International Centre for Radio Astronomy Research

OpenMP

Research Associate Professor
Kevin Vinsen

OpenMP: An API for Writing Multithreaded Applications

A set of compiler directives and library routines for parallel application programmers.

Designed to simplify writing multi-threaded programs in Fortran, C and C++.

Attempts to Standardise the last 20 years of Symmetric Multi-Processing (SMP) practice.

Supposed to be easier than PThreads

Lets try a thought experiment

If we wanted to create a “language” to write multi-threaded code. What would we need?

Shared Data

Shared memory variables might be declared as shared with, say,

```
shared int x1;
shared double x2[1000];
```

Private Data

Private memory variables might be declared as private with, say,

```
private int y1;
private double y2[1000];
```

par Construct

For specifying parallel statements:

```
par {
      S1;
      S2;
       .
       .
      Sn;
}
```

forall Construct

To start multiple similar processes together:

```
forall (i = 0; i < n; i++) {
        S1;
        S2;
        .
        .
        .
        Sm;
}
```

which generates *n* processes each consisting of the statements forming the body of the for loop, S_1, S_2, ..., S_m. Each process uses a different value of *i*.

Example

```
forall (i = 0; i < 5; i++)
      a[i] = 0;
```

clears **a[0]**, **a[1]**, **a[2]**, **a[3]**, and **a[4]** to zero concurrently.

Dependency Analysis

To identify which processes could be executed together.

Can see immediately in the code

```
forall (i = 0; i < 5; i++)
        a[i] = 0;
```

that every instance of the body is independent of other instances and all instances can be executed simultaneously.

However, it may not be that obvious. Need algorithmic way of recognising dependencies, for a *parallelising compiler*.

Dependency Analysis (cont.)

However, it may not be that obvious.

Can not see immediately in the code

```
forall (i = 0, j=5; i < 5; i++, j--)
        a[i] = a[j-i];
```

If every instance of the body is independent of other instances. It's unclear if all instances can be executed simultaneously.

Need algorithmic way of recognising dependencies, for a *parallelising compiler*.

OpenMP is

An Application Program Interface (API) that may be used to explicitly direct multi-threaded, shared memory parallelism

Comprised of three primary API components:

- Compiler Directives
- Runtime Library Routines
- Environment Variables

Portable:

- The API is specified for C/C++ and Fortran
- Most major platforms have been implemented including *nix platforms and Windows

OpenMP is

Standardised:

- Jointly defined and endorsed by a group of major computer hardware and software vendors

What does OpenMP stand for?

- Short version: Open Multi-Processing
- Long version: 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).

NOT necessarily implemented identically by all vendors.

NOT guaranteed to make the most efficient use of shared memory.

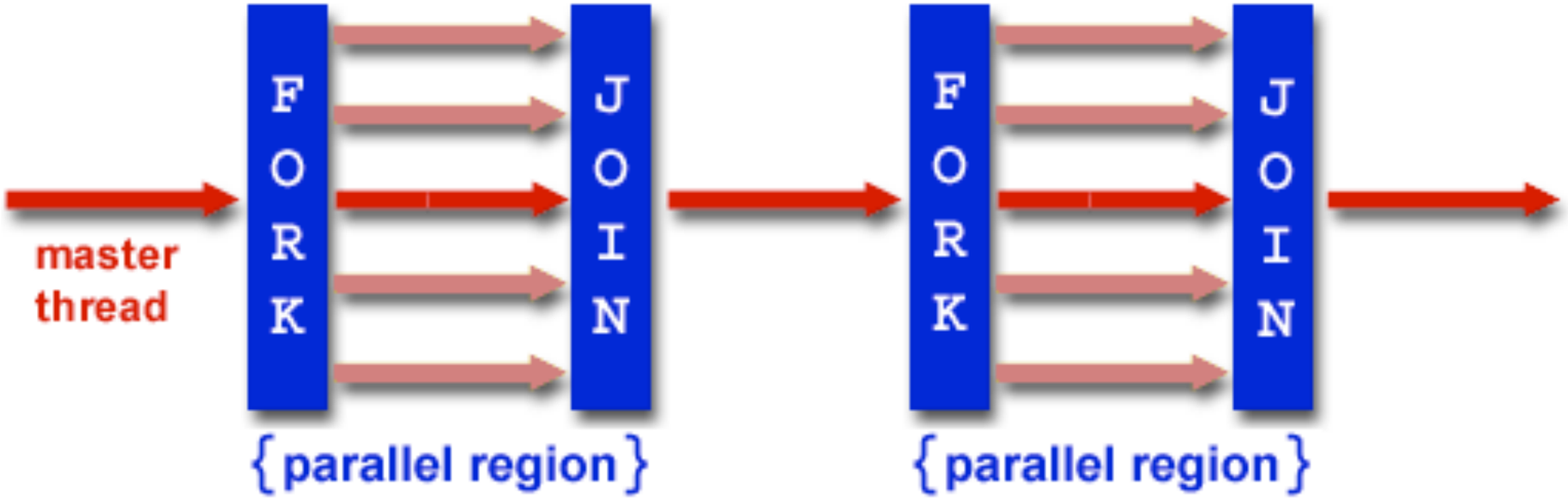
NOT a Silver Bullet

OpenMP programming model

Shared Memory, Thread Based Parallelism

Explicit Parallelism

Fork – Join Model



OpenMP programming model

- Compiler Directive Based
- Nested Parallelism Support
- Dynamic Threads
- No parallel I/O
- Memory Model – FLUSH often

OpenMP Core Syntax

Most of the constructs in OpenMP are compiler directives

```
#pragma omp parallel num_threads(4)
```

```
!$OMP PARALLEL num_threads(4)
```

Most OpenMP constructs apply to a block of code

It should have one point of entry at the top and one at the bottom

C/C++Example

```
#include <omp.h>
void main() {
   int var1, var2, var3;

   serial code
#pragma omp parallel private(var1, var2) shared(var3)
{
   parallel code
}
   resume serial code
}
```

#pragma omp

For C/C++, the OpenMP directives are contained in #pragma statements. The OpenMP #pragma statements have the format:

```
#pragma omp directive_name ...
```

where omp is an OpenMP keyword.

May be additional parameters (clauses) after the directive name for different options.

Some directives require code to be specified in a structured block (a statement or statements) that follows the directive and then the directive and structured block form a “construct”.

It is possible to write a parallel program with #pragma statements that a C/C++ compiler would compile to a sequential executable and an OpenMP parallel one.

Hello World

```
void main() {
   int ID = 0;
   printf("Hello (%d) ", ID);
   printf("World (%d)\n", ID);
}
```

```
Hello (0) World (0)
```

Hello World

```
#include "omp.h"
void main() {
#pragma omp parallel
   {
      int ID = omp_get_thread_num();
      printf("Hello (%d) ", ID);
      printf("World (%d)\n", ID);
   }
}
```

Hello World

```
Hello (1) Hello (0) World (1)

World (0)

Hello (2) Hello (3) World (3)

World (2)
```

Parallel Directive

C/C++
```
#pragma omp parallel
        structured_block
```

Fortran
```
!$OMP PARALLEL
        code
!$OMP END PARALLEL
```

creates multiple threads, each one executing the specified structured_block, either a single statement or a compound statement created with { …} with a single entry point and a single exit point.

There is an implicit barrier at the end of the construct.
The directive corresponds to forall construct.

How do threads interact

OpenMP is a multi-threading shared address model

Threads communicate by sharing variables

Unintended sharing of data causes race conditions

A race condition is when the program's outcome changes as the threads are scheduled differently

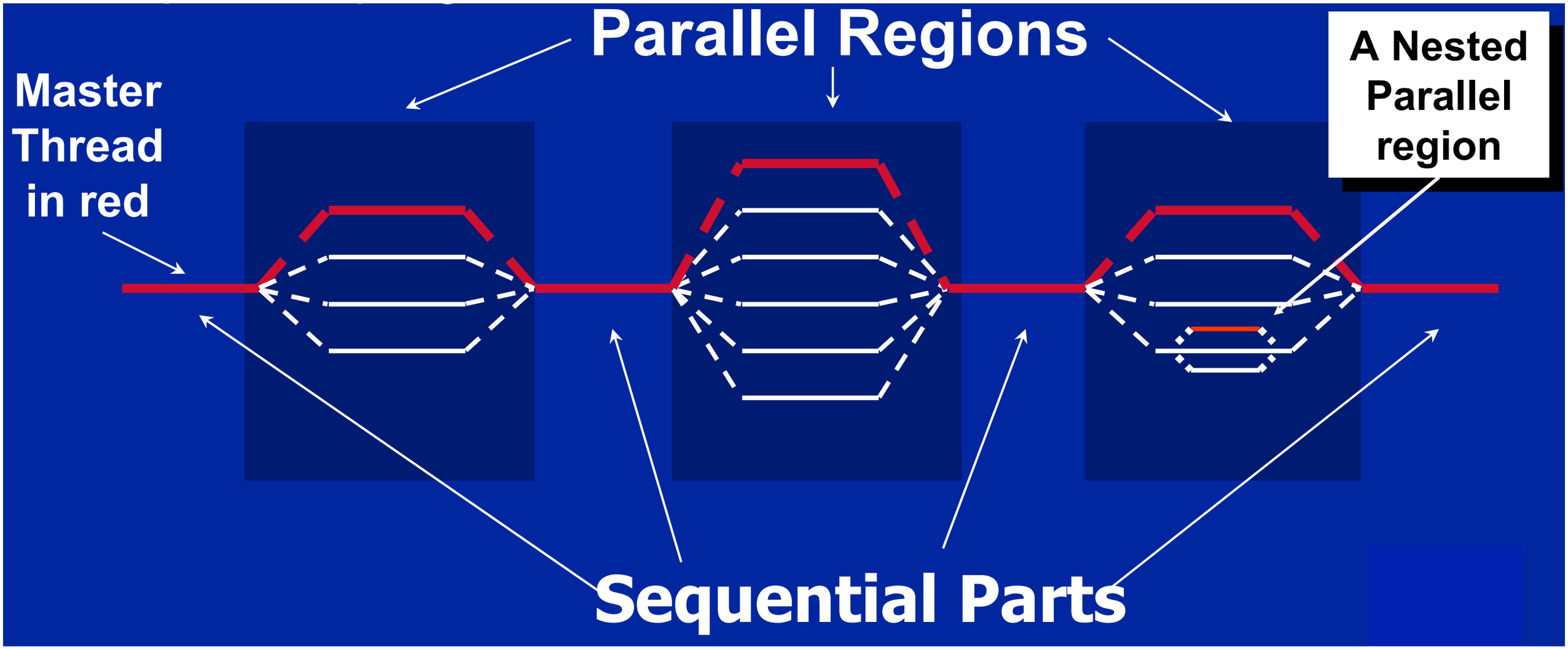
To control race conditions

Use synchronisation to protect data conflicts

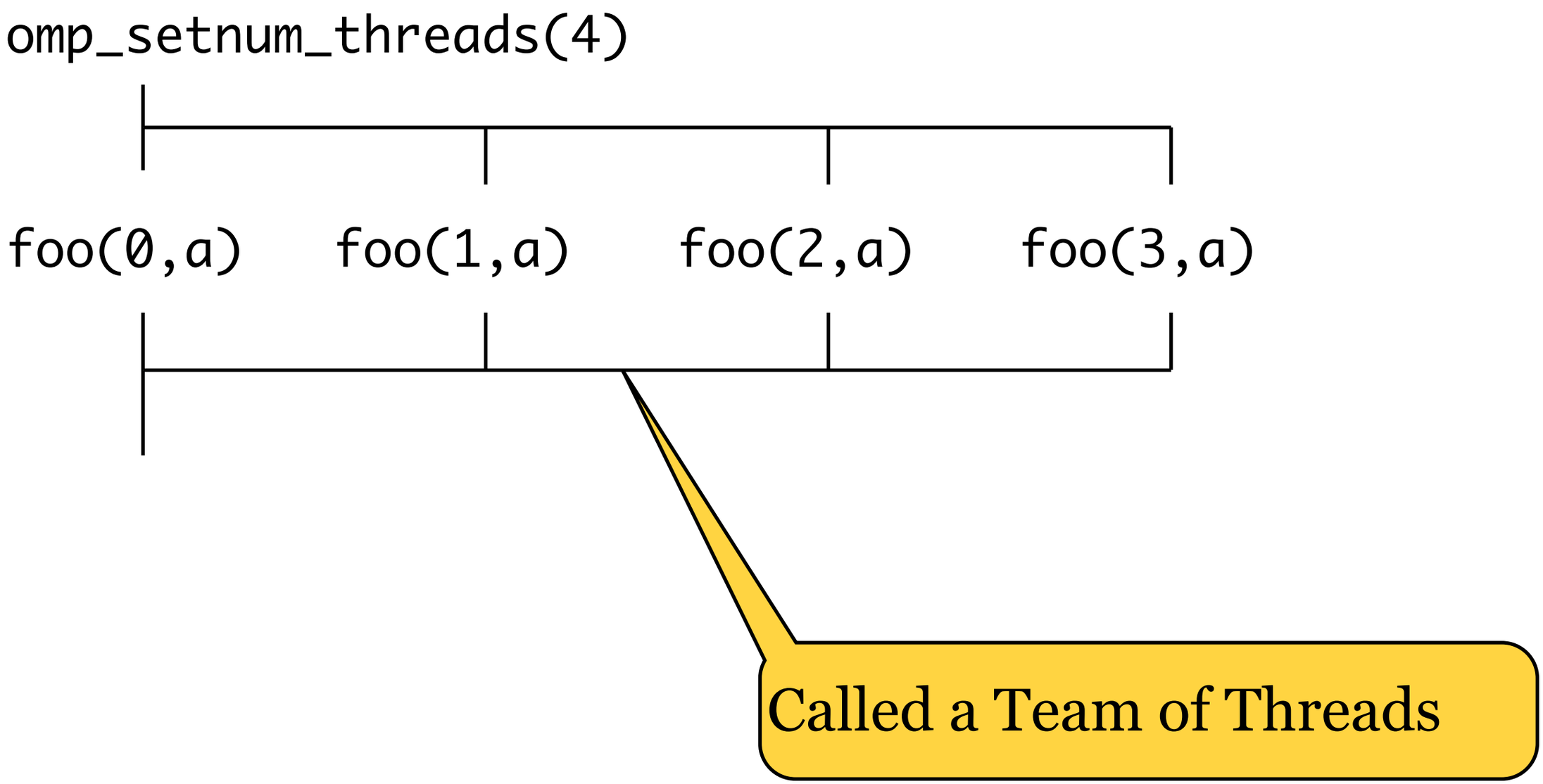
Synchronisation is expensive so

Change how data is accessed to minimise the need for synchronisation

Fork – Join Parallelism



Parallel Regions



Number of threads in a team

Established by either:

1. num_threads clause after the parallel directive, or
2. omp_set_num_threads() library routine being called,
3. the environment variable OMP_NUM_THREADS is defined
   in the order given or is system dependent if none of the above.

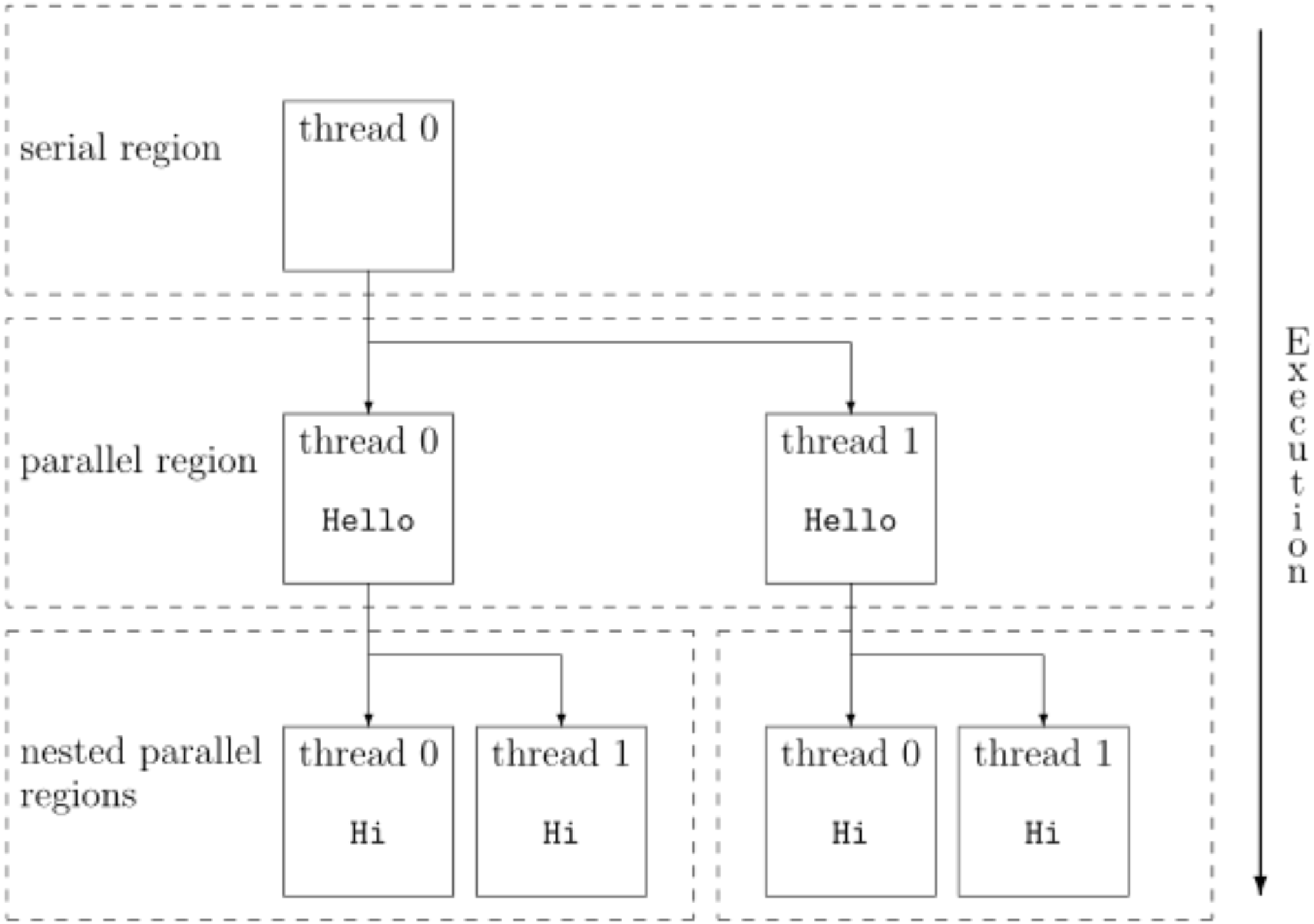
Number of threads available can also be altered automatically to achieve best use of system resources by a “dynamic adjustment” mechanism.

Nested Parallel Regions

It is possible to nest parallel regions into parallel regions. For example, if a thread in a parallel team encounters a new parallel region, then it creates a new team and it becomes the master thread of the new team. This second parallel region is called a nested parallel region.

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

Nested Parallel Regions



Synchronisation

High level synchronisation

critical

atomic

barrier

ordered

Low level synchronisation

flush

locks

Synchronisation Critical

The critical directive will only allow one thread execute the associated structured block. When one or more threads reach the critical directive:

```
#pragma omp critical name
    structured_block
```

```
!$OMP CRITICAL name
    ...
!$OMP END CRITICAL name
```

they will wait until no other thread is executing the same critical section (one with the same name), and then one thread will proceed to execute the structured block.

Note: name is optional. All critical sections with no name map to one undefined name.

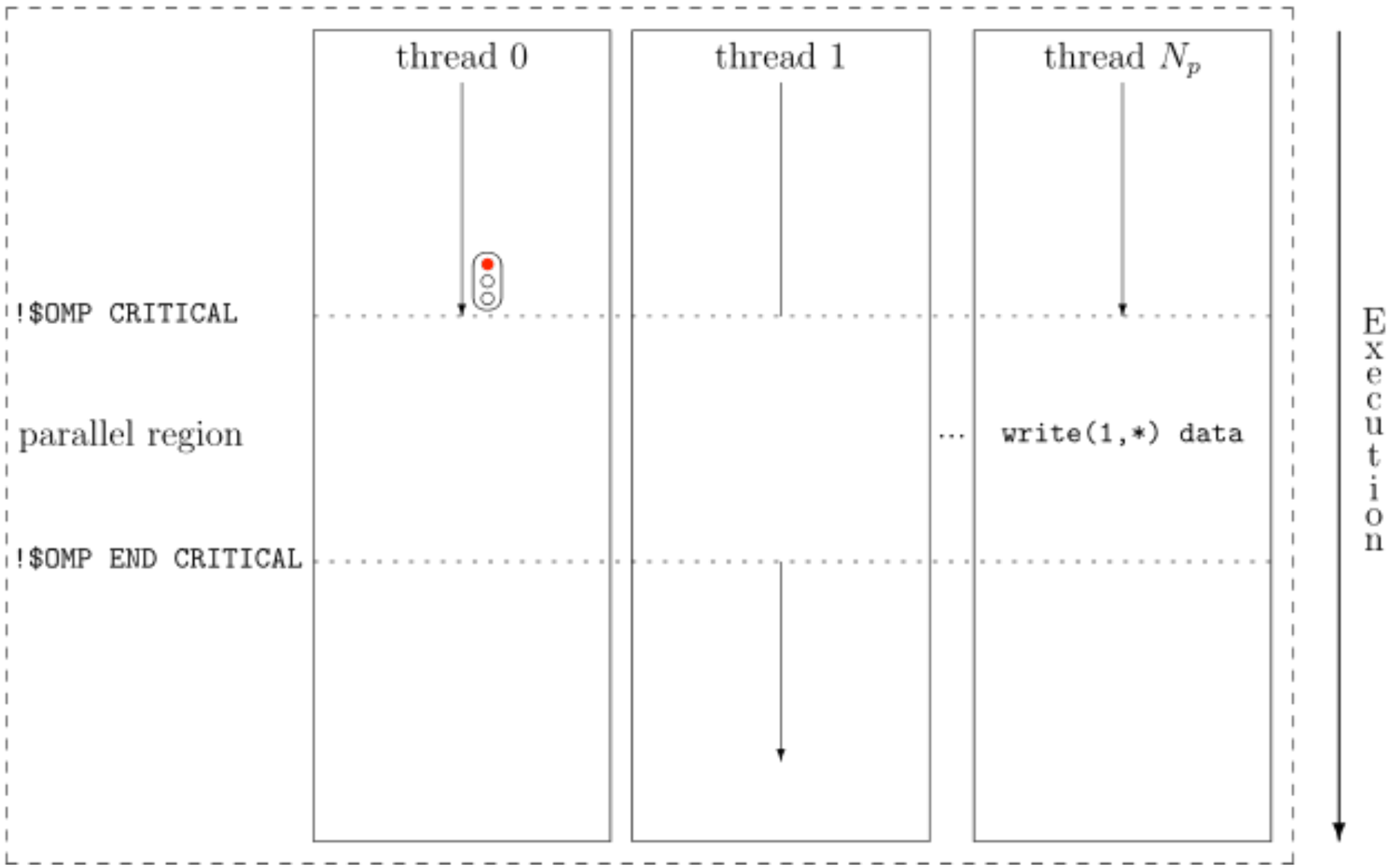
Critical

```
float res;
#pragma omp parallel
{
    float B; int i,id, nthrds;
    id = omp_get_thread_num();
    nthrds = omp_get_num_threads();
    for (i = id; i<niters; i+nthrds) {
        B = big_job(i);
#pragma omp critical
        consume(B, res);
    }
}
```

Threads wait their turn – only one at a time calls consume()

Critical

```
!$OMP CRITICAL write_file
  write(1,*) data
!$OMP END CRITICAL write_file
```



Synchronisation Constructs.

Atomic

The atomic directive

```
#pragma omp atomic
    expression_statement
```

```
!$OMP ATOMIC x
    ...
!$OMP END ATOMIC X
```

implements a critical section efficiently when the critical section simply updates a variable (adds one, subtracts one, or does some other simple arithmetic operation as defined by expression_statement).

Atomic

```
#pragma omp parallel
{
   double tmp, B;
   B = DOI_IT();
   tmp = big_ugly_function(B);
#pragma omp atomic
   X += tmp;
}
```

Atomic only applies to the update of a memory location. It only protects the read/update of X

SPMD vs Work-sharing

A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program
i.e. each thread executes redundantly executes the same code

How do we split up pathways through the code between threads within a team?

Called work-sharing

- Loop
- Sections
- Single
- Task – coming in OpenMP 3.0

Loop Constructs

The loop work-sharing construct splits up loop iterations among threads in a team

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

!$OMP DO
  do i = 1, 1000
    ...
  enddo
!$OMP END DO
```

Many different ways to do this

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

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

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

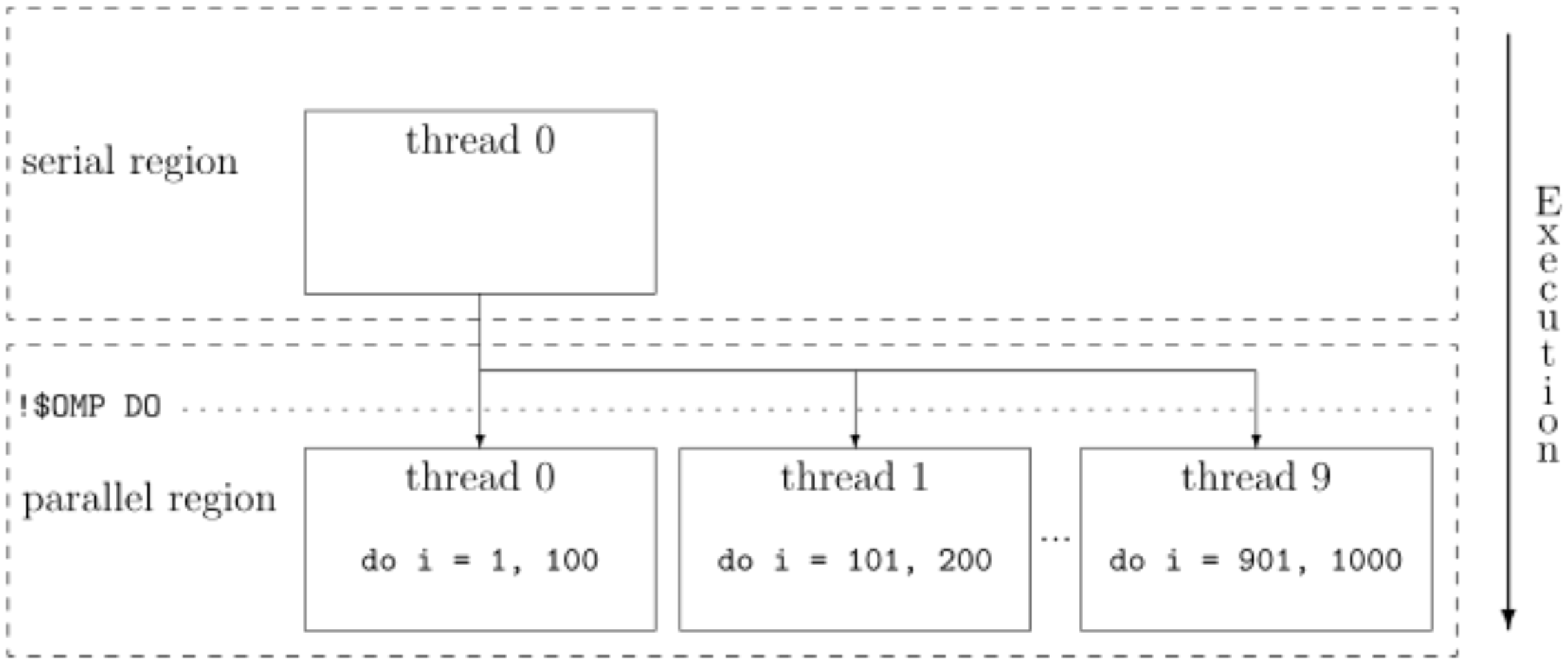
  GUIDED
  Iterations are dynamically assigned to threads in blocks as threads request them until no blocks remain to be assigned. Similar to DYNAMIC except that the block size decreases each time a parcel of work is given to a thread.

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

  AUTO
  The scheduling decision is delegated to the compiler and/or runtime system.

Loop Constructs

```
!$OMP DO
  do i = 1, 1000
     ...
  enddo
!$OMP END DO
```



Combining parallel/work-

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

Is equivalent to

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

Reduction

In large applications, you can often see the reduction operation inside hot loops.
Loops that reduce a collection of values to a single value are fairly common.

Reduction

How do we handle this case?

```
double ave = 0.0, A[MAX];
for (int i=0; i<MAX; i++) {
  ave += A[i];
}
```

```
double :: ave, A[MAX]
do i = 0, MAX
  ave = ave + A(i)
enddo
```

```
ave = ave/MAX;
```

```
ave = ave / MAX
```

We are combining values into a single accumulation variable

This is a very common situation - it is called reduction

Lock?

```
double ave = 0.0, A[MAX];
#pragma omp for
for (int i=0; i<MAX; i++) {
#pragma omp atomic
   ave += A[i];
}
ave = ave/MAX;
```

```
double :: ave, A[MAX]
!$OMP PARALLEL DO
do i = 0, MAX
!$OMP ATOMIC
   ave = ave + A(i)
enddo
!$OMP END DO
ave = ave / MAX
```

Lock?

```
double ave = 0.0, A[MAX];
#pragma omp for
for (int i=0; i<MAX; i++) {
#prag
  ave += A[i];
}
ave = ave/MAX;
```

```
double :: ave, A[MAX]
!$OMP PARALLEL DO
do i = 0, MAX
  ave = ave + A(i)
enddo
!$OMP END DO
ave = ave / MAX
```

LOCKS are EXPENSIVE!!

Many Loops

Thread 0:

```
double ave = 0.0, A[100];
double temp = 0.0;
for (int i=0;i<50;i++) {
  temp += A[i];
}
// lock ave
ave += temp
//unlock ave
```

Thread 1:

```
double ave = 0.0, A[100];
double temp = 0.0
for (int i=50;i<100;i++) {
  temp += A[i];
}
// lock ave
ave += temp
//unlock ave
```

Reduction

OpenMP reduction clause

```
reduction (op: list)
```

Inside a parallel or work-sharing construct

A local copy of each list variable is made

Compiler finds standard reduction expressions and uses them to update local copy

Local copies are reduced into a single value and combine with the global value

```
#pragma ... reduction (+:ave)
for (int i=0; i<MAX; i++) {
  ave += A[i];
}
ave = ave/MAX;
```

```
!$... REDUCTION(+:ave)
do i = 0, MAX
  ave = ave + A(i)
enddo
!$OMP END DO
ave = ave / MAX
```

Work-sharing Sections

The work-sharing sections construct directs the OpenMP compiler and runtime to distribute the identified sections of your application among threads in the team created for the parallel region.

Work-sharing Sections

```
#pragma omp parallel
{
    #pragma omp for
    for ( k = 0; k < m; k++ ) {
       x = fn1(k) + fn2(k);
    }
    #pragma omp sections private(y, z)
    {
      #pragma omp section
        { y = sectionA(x); fn7(y);
      #pragma omp section
        { z = sectionB(x); fn8(z);
    }
}
```

OpenMP first creates several threads

The iterations of the loop are divided among the threads

Once the loop is finished, the sections are divided among the threads so that each section is executed exactly once, but in parallel with the other sections.

Sections

The construct

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

cause the structured blocks to be shared among threads in team.
#pragma omp sections precedes the set of structured blocks.
#pragma omp section prefixes each structured block.

Barrier and Nowait

Barriers are a form of synchronisation method that OpenMP employs to synchronise threads.
Threads will wait at a barrier until all the threads in the parallel region have reached the same point.
You have been using implied barriers without realising it in the work-sharing for and work-sharing sections constructs.
At the end of the parallel, for, sections, and single constructs, an implicit barrier is generated by the compiler or invoked in the runtime library.
This barrier can be removed with the `nowait` clause

Nowait & sections

```
#pragma omp parallel
{
  #pragma omp for nowait
  for ( k = 0; k < m; k++ ) {
    fn10(k); fn20(k);
  }
#pragma omp sections private(y, z)
{
  #pragma omp section
    { y = sectionD(); fn70(y); }
  #pragma omp section
    { z = sectionC(); fn80(z); }
}
```

Nowait

In this example, since data is not dependent between the first work-sharing `for` loop and the second work-sharing `sections` code block, the threads that process the first work-sharing for loop continue immediately to the second work-sharing sections without waiting for all threads to finish the first loop.

```
#pragma omp parallel
{
  #pragma omp for nowait
  for ( k = 0; k < m; k++ ) {
    fn10(k); fn20(k);
  }
  #pragma omp sections private(y, z)
  {
    #pragma omp section
      { y = sectionD(); fn70(y); }
    #pragma omp section
      { z = sectionC(); fn80(z); }
  }
}
```

Synchronisation Barrier.

When a thread reaches the barrier

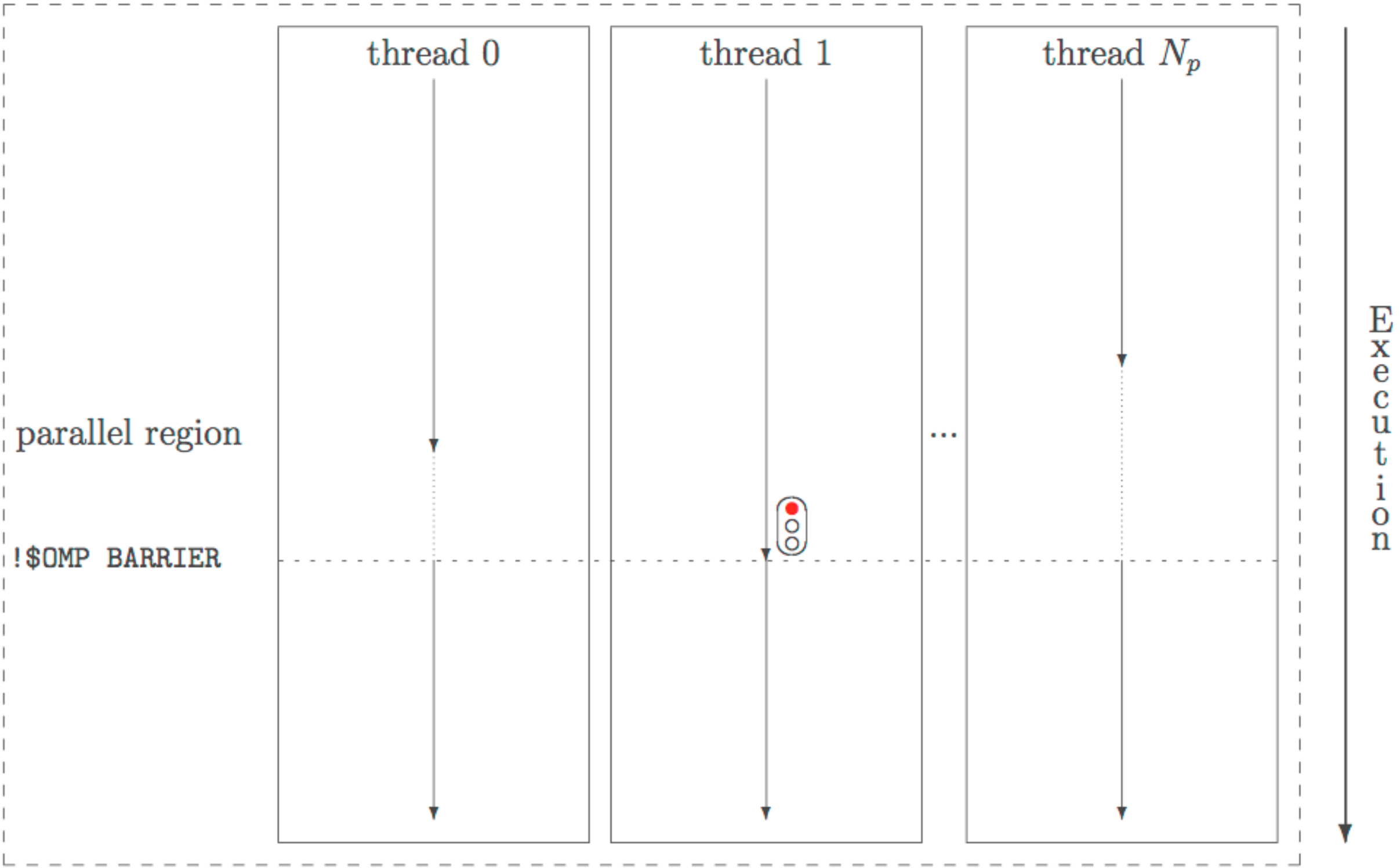
```
#pragma omp barrier                    !$OMP BARRIER
```

it waits until all threads have reached the barrier and then they all proceed together.

There two restrictions on the placement of barrier directive in a program.

1. All or none of threads in a team must be able to reach the barrier.
2. The barrier must be encountered in the same order by all threads in the team.

Barrier


thread 0
thread 1
thread N_p
parallel region
!$OMP BARRIER
...
Execution

Barrier

```
#pragma omp parallel shared(x, y, z) num_threads(2)
{
   int tid = omp_get_thread_num();
   if (tid == 0) {
      y = fn70(tid);
   }
   else {
      z = fn80(tid);
}
#pragma omp barrier
#pragma omp for
   for ( k = 0; k < 100; k++ ) {
      x[k] = y + z + fn10(k) + fn20(k);
   }
}
```

Barrier

The OpenMP code is to be executed by two threads; one thread writes the result to the variable y, and another thread writes the result to the variable z. Both y and z are read in the work-sharing for loop, hence, two flow dependences exist.
In order to obey the data dependence constraints in the code for correct threading, we add an explicit barrier pragma right before the work-sharing for loop to guarantee that the value of both y and z are ready for read.

In real applications, the barrier pragma is especially useful when all threads need to finish a task before any more work can be completed

```
#pragma omp parallel shared(x, y, z) num_threads(2)
{
   int tid = omp_get_thread_num();
   if (tid == 0) {
      y = fn70(tid);
   }
   else {
      z = fn80(tid);
}
#pragma omp barrier
#pragma omp for
   for ( k = 0; k < 100; k++ ) {
      x[k] = y + z + fn10(k) + fn20(k);
   }
}
```

Interleaving Single-thread and Multi-thread Execution

In large real-world applications, a program may consist of both serial and parallel code segments due to various reasons such as data dependence constraints and I/O operations.

A need to execute something only once by only one thread will certainly be required within a parallel region, especially because you are making parallel regions as large as possible to reduce overhead.

To handle the need for single-thread execution, OpenMP provides a way to specify that a sequence of code contained within a parallel section should only be executed one time by only one thread.

Interleaving Single-thread and Multi-thread Execution

```
#pragma omp parallel
{  // every thread calls this function
   int tid = omp_get_thread_num();
   // this loop is divided among the threads
   #pragma omp for nowait
   for ( k = 0; k < 100; k++ ) x[k] = fn1(k);

   #pragma omp master
   y = fn_input_only();// only the master threa

   #pragma omp barrier
   // again, this loop is divided among the thr
   #pragma omp for nowait
   for ( k = 0; k < 100; k++ ) x[k] = y + fn2(x

   #pragma omp single
   fn_single_print(y); // only one of threads

   #pragma omp master
   fn_print_array(x); // only one of threads
}
```

No implicit barrier at the end of the above loop causes all threads to synchronise here

Adding an explicit barrier to synchronise all threads to make sure x[0-99] and y is ready for use

The above loop does not have an implicit barrier, so threads will not wait for each other.
One thread – presumably the first one done with above, will continue and execute the following code.

The above single construct has an implicit barrier, so all threads synchronise here before printing x[]

Master Directive

The master directive:

```
#pragma omp master
    structured_block
```

causes the master thread to execute the structured block.

Different to those in the work sharing group in that there is no implied barrier at the end of the construct (nor the beginning). Other threads encountering this directive will ignore it and the associated structured block, and will move on.

Single

The directive

```
#pragma omp single
structured block
```

cause the structured block to be executed by one thread only.

Flush

A synchronisation point which causes thread to have a “consistent” view of certain or all shared variables in memory. All current read and write operations on the variables allowed to complete and values written back to memory but any memory operations in the code after flush are not started, thereby creating a “memory fence”. Format:

```
#pragma omp flush (variable_list)
```

Only applied to thread executing flush, not to all threads in the team.

Flush occurs automatically at the entry and exit of parallel and critical directives (and combined parallel for and parallel sections directives), and at the exit of for, sections, and single (if a no-wait clause is not present).

Flush

At first glance this seems unnecessary

In general writing/updating of shared variables is done in such a way that only one thread at a time access it

Only true in theory

how the OpenMP code does this is not specified

it leaves the door open for the OpenMP implementors to optimise the resulting code

Flush

Compilers are smart.

They routinely reorder instructions implementing a program

- this helps better exploit the functional units, keep machine busy, hide memory latencies, etc.

Compiler cannot move instructions

- past a barrier
- past a flush on variables

It can move them past a flush with a list of variables, so long as those variables are not accessed