THE UNIVERSITY of York

High Performance Computing - Advanced OpenMP

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

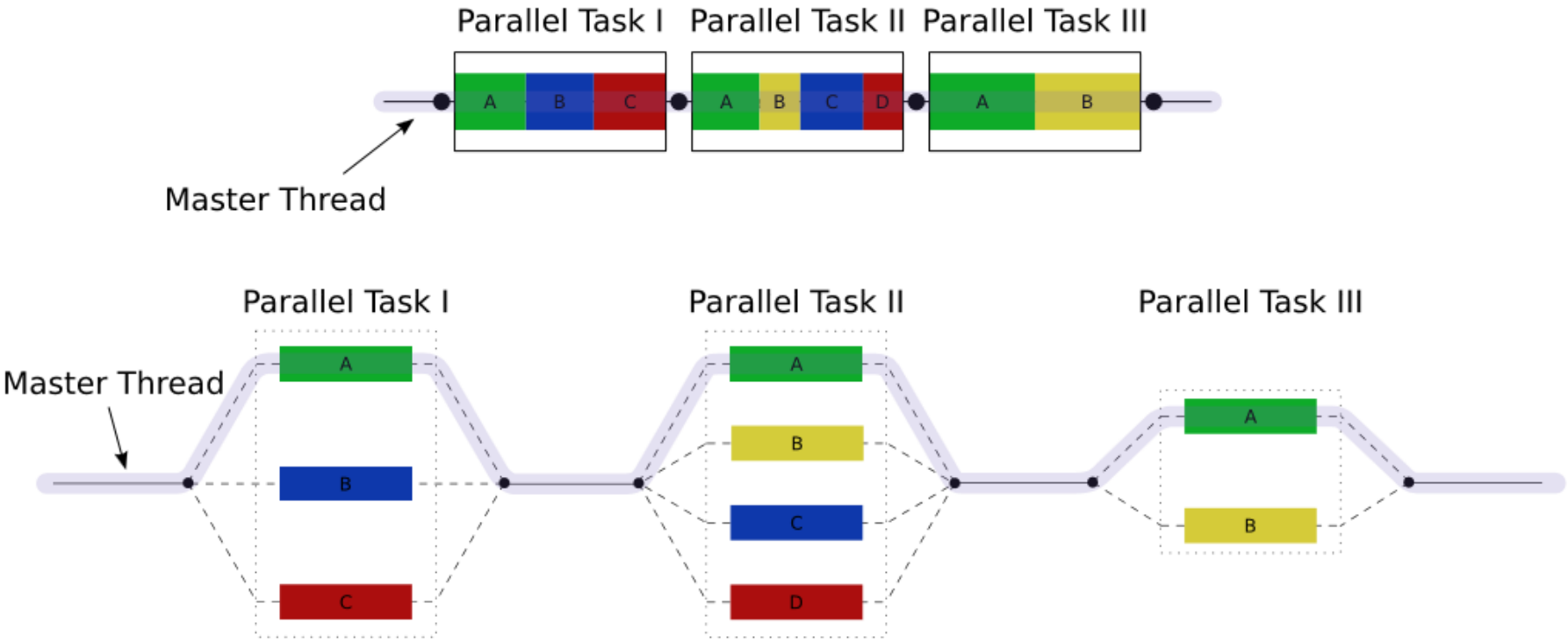
- History of OpenMP
- Advanced OpenMP
  - V1: synchronization and sections
  - V2: nested parallelism and workshare
  - V3: tasks
  - V4+ using accelerators, etc
- OpenMP Tips and Gotchas
- OpenMP Implementations

History of OpenMP

- Began as proprietary SGI directives
- Became open standard in 1997
  - OpenMP v1.0 (Fortran = 1997, C/C++ in 1998)
    - PARALLEL, PARALLEL DO, PRIVATE/SHARED/REDUCTION
    - OMP_GET_NUM_THREADS etc
    - (covered all this briefly in previous OpenMP lecture)
    - Also synchronization constructs such as MASTER/CRITICAL/BARRIER etc
    - Additional worksharing constructs such as SECTIONS and SINGLE

OpenMP Thread Model

- Program execution begins with single *master* thread
- Parallel construct creates *team* of threads (which includes the master) using *fork-join* model



picture from Wikipedia

Synchronization Constructs

- MASTER … END MASTER
  - Enclosed code is only executed by master thread
  - Other threads skip
  - No synchronization at end
  - No implied BARRIER at start or end
- SINGLE … END SINGLE
  - Execute on one thread but not necessarily master
  - Implies synchronization at start

Synchronization Constructs II

- CRITICAL … END CRITICAL
  - Enclosed code is only executed by one thread at a time
  - A thread will wait at start until no other thread is executing same block
  - No synchronization at end
- BARRIER (single statement)
  - Synchronizes all threads in a team
  - Must be reached by all or none of the threads

Synchronization Constructs III

- ATOMIC
  - Following single statement has protected memory update so that cannot have *race condition*
  - i.e. prevents multiple threads writing to same memory location
- FLUSH
  - Synchronization of all pending read/ writes to memory (or specified list of vars) for all threads
  - Often implicit, e.g. BARRIER, PARALLEL, END DO etc but not with MASTER or WORKSHARE

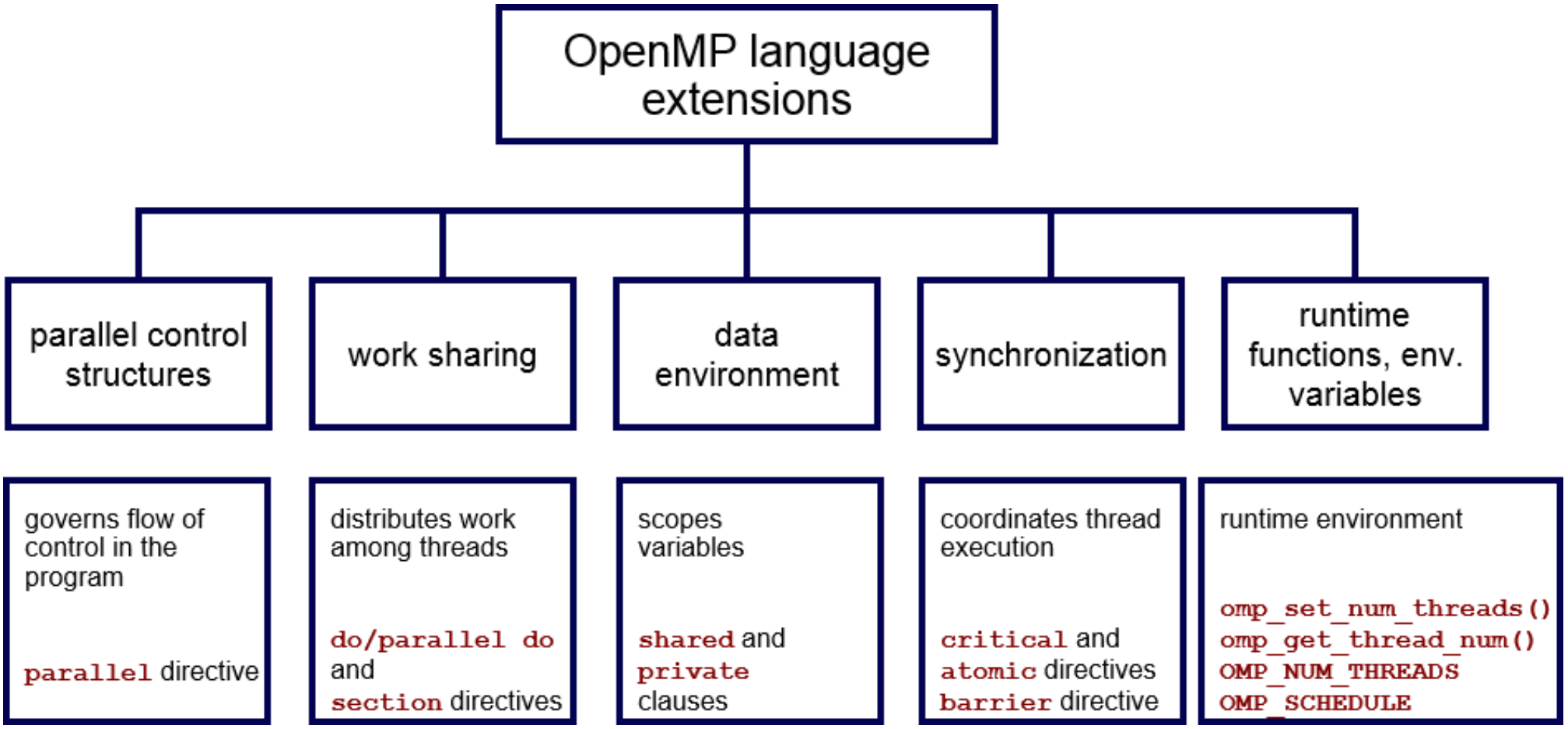
SECTIONS

- Specify a parallel block, wherein each designated SECTION is executed once by one thread in team

```
!$OMP SECTIONS [clause …]
!$OMP SECTION
… code …
!$OMP END SECTION
!$OMP SECTION
… code …
!$OMP END SECTION
!$OMP END SECTIONS [NOWAIT]
```

- Optional clause on entry specifies PRIVATE, REDUCTION etc.
- Arbitrary number of SECTION … END SECTION blocks
- Optional NOWAIT means no barrier – any thread can continue immediately

Structure of OpenMP



picture from Wikipedia

History of OMP (II)

- Began as proprietary SGI directives
- Became open standard in 1997
  - OpenMP v1.0 (Fortran = 1997, C/C++ in 1998)
    - PARALLEL, PARALLEL DO, PRIVATE/SHARED/REDUCTION
    - OMP_GET_NUM_THREADS etc
    - Also synchronization constructs such as MASTER/CRITICAL/BARRIER etc
  - OpenMP v2.0 (Fortran = 2000, C/C++ in 2002) added
    - NESTED parallelism
    - WORKSHARE, PARALLEL WORKSHARE
    - V2.5 (unified Fortran and C/C++ in 2005)

Nested Parallelism

- Can have a PARALLEL construct inside another one
  - Only if supported by this implementation
    - Use OMP_GET_NESTED( ) to test (returns true/false – also SET version available)
    - Also OMP_NESTED environment variable
  - If not supported then code only gets 1 thread per nested region i.e. no additional parallelism
  - Can create arbitrary number of new teams of threads

WORKSHARE

- “The WORKSHARE directive divides the work of executing the enclosed code into separate units of work, and causes the threads of the team to share the work of executing the enclosed code such that each unit is executed only once. The units of work may be assigned to threads in any manner as long as each unit is executed exactly once.”

```
!$OMP WORKSHARE
… block of code …
!$OMP END WORKSHARE [NOWAIT]
```

- The block of code is divided up between threads according to given rules
- Optional NOWAIT means no barrier – any thread can continue immediately

WORKSHARE example

```
INTEGER, DIMENSION(1:M,1:N) :: A,B,C,D
…
!$OMP PARALLEL WORKSHARE
   A=B+C
   WHERE (C/=0) D=1/C
!$OMP END PARALLEL WORKSHARE
```

- Can use WORKSHARE over array operations including WHERE and FORALL
- Can execute iterations in any order
- Distribution of threads up to compiler

History of OpenMP (III)

- Became open standard in 1997
  - OpenMP v1.0 (Fortran = 1997, C/C++ in 1998)
    - PARALLEL, PARALLEL DO, PRIVATE/SHARED/REDUCTION
    - OMP_GET_NUM_THREADS etc
    - MASTER/CRITICAL/BARRIER etc
  - OpenMP v2.0 (Fortran = 2000, C/C++ in 2002)
    - NESTED parallelism
    - WORKSHARE, PARALLEL WORKSHARE
    - V2.5 (unified Fortran and C/C++ in 2005)
  - OpenMP v3.0 (2008)
    - TASKS

TASKS

- TASK … END TASK defines a block of code which is packaged up for later execution
  - Some thread in the parallel region will execute the task at some point in future
  - Has an overhead but more efficient than forking a POSIX thread as OpenMP threads stay alive after join
- Tasks can be nested and by default, the parent task will not wait for children to finish unless use TASKWAIT directive

Linked List Example

```
p = listhead;
while (p) {
   process(p);
   p=next(p);
}
```

- Classic sequential linked list traversal
- Do some work on each item in list
- Assume that items can be processed independently
- How to parallelize? No loops!

Parallel pointer chasing

```
#pragma omp parallel
{
 #pragma omp single private(p)
 {
   p = listhead;
   while (p) {
    #pragma omp task firstprivate(p)
      {
       process(p);
      }
     p=next(p);
   }
 }
}
```

Only 1 thread packages the tasks – others free to do the work

TASK includes code and data – here we make a copy of p when the task is packaged as variable might be out of scope at time of execution.

End of parallel region implies a barrier and so ensures all tasks have completed.

Parallel pointer chasing on multiple lists

```
#pragma omp parallel
{
 #pragma omp for private(p)
 for (int i=0l; i<numlists; i++){
   p = listheads[i];
   while (p) {
    #pragma omp task firstprivate(p)
      {
       process(p);
      }
     p=next(p);
   }
 }
}
```

All threads package the tasks

Using multiple linked lists so can do parallel creation of tasks
NB packaging thread might decide to execute a task without packaging it at all!

End of parallel region implies a barrier and so ensures all tasks have completed.

TASKS and data sharing

- Default rule for data sharing is firstprivate to ensure value set at creation time
- OpenMP v3.0 changes some basic rules:
  - “private” now means per task not per thread
  - “shared” means “use existing storage” whereas “private” means “make new copy”
- Variables shared in enclosing parallel context are shared in the task

Nested tasks

- Data sharing can be complicated as here:
  - parent task has private B so each child task shares this value.
  - If parent task finishes then B may be deallocated which is trouble for children!
  - Hence need taskwait before exit …

```
!$OMP task private(B)
… B=…
!$OMP task shared(B)
… compute(B) …
!$OMP end task
…
!$OMP taskwait
!$OMP end task
```

Using TASKS

- Very powerful idea but:
  - Data scoping rules can be tricky – using default(none) to be explicit is a good idea!
  - Bigger overhead than using “parallel do” etc
  - Best if user controls number & granularity of task
- NB BARRIER affects threads not tasks and must make sure all threads in team reach it, so do NOT put inside a task! Ditto workshare.
- Can put parallel regions inside as nesting.

History of OpenMP (IV)

- Became open standard in 1997
  - OpenMP v1.0 (Fortran = 1997, C/C++ in 1998)
  - OpenMP v2.0 (Fortran = 2000, C/C++ in 2002)
  - OpenMP v3.0 (2008)
  - OpenMP v4.0 (2013)
    - Support for SIMD and accelerators, extensions to TASKS …
    - V4.5 (2015) adds TASKLOOP
  - OpenMP v5 (2018)
    - Full support for accelerators

SIMD

- Many compilers have proprietary directives to aid vectorization and generate SIMD code
- OpenMP v4 gives standard set
  - shows loop should be SIMDized
  - execute iterations in SIMD chunks
  - Each chunk executed concurrently over SIMD
  - NOT divided across threads

SIMD control

```
!$OMP PARALLEL DO SIMD [clauses]
…
!$OMP END PARALLEL DO SIMD
```

- In addition to usual clauses (e.g. private, shared, etc) can also have
  - SAFELEN(length) – max no. of iterations in chunk
  - ALIGNED – specify byte alignment of vars
- Also DECLARE SIMD directive to generate SIMDized version of functions, etc

Accelerators

- Similar to OpenACC directives but
  - Supports more than just loops (only option in OpenACC v1.0)
  - Less reliance on compiler to do the work
- Non-proprietary
  - Unlike CUDA
- More than GPUs
  - Support for Xeon Phi – preferred by Intel

Accelerator Model

- One host device with multiple target devices
  - *device* = execution engine with local storage – can have multiple but all must be of same type
  - *device data environment* = data associated with target data or target region
  - TARGET constructs control how data and code is offloaded to a device
  - Code in target region executes on the device
  - Executes in serial by default– can add extra OpenMP for device-level parallelism

Accelerator Directives

- TARGET DATA – move data (either to/from device) but does not execute code
- TARGET UPDATE – updates data during a target data region
- DECLARE TARGET – compiles version of subprogram to execute on device
- MAP clause – specifies how variables are accessible – can be TO/FROM/TOFROM

Accelerator Example

```
#pragma omp target map(to:B,C) map(tofrom:sum)
#pragma omp parallel for reduction(+:sum)
for (int i=0; i<N; i++){
   sum += B[i]+C[i];
}
```

- The ‘parallel for’ will execute on target device
  - Arrays B,C will be read-only
  - Sum will be read-write
- NB target region is blocking – thread must wait until device completes. Can embed targets inside tasks to be asynchronous …

OpenMP v4 and GPUs

- What if multiple GPUs? Many multiprocessors!
  - OpenMP synchronization not possible between multiprocessors
  - Solution is the TEAMS construct to create multiple master threads to execute in parallel, spawn parallel regions, but not synchronize or communicate with each other
  - Use DISTRIBUTE construct to spread iterations of a parallel loop across teams

TEAMS example

```
#pragma omp target teams distribute parallel for \
map(to:B,C), map(tofrom:sum), reduction (+:sum)
for (int i=0; i<N; i++){
   sum += B[i]+C[i];
}
```

- This now distributes iterations across multiprocessors AND across threads within a multiprocessor!
- NB Can have multiple GPUs or MICs but all must be of same type!

OpenMP Tips and Gotchas

- There is an overhead to executing a parallel region
  - Typically 10-100 microseconds
  - Hence region must contain enough work!
- Not all loops have independent iterations which can be OpenMP parallelized
  - Try running in reverse order & see if get same result!
- PRIVATE variables are *uninitialized* on entry
  - Can use FIRSTPRIVATE instead but probably bug
  - **Always** use DEFAULT(NONE) to force explicit
- REDUCTION variables must be initialized before OMP section as original value added to on exit …

OpenMP and Scheduling

- Default schedule is compiler dependent
  - Best to always specify – do not assume STATIC
- Hard to tune chunksize for static/dynamic
  - Often more robust to tune the “number of chunks per thread” and derive chunksize from that
  - Chunksize expression must be integer but need not be constant or compile-time expression

OpenMP and huge loops

- What if loop has many lines and lots of variables?
- Best to refactor loop body into a subprogram
  - Can then make many variables local and hence private by default
  - And only pass in those that are required so easier to spot and make shared/private etc
  - Much easier to test and correctly parallelize!

OpenMP and Static Variables

- Compiling a working serial code with OpenMP caused it to break – why?
  - Most likely your code assumed that the contents of a local variable were preserved
  - OpenMP forces all locals to be static on stack (not heap) *and* have separate stack per thread
  - Need to use SAVE or static correctly and consider if the variables should be THREADPRIVATE
  - Common issue with ‘first pass’ code – may need to move it outside the parallel region

OpenMP Implementations

- No full v5 yet – see supported versions at https://www.openmp.org/resources/openmp-compilers-tools/
- gcc/gfortran
  - Activate by `-fopenmp` flag
  - OpenMP v4.5 since v6
    - Full version in C/C++ but only partial Fortran until v11
  - Partial OpenMP v5 since v9
- icc/ifort
  - Activate by `-openmp` flag
  - v17.0 for OpenMP v4.5
  - v2021.1 has partial OpenMP v5.0

Further Reading

- Chapter 6 of “Introduction to High Performance Computing for Scientists and Engineers”, Georg Hager and Gerhard Wellein, CRC Press (2011).
- OpenMP at http://openmp.org includes lots of tutorials and guides
- Nice tutorial/guide at https://computing.llnl.gov/tutorials/openMP/
- Also guides on the HPC website …