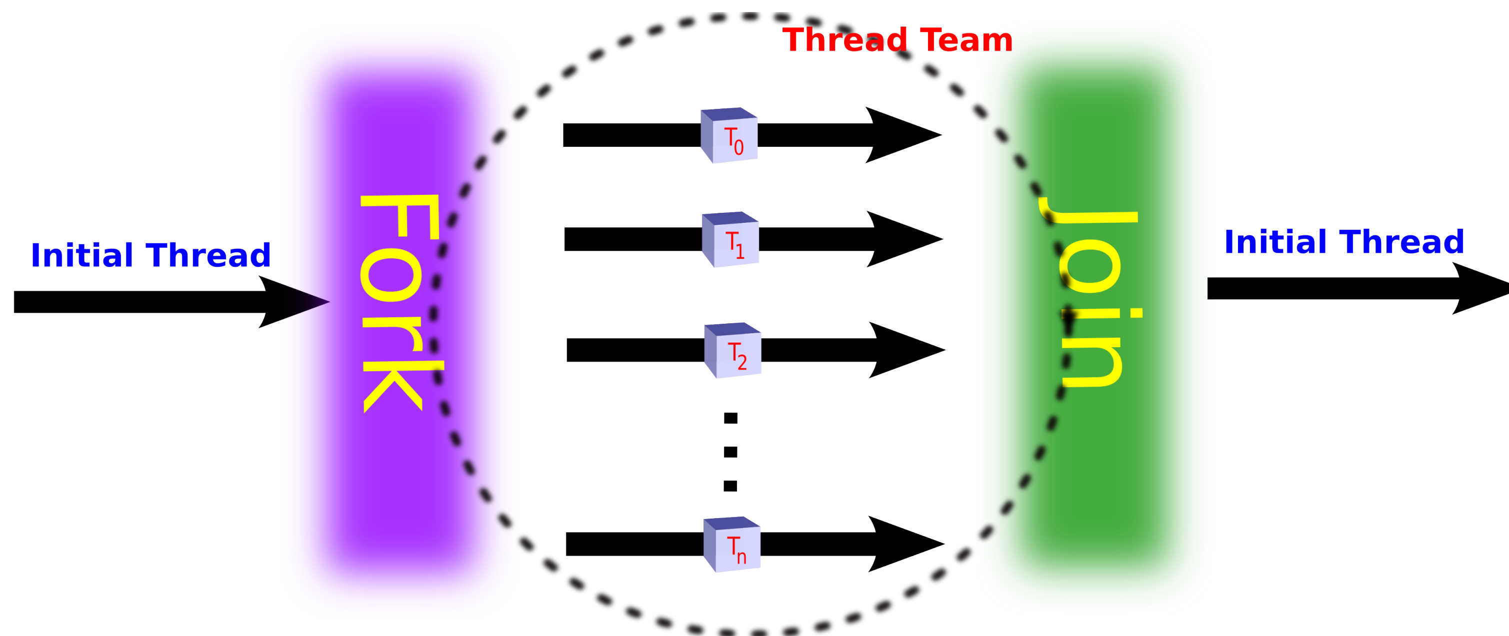
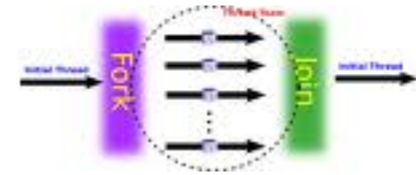
OpenMP Language Features



Agenda

- The parallel construct
- Work-sharing
- Data-sharing
- Synchronization
- Interaction with the execution environment
- More OpenMP clauses
- Advanced OpenMP constructs



The fork/join execution model

1. An OpenMP program starts as a single thread (*master thread*)
2. Additional threads are created when the master hits a parallel region.
3. When all threads have finished the parallel region, the new threads are given back to the runtime system.
4. The master continues after the parallel region.

All threads are *synchronized* at the end of a parallel region via a *barrier*.

OpenMP region

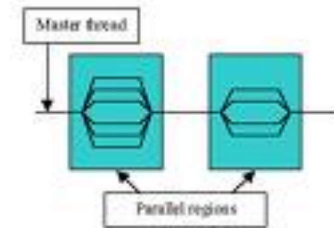
An OpenMP region of code consists of all code encountered during a specific instance of the execution of an OpenMP construct. A region includes any code in called routines.

In other words, a region encompasses all the code that is in the *dynamic* extent of a construct.

Structured block

Most OpenMP constructs apply to a *structured block* – a block of one or more statements with one entry point at the top and one point of exit at the bottom.

It is OK to have an `exit()` within the structured block.



Parallel region

```
#pragma omp parallel [clause[[,] clause]...]
      structured block
```

Figure 4.1: **Syntax of the parallel construct in C/C++** – The parallel region implicitly ends at the end of the structured block. This is a closing curly brace (}) in most cases.

The construct is used to specify computations that should be executed in parallel. Although it ensures that computations are performed in parallel it does not distribute the work among the threads in a team. In fact, if the programmer does not specify any work sharing, the work will be replicated.

Example of parallel region

```
#pragma omp parallel
   {
     printf("The parallel region is executed by thread %d\n",
        omp_get_thread_num());

     if ( omp_get_thread_num() == 2 ) {
        printf("  Thread %d does things differently\n",
               omp_get_thread_num());
     }
   }  /*-- End of parallel region --*/
```

Figure 4.3: **Example of a parallel region** – All threads execute the first `printf` statement, but only the thread with thread number 2 executes the second one.

Example output

```
The parallel region is executed by thread 0
The parallel region is executed by thread 3
The parallel region is executed by thread 2
  Thread 2 does things differently
The parallel region is executed by thread 1
```

Figure 4.4: **Output of the code shown in Figure 4.3** – Four threads are used in this example.

Parallel regions

OpenMP Team := Master + Workers

A parallel region is a block of code executed by all threads simultaneously

- The master thread always has ID 0
- Thread adjustment (if enabled) is only done before entering a parallel region
- Parallel regions can be nested, but support for this is implementation dependent
- An “if” clause can be used to guard the parallel region; in case the condition evaluates to “false”, the code is executed sequentially

Clauses supported by the parallel region

if *(scalar-expression)*
num_threads *(integer-expression)*
private *(list)*
firstprivate *(list)*
shared *(list)*
default(none | shared)
reduction *(operator:list)*
copyin *(list)*

Work-sharing

A work-sharing construct divides the execution of the enclosed code among the members of the team; in other words: they split the work.

Functionality	Syntax in C/C++
Distribute iterations over the threads	**#pragma omp for**
Distribute independent work units	**#pragma omp sections**
Only one thread executes the code block	**#pragma omp single**
Distribute tasks over the threads	**#pragma omp task**

Parallel loop

```
#pragma omp for [clause[[,] clause]...]
    for-loop
```

Figure 4.7: **Syntax of the loop construct in C/C++** – Note the lack of curly braces. These are implied with the construct.

```
for ( init-expr ; var relop b ; incr-expr )
```

Figure 4.9: **Format of C/C++ loop** – The OpenMP loop construct may be applied only to this kind of loop nest in C/C++ programs.

init-expr: initialization of the loop counter, *var*
relop: one of <, <=, >, >=.
incr-expr: one of ++, --, +=, -=, or a form such as *var* = *var* + *incr*.

Work-sharing in a parallel region

```
int main() {
    int a[100], i;
    #pragma omp parallel
    {
        #pragma omp for
        for (i = 0; i < 100; i++)
            a[i] = i;
    }
}
```

Parallel loop

- The iterations of the `for`-loop are distributed to the threads
- The scheduling of the iterations is determined by one of the scheduling strategies: **static**, **dynamic**, **guided**, and **runtime**.
- There is no synchronization at the beginning.
- All threads of the team synchronize at an implicit barrier at the end of the loop, unless the **nowait** clause is specified.
- The loop variable is by default private. It must not be modified in the loop body.

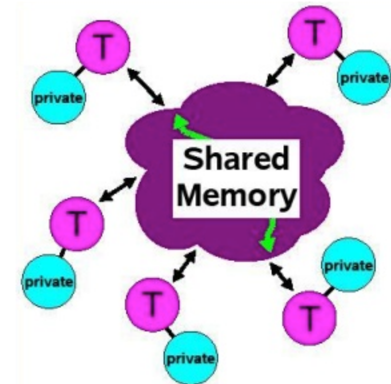
Shared and private data

Shared data are accessible by all threads.
A reference a[5] to a shared array accesses the same address in all threads.

Private data are accessible only by a single thread (the owner). Each thread has its own copy.

The default is shared.

Data-sharing attributes



- **Shared**
  - There is only one instance of the data
  - All threads can can read and write the data simultaneously, unless protected through a specific OpenMP construct
  - All changes made are visible to all threads, but not necessarily immediately, unless enforced.

- **Private**
  - Each thread has a copy of the data
  - No other thread can access this data
  - Changes are only visible to the thread owning the data

Private clause for parallel loop

```
int main() {
    int a[100], i, t;
    #pragma omp parallel
    {
        #pragma omp for private(t)
        for (i = 0; i < 100; i++) {
            t = f(i);
            a[i] = t;
        }
    }
}
```

Work-sharing loop

```
#pragma omp parallel shared(n) private(i)
{
   #pragma omp for
   for (i=0; i<n; i++)
      printf("Thread %d executes loop iteration %d\n",
             omp_get_thread_num(),i);
} /*-- End of parallel region --*/
```

Figure 4.10: **Example of a work-sharing loop** – Each thread executes a subset of the total iteration space $i = 0, \ldots, n-1$.

Example output

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

Figure 4.11: **Output from the example shown in Figure 4.10** – The example is executed for $n = 9$ and uses four threads.

Clauses supported by the loop construct

private *(list)*
firstprivate *(list)*
lastprivate *(list)*
reduction *(operator:list)*
ordered
schedule *(kind[,chunk_size)]*
nowait

The sections construct

```
#pragma omp sections [clause[[,] clause]...]
    {
    [#pragma omp section ]
        structured block
    [#pragma omp section
        structured block ]
    ...
    }
```

Figure 4.14: **Syntax of the sections construct in C/C++** – The number of sections controls, and limits, the amount of parallelism. If there are "n" of these code blocks, at most "n" threads can execute in parallel.

- Each section is executed once by a thread.
- Threads that have finished their section wait at the implicit barrier at the end of the sections construct.

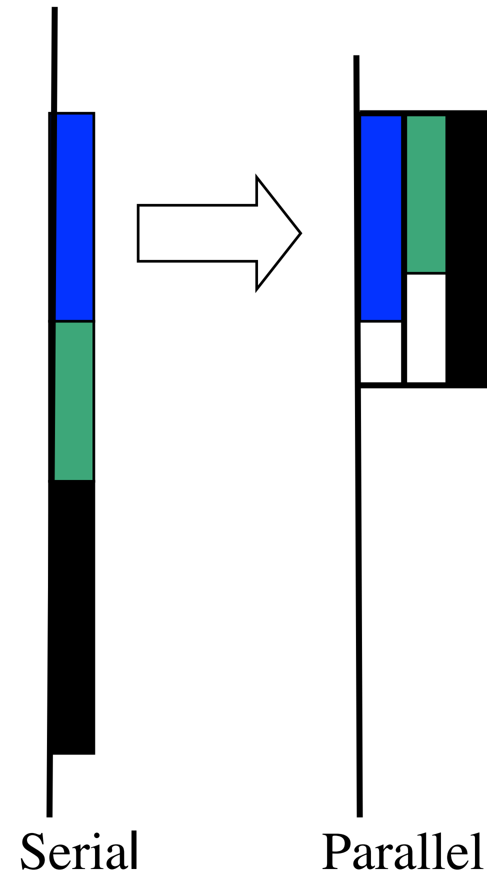
Parallel sections example

```
int main() {
    int a[100], b[100], i;
    #pragma omp parallel private(i)
    {
        #pragma omp sections
        {
            #pragma omp section
            for (i = 0; i < 100; i++)
                a[i] = 100;
            #pragma omp section
            for (i = 0; i < 100; i++)
                b[i] = 200;
        }
    }
}
```

Advantage of parallel sections

Independent sections of code can execute concurrently – reduce execution time

```
#pragma omp parallel sections
{
   #pragma omp section
   funcA();
   #pragma omp section
   funcB();
   #pragma omp section
   funcC();
}
```



Clauses supported by the sections construct

private *(list)*
firstprivate *(list)*
lastprivate *(list)*
reduction *(operator:list)*
nowait

The single and master constructs

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

```
#pragma omp master
        structured block
```

The master or single region enforces that only a single thread executes the enclosed code within a parallel region.

A master region is only executed by the master thread while the single region can be executed by any thread.

A master region is skipped by all other threads while all threads are synchronized at the end of a single region.

Single construct example

```
#pragma omp parallel shared(a,b) private(i)
{
   #pragma omp single
   {
      a = 10;
      printf("Single construct executed by thread %d\n",
             omp_get_thread_num());
   }
   /* A barrier is automatically inserted here */

   #pragma omp for
   for (i=0; i<n; i++)
       b[i] = a;

} /*-- End of parallel region --*/

   printf("After the parallel region:\n");
   for (i=0; i<n; i++)
       printf("b[%d] = %d\n",i,b[i]);
```

Figure 4.22: **Example of the single construct** – Only one thread initializes the shared variable **a**.

Combined parallel works-sharing constructs

<table>
<tr><th>Full version</th><th>Combined construct</th></tr>
<tr><td><pre>#pragma omp parallel
{
  #pragma omp for
    for-loop
}</pre></td><td><pre>#pragma omp parallel for
  for-loop</pre></td></tr>
<tr><td><pre>#pragma omp parallel
{
#pragma omp sections
{
  [#pragma omp section]
    structured block
  [#pragma omp section
    structured block]
  . . .
}
}</pre></td><td><pre>#pragma omp parallel sections
{
  [#pragma omp section]

    structured block
  [#pragma omp section
    structured block]
  . . .
}</pre></td></tr>
</table>

Figure 4.29: **Syntax of the combined constructs in C/C++** – The combined constructs may have a performance advantage over the more general **parallel** region with just one work-sharing construct embedded.

The shared clause

```
#pragma omp parallel for shared(a)
  for (i=0; i<n; i++)
   {
       a[i] += i;
   } /*-- End of parallel for --*/
```

Figure 4.31: **Example of the shared clause** – All threads can read from and write to vector a.

The private clause

```
#pragma omp parallel for private(i,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 --*/
```

Figure 4.32: **Example of the private clause** – Each thread has a local copy of variables `i` and `a`.

The lastprivate clause

```
#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);
```

Figure 4.34: **Example of the lastprivate clause** – This clause makes the sequentially last value of variable a accessible outside the parallel loop.

Assume n = 5:

```
Value of a after parallel for: a = 5
```

The firstprivate clause

```
for(i=0; i<vlen; i++) a[i] = -i-1;

indx = 4;
#pragma omp parallel default(none) firstprivate(indx) \
                     private(i,TID) shared(n,a)
   {
      TID = omp_get_thread_num();

      indx += n*TID;
      for(i=indx; i<indx+n; i++)
         a[i] = TID + 1;
   } /*-- End of parallel region --*/

printf("After the parallel region:\n");
for (i=0; i<vlen; i++)
    printf("a[%d] = %d\n",i,a[i]);
```

Figure 4.37: **Example using the firstprivate clause** – Each thread has a pre-initialized copy of variable `indx`. This variable is still private, so threads can update it individually.

The nowait clause

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

Figure 4.40: **Example of the nowait clause in C/C++** – The clause ensures that there is no barrier at the end of the loop.

The schedule clause

`schedule`(*kind [, chunk_size]*)

The schedule clause specifies how iterations of the loop are assigned to the team of threads.

The granularity of this workload is a *chunk*, a contiguous, non-empty subset of the iteration space.

The most straightforward schedule is *static*, which is the default on many OpenMP compilers. Both *dynamic* and *guided* schedules are useful for handling poorly balanced and unpredictable workloads.

Static scheduling

Schedule kind	Description
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. Each thread is assigned at most one chunk.

Static scheduling

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.

Guided scheduling

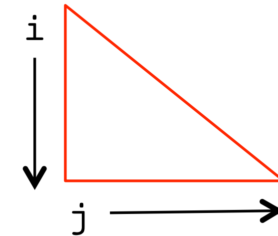
guided	The iterations are assigned to threads as the threads request them. The thread executes the chunk of iterations (controlled through the *chunk_size* parameter), then requests another chunk until there are no more chunks to work on. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations, divided by the number of threads, decreasing to 1. For a *chunk_size* of "k" ($k > 1$), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (with a possible exception for the last chunk to be assigned, which may have fewer than k iterations). When no *chunk_size* is specified, it defaults to 1.

Runtime scheduling

runtime	If this schedule is selected, the decision regarding scheduling kind is made at run time. The schedule and (optional) chunk size are set through the OMP_SCHEDULE environment variable.

Schedule example

Unbalanced workload




```
#pragma omp parallel for default(none) schedule(runtime) \
                         private(i,j) shared(n)
  for (i=0; i<n; i++)
   {
      printf("Iteration %d executed by thread %d\n",
          i, omp_get_thread_num());
      for (j=0; j<i; j++)
          system("sleep 1");
   } /*-- End of parallel for --*/
```

Figure 4.43: **Example of the schedule clause** – The runtime variant of this clause is used. The `OMP_SCHEDULE` environment variable is used to specify the schedule that should be used when executing this loop.

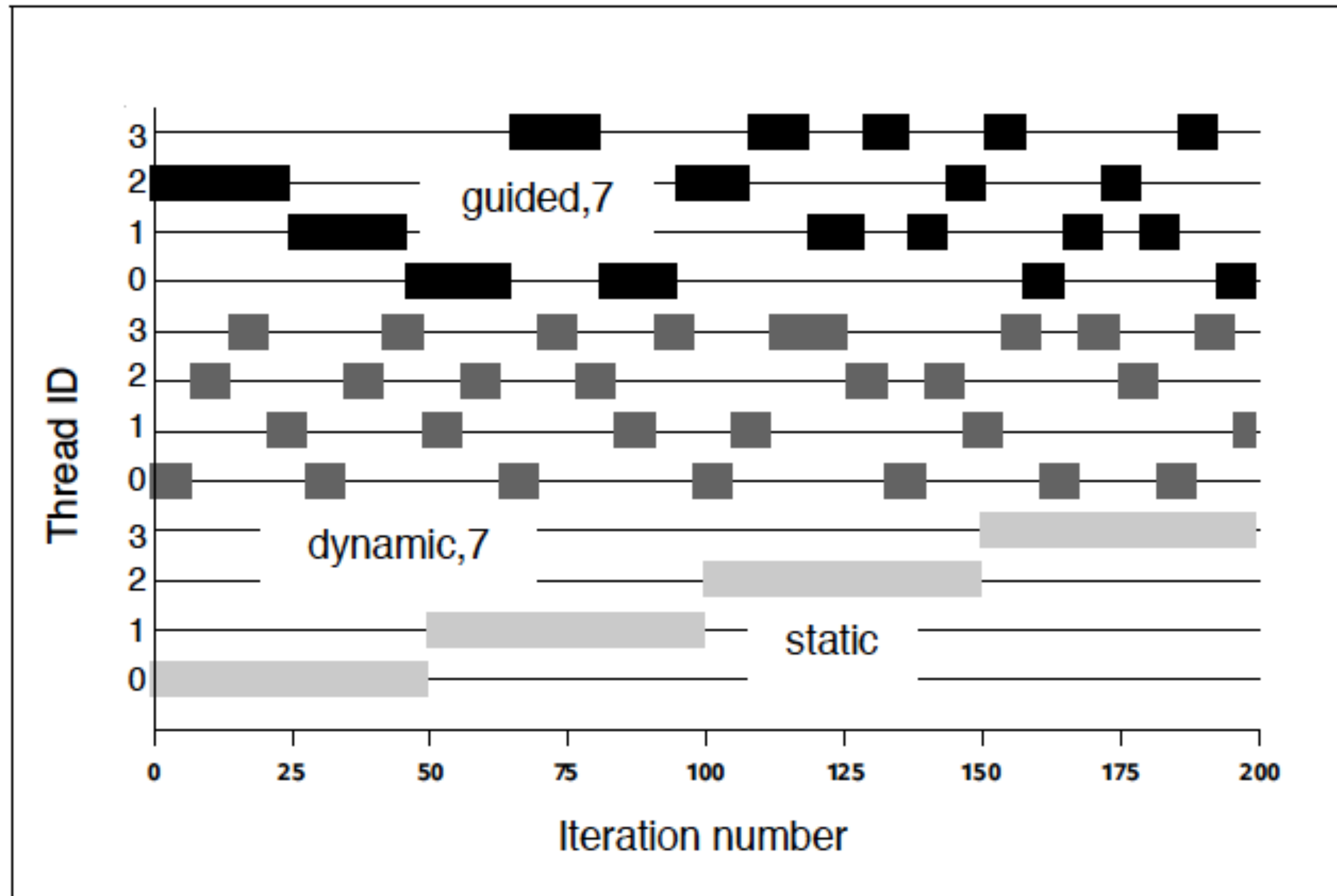


Figure 4.44: **Graphical illustration of the schedule clause** – The mapping of iterations onto four threads for three different scheduling algorithms for a loop of length $n = 200$ is shown. Clearly, both the `dynamic` and the `guided` policy give rise to a much more dynamic workload allocation scheme.

The barrier construct

```
#pragma omp barrier
```

Figure 4.45: **Syntax of the barrier construct in C/C++** – This construct binds to the innermost enclosing parallel region.

The barrier synchronizes all threads in a team.

When encountered each thread waits until all threads in that team have reached this point.

Many OpenMP constructs imply a barrier.

The most common use for a barrier is for avoiding a race condition.

The ordered construct

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

Figure 4.49: **Syntax of the ordered construct in C/C++** – This construct is placed within a parallel loop. The structured block is executed in the sequential order of the loop iterations.

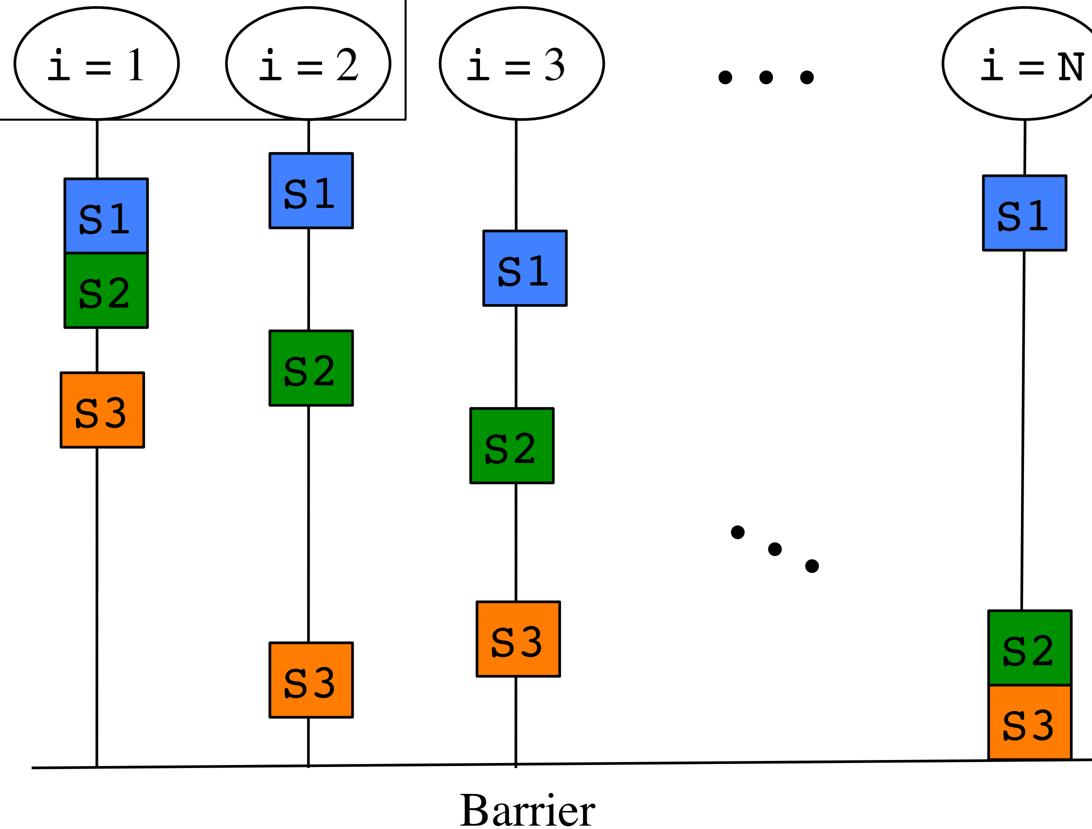
An ordered construct ensures that the code within the associated structured block is executed in sequential order.

An ordered clause has to be added to the parallel region in which this construct appears. For example,

```
#pragma omp parallel for ordered
```

Example with ordered clause

```
#pragma omp parallel for ordered
for (i = 1; i <= N; i++) {
    S1;
    #pragma omp ordered
    { S2; }
    S3;
}
```



The critical construct

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

Figure 4.51: **Syntax of the critical construct in C/C++** – The structured block is executed by all threads, but only one at a time executes the block. Optionally, the construct can have a name.

A thread waits at the beginning of the critical section until no other thread is executing a critical section with the same name.

All unnamed critical sections map to the same name.

Example with critical clause

```
#pragma omp parallel shared(n) private(TID)
   {
      TID = omp_get_thread_num();
      #pragma omp critical (print_tid)
      {
        printf("I am thread %d\n",TID);
      }
   } /*-- End of parallel region --*/
```

Figure 4.57: **Avoiding garbled output** – A critical region helps to avoid intermingled output when multiple threads print from within a parallel region.

The atomic construct

```
#pragma omp atomic
        statement
```

Figure 4.59: **Syntax of the atomic construct in C/C++** – The statement is executed by all threads, but only one thread at a time executes the statement.

An atomic construct ensures that a specific memory location is updated atomically (without interference).

The **atomic** construct may only be used together with an expression statement in C/C++, which essentially means that it applies a simple, binary operation such as an increment or decrement to the value on the left-hand side. The supported operations are: +, *, -, /, &, ^, |, <<, >>.

Locking library routines

```
void omp_func_lock (omp_lock_t *lck)
```

Figure 4.63: **General syntax of locking routines in C/C++** – For a specific routine, *func* expresses its functionality; *func* may assume the values `init, destroy, set, unset, test`. The values for nested locks are `init_nest, destroy_nest, set_nest, unset_nest, test_nest`.

Locks can be hold by only one thread at a time.

There are two types of locks: *simple locks*, which may not be locked if already in locked state, and *nestable locks*, which may be locked multiple times by the same thread. Nestable lock variables are declared with the special type `omp_nest_lock_t`.

Nestable locks

Unlike simple locks, nestable locks may be set multiple times by a single thread.

Each set operation increments a lock counter.

Each unset operation decrements the lock counter.

If the lock counter is 0 after an unset operation, the lock can be set by another thread.

General procedure to use locks

1. Define (simple or nested) lock variables.

2. Initialize the lock via a call to `omp_init_lock`.

3. Set the lock using `omp_set_lock` or `omp_test_lock`. The latter checks whether the lock is actually available before attempting to set it.

4. Unset a lock after the work is done via a call to `omp_unset_lock`.

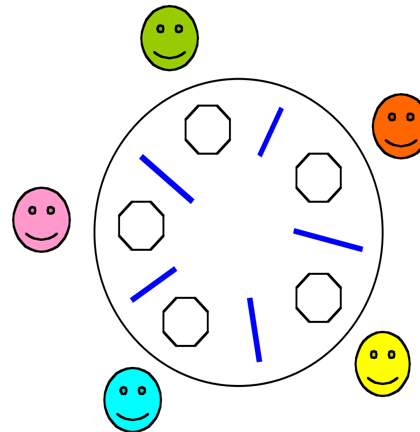
5. Remove the lock association by a call to `omp_destroy_lock`.

Lock example

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

int main() {
    omp_lock_t lock;
    omp_init_lock(&lock);
    #pragma omp parallel shared(lock)
    {
        int id = omp_get_thread_num();
        omp_set_lock(&lock);
        printf("My thread number is %d\n", id);
        omp_unset_lock(&lock);
        while (!omp_test_lock(&lock))
            other_work(id);   // lock not obtained
        real_work(id);     // lock obtained
        omp_unset_lock(&lock);
    }
    omp_destroy_lock(&lock);
}
```

The dining philosophers

Five philosopher are sitting around at a round table in deep thoughts. But of course, from time to time they must have something to eat. In front of each philosopher is a bowl of rice. Between each pair of philosophers is one chopstick. Before a philosopher can eat he must have two chopsticks, one taken from the left, and one taken from the right.



The philosophers must find some way to share chopsticks such that they all get to eat.

```c
#include <unistd.h>

#define N 5
int meals[N];
omp_lock_t chop_stick[N];

void think(int id) {
    printf("Philosopher #%d is thinking\n", id);
    sleep(rand() % 10 / 1000.0);
    printf("Philosopher #%d is hungry\n", id);
}

void eat(int id) {
    printf("Philospoher #%d is eating\n", id);
    sleep(rand() % 20 / 1000.0);
    printf("Philosopher #%d is stuffed\n", id);
}

omp_lock_t *left_chop_stick(int id) {
    return &chop_stick[(id - 1 + N) % N];
}

omp_lock_t *right_chop_stick(int id) {
    return &chop_stick[id];
}
```

cont'd on next page

```
main() {
    int i;
    for (i = 0; i < N; i++)
        omp_init_lock(&chop_stick[i]);
    #pragma opm parallel num_threads(N)
    {
        int meals, id = omp_get_thread_num();
        for (meals = 0; meals < 100; meals++) {
            think(id);
            if (id % 2 == 1) {
                omp_set_lock(left_chop_stick(id));
                omp_set_lock(right_chop_stick(id));
            } else {
                omp_set_lock(right_chop_stick(id));
                omp_set_lock(left_chop_stick(id));
            }
            eat(id);
            omp_unset_lock(left_chop_stick(id));
            omp_unset_lock(right_chop_stick(id));
        }
    }
}
```

The if clause

`if`(*scalar-logical-expression)*

The if clause is supported the parallel construct only.

If the logical expression evaluates to a non-zero value, the parallel region will be executed in parallel. Otherwise, the region is executed by a single thread only.

The clause is often used to test if there is enough work in a region to warrant its parallelization. Example,

```
#pragma omp parallel if(n>10)
```

The num_threads clause

`num_threads(`*scalar-integer-expression*`)`

The num_threads clause is supported by the parallel construct only.

The construct can be used to specify how many threads should be in a team executing a parallel region. Example,

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

The reduction clause

`reduction(`*operator*`:`*list*`)`

The reduction clause performs a reduction on the variables that appear in the list, with the operator *operator*.

The variables must be shared scalars (*scalar*: a variable that contains only one value).

Example with reduction clause

```
#pragma omp parallel for default(none) shared(n,a) \
                  reduction(+:sum)
        for (i=0; i<n; i++)
          sum += a[i];
   /*-- End of parallel reduction --*/
printf("Value of sum after parallel region: %d\n",sum);
```

Figure 4.77: **Example of the reduction clause** – This clause gets the OpenMP compiler to generate code that performs the summation in parallel. This is generally to be preferred over a manual implementation.

Supported reduction operators

Operator	Initialization value
+	0
*	1
-	0
&	~0
|	0
^	0
&&	1
||	0

Figure 4.80: **Operators and initial values supported on the reduction clause in C/C++** – The initialization value is the value of the local copy of the reduction variable. This value is operator, data type, and language dependent.

Reduction statements

```
x = x op expr
x binop= expr
x = expr op x (except for subtraction)
x++
++x
x--
--x
```

Figure 4.82: **Typical reduction statements in C/C++** – Here, `expr` has scalar type and does not reference `x`, *op* is not an overloaded operator, but one of `+`, `*`, `-`, `&`, `^`, `|`, `&&`, or `||`, and *binop* is not an overloaded operator, but one of `+`, `*`, `-`, `&`, `^`, or `|`.

The copyprivate clause

`copyprivate(`*list*`)`

The copyprivate clause is supported by the single construct only.

The variables in the list must be private in the enclosing parallel region.

The values of the executing thread are broadcasted to all other threads in the team.

Copyprivate example

```
#pragma omp parallel private(x)
{
    #pragma omp single copyprivate(x)
    {
        x = getValue();
    }
    useValue(x);
}
```

The flush directive

```
#pragma omp flush [(list)]
```

Figure 4.89: **Syntax of the flush directive in C/C++** – This enforces shared data to be consistent. Its usage is not always straightforward.

The flush directive synchronizes copies in register or cache of the executing thread with main memory.

It synchronizes those variables in the given list; if no list is specified, all shared variables in the region.

A flush is executed implicitly at all synchronization points.

Flush example

pipelining

```
#define MAX_THREADS 16

int iam, i, isync[MAX_THREADS];
for (i = 0; i < MAX_THREADS; i++) isync[i] = 0;
omp_set_num_threads(MAX_THREADS);

#pragma omp parallel private(iam)
{
    iam = omp_get_thread_num();
    if (aim != 0)
        while (!isync[iam - 1]) { // wait for neighbor
            #pragma omp flush(isync)
        }
    work(); // do my work
    isync[iam] = 1; // I am done
    #pragma omp flush(isync)
}
```

The threadprivate directive

```
#pragma omp threadprivate (list)
```

Figure 4.94: **Syntax of the threadprivate directive in C/C++** – The *list* consists of a comma separated list of file-scope, namespace-scope, or static block scope variables that have incomplete types. The `copyin` clause can be used to initialize the data in the threadprivate copies of the list item(s).

The effect of the threadprivate directive is that the named global-lifetime objects are replicated, so that each thread has its own copy.

Threadprivate variables differ from private variables because they are able to persist between different parallel sections of code.

Threadprivate data persistency

When the end of a parallel region is reached, the slave threads disappear, but they do not die. Rather, they park themselves on a queue waiting for the next parallel region. In addition, they retain their state, in particular their instances of the threadprivate variables. As a result the contents of threadprivate data persists for each thread from one parallel region to another.

The persistency is guaranteed as long as the number of threads does not change.

Runtime routines for threads

- Determine the number of threads for parallel regions

```
omp_set_num_threads(count)
```

- Query the maximum number of threads for team creation

```
maxthreads = omp_get_max_threads()
```

- Query the number of threads in current team

```
numthreads = omp_get_num_threads()
```

- Query own thread number

```
iam = omp_get_thread_num()
```

- Query number of processors

```
numprocs = omp_get_numprocs()
```

cont'd on next page

Runtime routines for threads (cont'd)

- Query state

```
logicalvar = omp_in_parallel()
```

- Allow the runtime system to determine the number of threads for team creation

```
omp_set_dynamic(logicalexp)
```

- Query whether runtime system can determine the number of threads

```
logicalvar = omp_get_dynamic()
```

cont'd on next page

Runtime routines for threads (cont'd)

- Query the wall clock time (in seconds) relative to an arbitrary reference time

```
time = omp_get_wtime()
```

- Allow nesting of parallel regions

```
omp_set_nested(logicalexp)
```

- Query nesting of parallel regions

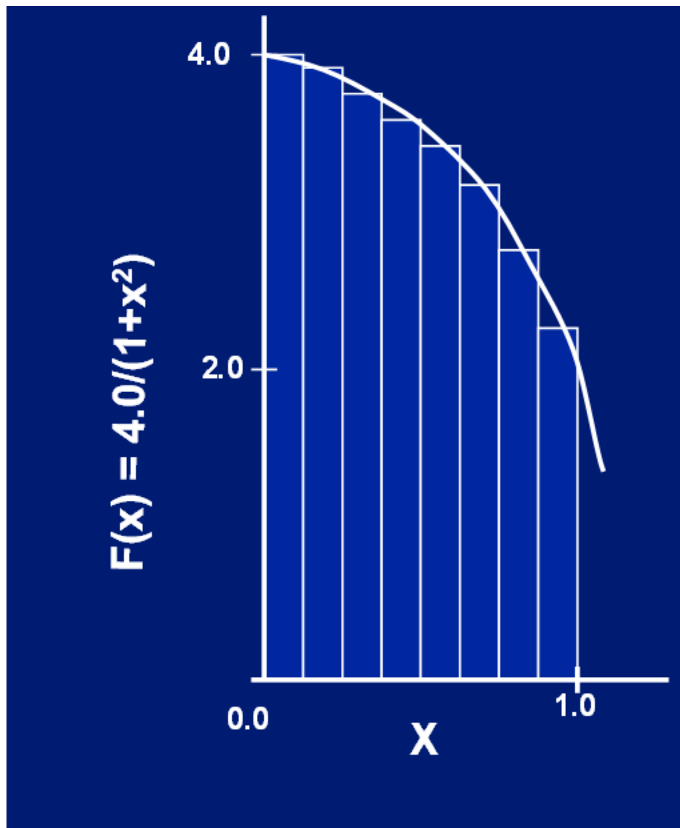
```
logicalvar = omp_get_nested()
```

Environment variables

```
OMP_NUM_THREADS=4

OMP_SCHEDULE="dynamic"

OMP_SCHEDULE="GUIDED,4"

OMP_DYNAMIC=TRUE

OMP_NESTED=TRUE
```

Numerical integration for estimating π

Mathematically, we know that $\int_0^1 \frac{4}{1+x^2} = \pi$



We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N-1} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

Serial π program

```
#include <stdio.h>

int main() {
    int N = 100000, i;
    double sum = 0.0;

    for (i = 0; i < N; i++) {
        double x = (i + 0.5) / N;
        sum += 4.0 / (1.0 + x * x);
    }
    printf("Estimate of pi   = %.15f\n", sum / N);
    printf("True value of pi = 3.141592653589793\n");
}
```

```
Estimate of pi   = 3.141592653598162
True value of pi = 3.141592653589793
```

Parallel π program

```
#include <stdio.h>

int main() {
    int N = 100000, i;
    double sum = 0.0;
    #pragma omp parallel reduction(+:sum)
    for (i = 0; i < N; i++) {
        double x = (i + 0.5) / N;
        sum += 4.0 / (1.0 + x * x);
    }
    printf("Estimate of pi   = %.15f\n", sum / N);
    printf("True value of pi = 3.141592653589793\n");
}
```

Finding the maximum value in an array

```
max = INT_MIN;
for (i = 0; i < n; i++) {
    if (a[i] > max)
        max = a[i];
}
```

Inefficient parallel code

```
max = INT_MIN;
#pragma omp parallel for shared(max)
for (i = 0; i < n; i++) {
    #pragma omp critical
    if (a[i] > max)
        max = a[i];
}
```

Improved parallel code

```
max = INT_MIN;
#pragma omp parallel for shared(max)
for (i = 0; i < n; i++) {
    #pragma omp flush(max)
    if (a[i] > max)
        #pragma omp critical
        if (a[i] > max)
            max = a[i];
}
```

Efficient parallel code

```
max = INT_MIN;
#pragma omp parallel shared(max)
{
    int private_max = max;
    #pragma for
    for (i = 0; i < n; i++)
        if (a[i] > private_max)
            private_max = a[i];
    #pragma omp flush(max)
    if (private_max > max)
        #pragma omp critical
        if (private_max > max)
            max = private_max;
}
```