Introduction to OpenMP

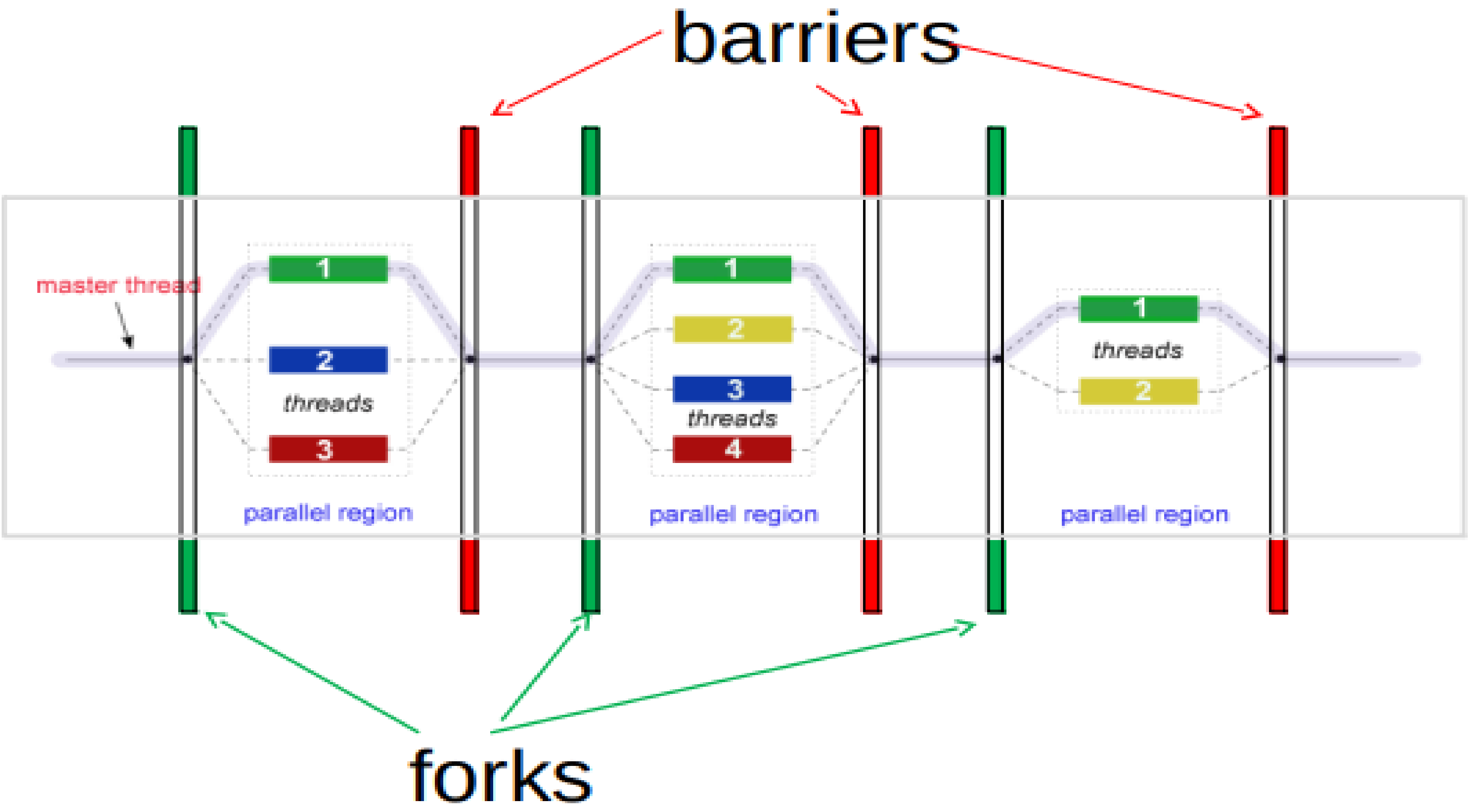
OpenMP introduction

- OpenMP stands for “Open Multi-Processing”
- OpenMP is a multi-vendor standard to perform shared-memory multithreading
- OpenMP uses fork-join model
- OpenMP is both directive and library based
- Each OpenMP thread has its own stack
- OpenMP probably gives you the biggest multithread benefit per amount of work you have to put in to using it

Threads

- A thread can be considered to be a lightweight process
- A more precise definition is that it is an execution path, a sequence of instructions, that is managed separately by the operating system scheduler as a unit
- Threads alleviate the overhead associated with the fork mechanism by copying only the bare essentials needed: the run-time stack
- The run-time stack cannot be shared between two threads

Fork-join model



OpenMP Consortium

What OpenMP Isn't

- OpenMP doesn't check for **data dependencies**, **data conflicts**, **deadlocks**, or **race conditions**. You are responsible for avoiding those yourself
- OpenMP doesn't check for **non-conforming** code sequences
- OpenMP doesn't **guarantee identical behavior** across vendors or hardware, or even between multiple runs on the same vendor's hardware
- OpenMP doesn't guarantee the **order in which threads execute**, just that they do execute
- OpenMP is not **overhead-free**
- OpenMP does not prevent you from writing code that triggers **cache performance problems**

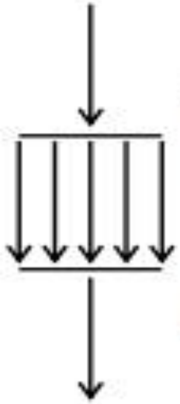
OpenMP: parallel regions

- A parallel region within a program is specified as

```
#pragma omp parallel [clause [[,] clause] …]
      Structured-block
```

- A team of threads is formed
- Thread that encountered the omp parallel directive becomes the **master thread** within this team
- **The structured-block is executed by every thread in the team**.
- At the end, there is an implicit **barrier**
- Only after **all threads have finished, the threads created by this directive are terminated and only the master resumes execution**
- A parallel region might be refined by a list of clause

OpenMP: parallel regions



```
#pragma omp parallel default(none)
{
            . . .
}
```

This creates a team of threads

Each thread then executes all lines of code in this block.

Think of it this way:

#pragma omp parallel default(none)

“Hello Word” Example/1

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

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


          printf("Hello World\n");


   return(0);
}
```

“Hello Word” Example/2

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

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

   #pragma omp parallel
   {
           printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

“Hello Word” Example/2

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

int main()
{
        #pragma omp parallel
        {
        printf("Hello World\n");
        } //End of parallel region
return(0);
}
```

```
"PHello.c" 12L, 168C written
sandhya@telnet:~/PP$ gcc PHello.c
sandhya@telnet:~/PP$ ./a.out
Hello World
sandhya@telnet:~/PP$
```

```
sandhya@telnet:~/PP$ gcc -fopenmp PHello.c
sandhya@telnet:~/PP$ ./a.out
Hello World
Hello World
Hello World
Hello World
sandhya@telnet:~/PP$
```

“Hello Word” Example/2

```
$ gcc -fopenmp hello.c


$ export OMP_NUM_THREADS=2
$ ./a.out
Hello World
Hello World

$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
$
```

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

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

   #pragma omp parallel
   {
          printf("Hello World\n");

   } // End of parallel region

   return(0);
}
```

```
sandhya@telnet:~/PP$ gcc -fopenmp PHello.c
sandhya@telnet:~/PP$ export OMP_NUM_THREADS=8
sandhya@telnet:~/PP$ ./a.out
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
sandhya@telnet:~/PP$
```

OpenMP Components

Directives

- Parallel region
- Worksharing constructs
- Tasking
- Offloading
- Affinity
- Error Handing
- SIMD
- Synchronization
- Data-sharing attributes

Runtime Environment

- Number of threads
- Thread ID
- Dynamic thread adjustment
- Nested parallelism
- Schedule
- Active levels
- Thread limit
- Nesting level
- Ancestor thread
- Team size
- Locking
- Wallclock timer

Environment Variable

- Number of threads
- Scheduling type
- Dynamic thread adjustment
- Nested parallelism
- Stacksize
- Idle threads
- Active levels
- Thread limit

“Hello Word” Example/3

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

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

#pragma omp parallel
  {
    int thread_id = omp_get_thread_num();
    int num_threads = omp_get_num_threads();

    printf("Hello World from thread %d of %d\n",
           thread_id, num_threads);
  }

   return(0);
}
```

Directives

Runtime Environment

“Hello Word” Example/3

```
$
$ gcc -fopenmp helloomp.c -o helloomp
$ ls helloomp
helloomp
```

```
:~$ ldd helloomp
linux-vdso.so.1 =>  (0x00007fff297c9000)
libgomp.so.1 => /usr/lib/x86_64-linux-gnu/libgomp.so.1 (0x00007f2b1de98000)
libpthread.so.0 => /lib/x86_64-linux-gnu/libpthread.so.0 (0x00007f2b1dc7b000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f2b1d8b1000)
libdl.so.2 => /lib/x86_64-linux-gnu/libdl.so.2 (0x00007f2b1d6ad000)
/lib64/ld-linux-x86-64.so.2 (0x00007f2b1e0c8000)
```

```
#pragma omp parallel
{
  int thread_id = omp_get_thread_num();
  int num_threads = omp_get_num_threads();

  printf("Hello World from thread %d of %d\n
          thread_id, num_threads);
}
```

Runtime library that provide the runtime environment

“Hello Word” Example/3

```
$
$ gcc -fopenmp helloomp.c -o helloomp
$ ls helloomp
helloomp
$
$ export OMP_NUM_THREADS=2
$  ./helloomp
Hello World from thread 1 of 2
Hello World from thread 0 of 2
$
$ export OMP_NUM_THREADS=4
$ ./helloomp
Hello World from thread 0 of 4
Hello World from thread 1 of 4
Hello World from thread 3 of 4
Hello World from thread 2 of 4
$
$ export OMP_NUM_THREADS=4
$ ./helloomp
Hello World from thread 1 of 4
Hello World from thread 2 of 4
Hello World from thread 3 of 4
Hello World from thread 0 of 4
```

```
#pragma omp parallel
{
  int thread_id = omp_get_thread_num();
  int num_threads = omp_get_num_threads();

  printf("Hello World from thread %d of %d\n"
         thread_id, num_threads);
}
```

Environment Variable

Environment Variable: it is similar to program arguments used to change the configuration of the execution without recompile the program.

NOTE: the order of print

OpenMP Programming Model

- Shared memory, thread-based parallelism
- –OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
- –A shared memory process consists of multiple threads.
- • Explicit Parallelism
- –Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- •Compiler directive based
- –Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code.

OpenMP controlling number of threads

- Once a program is compiled, the number of threads can be controlled using the following shell variables

  - **At the program level**, via the **omp_set_number_threads** function:

```
void omp_set_num_threads(int n)
```

  - At the pragma level, via the **num_threads** clause:

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

OpenMP controlling number of threads

- Asking how many cores this program has access to:

```
num = omp_get_num_procs( );
```

- Setting the number of available threads to the exact number of cores available:

```
omp_set_num_threads( omp_get_num_procs( ) );
```

- Asking how many OpenMP threads this program is using right now:

```
num = omp_get_num_threads( );
```

- Asking which thread number this one is:

```
me = omp_get_thread_num( )
```

Hello World in OpenMP

- Each thread has a unique integer “id”; master thread has “id” 0
- Other threads have “id” 1, 2, …
- OpenMP runtime function

`omp_get_thread_num()`

returns a thread’s unique “id”.

- What will be the programs output?

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

A sample OpenMP program

```
main( )
{
        omp_set_num_threads( 8 );
        #pragma omp parallel default(none)
        {
            printf( “Hello, World, from thread #%d ! \n” , omp_get_thread_num(  )   );
        }
        return 0;
}
```

A sample OpenMP program

First Run

```
Hello, World, from thread #6 !
Hello, World, from thread #1 !
Hello, World, from thread #7 !
Hello, World, from thread #5 !
Hello, World, from thread #4 !
Hello, World, from thread #3 !
Hello, World, from thread #2 !
Hello, World, from thread #0 !
```

Second Run

```
Hello, World, from thread #0 !
Hello, World, from thread #7 !
Hello, World, from thread #4 !
Hello, World, from thread #6 !
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #5 !
Hello, World, from thread #2 !
```

Third Run

```
Hello, World, from thread #2 !
Hello, World, from thread #5 !
Hello, World, from thread #0 !
Hello, World, from thread #7 !
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #4 !
Hello, World, from thread #6 !
```

Fourth Run

```
Hello, World, from thread #1 !
Hello, World, from thread #3 !
Hello, World, from thread #5 !
Hello, World, from thread #2 !
Hello, World, from thread #4 !
Hello, World, from thread #7 !
Hello, World, from thread #6 !
Hello, World, from thread #0 !
```

OpenMP: parallel loops

- A parallel for loops are declared as

  #pragma omp for [clause [[,] clause] ...]

  for-loops

Each *for loop* among ***for-loops*** associated with the ***omp for directive*** must be in the **canonical form**

- ✓ the loop variable is made **private** to each thread in the team and must be either (unsigned) **integer or a pointer**,
- ✓ the loop variable should not be modified during the execution of any iteration;
- ✓ the condition in the for loop must be a **simple relational expression**,
- ✓ the increment in the for loop must specify a change by constant additive expression;
- ✓ the number of iterations of all associated loops must be known before the start of the outermost for loop.

OpenMP: parallel loops

A clause is a specification that further describes a parallel loop, for instance,

- ✓ **collapse(integer)** specifies how many outermost *for loops* of *for-loops* are associated with the directive, and thus parallelized together;

- ✓ **nowait** eliminates the implicit barrier and thus synchronization at the end of *for-loops*.

Parallelizing Loops with Independent Iterations

- The **omp parallel for** directive in line 6 specifies that the for loop must be executed in parallel
- The iterations of the *for loop* will be divided among the threads
- Once all iterations have been executed, all threads in the team are synchronized at the implicit barrier at the end of the parallel for loop
- All slave threads are terminated
- Finally, the execution proceeds sequentially, and the master thread terminates the program by executing return 0

```
1  #include <stdio.h>
2  #include <omp.h>
3
4  int main (int argc, char *argv[]) {
5    int max; sscanf (argv[1], "%d", &max);
6    #pragma omp parallel for
7      for (int i = 1; i <= max; i++)
8        printf ("%d: %d\n", omp_get_thread_num (), i);
9    return 0;
10 }
```

Listing 3.3 Printing out all integers from 1 to *max* in no particular order.

Program does not specify how the iterations should be divided among threads.

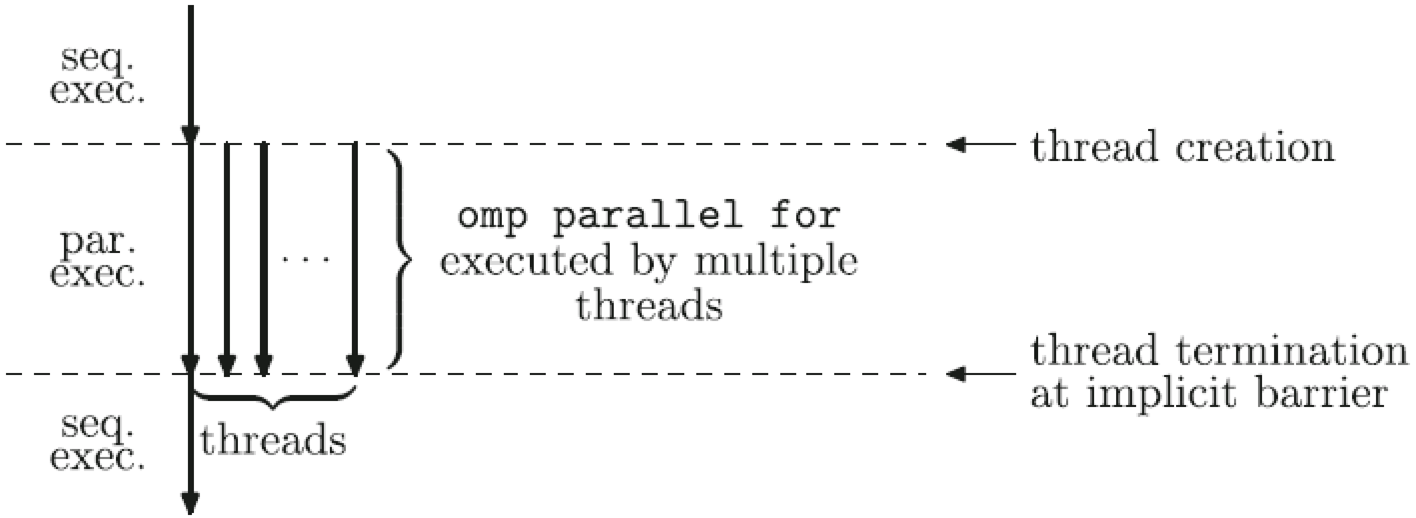


Fig. 3.5 Execution of the program for printing out integers as implemented in Listing 3.3

Divide for-loop for parallel sections

```
for (int i=0; i<8; i++) x[i]=0; //run on 4 threads
```

```
#pragma omp parallel
        {
        int numt=omp_get_num_thread();
        int id = omp_get_thread_num(); //id=0, 1, 2, or 3
        for (int i=id; i<8; i +=numt)
                                x[i]=0;

        }
```

// Assume number of threads=4

Thread 0

```
Id=0;
x[0]=0;
X[4]=0;
```

Thread 1

```
Id=1;
x[1]=0;
X[5]=0;
```

Thread 2

```
Id=2;
x[2]=0;
X[6]=0;
```

Thread 3

```
Id=3;
x[3]=0;
X[7]=0;
```

Use pragma parallel for

```
for (int i=0; i<8; i++) x[i]=0;
```

```
#pragma omp parallel for
{
            for (int i=0; i<8; i++)
                        x[i]=0;
}
```

System divides loop iterations to threads

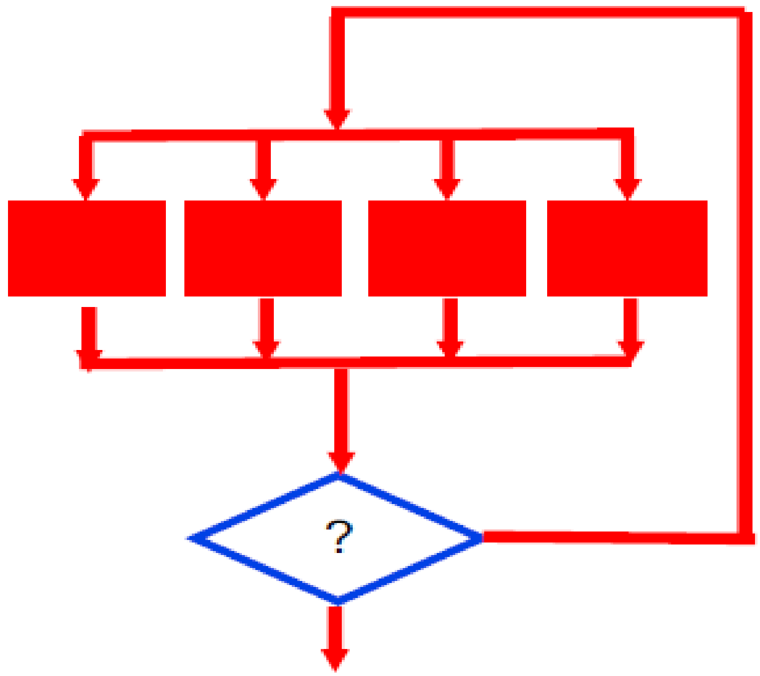
Id=0; **x[0]=0;** X[4]=0;	Id=1; **x[1]=0;** X[5]=0;	Id=2; **x[2]=0;** X[6]=0;	Id=3; **x[3]=0;** X[7]=0;

Programming Model – Parallel Loops

- Requirement for parallel loops
  - No data dependencies (reads/write or write/write pairs) between iterations!

- Preprocessor calculates loop bounds and divide iterations among parallel threads

```
#pragma omp parallel for

for( i=0; i < 25; i++ )
{

  printf("Foo");

}
```



Example

```
for (i=0; i<max; i++) zero[i] = 0;
```

- Breaks *for loop* into chunks, and allocate each to a separate thread
  - e.g. if `max` = 100 with 2 threads:
    assign 0-49 to thread 0, and 50-99 to thread 1
- Must have relatively simple “shape” for an OpenMP-aware compiler to be able to parallelize it
  - Necessary for the run-time system to be able to determine how many of the loop iterations to assign to each thread
- No premature exits from the loop allowed
  - i.e. No `break`, `return`, `exit`, `goto` statements

In general, don’t jump outside of any pragma block

OpenMP: parallel loops

- The loop variable is made private to each thread in the team and must be either (unsigned) integer or a pointer
- The loop variable should not be modified during the execution of any iteration
- The condition in the for loop must be a simple relational expression
- The increment in the for loop must specify a change by constant additive expression
- The number of iterations of all associated loops must be known before the start of the outermost for loop

OpenMP: parallel loops

- The start and terminate conditions must have compatible types
- Neither the start nor the terminate conditions can be changed during the execution of the loop
- The index can only be modified by the changed expression (i.e., not modified inside the loop itself)
- You cannot use a break or a goto to get out of the loop
- There can be no inter-loop data dependencies such as:

A[i]=a[i-1]+1;

OpenMP: parallel loops

```
x[ 0 ] = 0.;
y[ 0 ] *= 2.;
for( int i = 1; i < N; i++ )
{
        x[ i ] = x[ i-1 ] + 1.;
        y[ i ] *= 2.;
}
```

Because of the loop dependency, this whole thing is not parallelizable:

```
x[ 0 ] = 0.;
for( int i = 1; i < N; i++ )
{
        x[ i ] = x[ i-1 ] + 1.;
}

#pragma omp parallel for shared(y)
for( int i = 0; i < N; i++ )
{
        y[ i ] *= 2.;
}
```

But, it *can* be broken into one loop that is not parallelizable, plus one that is:

OpenMP: parallel loops

```
for( int i = 1; i < N; i++ )
{
            for( int j = 0; j < M; j++ )
            {
                        . . .
            }
}
```

Ah-ha – trick question. You put it on both!

How many for-loops to collapse into one loop

```
#pragma omp parallel for collapse(2)
for( int i = 1; i < N; i++ )
{
            for( int j = 0; j < M; j++ )
            {
                        . . .
            }
}
```

OpenMP: Data sharing

- Various data sharing clauses might be used in omp parallel directive to specify whether and how data are shared among threads:
  - **shared(list)** specifies that each variable in the list is shared by all threads in a team, i.e., all threads share the same copy of the variable
  - **private(list)** specifies that each variable in the list is private to each thread in a team, i.e., each thread has its own local copy of the variable
  - **firstprivate(list)** is like private but each variable listed is initialized with the value it contained when the parallel region was encountered
  - **lastprivate(list)** is like private but when the parallel region ends each variable listed is updated with its final value within the parallel region

OpenMP: data sharing

```
float x = 0.;
#pragma omp parallel for ...
for( int i = 0; i < N; i++ )
{
        x = (float) i;
        float y = x*x;
        << more code... >
}
```

i and **y** are automatically *private* because they are defined within the loop.

Good practice demands that **x** be explicitly declared to be shared or private!

private(x)
Means that each thread will get its own version of the variable

shared(x)
Means that all threads will share a common version of the variable

default(none)
I recommend that you include this in your OpenMP for-loop directive. This will force you to explicitly flag all of your externally-declared variables as *shared* or *private*. Don't make a mistake by leaving it up to the default!

OpenMP: data sharing

- If not specified otherwise,
  - Automatic variables declared outside a parallel construct are shared by default
  - Automatic variables declared within a parallel construct are private
  - Static and dynamically allocated variables are shared

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

int main()
{
    int a = 5;
#pragma omp parallel num_threads(5)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region: %d\n", a);
    }
    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

Microsoft Visual Studio Debug Console

```
Parallel region: 15
Parallel region: 35
Parallel region: 45
Parallel region: 55
Parallel region: 25
Outside parallel region: 55

D:\DSCA\Parallel_Computing\Lab_Programs\Examp
ith code 0.
To automatically close the console when debug
le when debugging stops.
Press any key to close this window . . .
```

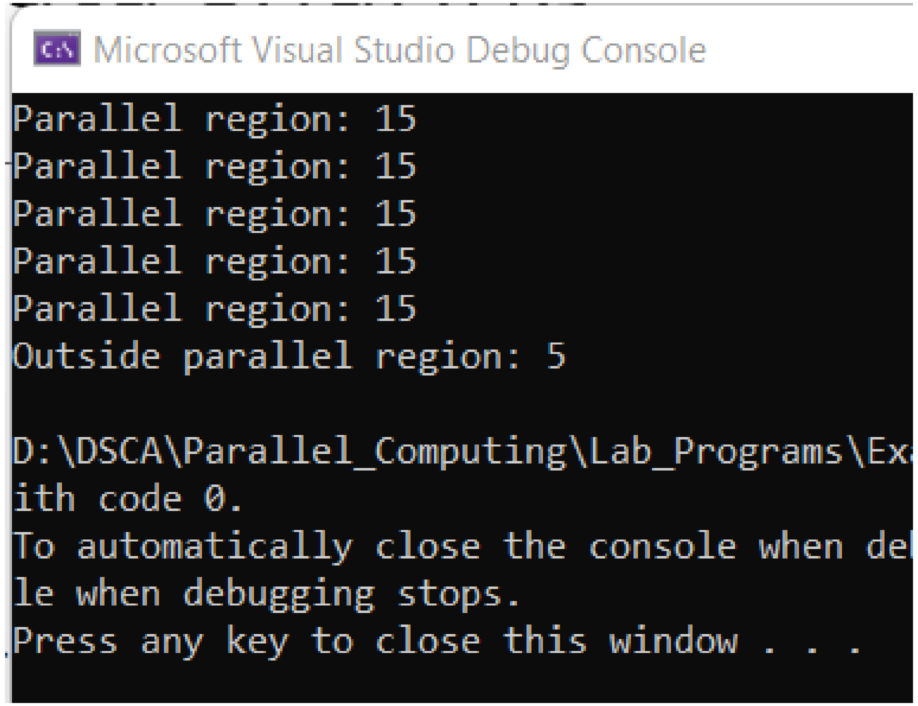
OpenMP: data sharing

- Private clause

```
int main()
{
    int a =5;
    #pragma omp parallel num_threads(4) private(a)
    {

        int b = 10;
        a = a + b;
        printf("%d \n  ", a);
    }


    return 0;
}
```

OpenMP: data sharing

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

int main()
{
    int a = 5;
    #pragma omp parallel num_threads(5) firstprivate(a)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region: %d\n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```



OpenMP: data sharing

```
int main()
{
    int a =5;
    #pragma omp parallel num_threads(4) lastprivate(a)
    {
        int b = 10;
        a = a + b;
        printf("%d \n  ", a);
    }
    printf("outside: %d\n", a);

    return 0;
}
```

OpenMP: data sharing

```
#pragma omp parallel for private(i) lastprivate(a)
  for (i=0; i<n; i++)
   {
        a = i+1;
        printf("Thread %d has a value of a = %d for i = %d\n",
               omp_get_thread_num(),a,i);
   } /*-- End of parallel for --*/

  printf("Value of a after parallel for: a = %d\n",a);
```

```
Thread 0 has a value of a = 1 for i = 0
Thread 0 has a value of a = 2 for i = 1
Thread 2 has a value of a = 5 for i = 4
Thread 1 has a value of a = 3 for i = 2
Thread 1 has a value of a = 4 for i = 3
Value of a after parallel for: a = 5
```

OpenMP: data sharing

- **Default clause:**
  - The default clause is used to give variables a default data-sharing attribute
  - For example, default(shared) assigns the shared attribute to all variables referenced in the construct.
  - The default(private) clause makes all variables private by default
    - Not supported in C/C++ (supported only in Fortran)
  - This clause is most often used to define the data-sharing attribute of most of the variables in a parallel region
  - If **default(none)** is specified instead, the programmer is forced to specify a data-sharing attribute for each variable in the construct

OpenMP: data sharing

```
int main()
{
    int a = 5;
    #pragma omp parallel num_threads(5) default(none)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region:a= %d \n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

⚠	C6993	Code analysis ignores OpenMP constructs; analyzing single-threaded code.	Example_Data_Sharing	Source.cpp	7
✖	C3052	'a': variable reference occurs under a default(none) clause and must have explicitly specified data sharing attributes	Example_Data_Sharing	Source.cpp	10

OpenMP: data sharing

```c
int main()
{
    int a = 5;
    #pragma omp parallel num_threads(5) default(none) shared(a)
    {
        int b = 10;
        a = a + b;
        printf("Parallel region:a= %d \n", a);
    }

    printf("Outside parallel region: %d\n", a);
    return 0;
}
```

```
Parallel region:a= 15
Parallel region:a= 25
Parallel region:a= 45
Parallel region:a= 35
Parallel region:a= 55
Outside parallel region: 55

D:\DSCA\Parallel_Computing\Lab_Programs\Example_Dat
ith code 0.
To automatically close the console when debugging st
le when debugging stops.
Press any key to close this window . . .
```

OpenMP: Nowait clause

- The **nowait** clause allows the programmer to fine-tune a program’s performance
- This clause overrides that feature of OpenMP
- If it is added to a construct, the barrier at the end of the associated construct will be suppressed
- **Note:** however, that the barrier at the end of a parallel region cannot be suppressed
- One can try to identify places where a barrier is not needed and insert the **nowait** clause

OpenMP: Nowait clause

```c
int main()
{
#pragma omp parallel
    {
    #pragma omp for
        for (int i = 0; i <5; i++)
        {
            printf("first loop i= %d\n", i);
        }

        printf("outside\n");
    }


    return 0;
}
```

```
first loop i= 2
first loop i= 3
first loop i= 0
first loop i= 4
first loop i= 1
outside
outside
outside
outside
outside
outside
outside
outside

D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug\Now
To automatically close the console when debugging stops, enable To
le when debugging stops.
Press any key to close this window . . .
```

```c
int main()
{
#pragma omp parallel
    {
    #pragma omp for nowait
        for (int i = 0; i <5; i++)
        {
            printf("first loop i= %d\n", i);
        }

        printf("outside\n");
    }


    return 0;
}
```

```
first loop i= 1
outside
first loop i= 3
outside
first loop i= 0
outside
first loop i= 2
outside
outside
first loop i= 4
outside
outside
outside

D:\DSCA\Parallel_Computing\Lab_Programs\Nowait_trial\x64\Debug
To automatically close the console when debugging stops, enabl
le when debugging stops.
Press any key to close this window . . .
```

OpenMP: Schedule Clause

- The schedule clause is supported on the loop construct only
- It is used to control the manner in which loop iterations are distributed over the threads, which can have a major impact on the performance of a program
- The syntax is: **schedule(kind [,chunk_size])**
- The granularity of this workload distribution is a chunk
- **Note** that the chunk_size parameter need not be a constant

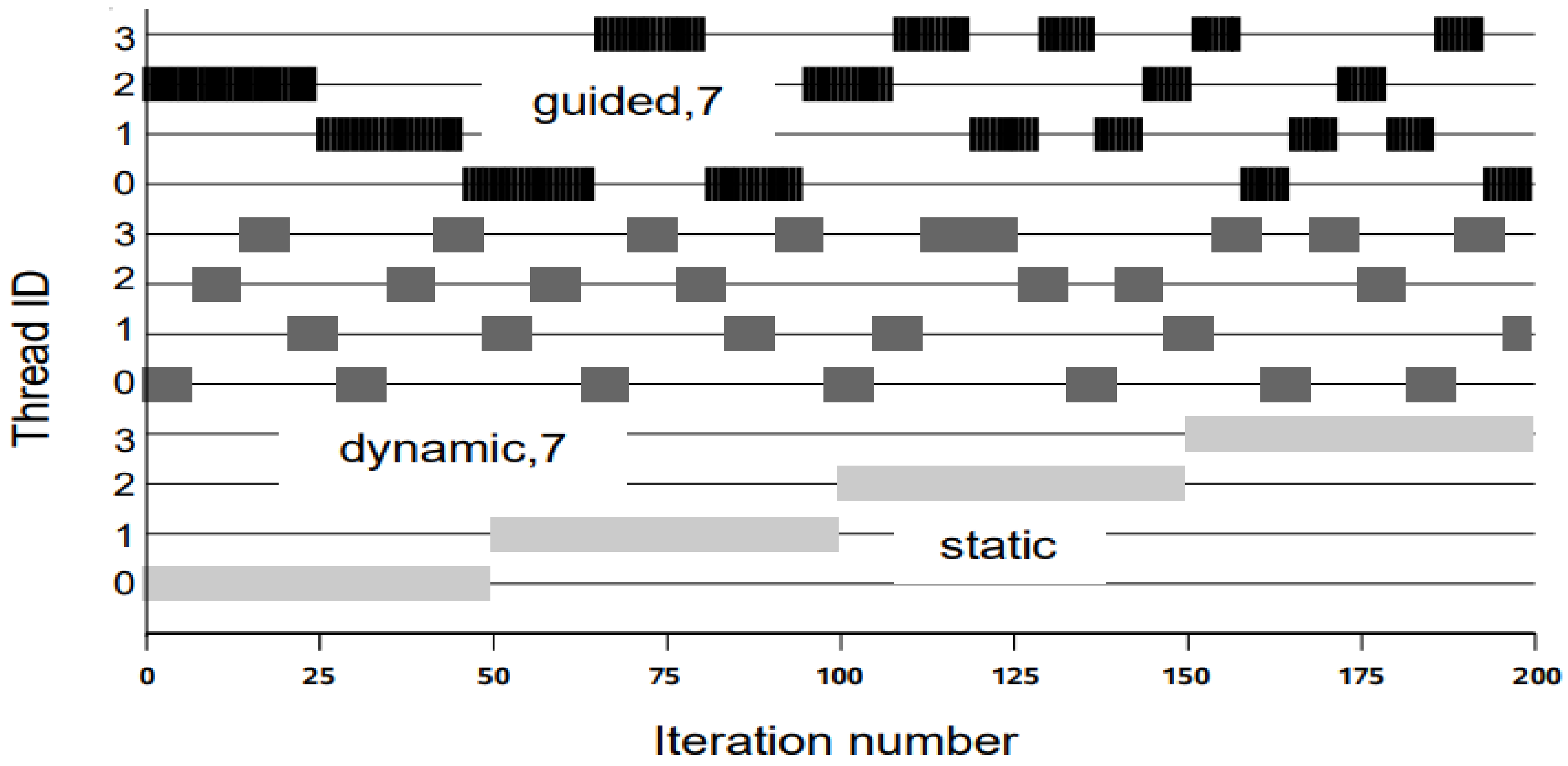
OpenMP: Schedule Clause

- **Static:**
  - Iterations are divided into chunks of size chunk size
  - The chunks are assigned to the threads statically in a **round-robin** manner, in the **order of the thread number**
  - The last chunk to be assigned may have a smaller number of iterations
  - When no chunk size is specified, the iteration space is divided into chunks that are approximately equal in size
- **Dynamic:**
  - The iterations are assigned to threads as the threads request them
  - The thread executes the chunk of iterations (controlled through the chunk size parameter), then requests another chunk until there are no more chunks to work on
  - The last chunk may have fewer iterations than chunk size. When no chunk size is specified, it defaults to 1

OpenMP: Schedule Clause

- **Guided**
  - The iterations are assigned to threads as the threads request them
  - The thread executes the chunk of iterations (controlled through the chunk size parameter), then requests another chunk until there are no more chunks to work on.
  - For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1.
  - For a chunk size of “k” (k > 1), chunks do not contain fewer than k iterations
- **Runtime**
  - If this schedule is selected, the decision regarding scheduling kind is made at run time

OpenMP: Schedule Clause



OpenMP: Synchronization Constructs

- An algorithm may require us to manipulate the actions of multiple threads to ensure that updates to a shared variable occur in a certain order
- it may simply need to ensure that two threads do not simultaneously attempt to write a shared object
- Synchronization Constructs can be used when implicit barrier are not sufficient enough

Synchronization Constructs: Barrier

- A barrier is a point in the execution of a program where threads wait for each other
  - no thread in the team of threads it applies to may proceed beyond a barrier until all threads in the team have reached that point
- the compiler automatically inserts a barrier at the end of the construct
- Two important restrictions apply to the barrier construct:
  - Each barrier must be encountered by all threads in a team, or by none at all
  - The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in the team.

```
#pragma omp barrier
```

Synchronization Constructs: Barrier

```c
#pragma omp parallel
    {
        int TID;
        TID = omp_get_thread_num();
        if (TID < omp_get_num_threads() / 2)
        {
            for (int i = 0; i < 9000; i++)
            {
            }
            printf("Sleeping\n");
        }
        printf("Reached barrier....waiting\n");
        #pragma omp barrier
        printf("After barrier....\n");
    }
    return 0;
}
```

```
Sleeping
Sleeping
Reached barrier....waiting
Reached barrier....waiting
Reached barrier....waiting
Reached barrier....waiting
Reached barrier....waiting
Sleeping
Reached barrier....waiting
Reached barrier....waiting
Sleeping
Reached barrier....waiting
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....
After barrier....

D:\DSCA\Parallel_Computing\Lab_Programs\Barrier_exa
.
To automatically close the console when debugging s
le when debugging stops.
Press any key to close this window . . .
```

Synchronization Constructs: Barrier

- The most common use for a barrier is to avoid a **data race condition**
- Inserting a barrier between the **writes to** and **reads from a shared variable** guarantees that the accesses are appropriately ordered

Synchronization Constructs: Ordered

- allows one to execute a structured block within a parallel loop in sequential order
- This is sometimes used, for instance, to enforce an ordering on the printing of data computed by different threads
- help determine whether there are any data races in the associated code

```
#pragma omp ordered
            structured block
```

Synchronization Constructs: Ordered

```c
//Ordered Clause
int n = 8;
int a[8] = {};
#pragma omp parallel for default(none) ordered schedule(runtime)  shared(n,a)
for (int i = 0; i < n; i++)
{
    int TID = omp_get_thread_num();
    printf("Thread %d updates a[%d]\n", TID, i);
    a[i] += i;
    #pragma omp ordered
    {
        printf("Thread %d prints value of a[%d] = %d\n", TID, i, a[i]);
    }
}
```

```
Thread 3 updates a[3]
Thread 6 updates a[6]
Thread 7 updates a[7]
Thread 0 updates a[0]
Thread 0 prints value of a[0] = 0
Thread 1 updates a[1]
Thread 1 prints value of a[1] = 1
Thread 5 updates a[5]
Thread 4 updates a[4]
Thread 2 updates a[2]
Thread 2 prints value of a[2] = 2
Thread 3 prints value of a[3] = 3
Thread 4 prints value of a[4] = 4
Thread 5 prints value of a[5] = 5
Thread 6 prints value of a[6] = 6
Thread 7 prints value of a[7] = 7

D:\DSCA\Parallel_Computing\Lab_Programs\Barr
.
To automatically close the console when debu
le when debugging stops.
Press any key to close this window . . .
```

Synchronization Constructs: Critical

- The critical construct provides a means to ensure that multiple threads do not attempt to update the same shared data simultaneously
- The associated code is referred to as a critical region, or a critical section
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name

```
#pragma omp critical [(name)]
    structured block
```

Synchronization Constructs: Critical

```c
int sum = 1;
int n = 8;
int a[8] = {};
#pragma omp parallel shared(n,a,sum)
{
    int TID = omp_get_thread_num();
    int sumLocal = 1;
    #pragma omp for
    for (int i = 0; i < n; i++)
        sumLocal += a[i];
    #pragma omp critical
    {
        sum += sumLocal;
        printf("TID=%d: sumLocal=%d sum = %d\n", TID, sumLocal, sum);
    }
}
printf("Value of sum after parallel region: %d\n", sum);
```

```
TID=5: sumLocal=1 sum = 2
TID=7: sumLocal=1 sum = 3
TID=6: sumLocal=1 sum = 4
TID=3: sumLocal=1 sum = 5
TID=2: sumLocal=1 sum = 6
TID=4: sumLocal=1 sum = 7
TID=1: sumLocal=1 sum = 8
TID=0: sumLocal=1 sum = 9
Value of sum after parallel region: 9

D:\DSCA\Parallel_Computing\Lab_Programs\Barrier_example
To automatically close the console when debugging stops
le when debugging stops.
Press any key to close this window . . .
```