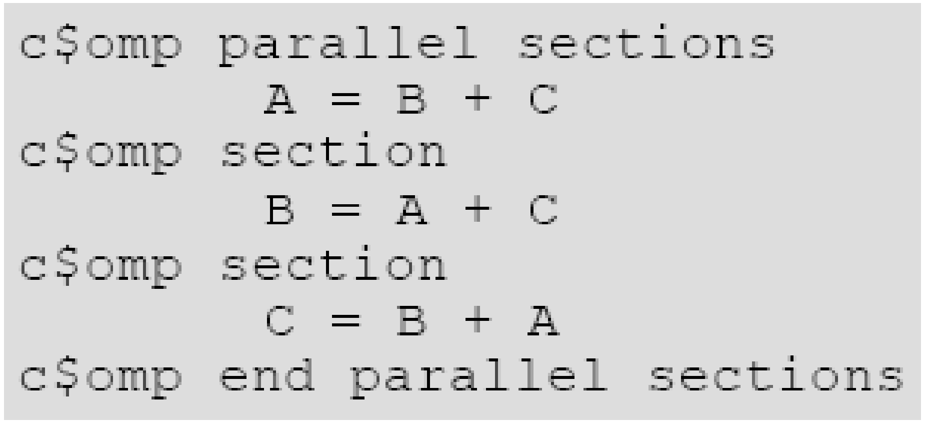
- **Race Conditions**

  – The outcome of a program depends on the detailed timing of the threads in the team.

- **Deadlock**

  – Threads lock up waiting on a locked resource that will never become free.

Race Conditions: Examples



```
c$omp parallel sections
        A = B + C
c$omp section
        B = A + C
c$omp section
        C = B + A
c$omp end parallel sections
```

- The result varies unpredictably depending on the order in which threads execute the sections.
- Wrong answers are produced without warning!

Race Conditions: Examples

```
c$omp parallel shared(x) private(tmp)
       id = OMP_GET_THREAD_NUM()
c$omp do reduction(+:x)
       do j=1,100
           tmp = work(j)
           x = x + tmp
       enddo
c$omp end do nowait
       y(id) = work(x,id)
c$omp end parallel
```

- The result varies unpredictably because the value of x isn’t correct until the barrier at the end of the do loop is reached.
- Wrong answers are produced without warning!
- Be careful when using nowait!

Race Conditions: Examples

```
       real tmp,x
c$omp parallel do reduction(+:x)
       do j=1,100
           tmp = work(j)
           x = x + tmp
       enddo
c$omp end do
       y(id) = work(x,id)
```

- The result varies unpredictably because access to the shared variable tmp is not protected.
- Wrong answers are produced without warning!
- Probably want to make tmp private.

Data Dependences

- Detection
- Classification
- Removal

Detection

- Loop-carried dependence: dependency between statements executed in different iterations of the loop

- Dependences are always associated with a particular memory location, we can detect them by analyzing how each variable is used within the loop

  - Is the variable only read and never assigned within the loop body? If so, there are no dependences involving it

  - Otherwise, consider the memory locations that make up the variable and that are assigned within the loop. For each such location, is there exactly one iteration that accesses the location? If so, there are no dependences involving the variable. If not, there is a dependence.

Loops with or without data dependence

```
10      do i = 2, n
             a(i) = a(i) + a(i-1)
        enddo

20      do i = 2, n, 2
             a(i) = a(i) + a(i-1)
        enddo

30      do i = 2, n/2
             a(i) = a(i) + a(i + n/2)
        enddo

40      do i = 2, n/2+1
             a(i) = a(i) + a(i + n/2)
        enddo
```

10 yes

each iteration writes an element of a that is read by the next iteration

20 no

loop has a stride of 2, it writes every other element

30 no

each iteration read only the element it writes plus an element that is not written by the loop since it has a subscript greater than n/2

40 yes

the first iteration read a(n/2+1), while that last iteration write this element

Classification

- Loop-carried dependence

- Dataflow dependency:
  Dataflow relation between the two dependent statements, i.e., whether or not the two statements communicate values through the memory location

S1 – earlier statement, write the memory location
S2 – later statement, read the memory location
The value read by S2 in a serial execution is the same as that written by S1. In this case, the result of a computation by S1 is communicated, or ‘flows’ to S2, called flow dependence

S1 must execute first to produce the value that is consumed by S2

Generally, it’s hard to remove this dependence

Classification: continue

- Dataflow dependency:

  two other kinds of dependences which can be removed; they are not communication of data between S1 and S2, but reuse of the memory for different purpose at different points in the program

- anti dependence

  S1 read the location
  S2 write the location

  make a private copy of the location and initializing the copy belonging to S1

- output dependence

  both S1 and S2 write the location

  privatizing the memory location and in addition copying the last value back to the shared copy of the location

A loop containing multiple data dependences

```
      do i = 2, n-1
10       x = d(i) + i
20      a(i) = a(i + 1) + x
30      b(i) = b(i) +b(i - 1) + d(i - 1)
40      c(2) = 2 * i
      enddo
```

Memory location	Line	Iteration earlier	Access	Line	Iteration later	Access	Loop carried	Kind of dataflow
x	10	i	write	20	i	r	no	flow
x	10	i	w	10	i+1	w	y	output
x	20	i	read	10	i+1	w	y	anti
a(i+1)	20	i	r	20	i+1	w	y	anti
b(i)	30	i	w	30	i+1	r	y	flow
c(2)	40	i	w	40	i+1	w	y	output

Remove dependences

- **removal of anti dependences**

Serial version containing anti dependences

```
! Array is assigned before start of loop
    do i = 1, n-1
        x = (b(i) + c(i))/2
10      a(i) = a(i+1) +x
    enddo
```

Parallel version with dependences removed

```
! $omp parallel do shared(a, a2)
    do i = 1, n-1
       a2(i) = a(i+1)                    - make a copy of the array
    enddo
! $omp parallel do shared(a, a2) private(x)
    do i = 1, n-1
        x = (b(i) + c(i))/2
10      a(i) = a2(i) +x
    enddo
```

Remove dependences

- **removal of output dependences**

Serial version containing output dependences

```
do i = 1, n
     x = (b(i) + c(i))/2
    a(i) = a(i) +x
    d(1) = 2 * x
enddo
y = x + d(1) + d(2)
```

Parallel version with dependences removed

```
! $omp parallel do shared(a) lastprivate(x, d1)
     do i = 1, n
          x = (b(i) + c(i))/2
         a(i) = a(i) +x
         d1 = 2 * x
     enddo
     d(1) = d1
     y = x + d(1) + d(2)
```

Remove dependences

- **removal of flow dependences caused by a reduction**

Serial version containing a flowdependence

```
    x = 0
    do i = 1, n
       x = x + a(i)
    enddo
```

Parallel version with dependences removed by reduction clause

```
   x = 0
! $omp parallel do reduction(+: x)
    do i = 1, n
       x = x + a(i)
    enddo
```

Remove dependences

- **removal of flow dependences using loop skewing**

Serial version containing a flow dependence

```
      do i = 2, n
 10      b(i) = b(i) + a(i-1)
 20      a(i) = a(i) + c(i)
      enddo
```

Parallel version with dependences removed by reduction clause

```
     b(2) = b(2) + a(1)
! $omp parallel do shared(a, b, c)
      do i = 1, n-1
 20      a(i) = a(i) + c(i)
 10      b(i+1) = b(i+1) + a(i)
      enddo
      a(n) = a(n) + c(n)
```

Dealing with non-removable dependences

- **parallelization of a loop nest containing a recurrence**

Serial version containing a recurrence

```
      do j = 1, n
         do i = 1, n
              a(i, j) = a(i, j) + a(i, j-1)
         enddo
      enddo
```

Parallel version to the loop in the nest

```
     do j = 1, n
!$omp parallel do shared (a)
         do i = 1, n
             a(i, j) = a(i, j) + a(i, j-1)
         enddo
      enddo
```

Dealing with non-removable dependences

- **parallelization of part of a loop using fissioning**

Serial version containing a recurrence

```
        do i = 1, n
10            a(i, j) = a(i, j) + a(i, j-1)
20            y = y + c(i)
        enddo
```

Parallel version

```
        do i = 1, n
10            a(i, j) = a(i, j) + a(i, j-1)
        enddo

!$omp parallel do reduction(+: y)
        do i = 1, n
 20           y = y + c(i)
        enddo
```

Deadlock Examples

```
        call OMP_INIT_LOCK(lcka)
        call OMP_INIT_LOCK(lckb)
c$omp parallel sections
        call OMP_SET_LOCK(lcka)
        call OMP_SET_LOCK(lckb)
        call useAandB(res)
        call OMP_UNSET_LOCK(lckb)
        call OMP_UNSET_LOCK(lcka)
c$omp section
        call OMP_SET_LOCK(lckb)
        call OMP_SET_LOCK(lcka)
        call useBandA(res)
        call OMP_UNSET_LOCK(lcka)
        call OMP_UNSET_LOCK(lckb)
c$omp end parallel sections
```

- If A is locked by one thread and B by another, you have deadlock.
- If both are locked by the same thread, you have a race condition!
- Avoid nesting different locks.

Deadlock Examples

```
       call OMP_INIT_LOCK(lcka)
c$omp parallel sections
       call OMP_SET_LOCK(lcka)
       ival = work()
       if (ival.eq.tol) then
         call OMP_UNSET_LOCK(lcka)
       else
         call error(ival)
       endif
c$omp section
       call OMP_SET_LOCK(lcka)
       call useBandA(res)
       call OMP_UNSET_LOCK(lcka)
c$omp end parallel sections
```

- If A is locked in the first section and the if statement branches around the unset lock, then threads in the other section will deadlock waiting for the lock to be released.
- Make sure you release your locks!

Example: Calculating π

- **Numerical integration**

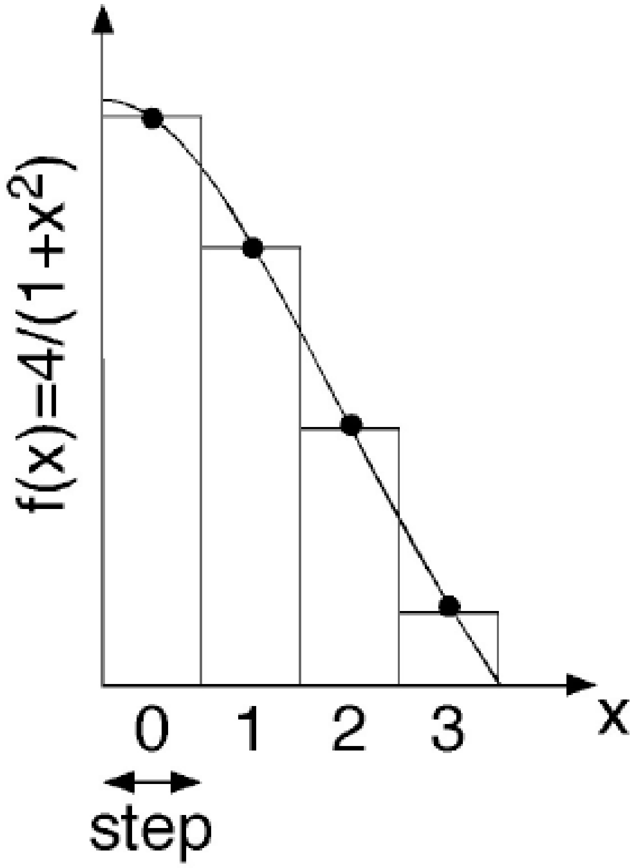
$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`
$x_i = (i+0.5)\Delta$ $(i = 0,\ldots,N-1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



OpenMP Program: `omp_pi_critical.c`

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
void main() {
  double step,sum=0.0,pi;
  step = 1.0/NBIN;
#pragma omp parallel
  {
    int nthreads,tid,i;
    double x;
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    for (i=tid; i<NBIN; i+=nthreads) {
      x = (i+0.5)*step;
#pragma omp critical
      sum += 4.0/(1.0+x*x);
    }
  }
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```

Shared variables

Private (local) variables

This has to be atomic

Avoid Critical Section: `omp_pi.c`

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
void main() {
  int nthreads,tid;
  double step,sum[MAX_THREADS]={0.0},pi=0.0;
  step = 1.0/NBIN;
#pragma omp parallel private(tid)
  {
    int i;
    double x;
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    for (i=tid; i<NBIN; i+=nthreads) {
      x = (i+0.5)*step;
      sum[tid] += 4.0/(1.0+x*x);
    }
  }
  for(tid=0; tid<nthreads; tid++) pi += sum[tid]*step;
  printf("PI = %f\n",pi);
}
```

Array of partial sums for multi-threads

OpenMP on SHARCNET

- SHARCNET systems

  http://www.sharcnet.ca/Facilities/index.php

  2 - Shared memory systems (silky, typhon)

  Many Hybrid Distributed-Shared Memory clusters

   - clusters with multi-core nodes

- Consequence: all systems allow for SMP- based parallel programming (i.e., OpenMP) applications

Size of OpenMP Jobs on specific system

System	Nodes	CPU/Node	OMP_NUM_THREADS (max)
bala, bruce, bull, dolphin, narwhal, megaladon, tiger, whale, zebra	Opteron	4	4
gulper,goblin, requin, wobbe, cat	Opteron (cat is mixed)	2	2
greatwhite	Alpha	4	4
coral, spinner	Itanium2	2	2
mako	Xeon	2	2
silky	SGI Altix SMP	128	128
typhon	Alpha SMP	16	16

OpenMP: compile and run

- **Compiler flags:**

Intel (icc, ifort)	-openmp
Pathscale (cc, c++, f77, f90)	-openmp
PGI (pgcc, pgf77, pgf90)	-mp

e.g.,
```
f90 –openmp –o hello_openmp hello_openmp.f
```

- **Run OpenMP jobs in the threaded queue**

Submit OpenMP job on a cluster with 4-cpu nodes
(The size of threaded jobs varies on different systems as discussed in the previous page)

```
sqsub –q threaded –n 4 –o hello_openmp.log ./hello_openmp
```

References

1) Parallel Programming in OpenMP by Rohit Chandra, Morgan Kaufman Publishers, ISBN 1-55860-671-8

2) OpenMP specifications for C/C++ and Fortran, http://www.openmp.org/

3) http://www.openmp.org/presentations/sc99/sc99_tutorial_files/v3_document.htm

4) http://www.llnl.gov/computing/tutorials/openMP/

5) http://www.nic.uoregon.edu/iwomp2005/iwomp2005_tutorial_openmp_rvdp.pdf

6) http://www.osc.edu/hpc/training/openmp/big/fsld.001.html

7) http://cacs.usc.edu/education/cs596/06OMP.pdf

8) http://www.ualberta.ca/AICT/RESEARCH/Courses/2002/OpenMP/omp-from-scratch.pdf