SHARED-MEMORY PARALLEL PROGRAMMING WITH OPENMP

Dr Noor Mahammad Sk

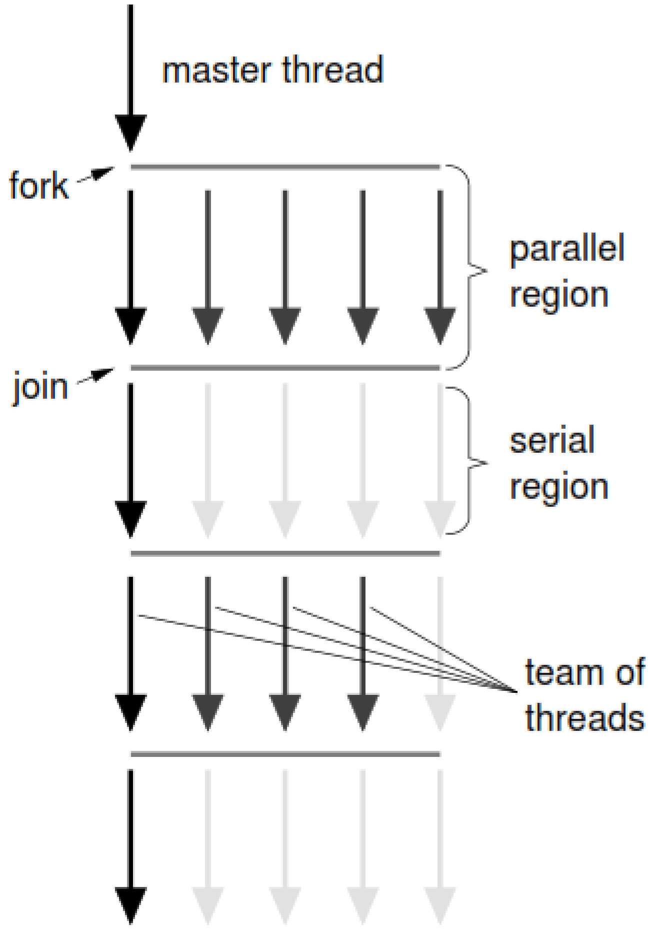
Hprcse

OpenMP

- Shared memory opens the possibility to have immediate access to all data from all processors without explicit communication.
- OpenMP is a set of *compiler directives that a non-OpenMP-capable compiler would just regard as comments and* ignore.
- A well-written parallel OpenMP program is also a valid serial program (with Compiler Directives of OpenMP).
- The central entity in an OpenMP program is not a process but a *Thread.*
- Threads are also called "lightweight processes" because several of them can share a common address space and mutually access data.
- Spawning a thread is much less costly than forking a new process, because threads share everything but instruction pointer (the address of the next instruction to be executed), stack pointer and register state.

Uses Fork and Join Model

- The master thread "forks" team of threads, which work on shared memory in a parallel region.
- After the parallel region, the threads are "joined," i.e., terminated or put to sleep, until the next parallel region starts.
- The number of running threads may vary among parallel regions.



Thread – Private Variable

- Each thread can, by means of its local stack pointer, also have “private” variables,
- Since all data is accessible via the common address space
- It is only a matter of taking the address of an item to make it accessible to all other threads as well.
- However, the OpenMP standard actually *forbids making a private object available* to other threads via its address.

Parallel Execution

- A single thread is called the *master thread, runs immediately* after startup.
- Truly parallel execution happens inside *parallel regions, of which an arbitrary* number can exist in a program.
- Between two parallel regions, no thread except the master thread executes any code.
- Inside a parallel region, a *team of threads executes instruction streams* concurrently.
- The number of threads in a team may vary among parallel regions.

Example

- OpenMP is a layer that adapts the raw OS thread interface to make it more usable with the typical structures that numerical software tends to employ.

```
  #include <omp.h>

  std::cout << "I am the master, and I am alone";
#pragma omp parallel
{
  do_work_package(omp_get_thread_num(),omp_get_num_threads());
}
```

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

Data Scoping

- Any variables that existed before a parallel region still exist inside, and are by default shared between all threads.
- True work sharing, however, makes sense only if each thread can have its own, *private variables.*
- *OpenMP supports this concept* by defining a separate stack for every thread.
- There are three ways to make private variables.

Private variable

- A variable that exists before entry to a parallel construct can be privatized, i.e., made available as a private instance for every thread, by a PRIVATE clause to the OMP PARALLEL directive.
- The private variable's scope extends until the end of the parallel construct.
- The index variable of a work sharing loop is automatically made private.
- Local variables in a subroutine called from a parallel region are private to each calling thread.

Hprcse

Private Clause

- The PRIVATE clause to the PARALLEL directive privatizes all specified variables, i.e., each thread gets its own instance of each variable on its local stack, with an undefined initial value

OpenMP worksharing for loops

- Loops are natural candidates for parallelization if individual iterations are independent.
- This corresponds to the medium-grained data parallelism
- The iterations of the loop are distributed among the threads (which are running because we are in a parallel region).
- Each thread gets its own iteration space, i.e., is assigned to a different set of i values.
- How threads are mapped to iterations is implementation-dependent by default, but can be influenced by the programmer.
- Although shared in the enclosing parallel region, the loop counter i is privatized automatically.
- Loop counters are restricted to integers (signed or unsigned), pointers, or random access iterators.

Example

```
   double precision :: pi,w,sum,x
   integer :: i,N=1000000

   pi = 0.d0
   w = 1.d0/N
   sum = 0.d0
!$OMP PARALLEL PRIVATE(x) FIRSTPRIVATE(sum)
!$OMP DO
   do i=1,n
     x = w*(i-0.5d0)
     sum = sum + 4.d0/(1.d0+x*x)
   enddo
!$OMP END DO
!$OMP CRITICAL
   pi= pi + w*sum
!$OMP END CRITICAL
!$OMP END PARALLEL
```

Race Condition

- In a parallel loop, each thread executes "its" share of the loop's iteration space, accumulating into its private *sum* variable.
- After the loop, and still inside the parallel region, the partial sums must be added to get the final result
- Because the private instances of *sum* will be gone once the region is left.
- **There is a problem**, however: Without any countermeasures, threads would write to the result variable pi concurrently.
- The result would depend on the exact order the threads access pi, and it would most probably be wrong.
- This is called a *race condition*.

Synchronization - Critical Regions

- Concurrent write access to a shared variable or, in more general terms, a shared resource, must be avoided by all means to circumvent race conditions.
- *Critical regions* solve this problem by making sure that at most one thread at a time executes some piece of code.
- If a thread is executing code inside a critical region, and another thread wants to enter, the latter must wait (block) until the former has left the region.
- Note that the order in which threads enter the critical region is undefined, and can change from run to run.
- Consequently, the definition of a “correct result” must encompass the possibility that the partial sums are accumulated in a random order, and the usual reservations regarding floating-point accuracy do apply.

Deadlock

- Critical regions hold the danger of *deadlocks when used inappropriately.*
- *A deadlock* arises when one or more “agents” (threads in this case) wait for resources that will never become available, a situation that is easily generated with badly arranged CRITICAL directives.
- When a thread encounters a CRITICAL directive inside a critical region, it will block forever.
- Since this could happen in a deeply nested subroutine, deadlocks are sometimes hard to pin down.

Solution to Deadlock

- OpenMP has a simple solution for this problem:
- A critical region may be given a *name that distinguishes it from others.*

- Without the names on the two different critical regions, this code would deadlock because a thread that has just called func(),
- Already in a critical region, would immediately encounter the second critical region and wait for itself indefinitely to free the resource.
- With the names, the second critical region is understood to protect a different resource than the first.

```
!$OMP PARALLEL DO PRIVATE(x)
  do i=1,N
     x = SIN(2*PI*x/N)
!$OMP CRITICAL (psum)
     sum = sum + func(x)
!$OMP END CRITICAL (psum)
  enddo
!$OMP END PARALLEL DO
  ...
  double precision func(v)
  double precision :: v
!$OMP CRITICAL (prand)
  func = v + random_func()
!$OMP END CRITICAL (prand)
  END SUBROUTINE func
```

Disadvantage of Named Critical Regions

- The names are unique identifiers.
- It is not possible to have them indexed by an integer variable, for instance.
- There are OpenMP API functions that support the use of *locks for protecting shared resources.*
- The advantage of locks is that they are ordinary variables that can be arranged as arrays or in structures.
- That way it is possible to protect each single element of an array of resources individually, even if their number is not known at compile time.

Synchronization - **Barriers**

- If, at a certain point in the parallel execution, it is necessary to synchronize *all* threads.
- The barrier is a *synchronization point, which guarantees that all threads have reached* it before any thread goes on executing the code below it.
- Certainly it must be ensured that every thread hits the barrier, or a deadlock may occur.
- Note also that every parallel region executes an implicit barrier at its end, which cannot be removed.
- There is also a default implicit barrier at the end of work sharing loops and some other constructs to prevent race conditions.

Reductions

- the loop implements a *reduction* operation:
- Many contributions (the updated elements of a()) are accumulated into a single variable.
- We have previously solved this problem with a critical region, but OpenMP provides a more elegant alternative by supporting reductions directly via the REDUCTION clause.
- It automatically privatizes the specified variable(s) and initializes the private instances with a sensible starting value.
- At the end of the construct, all partial results are accumulated into the shared instance of s, using the specified operator (+ here) to get the final result.
- The most common cases (addition, subtraction, multiplication, logical, etc.) are covered.

Example

- Example with reduction clause for adding noise to the elements of an array and calculating its vector norm.

```
  double precision :: r,s
  double precision, dimension(N) :: a

  call RANDOM_SEED()
!$OMP PARALLEL DO PRIVATE(r) REDUCTION(+:s)
  do i=1,N
    call RANDOM_NUMBER(r)  ! thread safe
    a(i) = a(i) + func(r)  ! func() is thread safe
    s = s + a(i) * a(i)
  enddo
!$OMP END PARALLEL DO

  print *,'Sum = ',s
```

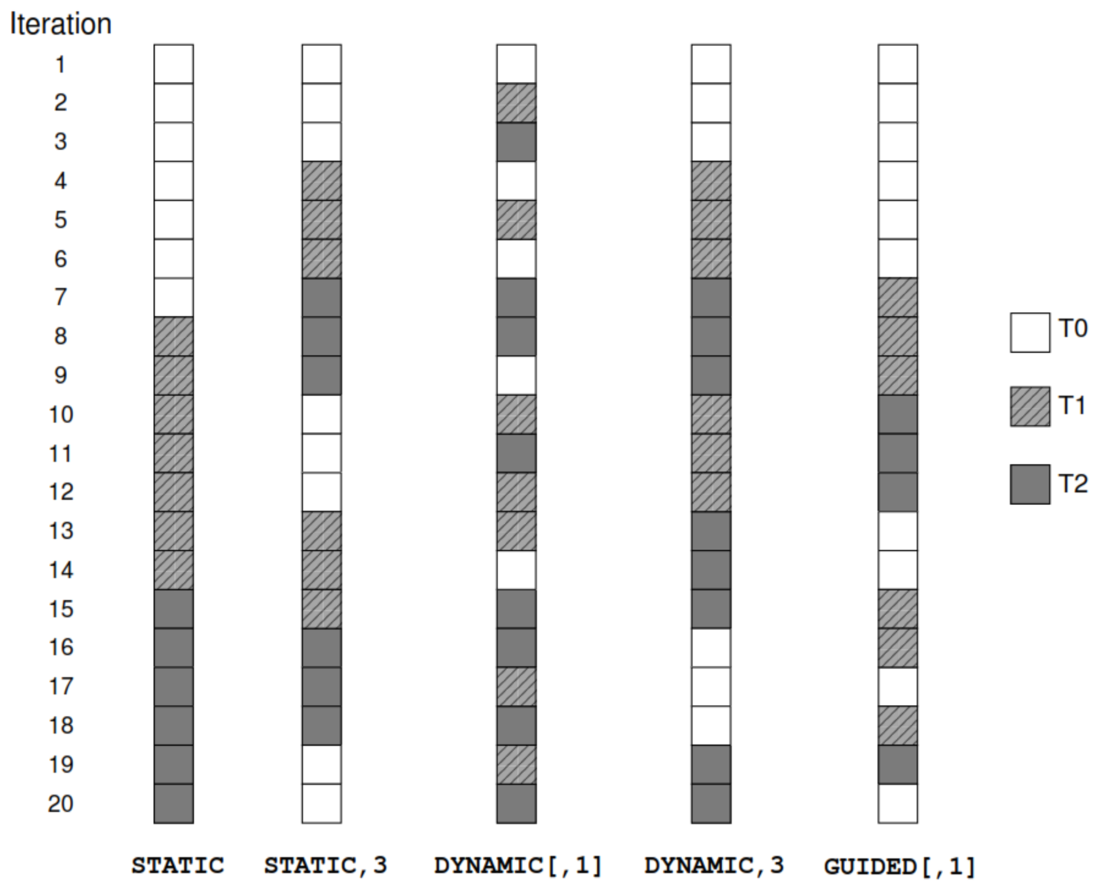
Loop Scheduling

```
!$OMP DO SCHEDULE(STATIC)
  do i=1,N
    a(i) = calculate(i)
  enddo
!$OMP END DO
```

- As mentioned earlier, the mapping of loop iterations to threads is configurable.
- It can be controlled by the argument of a SCHEDULE clause to the loop work sharing directive:
- The simplest possibility is STATIC, which divides the loop into contiguous chunks of (roughly) equal size.
- Each thread then executes on exactly one chunk.
- If for some reason the amount of work per loop iteration is not constant but, e.g., decreases with loop index, this strategy is suboptimal because different threads will get vastly different workloads, which leads to load imbalance.
- One solution would be to use a *chunk size like in “STATIC,1,” dictating that chunks of size 1 should be distributed* across threads in a round-robin manner.
- The chunk size may not only be a constant but any valid integer-valued expression.

Loop schedules in OpenMP

- The example loop has 20 iterations and is executed by three threads (T0, T1, T2).
- The default chunksize for DYNAMIC and GUIDED is one.
- If a chunksize is specified, the last chunk may be shorter.
- Note that only the STATIC schedules guarantee that the distribution of chunks among threads stays the same from run to run.

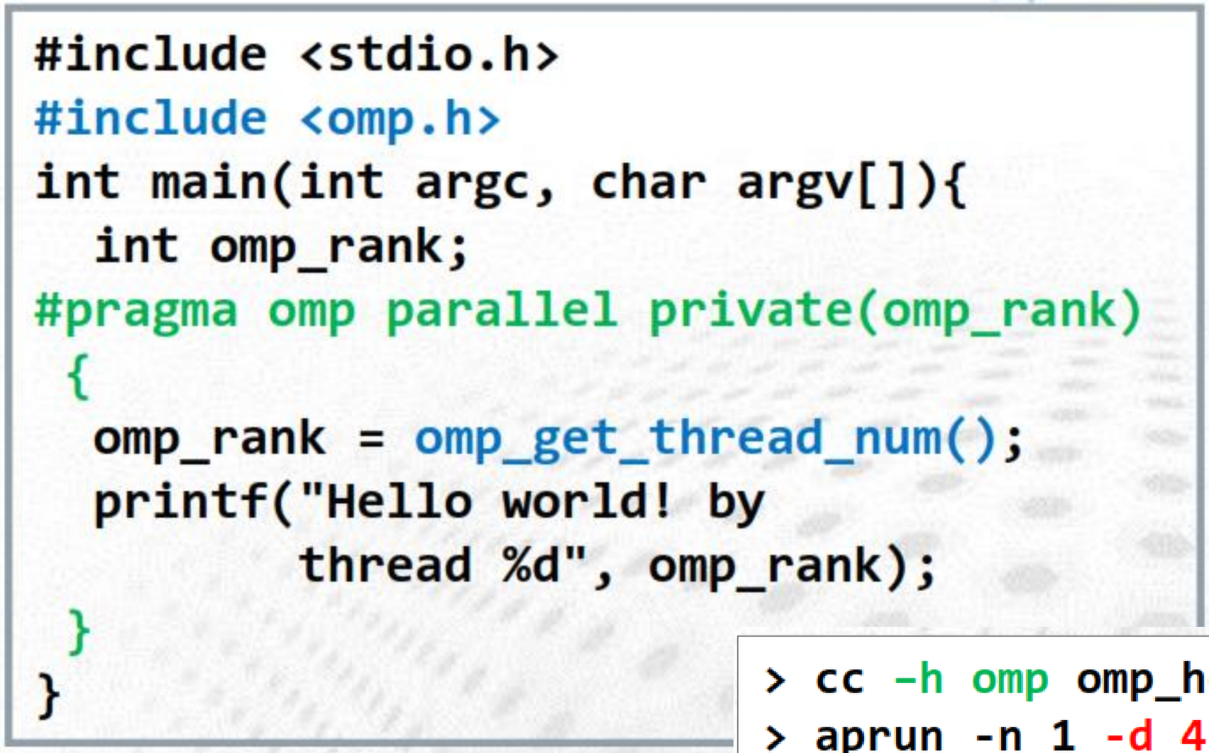


Dynamic Scheduling

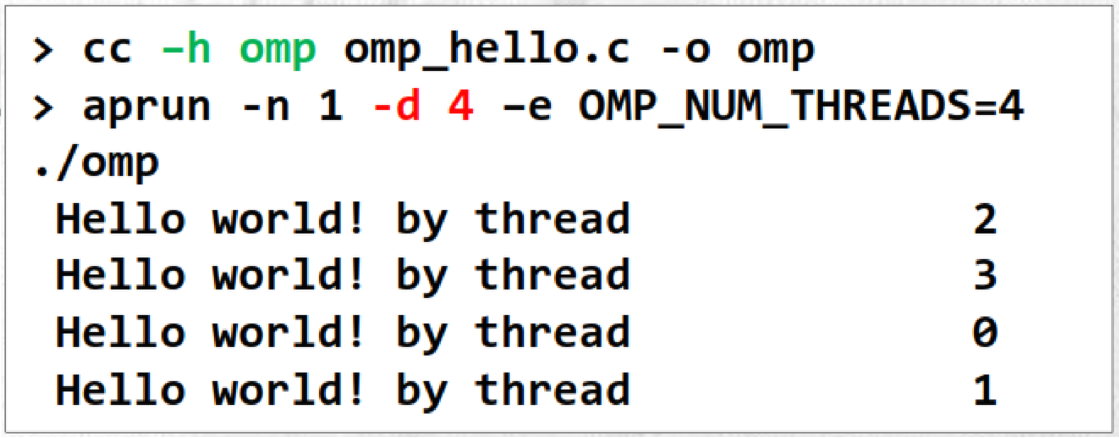
- *Dynamic scheduling assigns a chunk of work, whose size is defined by the* chunk size, to the next thread that has finished its current chunk.
- This allows for a very flexible distribution which is usually not reproduced from run to run.
- Threads that get assigned “easier” chunks will end up completing more of them, and load imbalance is greatly reduced.
- The dynamic scheduling generates significant overhead if the chunks are too small in terms of execution time.
- This is why it is often desirable to use a moderately large chunk size on tight loops, which in turn leads to more load imbalance.
- In cases where this is a problem, the *guided schedule may help.*

DATA RACE

Example: Helloworld with OpenMP



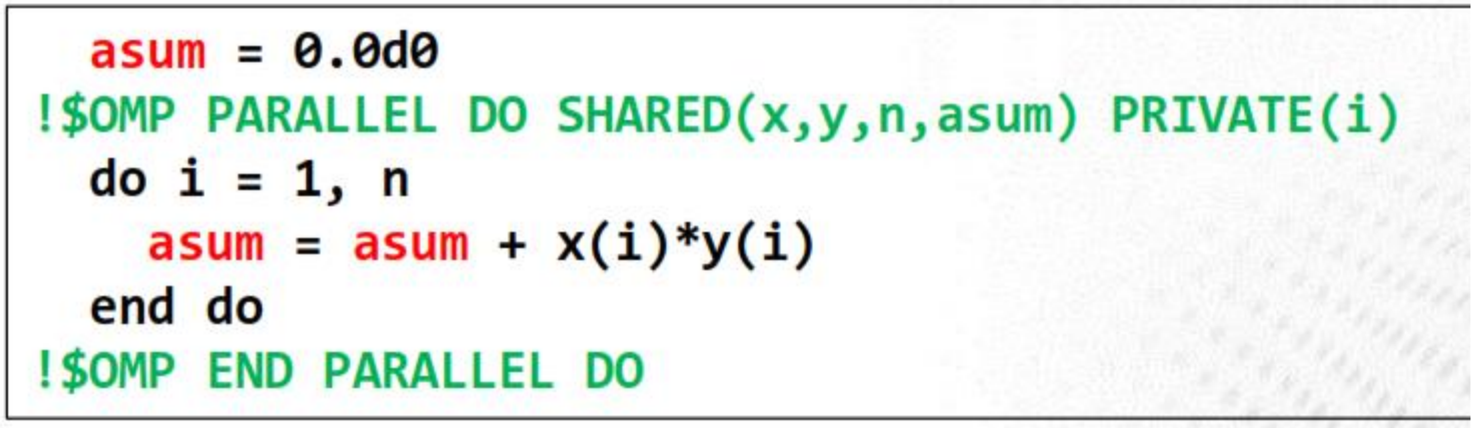
```c
#include <stdio.h>
#include <omp.h>
int main(int argc, char argv[]){
  int omp_rank;
#pragma omp parallel private(omp_rank)
 {
  omp_rank = omp_get_thread_num();
  printf("Hello world! by
         thread %d", omp_rank);
 }
}
```



```
> cc -h omp omp_hello.c -o omp
> aprun -n 1 -d 4 -e OMP_NUM_THREADS=4
./omp
 Hello world! by thread            2
 Hello world! by thread            3
 Hello world! by thread            0
 Hello world! by thread            1
```

Race Condition

- Race conditions take place when multiple threads read and write a variable simultaneously, for example

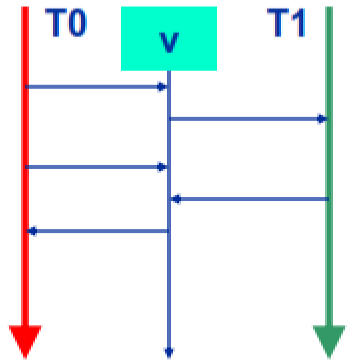


```
  asum = 0.0d0
!$OMP PARALLEL DO SHARED(x,y,n,asum) PRIVATE(i)
  do i = 1, n
    asum = asum + x(i)*y(i)
  end do
!$OMP END PARALLEL DO
```

- Random results depending on the order the threads access asum
- We need some mechanism to control the access

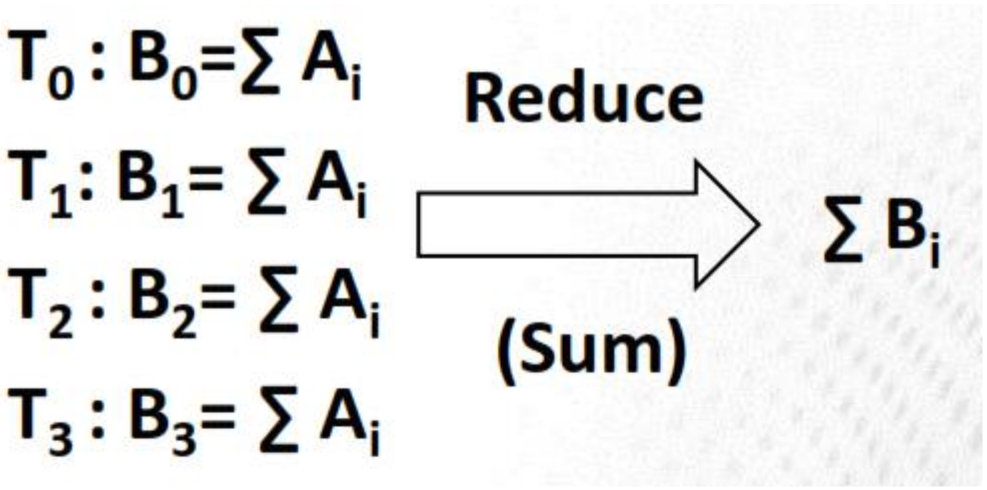
Race Condition

- Two threads access the same shared variable and at least one thread modifies the variable and the sequence of the accesses is undefined, i.e. unsynchronized
- The result of a program depends on the detailed timing of the threads in the team.
- This is often caused by unintended sharing of data
- Consequence: Wrong results



Reductions

- Summing elements of array is an example of reduction operation



- OpenMP provides support for common reductions with the reduction clause

Reduction Clause

- **reduction(operator:var_list)**
- Performs reduction on the (scalar) variables in list
- Private reduction variable is created for each thread's partial result
- Private reduction variable is initialized to operator's initial value
- After parallel region the reduction operation is applied to private variables and result is aggregated to the shared variable

Reduction Operators

Operator	Initial value
+	0
-	0
*	1

C/C++ only

Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran only

Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEGV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN	max pos.
MAX	min neg.

Race Condition with Reduction

```
!$OMP PARALLEL DO SHARED(x,y,n) PRIVATE(i) REDUCTION(+:asum)
  do i = 1, n
    asum = asum + x(i)*y(i)
  end do
!$OMP END PARALLEL DO
```

Execution Controls

- Sometimes a part of parallel region should be executed only by the master thread or by a single thread at time
  - IO, initializations, updating global values, etc.
  - Synchronization
- OpenMP provides clauses for controlling the execution of code blocks

Execution Controls: barrier

- Synchronizes all threads at this point
- When a thread reaches a barrier it only continues after **ALL** threads have reached it
- Implicit barrier at the end of parallel do/for, single
- **Restrictions**:
- Each barrier must be encountered by all threads in a team, or none at all
- The sequence of work-sharing regions and barrier regions encountered must be same for all threads in team

Execution Controls

- **master**
- Specifies a region that should be executed only by the master thread
- Note that there is no implicit barrier at end

- **single**
- Specifies that a regions should be executed only by a single (arbitrary) thread
- Other threads wait (implicit barrier)

Execution Controls

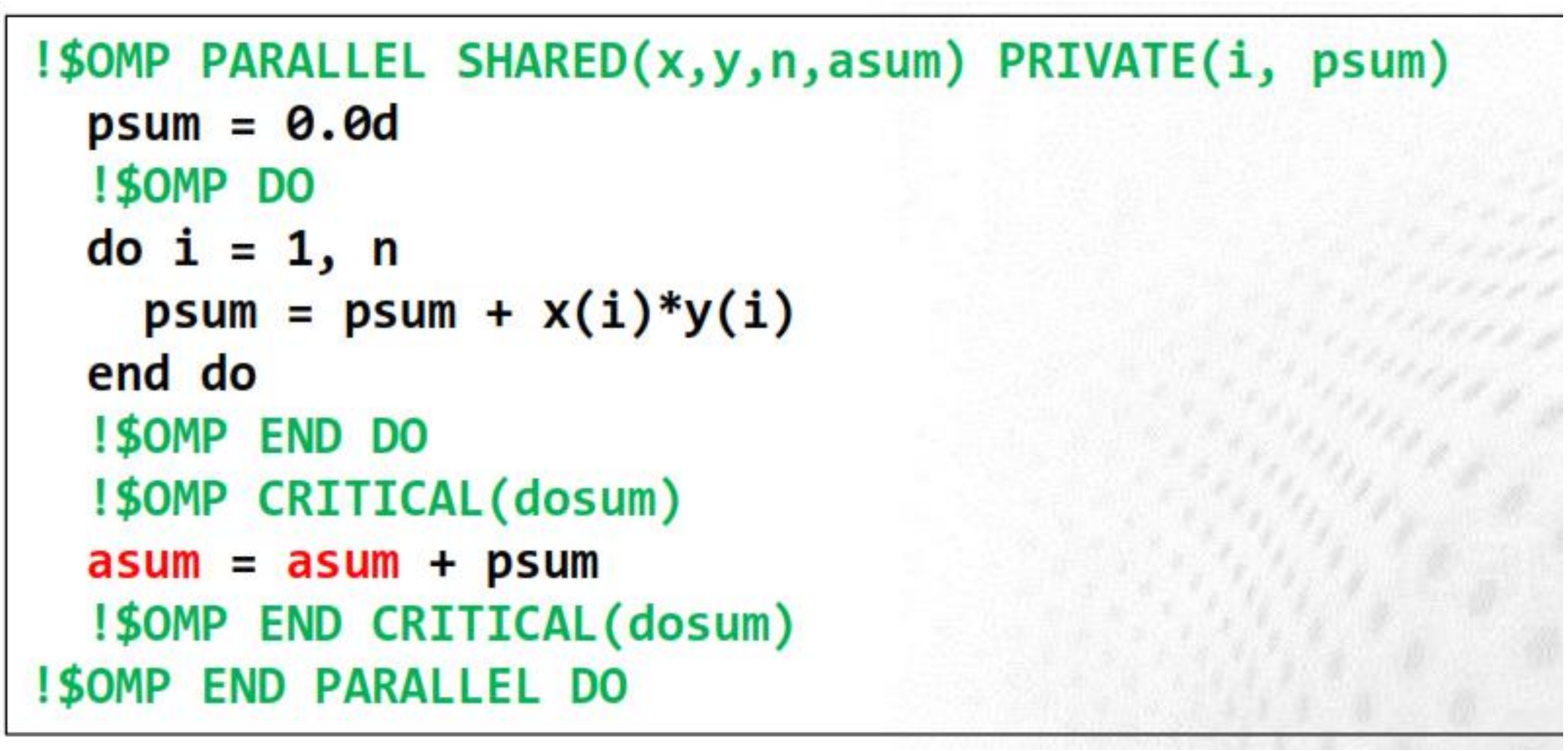
- **critical [(name)]**
- A section that is executed by only one thread at a time
- Optional name specifies different critical section
- Unnamed critical sections are treated as the **same section**

- **flush [(name)]**
- Synchronizes the memory of all threads
- Makes sure each thread has a consistent view of memory
- Implicit flush at barrier, critical

Execution Controls

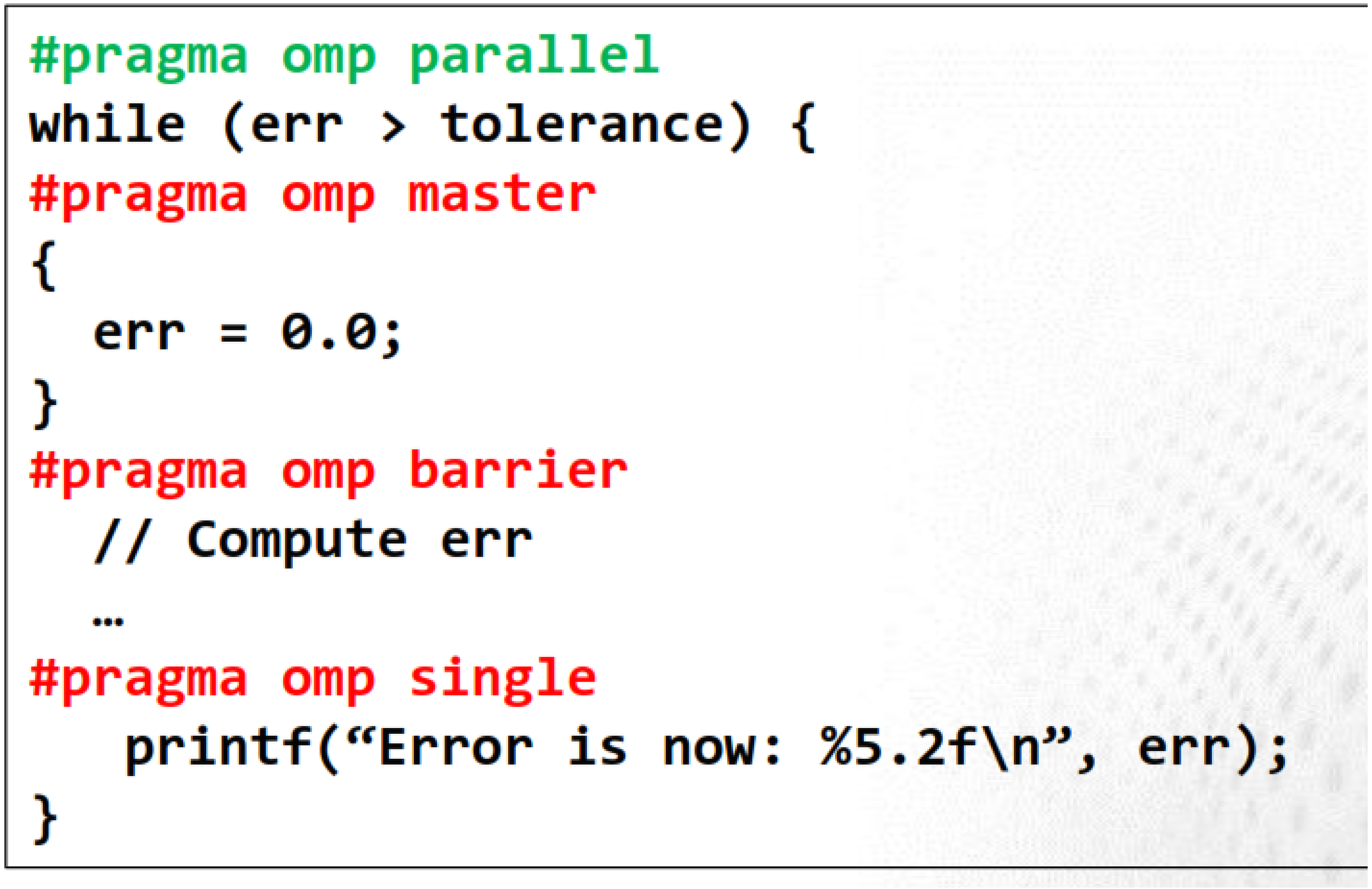
- **atomic**
- Strictly limited construct to update a single value, can not be applied to code blocks
- Can be faster on hardware platforms that support atomic updates

Example: Reduction using Critical Section



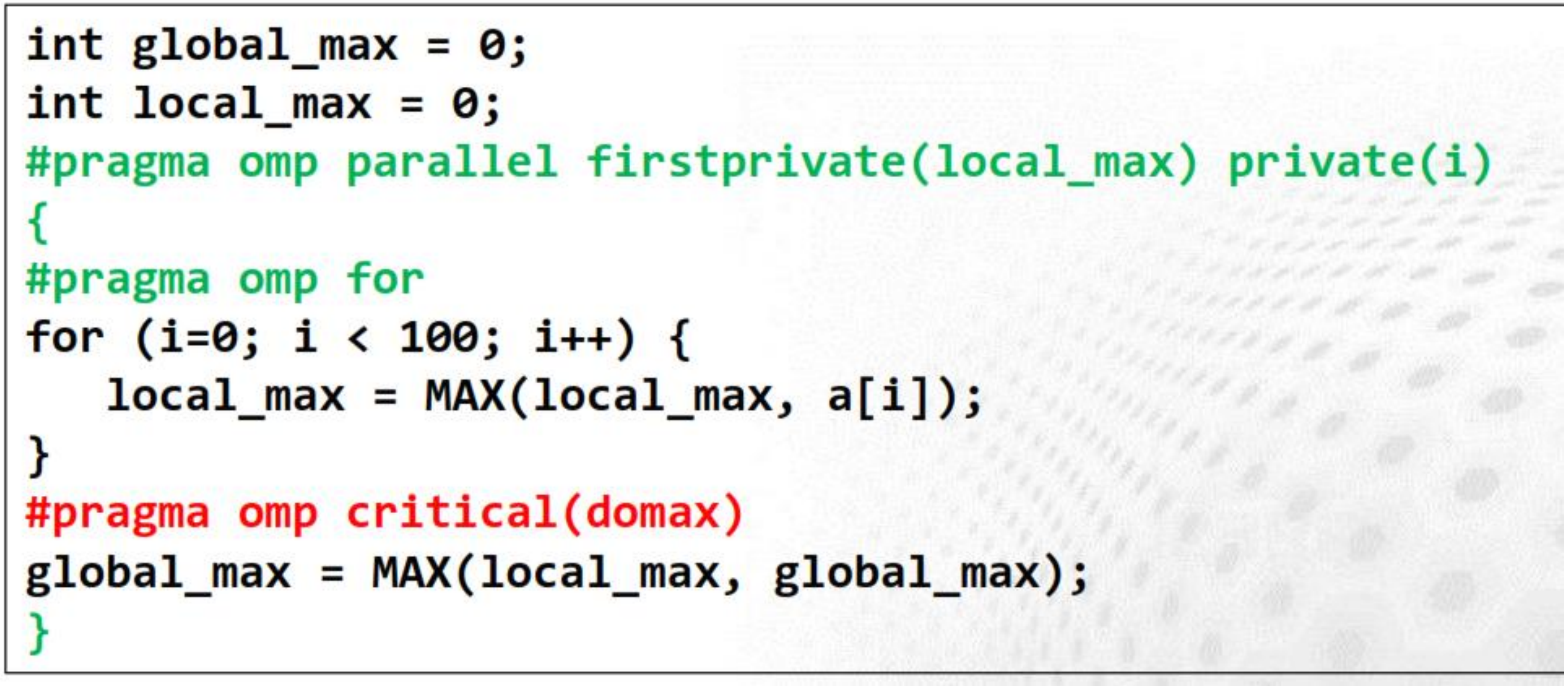
```
!$OMP PARALLEL SHARED(x,y,n,asum) PRIVATE(i, psum)
  psum = 0.0d
  !$OMP DO
  do i = 1, n
    psum = psum + x(i)*y(i)
  end do
  !$OMP END DO
  !$OMP CRITICAL(dosum)
  asum = asum + psum
  !$OMP END CRITICAL(dosum)
!$OMP END PARALLEL DO
```

Example: Initialization and Output



```
#pragma omp parallel
while (err > tolerance) {
#pragma omp master
{
  err = 0.0;
}
#pragma omp barrier
  // Compute err
  …
#pragma omp single
   printf("Error is now: %5.2f\n", err);
}
```

Example: Updating Global Variable



```
int global_max = 0;
int local_max = 0;
#pragma omp parallel firstprivate(local_max) private(i)
{
#pragma omp for
for (i=0; i < 100; i++) {
   local_max = MAX(local_max, a[i]);
}
#pragma omp critical(domax)
global_max = MAX(local_max, global_max);
}
```

Deadlock in Critical Construct OpenMP

- The code inside a CRITICAL region is executed by only one thread at a time.
- The order is not specified.
- This means that if a thread is currently executing inside a CRITICAL region and another thread reaches that CRITICAL region and attempts to execute it, it will block until the first thread exits that CRITICAL region.

```
#pragma omp critical
x = x + 1;
```

Example

```
subroutine foo
          !$OMP PARALLEL
          !$OMP CRITICAL
                    print*, 'Hallo i am just a single thread and I like it that way'
          !$OMP END CRITICAL
          !$OMP END PARALLEL
end subroutine foo

program deadlock
implicit none
integer :: i,sum = 0
          !$OMP PARALLEL
          !$OMP DO
          do i = 1, 100
          !$OMP CRITICAL
                    sum = sum + i
                    call foo()
          !$OMP END CRITICAL
          enddo
          !$OMP END DO
          !$OMP END PARALLEL

          print*, sum
end program deadlock
```

Deadlock

- In OpenMP deadlock can happen if inside a critical region a function is called which contains another critical region.
- In this case the critical region of the called function will wait for the first critical region to terminate - which will never happen.
- *When a thread encounters a CRITICAL directive inside a critical region, it will block forever*.
- Two possible ways - deadlock occurs in a nested critical region

First Approach

- If two threads arrive at a nested critical construct (one critical region inside another), thread one enters the "outer" critical region and thread two waits.
- When a thread encounters a critical construct, it waits until no other thread is executing a critical region with the same name.
- Now, thread one DOES NOT enter the nested critical region, because it is a synchronization point where threads wait for all other threads to arrive before proceeding.
- And since the second thread is waiting for the first thread to exit the "outer" critical region they are in a deadlock.

Second Approach

- Both threads arrive at the "outer" critical construct.
- Thread one enters the "outer" critical construct, thread two waits.
- Now, thread one ENTERS the "inner" critical construct and stops at it's implied barrier, because it waits for thread two.
- Thread two on the other hand waits for thread one to exit to "outer" thread and so both are waiting forever.

Deadlock - Example

```
!$OMP PARALLEL DO PRIVATE(x)
  do i=1,N
    x = SIN(2*PI*x/N)
!$OMP CRITICAL
    sum = sum + func(x)
!$OMP END CRITICAL
  enddo
!$OMP END PARALLEL DO
...

double precision function func(v)
double precision :: v
!$OMP CRITICAL
  func = v + random_func()
!$OMP END CRITICAL
end function func
```

Nested critical regions deadlock even a *single* thread, since they are understood to protect the same resource

Deadlock – Example: Remedies

- **Named critical regions decouple independent resources:**

```
!$OMP PARALLEL DO PRIVATE(x)
  do i=1,N
    x = SIN(2*PI*x/N)
!$OMP CRITICAL (psum)
    sum = sum + func(x)
!$OMP END CRITICAL (psum)
  enddo
!$OMP END PARALLEL DO
...

double precision function func(v)
double precision :: v
!$OMP CRITICAL (prand)
  func = v + random_func()
!$OMP END CRITICAL (prand)
end function func
```

✓

Names make the critical regions protect disjoint resources
→ deadlock gone

OPENMP SCHEDULING

OpenMP: Scheduling

- OpenMP specialty is a parallelization of loops.
- The loop construct enables the parallelization.

```
#pragma omp parallel for
for (...)
{ ... }
```

- OpenMP then takes care about all the details of the parallelization.
- It creates a team of threads and distributes the iterations between the threads.
- OpenMP schedules the iterations between the threads

Scheduling: Explicit

- If the loop construct has explicit schedule clause
- OpenMP uses **scheduling-type** for scheduling the iterations of the for loop.

#pragma omp parallel for schedule(scheduling-type)

for (...)

{ ... }

Scheduling: Runtime

- If the scheduling-type is equal to runtime then OpenMP determines the scheduling by the internal control variable run-sched-var.
- We can set this variable by setting the environment variable OMP_SCHEDULE to the desired scheduling type.
- For example, in bash-like terminals:
- $ **export** OMP_SCHEDULE**=**sheduling**-**type
- Another way to specify run-sched-var is to set it with **omp_set_schedule** function.

```
…
omp_set_schedule(sheduling-type);
…
```

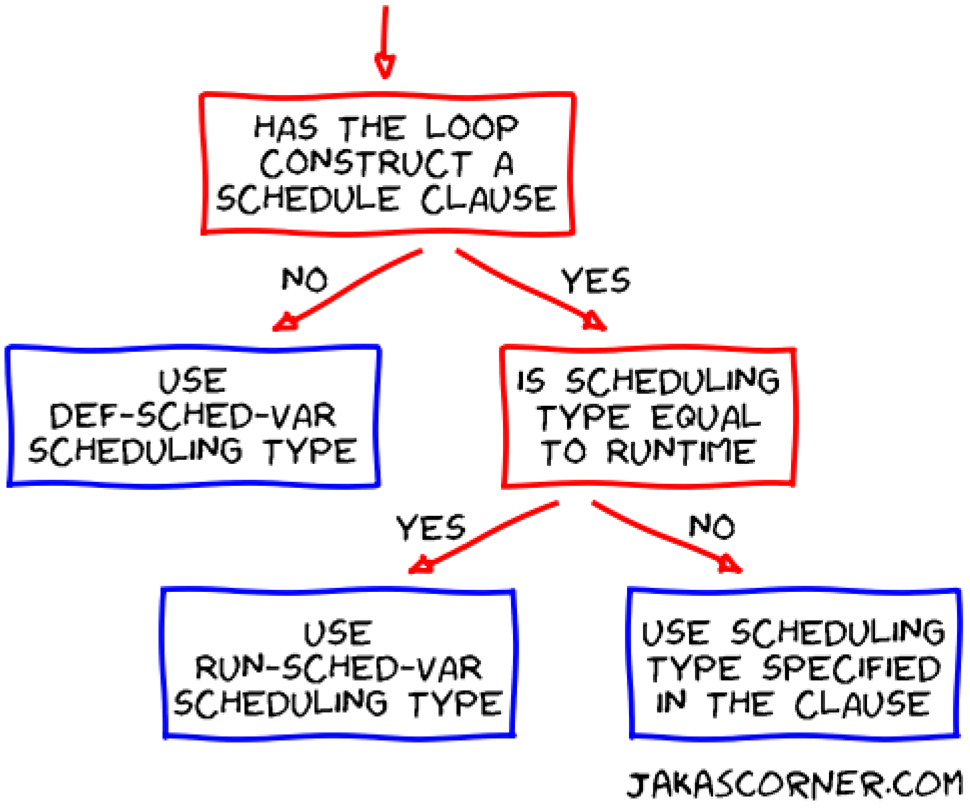
Schedule: Default

- If the loop construct does not have an explicit schedule clause hen OpenMP uses the default scheduling type.
- It is defined by the internal control variable def-sched-var and it is implementation dependent.

```
#pragma omp parallel for
for (...)
{ ... }
```

Scheduling

- OpenMP determines a scheduling type of a for loop.



HAS THE LOOP CONSTRUCT A SCHEDULE CLAUSE
NO
YES
USE DEF-SCHED-VAR SCHEDULING TYPE
IS SCHEDULING TYPE EQUAL TO RUNTIME
YES
NO
USE RUN-SCHED-VAR SCHEDULING TYPE
USE SCHEDULING TYPE SPECIFIED IN THE CLAUSE
JAKASCORNER.COM

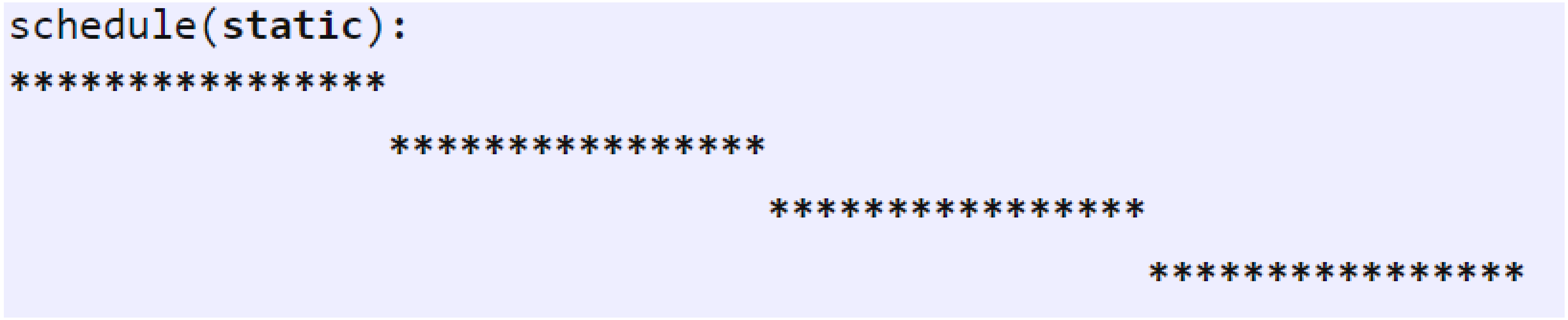
The scheduling types

- We can choose between five different scheduling types:
- static,
- dynamic,
- guided,
- auto and
- runtime.

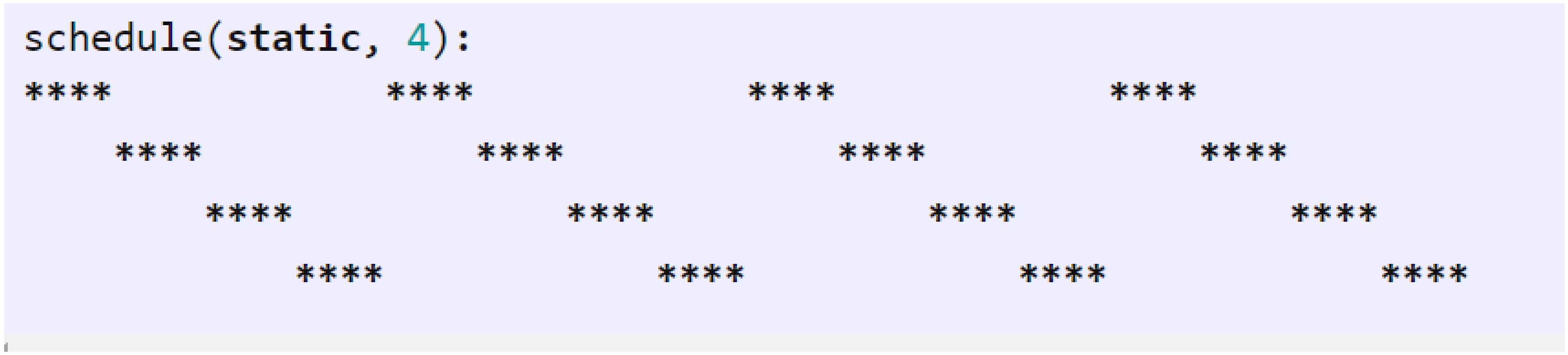
Scheduling: Static

- The schedule(static, chunk-size) clause of the loop construct specifies that the for loop has the static scheduling type.
- OpenMP divides the iterations into chunks of size **chunk-size** and it distributes the chunks to threads in a circular order.
- When no **chunk-size** is specified, OpenMP divides iterations into chunks that are approximately equal in size and
- It distributes at most one chunk to each thread.

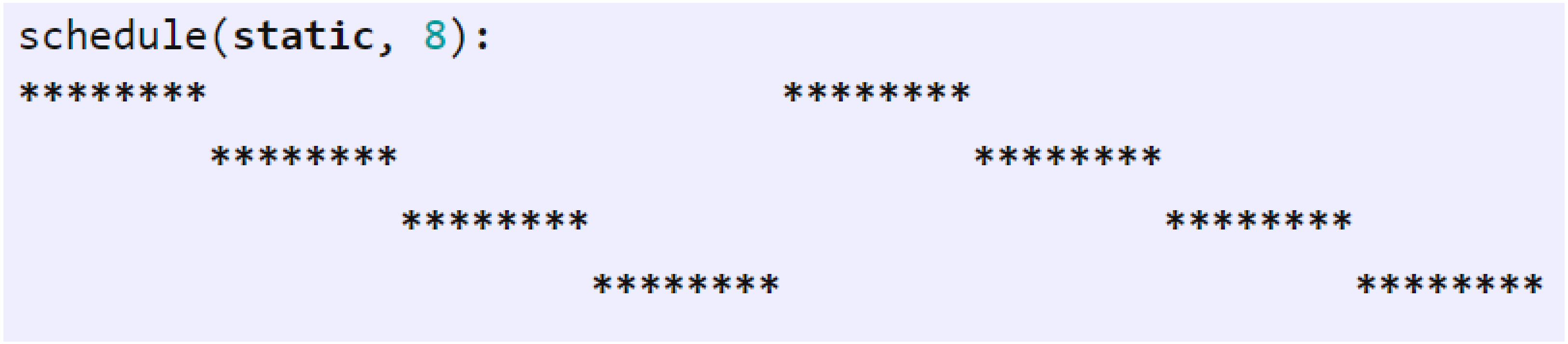
Examples of Static Scheduling



```
schedule(static):
****************
                ****************
                                ****************
                                                ****************
```



```
schedule(static, 4):
****            ****            ****            ****
    ****            ****            ****            ****
        ****            ****            ****            ****
            ****            ****            ****            ****
```



```
schedule(static, 8):
********                        ********
        ********                        ********
                ********                        ********
                        ********                        ********
```

Example of Static Scheduling

- We parallelized a for loop with 64 iterations and we used four threads to parallelize the for loop.
- Each row of stars in the examples represents a thread.
- Each column represents an iteration.
- The first example (schedule(static)) has 16 stars in the first row.
- This means that the first tread executes iterations 1, 2, 3, …, 15 and 16.
- The second row has 16 blanks and then 16 stars.
- This means that the second thread executes iterations 17, 18, 19, …, 31, 32. Similar applies to the threads three and four.
- We see that for schedule(static) OpenMP divides iterations into four chunks of size 16 and it distributes them to four threads.
- For schedule (static, 4) and schedule (static, 8) OpenMP divides iterations into chunks of size 4 and 8, respectively.
- The static scheduling type is appropriate when all iterations have the same computational cost.

Scheduling: Dynamic

- The **schedule(dynamic, chunk-size)** clause of the loop construct specifies that the for loop has the dynamic scheduling type.
- OpenMP divides the iterations into chunks of size chunk-size.
- Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.
- There is no particular order in which the chunks are distributed to the threads.
- The order changes each time when we execute the for loop.
- If we do not specify **chunk-size**, it defaults to one.

Examples of dynamic scheduling

```
schedule(dynamic):
*   ** **  * * *  *      *  *    **   *  *  * *       *  *   *
  *       *     *    * *     * *   *    *        * *   *    *
 *       *    *     * *   *   *     *  *       *  *  *  *  *   *
   *  *     *    * *    *  *    *    *    ** *  *   *     *   *
```

```
schedule(dynamic, 1):
    *    *     *        *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *      *   ***  *   *         *
 *   *  *  *  *    ** *    *      *  *  * *   *  *   *   *
  *    *     *  **        *  * *    *          *  *    *  * *  *
```

- We can see that for **schedule(dynamic)** and **schedule(dynamic, 1)** OpenMP determines similar scheduling.
- The size of chunks is equal to one in both instances.
- The distribution of chunks between the threads is arbitrary.

Examples of dynamic scheduling

```
schedule(dynamic, 4):
            ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

```
schedule(dynamic, 8):
                ********                                ********
                        ********        ********
********                        ********        ********
        ********
```

- For **schedule(dynamic, 4)** and **schedule(dynamic, 8)** OpenMP divides iterations into chunks of size four and eight, respectively.
- The distribution of chunks to the threads has no pattern.

Scheduling: Dynamic

- The dynamic scheduling type is appropriate when the iterations require different computational costs.
- This means that the iterations are poorly balanced between each other.
- The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.

Scheduling: Guided

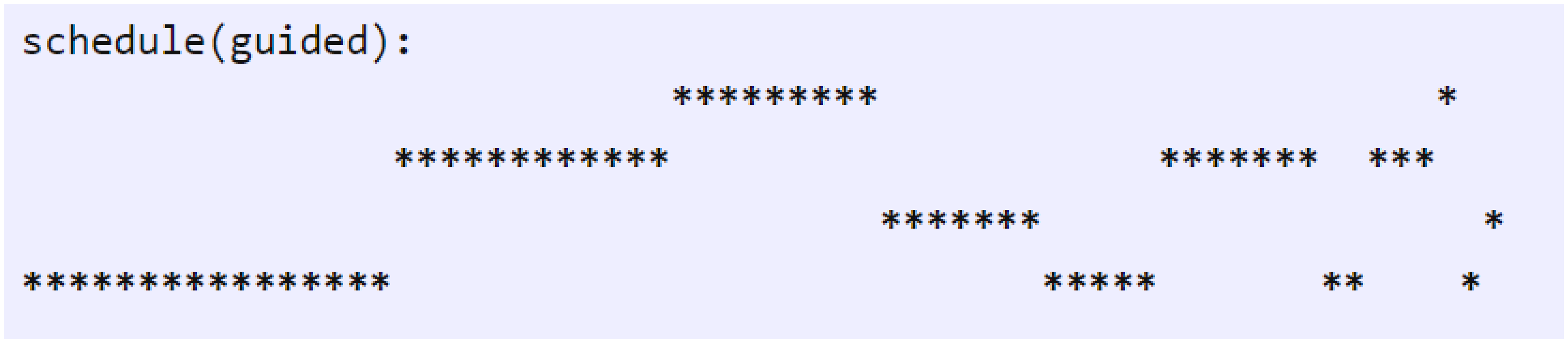
- The guided scheduling type is similar to the dynamic scheduling type.
- OpenMP again divides the iterations into chunks.
- Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.
- The difference with the dynamic scheduling type is in the size of chunks.
- The size of a chunk is proportional to the number of unassigned iterations divided by the number of the threads.
- Therefore the size of the chunks decreases.

Hprcse

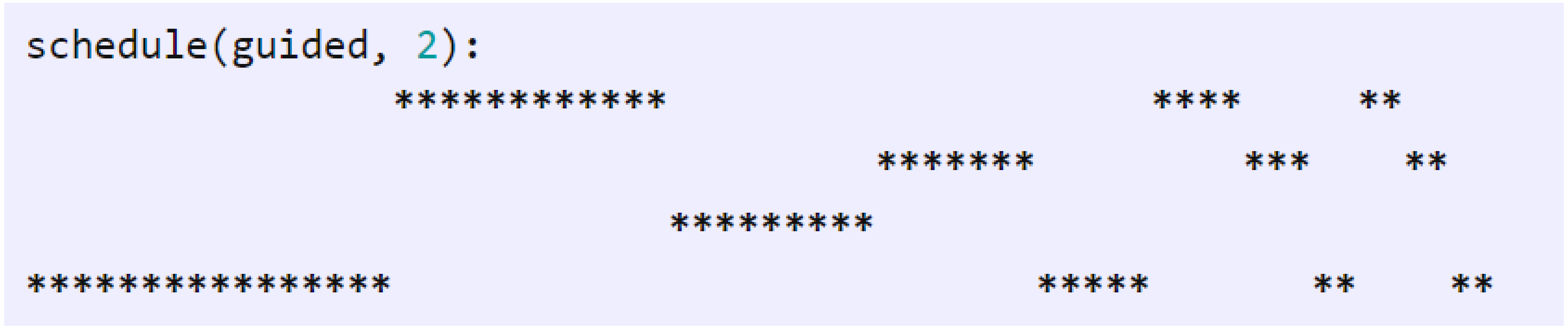
Scheduling: Guided

- The minimum size of a chunk is set by **chunk-size**.
- We determine it in the scheduling clause: **schedule(guided, chunk-size)**.
- However, the chunk which contains the last iterations may have smaller size than **chunk-size**.
- If we do not specify chunk-size, it defaults to one.

Examples of the Guided Scheduling



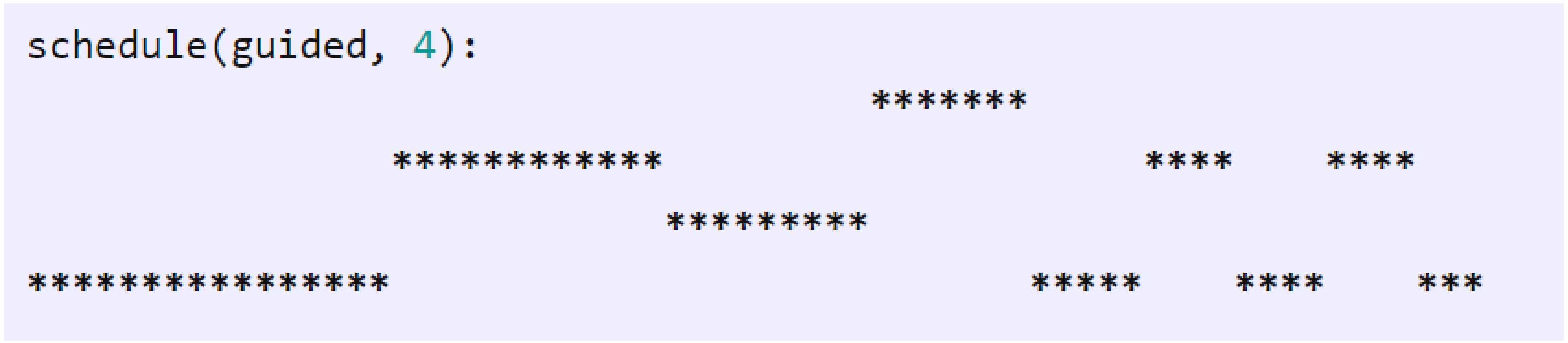
```
schedule(guided):
                            *********                        *
                ************                     *******  ***
                                     *******                   *
****************                            *****       **    *
```



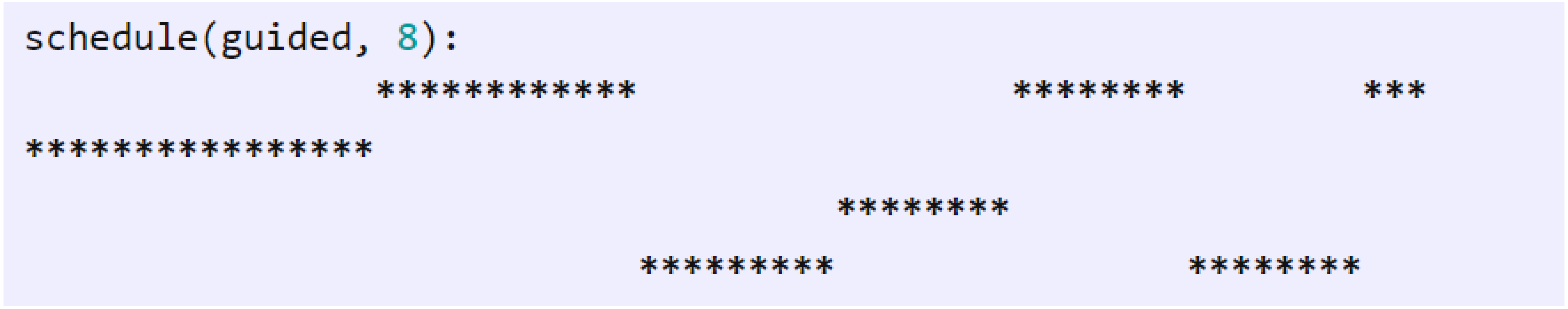
```
schedule(guided, 2):
                ************                     ****     **
                                     *******         ***    **
                            *********
****************                            *****       **    **
```

- We can see that the size of the chunks is decreasing.
- First chunk has always 16 iterations.
- This is because the for loop has 64 iterations and we use 4 threads to parallelize the for loop.
- If we divide 64 / 4, we get 16.

Examples of the Guided Scheduling



```
schedule(guided, 4):
                                     *******
                ************                     ****    ****
                            *********
****************                            *****    ****    ***
```



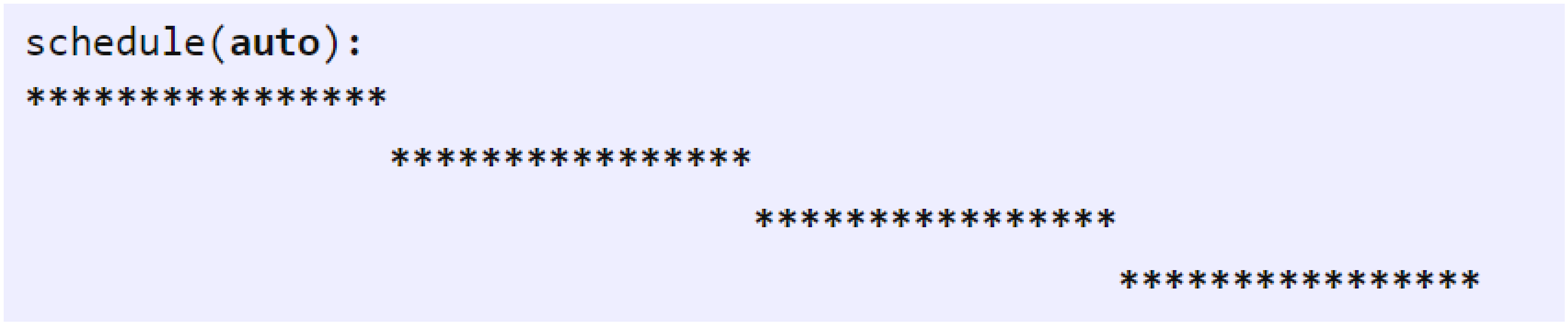
```
schedule(guided, 8):
                ************                 ********        ***
****************
                                     ********
                            *********                ********
```

Scheduling: Guided

- We can also see that the minimum chunk size is determined in the schedule clause.
- The only exception is the last chunk. Its size might be lower then the prescribed minimum size.
- The guided scheduling type is appropriate when the iterations are poorly balanced between each other.
- The initial chunks are larger, because they reduce overhead.
- The smaller chunks fills the schedule towards the end of the computation and improve load balancing.
- This scheduling type is especially appropriate when poor load balancing occurs toward the end of the computation.

Scheduling: Auto

- he auto scheduling type delegates the decision of the scheduling to the compiler and/or runtime system.
- In the following example, the compiler/system determined the static scheduling.



```
schedule(auto):
****************
                ****************
                                ****************
                                                ****************
```

Scheduling: Runtime

- The runtime scheduling type defers the decision about the scheduling until the runtime.
- We already described different ways of specifying the scheduling type in this case.
- One option is with the environment variable **OMP_SCHEDULE** and the other option is with the function **omp_set_schedule**.

Scheduling: Default

- If we do not specify the scheduling type in a for loop

```
#pragma omp parallel for
for (...)
{ ... }
```

- OpenMP uses the default scheduling type (defined by the internal control variable def-sched-var).
- If we do not specify the scheduling type in my machine, we get the following result.

```
default:
****************
                ****************
                                ****************
                                                ****************
```