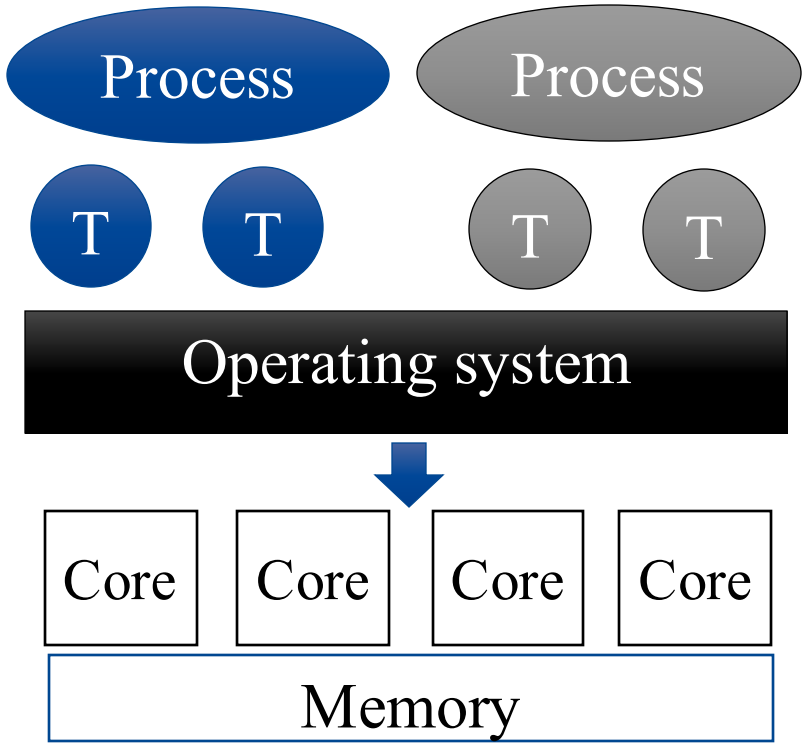
Shared Memory Programming with OpenMP

Shared Memory Programming

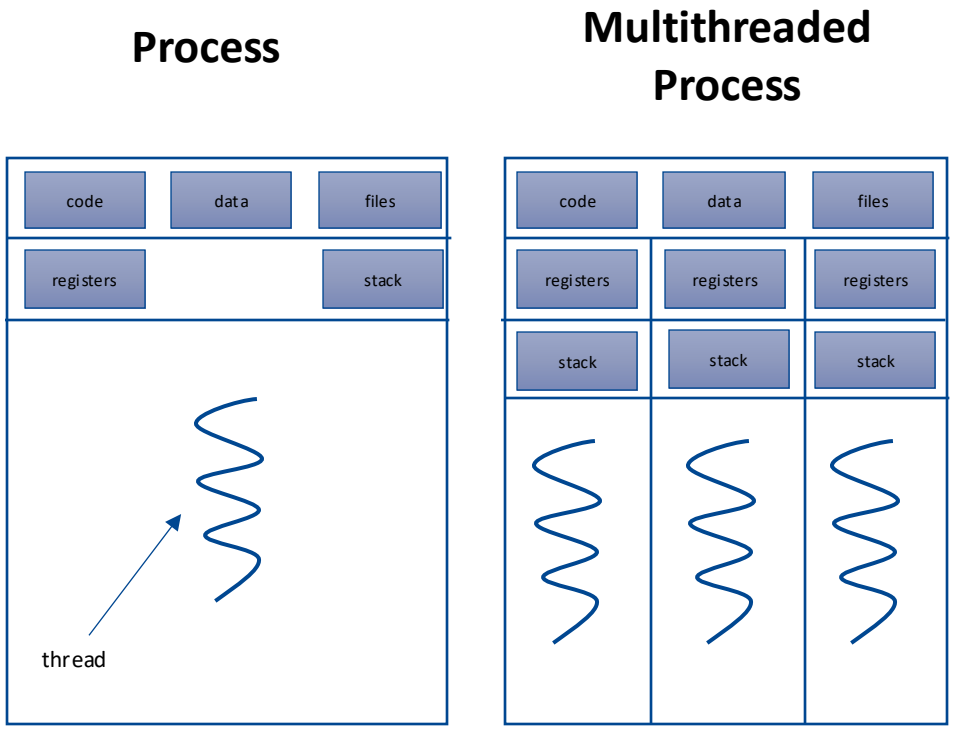


Basic assumptions:

- Shared memory hardware support
- An operating systemt (OS) that provides
  - **Processes** with indivudal address spaces
  - **Threads** that share address space within a process
  - An **OS** that **schedules** threads for execution on CPUs/cores

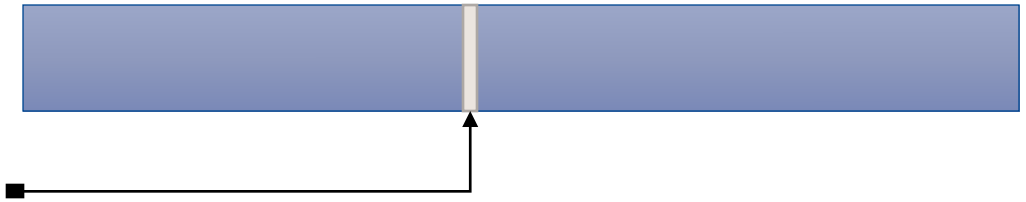
Processes and Threads

- A **thread** is a runtime entity that is able to independently execute a stream of instructions, consiting of a **program counter**, registers and stack
- The **OS** creates a **process** to execute a program. It can consists of **one or several threads**, but only one stream of instructions can be executed at any given time

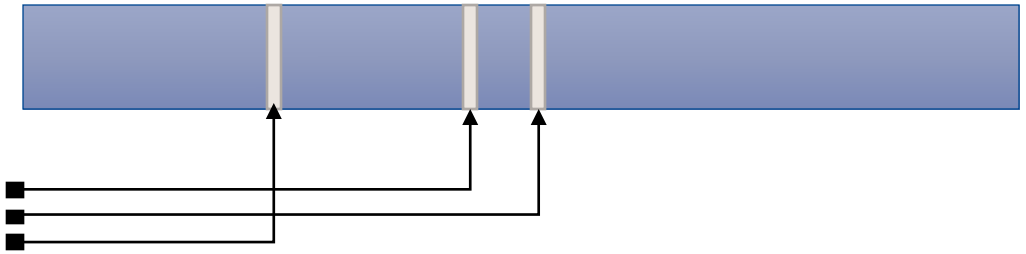


Processes and Threads

- A process has its own address space and progam counter



- Threads have their **individual program counters** but **share** the same address space



- What could happen if all threads (gray boxes) overlaps?

Data Race Condition

One of the major difficulties of multithreaded programming

Time

Thread 1	Register	Thread 2	Register	Variable in memory
				0
Read into register	0			0
Increase register	1			0
Write to memory	1			1
		Read into register	1	1
		Increase register	2	1
		Write to memory	2	2

Data Race Condition

One of the major difficulties of multithreaded programming

Time

Thread 1	Register	Thread 2	Register	Variable in memory
				0
Read into register				
		Read into register		
Increase register				
		Increase register		
Write to memory				
		Write to memory		

Data Race Condition

One of the major difficulties of multithreaded programming

Time

Thread 1	Register	Thread 2	Register	Variable in memory
				0
Read into register	0		0	0
		Read into register	0	0
Increase register	1		0	0
		Increase register	1	0
Write to memory			1	1
		Write to memory	1	1

OpenMP
Core Concepts

OpenMP – An API for Writing Parallel Applications

- A set of compiler directives and library routines for writing parallel applications
  - Use pre-processor **directives** and **code generation**

    Not to be confused with auto parallelisation! Parallelism must be specified by the programmer
- Compiler is free to generate any kind of threads
- **Portable** (but one has to trust the compiler...)
- OpenMP used to greatly simplifies writing multithreaded programs in Fortran, C or C++ (This lecture)
- … later versions of OpenMP adds support for tasking, vectorisation and GPU programming (Next lecture)

Compiler Directives

A directive **pragma** is a language construct that instructs (*or hints*) the compiler on how to process its input

Examples of directives to C/C++ compilers

- `#define` substitutes a pre-process macro
- `#include` inserts the content of a particular file
- `#pragma` issues special commands to the compiler

Examples in Fortran

- `!DIR$` issues special commands to the compiler

OpenMP Directives

Directives are used to express parallelism in OpenMP

In C/C++ directives begin with

- `#pragma omp`

In Fortran directives begin with

- `!$omp`

OpenMP Construct and Clauses

- A **construct** is a specific OpenMP executable directive

- OpenMP directives may include various **clauses** to provide further information to the expected behaviour of the OpenMP implementation

```
!$omp construct [clause [clause]…]
```

- Example **core syntax** in Fortran

```
!$omp parallel num_threads(4)
```

Compiler directive Construct Clause

OpenMP Parallel Region

- A parallel region is a region executed by **all threads**
  - Default storage attributes are defined by a **data environment** (we come back to this later)
- In C/C++ a region is included in {…} after a directive
- In Fortran a region is included between a directive pair

```
… code executed by a single thread
#pragma omp parallel
{
… code executed by all threads
}
… code executed by a single thread
```

```
… code executed by a single thread
!$omp parallel
… code executed by all threads
!$omp end parallel
… code executed by a single thread
```

Question

Which thread is going to execute `hello()`?

```
program test

!$omp parallel

  call hello

!$omp end parallel

end program test
```

```
subroutine hello

  write(*,*) 'Hello world!'

end subroutine hello
```

OpenMP Library Functions

OpenMP function prototypes and types are defined in

- `#include <omp.h> (C/C++)`
- `use omp_lib (Fortran)`

Some of the library function we will use to:

- Modify/check the number of threads:

```
omp_set_num_threads()
omp_get_num_threads()
omp_get_thread_num()
omp_get_max_threads()
```

- Check if we are in an active parallel region:

```
omp_in_parallel()
```

- Dynamically vary the number of threads:

```
omp_set_dynamic()
omp_get_dynamic()
```

- Check the number of processors in the system

```
omp_num_procs()
```

OpenMP Environment Variables

The number of threads are controlled by the environment variable OMP_NUM_THREADS

```
export OMP_NUM_THREADS=<number of threads to use> (sh/bash)
```

```
setenv OMP_NUM_THREADS <number of threads to use> (csh/tcsh)
```

Or when executing a program (e.g. `a.out`)

```
env OMP_NUM_THREADS=<number of threads to use> ./a.out
```

Our First OpenMP Program

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

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


#pragma omp parallel
  {
    int tid = omp_get_thread_num();
    printf("Hello World from thread %d!\n", tid);
  }
  return 0;
}
```

```fortran
program hello_world
  use omp_lib


!$omp parallel
  write(*,*) 'Hello World from thread', &
             omp_get_thread_num()
!$omp end parallel

end program hello_world
```

Compile OpenMP Programs

Different compilers have different flags for OpenMP

```
gcc/g++ -fopenmp hello_world.c        (GNU)

icx/icpx -qopenmp hello_world.c       (Intel)

cc/CC -fopenmp hello_world.c          (Cray)

nvc -mp=multicore hello_world.c       (NVIDIA)
```

C/C++

```
gfortran -fopenmp hello_world.f90             (GNU)

ifort/ifx -qopenmp hello_world.f90            (Intel)

ftn -homp hello_world.f90                     (Cray)

nvfortran -mp=multicore hello_world.f90       (NVIDIA)

nagfor -openmp hello_world.f90                (NAG)
```

Fortran

Our First OpenMP Program

Sample output

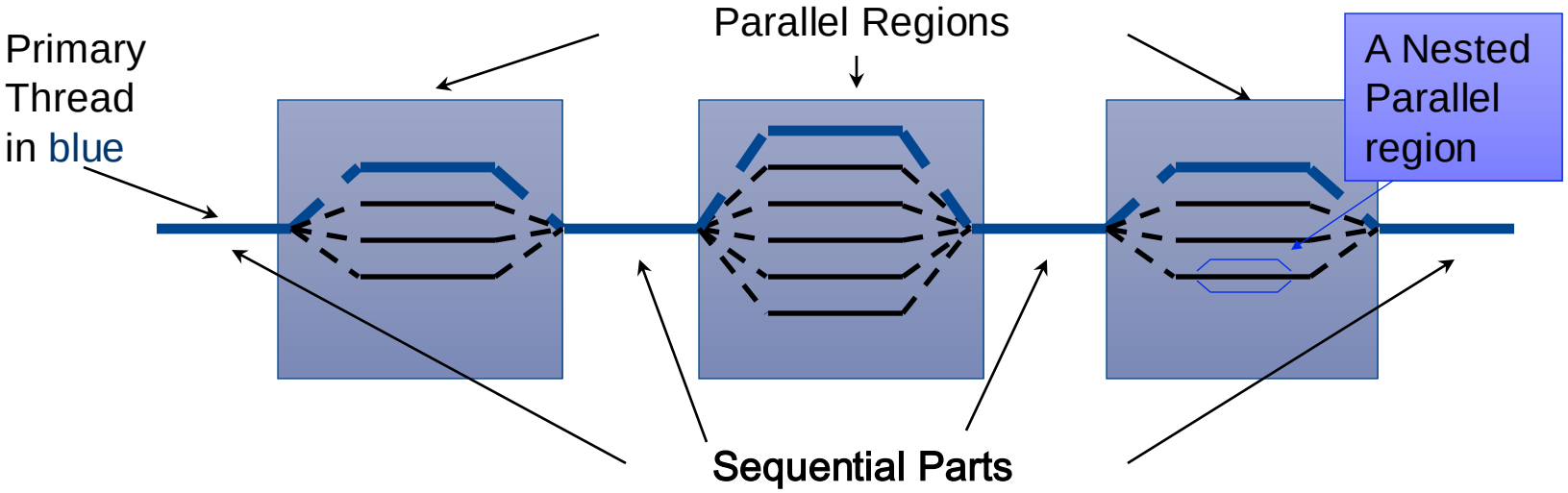
```
> nagfor -openmp hello.f90 -o hello
> ./hello
 Hello World from thread 1
 Hello World from thread 5
 Hello World from thread 6
 Hello World from thread 3
 Hello World from thread 7
 Hello World from thread 4
 Hello World from thread 2
 Hello World from thread 9
 Hello World from thread 8
 Hello World from thread 0
```

- Why is the order different?
- How do I change the number of threads?
- What happens without the OpenMP flag?

OpenMP's Model of Computation

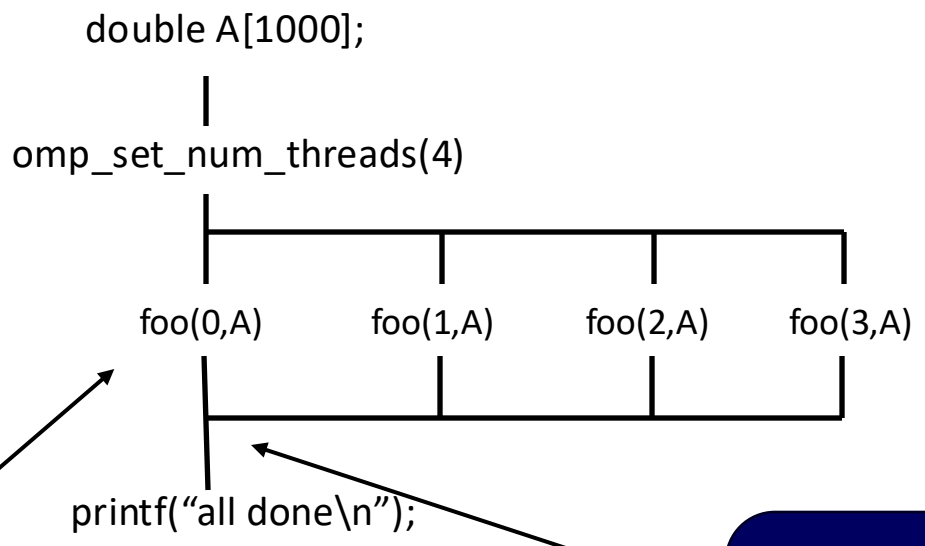
Fork-Join Parallelism

- **Primary thread** spawns a **team of threads**
- Parallelism can be added incrementally until the performance goals are met i.e. the sequential program evolves into a parallel program



OpenMP – Thread Creation

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
  int tid = omp_get_thread_num();
  foo(tid, A);
}
printf("all done\n");
```



Each thread computes the same code with **different** tid. A **single copy of A** is shared among all threads.

Threads wait here for all threads to finish before proceeding (i.e. an implicit *barrier*)

OpenMP Memory and Synchronisation

OpenMP Memory Organisation

Shared Variables (default)

- A variable occupies **the same memory location** on all threads, updates are eventually visible for all the threads
- Memory consistency is implementation-dependent but programmable

```
#pragma omp parallel shared (var_list)
```

Private Variables (need to specify)

- Replicates each specified variable on each thread's stack
- Private to thread, implicitly created inside the parallel region (and destroyed afterwards, why?)

```
#pragma omp parallel private (var_list)
```

OpenMP Memory Organisation

Initialisation and sharing of private variables

- `firstprivate` is a special case of private, initialises each private copy with the corresponding value from the primary thread

```
#pragma omp parallel firstprivate (tmp)
```

- lastprivate passes the value of a private variable from the last iteration to a global variable

```
#pragma omp parallel for lastprivate (tmp)
```

(*lastprivate requires work-sharing directives which we will cover later)*

OpenMP Memory Organisation

Default shared settings

- **Variables allocated prior to entering a parallel region** are shared inside the region by default
- **Loop index** variables of parallel do/for loops in parallel regions are private by default
- Global variables (i.e. common blocks) are shared by default

NOTE: dynamically allocated memory can’t be used with private. Each thread needs to allocate its own chunk. Why?

OpenMP Shared Variables

OpenMP provides a **relaxed-consistency** memory model

- Shared variables might be accessed by multiple threads
- Each thread has its **own temporary view** of the shared variables, as it might access from a local copy stored in cache or from a register
- There is **no guarantee** that the local copy is immediately flushed to the shared memory space after an update

Synchronisation is needed to impose ordering constraint and correctness

- High level synchronisation
  - Critical section, atomic update, ordered execution and barriers
- Low level synchronisation
  - Memory flush and locks

Access To Shared Variables

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

res is shared because it is declared before the parallel region

All threads update to res

Result depends on **relative speed** of the accessing threads (**race condition**)

Access To Shared Variables – OpenMP Critical

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

res is shared because it is declared before the parallel region

All threads update to res

Result depends on **relative speed** of the accessing threads (**race condition**)

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

Threads wait their turn – only one at a time performs res = res + B

Mutual exclusion: Only one thread at a time can enter a **critical** region.

Access To Shared Variables – OpenMP Atomic

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

res is shared because it is declared before the parallel region

All threads update to res

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

Atomic only protects the read/update of res

Result depends on **relative speed** of the accessing threads (**race condition**)

Atomic provides mutual exclusion but only applies to the **read/update** of a memory location

OpenMP Barrier Construct

Used to synchronise threads, each thread waits until all threads has arrived at the **barrier**

```
#pragma omp parallel shared (A, B, C) private(id)
{
  id = omp_get_thread_num();
  A[id] = big_calc1(id);
  #pragma omp barrier
  for(i = 0; i < N; i++){
    C[i] = big_calc3(i, A);
  }
}
```

OpenMP Master Construct

A region of code that only the **primary** thread executes. The other skips over the section (**no synchornisation**)

```
#pragma omp parallel
{
#pragma omp master
  {
    int k = omp_get_num_threads();
    printf ("Number of Threads requested = %d\n",k);
  }
  /* other work */
}
```

OpenMP Single Construct

A region of code that is executed by the **first thread** who encounters it (not necessarily the primary thread). An **implicit barrier** is implied at the end of the block (can be removed with a nowait clause)

```
#pragma omp parallel
{
#pragma omp single nowait
  {
    int k = omp_get_num_threads();
    printf ("Number of Threads requested = %d\n",k);
  }
  /* other work */
}
```

OpenMP Locks

Low level general purpose locks provided by the OpenMP runtime, similar in use to **semaphores**, available if unset

```
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
  id = omp_get_thread_num();
  tmp = do_lots_of_work(id);
  omp_set_lock(&lck);
  printf("%d %d\n", id, tmp);
  omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
```

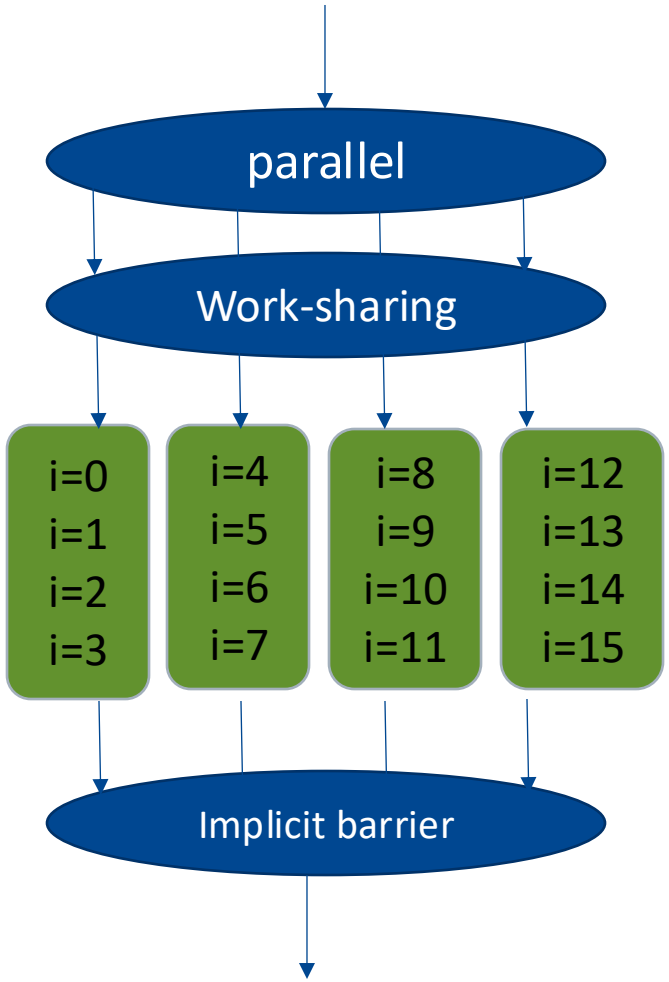
What is the difference between a critical section and locks ?

OpenMP
Work-sharing

OpenMP – Work-sharing

Threads are assigned, an independent subset of the total workload

For example, different chunks of an iteration is distributed among the threads



OpenMP loop work-sharing construct

OpenMP's loop work-sharing construct divide loop iterations among all active threads

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

C/C++

```fortran
!$omp parallel
!$omp do
do i = 1, N
  a(i) = a(i) + b(i)
end do
!$omp end do
!$omp end parallel
```

Fortran

The variable i is made "private" to each thread by default. You could do this explicitly with a "private(i)" clause

Combined OpenMP Constructs

OpenMP allows for a combined parallel and work-sharing directive (do/for)on the same line

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

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

However, for performance reasons one often aims at having as large as possible parallel regions

Working with Loops

Basic approach:

- Find compute intensive loops (use a profiler!)
- Make the loop iterations independent, such that they can safely be executed in any order without loop-carried dependencies
- Place the appropriate OpenMP work-sharing directive, test and debug

Reductions

A **reduction** is used to create code for recurrence calculations (associative and communitave operators) so that they can be performed in parallel

- Very common in numerical methods e.g. computing averages, norms or finding min/max

```
double avg = 0.0;
double A[MAX];
for(int i = 0; i < MAX; i++) {
  avg + = A[i]; /* <- reduction */
}
avg /= MAX;
```

- Support in most parallel programming environments

Reductions in OpenMP

OpenMP reduction clause:

reduction (op : list)

Inside a parallel or a work-sharing construct:

- A **local copy** of each variable in the list is created and initialised depending on the “op” (e.g. 0 for “+”)
- **Updates** to the **local copy**
- Local copies are **reduced into a single value** and combined with the original global value

```
double avg = 0.0;
double A[MAX];
#pragma omp parallel for reduction (+:avg)
for(int i = 0; i < MAX; i++) {
  avg + = A[i]; /* <- reduction */
}
avg /= MAX;
```

The variables in the “list” must be shared in the enclosing parallel region

OpenMP Loop Scheduling

The OpenMP runtime decides how the loop iterations are distributed among all threads – scheduling

OpenMP defines the following loop scheduling choices:

- **Static** – predefined at compile time. Lowest overhead, predictable
- **Dynamic** – Selection made at runtime
- **Guided** – Special case of dynamic; attempts to reduce overhead
- **Auto** – When the runtime can “learn” from previous executions of the same loop nest

The schedule clause

The schedule clause affects how loop iterations are mapped onto threads:

- `schedule (static [chunk])`
  - Deal-out blocks of iterations of size “chunk” to each thread
- `schedule (dynamic [chunk])`
  - Each thread grabs “chunk” iterations off a queue until all iterations have been given out
- `schedule (guided [chunk])`
  - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds
- `schedule (runtime)`
  - Schedule and chunk size is taken from the OMP_SCHEDULE environment variable (or the runtime)
- `schedule (auto)`
  - Schedule is up to the runtime to choose (does not have to be any of the above)

The schedule clause

schedule.f90

```fortran
program schedule
  use omp_lib
  use, intrinsic :: iso_c_binding
  integer :: i
  integer, parameter :: n = 1000
  integer :: buffer(n)


  interface
     subroutine usleep(u) bind(c)
       use, intrinsic :: iso_c_binding
       integer(kind=c_long), value :: u
     end subroutine usleep
  end interface


  !$omp parallel do schedule(runtime)
  do i = 1,n
     buffer(i) = omp_get_thread_num()
     call usleep(int(rand(buffer(i))*2000, kind=c_long))
  end do
  !$omp end parallel do

  open(1, file="schedule.dat")
  do i = 1, n
     write(1, *) i, buffer(i)
  end do
  close(1)
end program schedule
```

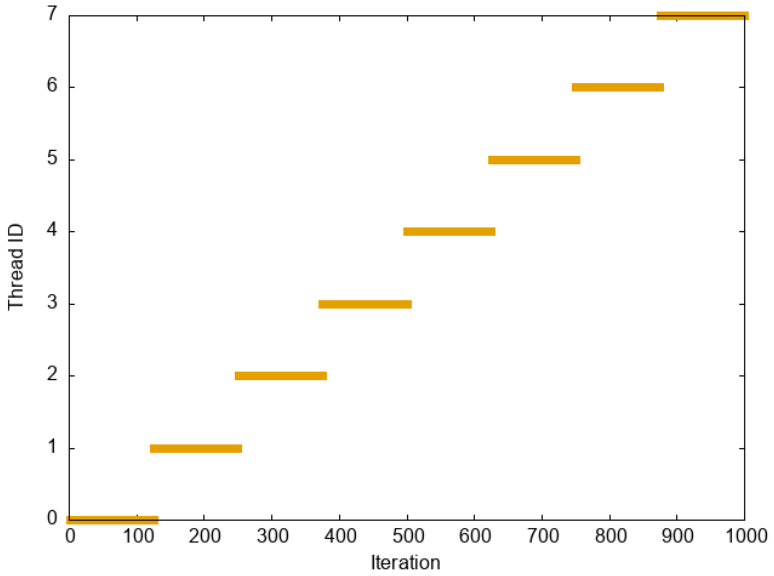
schedule.gp

```
set terminal png
set autoscale
set output "schedule.png"
set xlabel "Iteration"
set ylabel "Thread ID"
unset key
plot "schedule.dat" using 1:2 ls 4
```

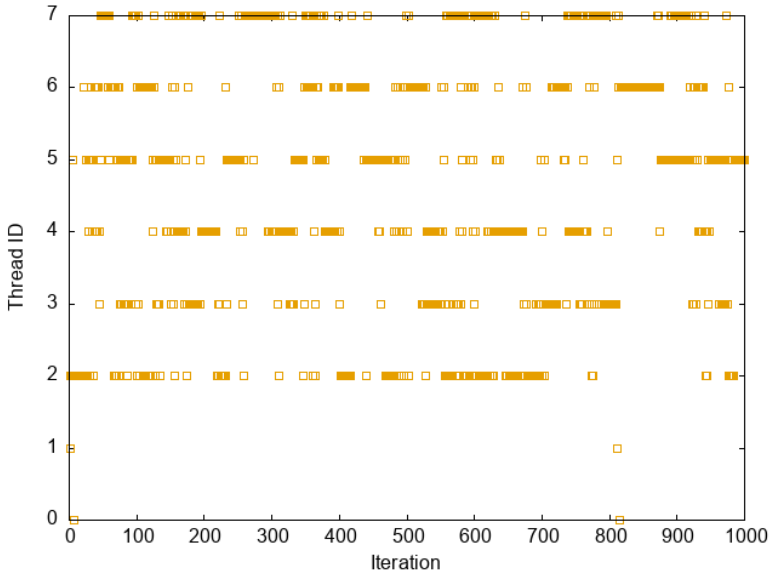
```
> gfortran -fopenmp schedule.f90
> env OMP_SCHEDULE=static ./a.out
> gnuplot schedule.gp

Now check schedule.png
```

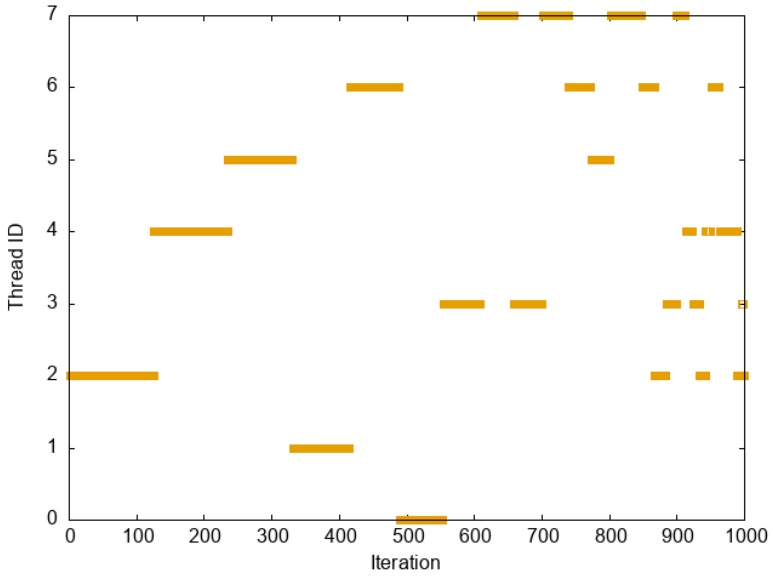
The schedule clause



Static



Dynamic



Guided

Sections Work-sharing Construct

The sections work-sharing construct assigns a different part of the code to each thread

```
#pragma omp parallel
{
  #pragma omp sections
  {
  #pragma omp section
    x_calculation();
  #pragma omp section
    y_calculation();
  #pragma omp section
    z_calculation();
  }
}
```

C/C++

Ideal for when each subproblem doesn't have enough parallelism for work-sharing do/for

References

OpenMP specifications:

http://www.openmp.org/specifications/

OpenMP API syntax reference guide

https://www.openmp.org/wp-content/uploads/OpenMPRefGuide-5.2-Web-2024.pdf