COMP90025 Parallel and Multicore Computing

OpenMP

Lachlan Andrew

School of Computing and Information Systems

2023 Semester II

OpenMP Specification

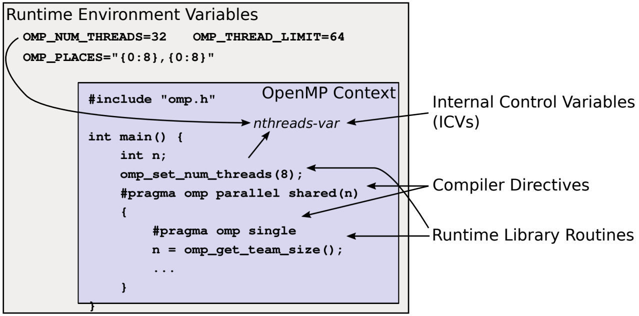
https://www.openmp.org/specifications/

- These slides draw significantly from the OpenMP 5.1 Specification (Nov 2020). You are encouraged to read the specification as well.
- The OpenMP Specification defines a parallel programming model that abstracts a *single process, multi-threaded* program execution on a (single) machine:
  - an abstract view of the machine's resources, including multi-socket, multi-core processors, threads and thread synchronization, memory hierarchy, SIMD, NUMA and devices, e.g., offloading to GPUs;
  - an API providing high-level parallel programming constructs that consistently encapsulate the machine's resources, e.g., by *transparently* making use of the OS threading library, processor SIMD vector registers and instructions, and compiling of target code for supported devices.
- *It is not an implementation*. Individual hardware and compiler vendors are required to support and implement the OpenMP Specification. An OpenMP program should run consistently across different machines/compilers, however some aspects of the specification remain *implementation dependent* and not all hardware/compiler implementations support or distinguish all of the specifications. Similarly the specifications may not cover every kind of machine implementation that exists.

Getting started

OpenMP is commonly supported on commercial OSes and hardware. Linux distributions support it via the gcc compiler. You can therefore readily develop and test OpenMP programs. On HPC facilities, the OpenMP compiler implementation should be vendor specific and support everything that the machine offers; e.g., using an Intel compiler for Intel processors.

- If you use the gcc compiler then gcc-11 is required for OpenMP 5:
  - you can build the compiler if necessary following e.g., instructions at `https://iamsorush.com/posts/build-gcc11/`
  - compiling the compiler with offloading support to GPUs is more involved, however.
- A minimal simple compile line for your OpenMP program might be: `g++-11.2 -fopenmp prog.cpp -o prog`
- You could install some other useful stuff:
  - libnuma – for C++ NUMA support functions (may need the `-lnuma` compile option) will allow you to find out which NUMA node a thread is running on
  - `hwloc-ls` and `numactl` – utilities to inspect and control NUMA on your OS, will give you a definitive overview of the processor resources, such as RAM, cache, cores, hyperthreads and NUMA characteristics



```
#include "omp.h"

int main() {
    int n;
    omp_set_num_threads(8);
    #pragma omp parallel shared(n)
    {
        #pragma omp single
        n = omp_get_team_size();
        ...
    }
}
```

- Runtime Environment Variables allow you to set the value of OpenMP *internal control variables* (ICVs) *prior* to the program main execution. E.g. in bash-like shells, using `export OMP_NUM_THREADS=16` prior to running your program or, if your program is called `myprog` then `OMP_NUM_THREADS=16 ./myprog` will set the variable for that run only. Some ICVs must be set this way.
- Compiler Directives allow specifying parallelism without changing the semantics of the base language. In theory, if the directives were removed the program would produce the same output, sequentially. In practice, there can be differences because parallel computation can introduce artefacts.
- Runtime Library Routines allow you to inspect and modify ICVs, and to interact with the OpenMP implementation, e.g., to allocate memory with specified properties or to control parallel constructs.

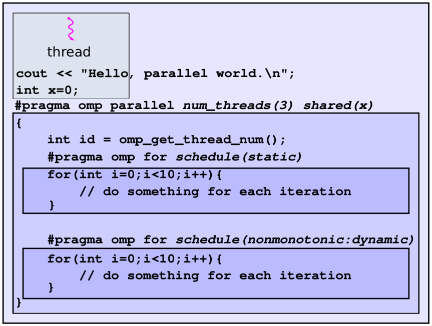
```
// start of main
int n;
#pragma omp parallel shared(n)
{
    #pragma omp single
    {
        n = omp_get_team_size();
        …
    }
    Int id = omp_get_thread_num();
    #pragma omp parallel
    {
        ….
    }
}
```

- At any point in time, an OpenMP program's *context* defines *traits* that

Synchronization, SIMD and NUMA

- Synchronization between threads, that may ordinarily be done by the programmer using primitive concurrency control techniques such as locks and signals, are provided via directives.
  - A *thread team* consists of a number of threads. Some synchronization operations are limited to the team of threads, i.e., it does not affect other teams, while some affect all teams.
  - A `barrier` directive marks a point of execution of a program encountered by a team of threads, beyond which no thread in the team may execute until all threads in the team have reached the barrier, and all explicit tasks generated by the team have executed to completion.
  - A `flush` directive tells a thread to enforce consistency between its view and other threads' view of memory.
  - A `critical` directive ensures that only one thread can execute a given structured code block at a time.
  - An `atomic` directive ensures that memory reads, writes, and updates are done by a thread without interference from other threads. Faster, but only applies for some simple operations.
- A `simd` directive makes use of SIMD processor instructions.
- Synchronization *hints* can enable processor optimization techniques like speculative execution to be used.
- Memory management and thread affinity provide for optimization over NUMA architectures, using an abstract model of placement, memory categories and partitioning.
  - Programmers can specify how the available processing units on the architecture are partitioned in *places*.
  - Programmers can specify how and where memory is to be allocated, and what type of

Program Execution



```
cout << "Hello, parallel world.\n";
int x=0;
#pragma omp parallel num_threads(3) shared(x)
{
    int id = omp_get_thread_num();
    #pragma omp for schedule(static)
    for(int i=0;i<10;i++){
        // do something for each iteration
    }

    #pragma omp for schedule(nonmonotonic:dynamic)
    for(int i=0;i<10;i++){
        // do something for each iteration
    }
}
```

- Understanding an OpenMP program execution is about understanding what happens when a thread of execution encounters a directive.
- In the program above, the main or primary thread of the program starts executing code that is part of its task.
  - It writes some text to the standard output.
  - It allocates an automatic variable and initializes it to 0.
  - It encounters a `parallel` directive...

The `parallel` directive and common clauses

https://www.openmp.org/spec-html/5.1/openmpse14.html#x59-590002.6

- Any thread that encounters the `parallel` directive will create a *parallel region* and create a *thread team* in that region, with itself as the *primary* thread, where every thread in the team executes the structured block. An implicit barrier exists at the end of the parallel region.

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

- `num_threads(integer-expression)`
  - Overrides the `nthreads-var` ICV. The actual number of threads created is limited by the `thread-limit-var` ICV and other conditions.
  - The `OMP_NUM_THREADS` environment variable sets the `nthreads-var` and optionally also the `max-active-levels-var` ICVs. Programmers can use `omp_set_num_threads(int num_threads)` to set `nthreads-var` at runtime, and respectively `int omp_get_num_threads()` to obtain its value.
  - Similarly `OMP_THREAD_LIMT`, `OMP_DYNAMIC` and `OMP_MAX_ACTIVE_LEVELS` affect the number of threads created and can be controlled at runtime.
  - For details see https://www.openmp.org/spec-html/5.1/openmpsu40.html#x60-600002.6.1
- `private(list)` – a list of program variables that each thread in the team will have its own memory allocated for
- `shared(list)` – a list of program variables that will be shared among the threads in the team

#pragma omp syntax

In C/C++:

```
#pragma omp directive-name [[,] clause[[,] clause]... ] new-line
```

In C+11 and higher with C++ attribute specifiers:

```
[[ omp :: directive( directive-name [[,] clause[[,] clause]... ] ) ]]
```

or

```
[[ using omp :: directive( directive-name [[,] clause[[,] clause]... ] ) ]]
```

- Only one directive-name can be specified per directive, but multiple directives can be applied to one following block.
- The order in which clauses appear on directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause or the directives on which they can appear.
- Some directives are *stand alone* in that they instruct the runtime to do something, and some directives must be followed by a *structured block* or structured block sequence.