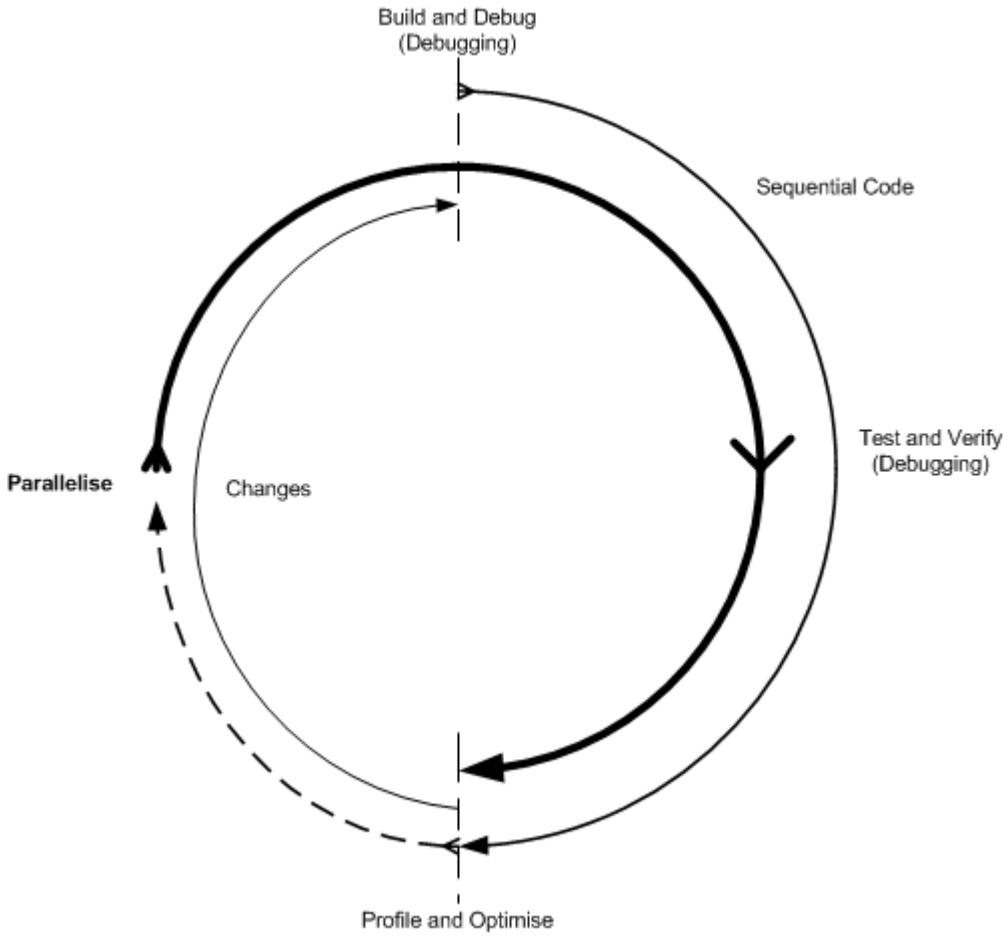
University of East Anglia

Parallelising Scientific Codes Using OpenMP

Wadud Miah
Research Computing Group

Software Performance Lifecycle



Scientific Programming

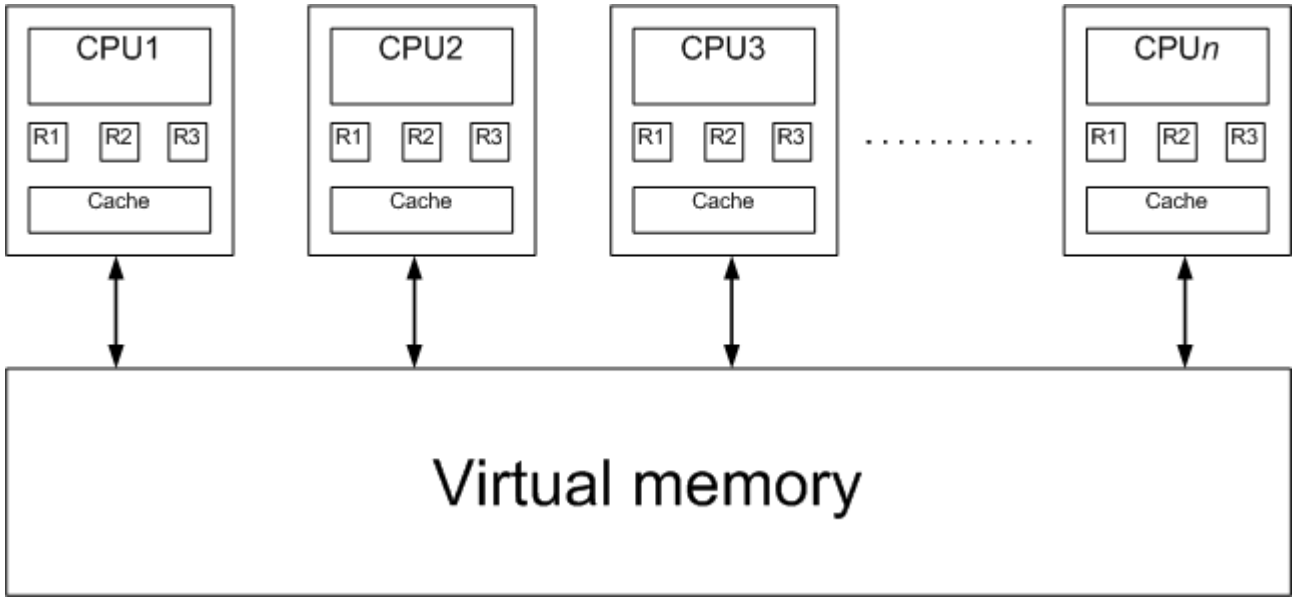
- Early scientific codes were mainly sequential and were executed on single core CPUs;
- CPU performance accelerated from 1 MHz and have peaked at around 3.2 GHz;
- However, scientists want to a) refine and expand computational domains b) explore challenging scientific problems;
- Solution: parallel programming many-core architectures.

Virtual Memory Machines

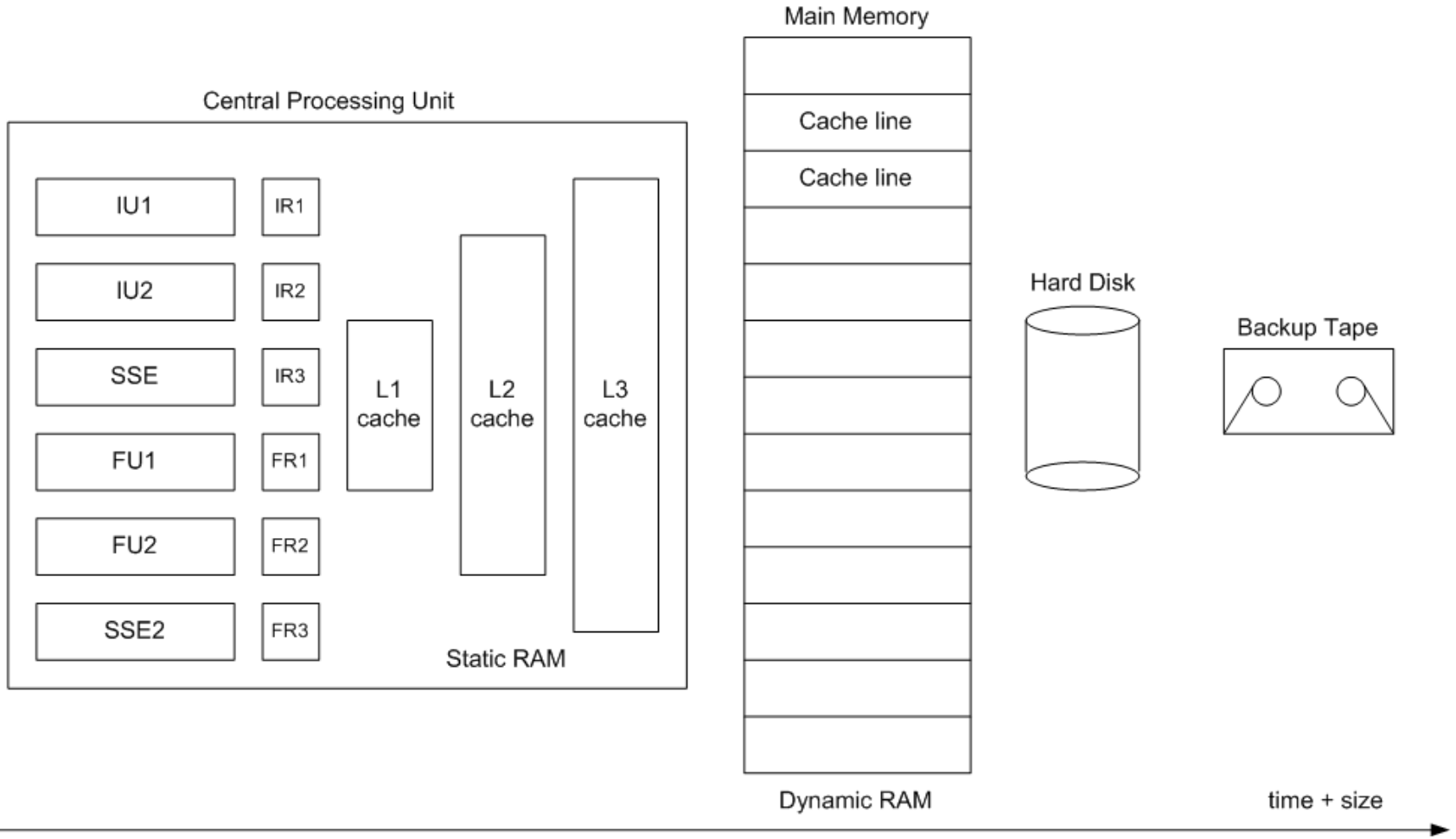
- OpenMP provides a multi-threaded parallel programming mechanism that executes on virtual memory machines;
- A virtual memory machine is where all processing units access the entire memory space in the same manner;
- This includes shared memory nodes, large parallel machines and even clusters;
- Each processing unit still has its own register set and cache memory.

Virtual Memory Diagram

- A simplistic view of virtual memory:

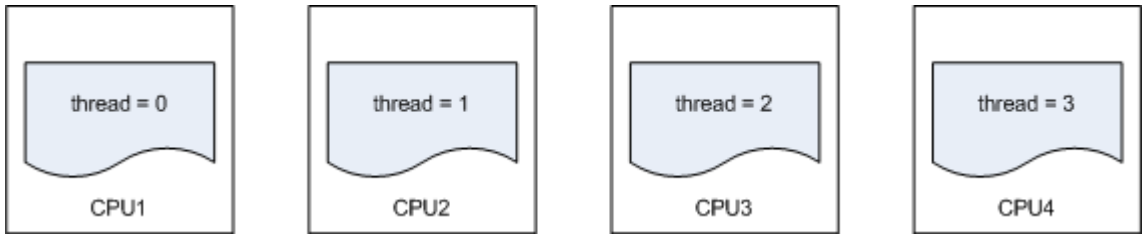


Memory Hierarchy



Parallel Programming Paradigms

- Loop parallelism - computationally intensive independent loops are parallelised;
- Single program multiple data (SPMD) - the same program is spawned on different CPUs with a different thread number;



- As OpenMP runs on a shared virtual memory environment, race conditions can occur.

Thread Structure

- A process is an instance of a program in execution;
- A thread is an independent execution path in a process;
- All threads can share the same memory space;
- Private thread memory is stored in a stack;
- Threads are quicker to create and destroy compared to processes.

OpenMP

- OpenMP (Open Multi-Processing) is a specification for virtual memory parallelism;
- It parallelises codes by spawning multiple threads and functionality is provided by the compiler;
- The developer prefixes code for parallelism with compiler directives;
- The code uses environment variables to control behaviour;
- Code is linked against parallel libraries.

Developer Assisted Acceleration

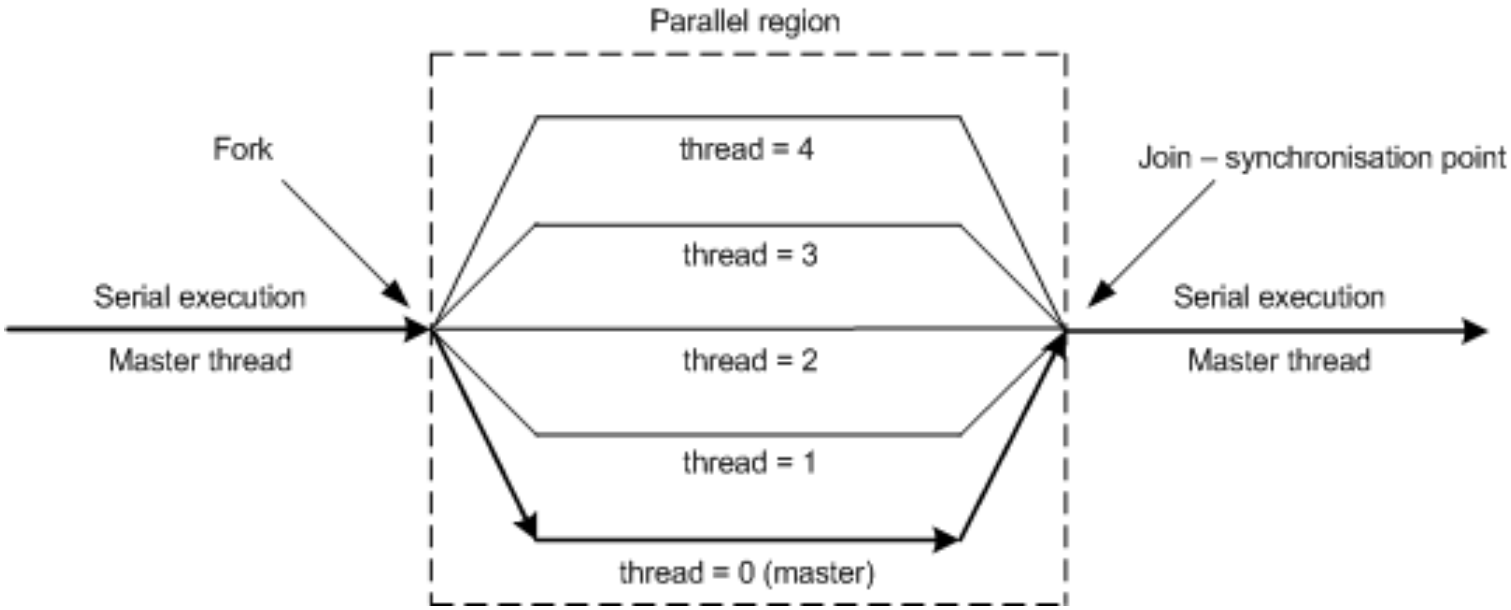
- Developers will have to assist the compiler to accelerate their codes;
- Auto parallelisation just does not work - the days of writing generic codes are history;
- Developer has intimate knowledge of how their codes work which helps the compiler;
- This is the path to code scalability for exascale computing (10^{18} FLOP/s);
- Compilers that support OpenMP: GNU, PGI and Intel.

Incremental Code Development

- OpenMP provides an incremental style of parallel software development;
- Test individual modifications to code;
- OpenMP parallelisation is added during the compilation and linking phase;
- GNU: `-fopenmp`
- PGI: `-mp`
- Intel: `-openmp`
- To switch off OpenMP, simply re-compile without the above switches.

Execution Model

- OpenMP uses the fork join model;



Parallel Region - Structured Block

- Has a single point of entry and a single point of exit and a call to exit is allowed;
- Number of threads fixed during execution;
- No jumps or goto statements are allowed within structured blocks;
- Parallel region is also known as the lexical scope or lexical extent;
- All threads synchronise at the end and the thread that finishes first is completely arbitrary.

Number of Threads

- The total number of threads is controlled by the shell environment variable `$OMP_NUM_THREADS`;
- Master thread is always number zero;
- Slave threads are numbered from `1` to `$OMP_NUM_THREADS - 1`;
- Number of threads is fixed in a parallel region;
- Different parallel regions can have different number of threads.

OpenMP Syntax

- OpenMP uses compiler directives to control thread parallelism;
- In Fortran we use a sentinel and directive with optional clauses (case insensitive):

```
!$OMP directive [clauses]
C$OMP directive [clauses]
*$OMP directive [clauses]
```

- In C/C++ we use pragmas and directive with optional clauses (case sensitive):

```
#pragma omp directive [clauses]
```

Parallel Regions (1)

- Block of code that runs independently and is a candidate for parallelisation;
- Loops in the form cannot be parallelised

$u^{n+1} = f(u^n, u^{n-1}, u^{n-2}, \ldots, u^{n-k});$

- This has to be identified by the developer;
- In Fortran:

```
!$OMP PARALLEL [clauses]
    structured block
!$OMP END PARALLEL
```

- Directives are case insensitive;

Parallel Regions (2)

- In C and C++:

```
#pragma omp parallel [clauses]
{
    structured block
}
```

- Directives are case sensitive.

OpenMP Clauses

- Clauses control how variables in the structured block are controlled and accessed;
- Clauses apply to the parallel region only;
- The entire variable must be scoped - cannot be applied to certain array elements;
- Variables are typically shared, private or protected between threads;
- Shared variables can be used to communicate between threads, but they must be protected against race conditions.

Race Conditions

- Race condition is a type of bug that exists in parallel programs;
- Different execution runs of the parallel program produces different results;
- This occurs when more than one thread changes a value of a variable - each thread races to change it;
- Shared variables require protection to prevent this type of bug;
- Usually very difficult to debug.

Race Condition Example (1)

- Consider the following code example:

```
#pragma omp parallel
{
  i = omp_get_thread_num( ) + 1;
  race = race + i;
}
```

- For four threads, we should get `race = 10;`
- However, we might get this value but the outcome is completely arbitrary.

Race Condition Example (2)

- Starting sequence is completely arbitrary;

thread 0	thread 1	thread 2	thread 3	race
read 0				0
i = 1	read 0			0
add 0 + 1	i = 2			0
store 1	add 0 + 2		read 1	1
	store 2	read 2	i = 4	2
		i = 3	add 1 + 4	2
		add 2 + 3	store 5	5
		store 5		5

- The last value of 5 is obviously incorrect.

Private Clause

- A private variable is local only to a thread - other threads cannot access it;
- The syntax for a private clause is:

  `private (variables)`

- Variables are declared on the thread memory stack;
- Variables are undefined upon entry and exit to the parallel region even if they have been initialised.

Shared Clause

- A shared variable is shared by all threads and is the same value for all threads;
- The syntax for a shared clause is:

  `shared (`*`variables`*`)`

- Threads can communicate with each other using shared variables;
- Shared data needs to be protected to prevent race conditions;
- Arrays are generally shared between threads where they work on their chunk of elements.

Default Clause

- Allows a default scoping for variables;
- Syntax is:

```
default(type)
```

- Values include `none`, `shared` and `private`;
- Default is usually `shared` and loop variables are `private`;
- Do not rely on defaults. Always use `default(none)` and explicitly scope all variables. Easier to read and debug.

IF Clause

- Branches into the parallel region if a condition is true;
- Can be used to decide if sufficient work is available to parallelise;
- Syntax is:

```
!$OMP PARALLEL IF (cond)
#pragma omp parallel if (cond)
```

Work Sharing Constructs

- Work sharing constructs are used to divide the work in a parallel region;
- `do` or `for` - share loop iterations between threads for Fortran and C/C++ respectively;
- `sections` - share blocks of code between threads;
- `single` - only one thread to carry out the block;
- `workshare` - share Fortran 90 array operations between threads.

Fortran DO Construct

- The syntax for a Fortran DO construct is:

```
!$OMP DO [clauses]
      do
            ! loop body
      end do
!$OMP END DO [NOWAIT]
```

- The loop is executed in parallel by the threads;
- The loop must iterate a fixed number of times with no branching out.

Fortran DO Example

```
real*8 :: a(n), b(n)
integer :: i

!$OMP PARALLEL default(none) &
!$OMP shared(a, b) private(i)
!$OMP DO
  do i = 1, n
    a(i) = a(i) + b(i)
  end do
!$OMP END DO
!$OMP END PARALLEL
```

C/C++ FOR Construct

- The syntax for a C/C++ for construct is:

```
#pragma omp for [nowait] \
  [clauses]
for ( ; ; ) {
  /* loop body */
}
```

- The loop is executed in parallel by the threads;
- The loop must iterate a fixed number of times with no branching out.

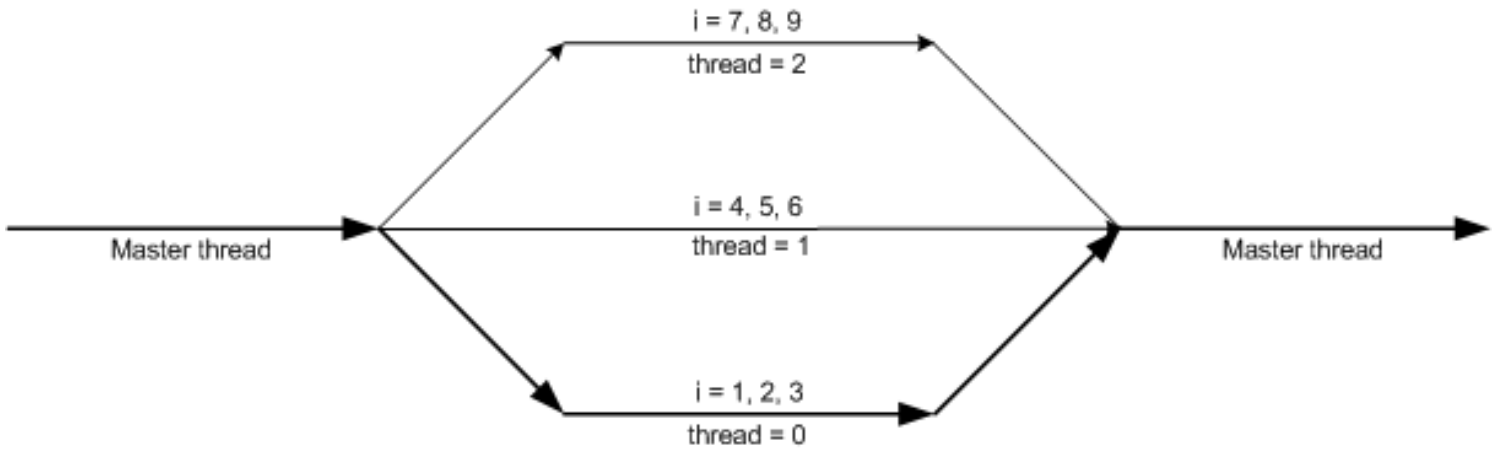
C/C++ FOR Example

```
double a[n], b[n];
int i;

#pragma omp parallel default(none) \
  shared(a, b) private(i)
#pragma omp for
  for ( i = 0 ; i < n; i++ ) {
    a[i] = a[i] + b[i];
  }
```

Loop Construct Diagram

- Example loop that iterates i = 1 to 9 with three threads:



- Above schedule is `schedule(static,3)`
- Default schedule is implementation dependent.

Clauses for DO and FOR Loops

- `private` - variable is private to thread;
- `firstprivate` - variable holds its value when entering parallel region;
- `lastprivate` - variable holds its value when leaving parallel region;
- `reduction` - collating results from threads;
- `ordered` - sequential ordering of execution;
- `schedule` - changing execution schedule;
- `nowait` - remove synchronisation.

NOWAIT Clause

- The `nowait` clause removes the implicit barrier at the end;
- Threads that complete early move on to the next work sharing construct;
- A method to keep processing units busy rather than wait for all threads to complete.

Sections Construct

- The sections construct allows for non-iterative execution;
- Used for blocks of code that are independent;
- Useful for functional parallelism, e.g. *a = f(a) + g(a) + h(a)*;
- Each block is executed by a single thread;
- Implied barrier at the end which can be removed with a `nowait` clause;
- This feature should be avoided as load balancing is difficult to obtain.

Fortran Sections

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
   first_block
!$OMP SECTION
   second_block
!$OMP SECTION
   third_block
!$OMP END SECTIONS [NOWAIT]
!$OMP END PARALLEL
```

C/C++ Sections

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
   first_block
#pragma omp section
   second_block
#pragma omp section
   third_block
}
```

Sections Clauses

- `private`
- `reduction`
- `nowait`
- `firstprivate`
- `lastprivate`

Single Clause

- The single construct ensures only one thread executes a block of code;
- All threads will wait at the end of the single block unless `nowait` is specified;
- Fortran syntax:

```
!$OMP SINGLE [clauses]
  block
!$OMP END SINGLE
```

- C/C++ syntax:

```
#pragma omp single [clauses]
  block
```

Single Clauses

- `private` - variable is private
- `nowait` - synchronisation is removed;
- `firstprivate` - variable holds its value when entering parallel region;
- `copyprivate` - a private variable is broadcasted to all other threads not executing.

Fortran 90 Workshare Construct

- The workshare parallelises Fortran 90 array operations;
- Example:

```
!$OMP PARALLEL SHARED(a, b, n)
!$OMP WORKSHARE
  a(1:n) = b(1:n) + 1.0
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

Reduction Clause

- Used for safely collating results from all threads using associative and commutative operations;
- Requires the developer to specify the variable(s) and operation;
- Always use the reduction clause instead of implementing your own;
- This feature has been optimised, so will give better performance.

Fortran Reduction

- Clause - `reduction(op/intrinsic : list)`
- Statement form:

```
x = x op exp
x = exp op x (op ≠ -)
x = intrinsic(x, exp)
x = intrinsic(exp, x)
```

`x` is scalar variable

`exp` is a scalar expression

`op` is `+ - * .AND. .OR. .EQV. .NEQV.`

`intrinsic` is `MAX MIN IAND IOR IEOR`

C/C++ Reduction

- Clause - `reduction(op : list)`
- Statement form:

  `x = x` *`op`* `exp;` `x` *`binop`* `= exp`

  `x = exp` *`op`* `x` (*`op`* ≠ subtraction)

  `x++; ++x`

  `x--; --x`

`x` is a scalar variable of intrinsic type (no pointers)

`exp` is a scalar expression not referencing `x`

`op` (not overloaded) is `+ - * & ^ | && ||`

`binop` (not overloaded) is `+ * - & ^ |`

Fortran Reduction Initialisation

Operator	Value
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
MAX	Smallest negative number
MIN	Largest positive number
IAND	All bits on
IOR	0
IEOR	0

C/C++ Reduction Initialisation

Operator	Value
+	0
*	1
–	0
&	~0
|	0
^	0
&&	1
||	0

Fortran Reduction Example

```
!$OMP PARALLEL default(none) &
!$OMP shared(sum, a) private(i)
!$OMP DO reduction(+ : sum)
do i = 1, n
  sum = sum + a(i)
end do
!$OMP END DO
!$OMP END PARALLEL
```

C/C++ Reduction Example

```
#pragma omp parallel default(none) \
  shared(sum, a) private(i)
#pragma omp for reduction(+ : sum)
  for ( i = 0; i < n; i++ ) {
    sum += a[i];
  }
```

Synchronisation

- Synchronisation is used to ensure proper ordering of executions;
- This is to ensure correctness and to prevent race conditions;
- All threads will be at a single point;
- Should only be used when absolutely necessary as this slows down program execution and subsequently reduces code scalability.

Synchronisation Constructs

- Barrier;
- Master;
- Critical;
- Atomic;
- Flush;
- Ordered.

Barrier Construct

- A barrier construct implements a synchronisation point:

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

- This is contained within a parallel block;
- No thread will proceed any further than this point until all threads have reached this point.

Master Construct

- Code block is only executed by master thread;
- No implicit barrier at the end - add a barrier if one is required:

```
!$OMP MASTER
  block
!$OMP END MASTER

#pragma omp master
  block
```

Critical Construct

- Critical safely executes a block of code to ensure no race conditions occur;
- All threads execute the block one at a time;
- Thread execution sequence is completely arbitrary;

```
!$OMP CRITICAL [label]
  block
!$OMP END CRITICAL

#pragma omp critical [label]
  block
```

Atomic Construct

- Atomic safely updates variables by multiple threads - preventing race conditions:

```
!$OMP ATOMIC
  expression
```

```
#pragma omp atomic
  expression
```

- Atomic operations are better optimised than critical sections.

Atomic - Fortran

- Fortran `expression` is in the form:

`x = x` *`op`* `exp` or `x = exp` *`op`* `x`

`x =` *`intrinsic`*`(x, exp)` or

`x =` *`intrinsic`*`(exp, x)`

`x` is scalar variable

`exp` is a scalar expression not referencing `x`

`intrinsic` is `MAX MIN IAND IOR IEOR`

`op` is `+ - * / .AND. .OR. .EQV. .NEQV.`

Atomic - C/C++

- C/C++ `expression` is in the form:

```
x op = exp;
x++; ++x;
x--; --x;
```

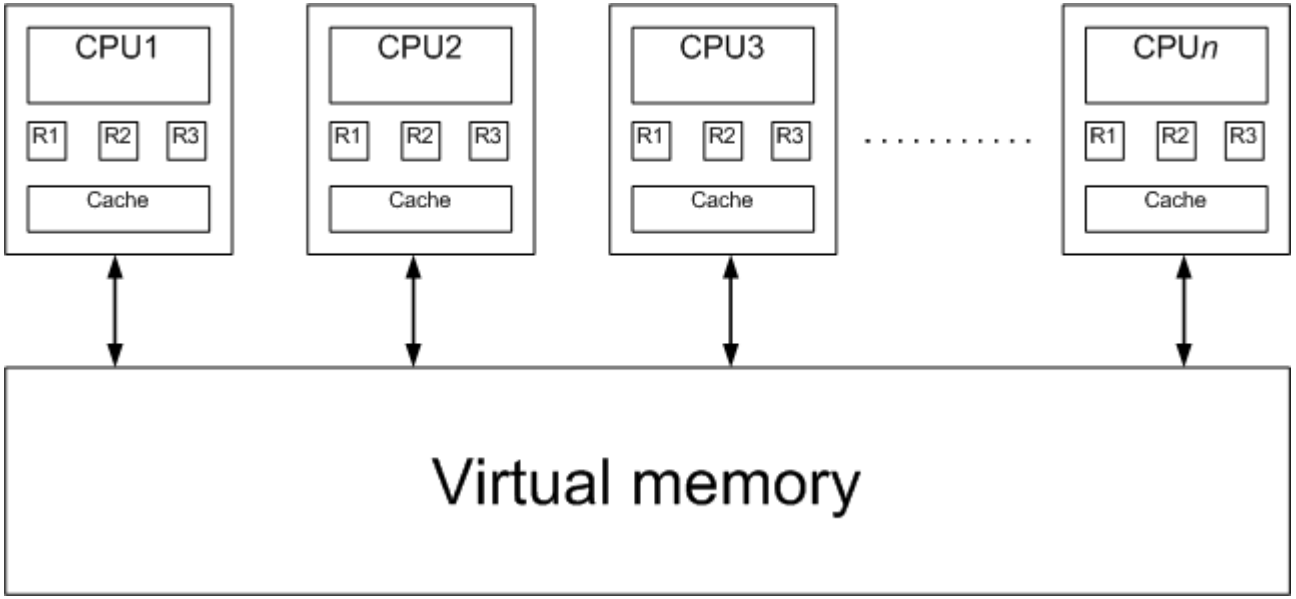
`x` is scalar variable

`exp` is a scalar expression not referencing `x`

`op` is (not overloaded) `+ - * / & ^ | << >>`

Flush Construct (1)

- Recall the virtual memory structure:



- Local CPU cache and registers improve performance;

Flush Construct (2)

- When a thread updates a value, it may need to broadcast this value to all other threads;
- When a thread requires the latest value, it requests it from the other threads;
- This process is known as cache coherency;
- Flush is implied at entry to: critical and ordered;
- Flush is implied at exit from: critical, ordered, do (and for), parallel, sections, single and workshare. Also implied at barrier;
- Cannot have the nowait clause;

Flush Construct (3)

- Syntax for flush construct:

```
!$OMP FLUSH [list]
#pragma omp flush [list]
```

- If list is omitted, all shared variables are flushed;
- If list is provided, only listed variables are flushed;
- To increase efficiency or program, only specify variable(s) that require flushing;
- No implied synchronisation point at flush.

Ordered Clause and Construct

- Ordered allows blocks of code to be executed as if they were executed sequentially;
- Only valid within a loop construct;
- Utilise if certain blocks require correct sequencing;
- To ensure greater scalability, ensure construct contains a small amount of work;
- Compute intensive code can be outside the construct.

Ordered - Fortran

```
!$OMP PARALLEL
!$OMP DO ORDERED
do
   normal-block
   !$OMP ORDERED
     ordered-block
   !$OMP END ORDERED

end do
!$OMP END DO
!$OMP END PARALLEL
```

Ordered - C/C++

```
#pragma omp parallel
#pragma omp for ordered
for ( ; ; ) {
  normal-block
  #pragma omp ordered
  {
    ordered-block
  }

}
```

Scheduling

- The scheduling feature allows the mapping of loop iterations to threads;
- The purpose of different scheduling strategies is to ensure good load balancing between threads;
- If properly load balanced, code performance will increase resulting in greater scalability;

```
!$OMP DO SCHEDULE(type, chunk)
#pragma omp for schedule(type, chunk)
```

Scheduling Types (1)

- `schedule( static, chunk )`

iterations are divided into pieces of chunk size and assigned statically to threads in a round robin manner. Last thread may have fewer than chunk size. Has the lowest overhead;

- `schedule( dynamic, chunk )`

iterations are broken into pieces of chunk size. When a thread completes processing one chunk, it dynamically fetches another. Has the highest overhead so ensure there is plenty of work;

Scheduling Types (2)

- `schedule( guided, chunk )`

chunk size exponentially decreased with each iteration. Chunk size specifies the minimum iterations to be dispatched;

- `schedule( runtime )`

schedule type obtained via the environment variable `$OMP_SCHEDULE`

Runtime Functions (1)

- `omp_set_num_threads(`*`value`*`)` - sets the number of threads for a parallel region;
- `omp_get_num_threads()` - gets the current number of threads inside a parallel region. Returns 1 outside a parallel region;
- `omp_get_max_threads()` - returns the maximum number of threads, namely the value of `$OMP_NUM_THREADS`;

Runtime Functions (2)

- `omp_get_thread_num()` - get unique thread ID. Ranges from 0 to `$OMP_NUM_THREADS` - 1;
- `omp_get_num_procs()` - gets the number of processors available;
- `omp_in_parallel()` - returns true if in parallel region;
- `omp_set_dynamic(`*`value`*`)` - enables or disables dynamic adjustment of threads;
- `omp_get_dynamic()` - returns the value of the above.

Timing Functions

- OpenMP has timing routines to measure performance gains;
- They measure wall clock time and return double precision values;
- `omp_get_wtime()` - returns time measured in seconds from some fixed point;
- `omp_get_wtick()` - returns clock resolution.

Timing Example

```
t1 = omp_get_wtime();
// lots_of_work
t2 = omp_get_wtime();

// how long lots_of_work took in
// seconds
dt = t2 - t1;
```

Conditional Compilation

- The above subroutines will not be resolved if the OpenMP compilation flag is switched off;
- Embed subroutine calls in pre-processor conditionals:

```
#ifdef _OPENMP
  thread = omp_get_thread_num();
#else
  thread = 0;
#endif
```

- This is for both Fortran and C/C++.

Header Includes

- For Fortran include the file:

```
include "omp_lib.h"
```

or use pre-compiled header (or module):

```
USE omp_lib
```

- For C/C++ include the file:

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

- No need to specify the `-I` flag for header directory search during compilation.

Conclusion

- OpenMP provides a directive based programming specification for multi-threaded parallelism;
- OpenMP runs on virtual memory machines, e.g. a single node on a cluster;
- Provides an incremental style of programming which makes it easy to test;
- OpenMP is easier than MPI but does not scale as well. However, large memory nodes can increase scalability with intelligent scheduling.

Practical Exercises

- Please use the Intel compilers:

```
module load icc/intel/12.1
```

- To compile an OpenMP C program:

```
icc -openmp program.c -o program
```

- To compile an OpenMP C++ program:

```
icpc -openmp program.cc -o program
```

- To compile an OpenMP Fortran program:

```
ifort -openmp program.f90 -o program
```

- Ensure module is loaded before running program unless the `-static` flag is used.

Reference

1. *Using OpenMP*, B. Chapman, *et al*;
2. *Parallel Programming in OpenMP*, R. Chandra, *et al*;
3. *Parallel Programming in C with MPI and OpenMP*, M. Quinn;
4. *Introduction to High Performance Computing for Scientists and Engineers*, G. Hager and G. Wellein.