OpenMP

Enabling HPC since 1997

GPUMC, Spring 2022

Hank Dietz

http://aggregate.org/hankd/

University of Kentucky

References

- OpenMP primary WWW site
  `http://openmp.org/`
- The latest reference "card" (16 pages!)

  `https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf`
- Various links at the course WWW site

What Is OpenMP?

- Took a while to develop: started in 1997
- Not a library, but also not a language
  - Compiler directives for Fortran, C, C++ …
  - Ignoring directives gives sequential code, except where it doesn't ;-)
  - Associated libraries, some explicitly used
- Compilers supporting include: GCC, LLVM/Clang, Intel, Microsoft

What Is OpenMP?

- Mostly SPMD (Single Program, Multiple Data) programming model, execution using threads
- Intended to target shared memory multi-core and multi-processor systems, now also
  - Logically-shared memory systems
  - SIMD and GPUs
- Designed to make it easy to parallelize an existing sequential program

Hello, World

- Make `hello.c` contain:

```
#include “omp.h”
void main() {
#pragma omp parallel
{ int iproc=omp_get_thread_num();
  printf(“I am PE%d\n”, iproc); } }
```

- Compile and run by:
  `gcc -fopenmp hello.c -o hello`

What OpenMP Does

- Makes threads for you
- Handles assigning work to each thread for you (does scheduling)
- Lets threads communicate via shared access to memory, but also can make local variables
- Provides means for synchronization
  - Avoid races
  - Enforce desired orderings

Thread Creation

- A sequential master thread always working
  - Spawns a team of threads as needed
  - Threads may be forked/joined for each parallel code region or may be idled between
- Can request a specific number of threads:
  - `OMP_NUM_THREADS` environment variable
    `nproc=omp_get_num_threads();`
  - `omp_num_procs()` gives physical PE count
  - `omp_set_num_threads(nproc);`

A Bit About Hello, World

```
#include “omp.h”
void main() {
#pragma omp parallel
{ int iproc=omp_get_thread_num();
  printf(“I am PE%d\n”, iproc); } }
```

- Each thread has its own stack (own **`iproc`**)
- Team activates for each parallel region
- Barrier syncs implicitly bracket each region

Mutual Exclusion

- A single memory update can be made atomic:

```
#pragma omp atomic
a += 1;
```

- Critical protects a larger operation or block:

```
#pragma omp critical
myfunction(a, &b);
```

Explicit Locking

- Can use locks to force general exclusion

```
omp_lock_t m;
omp_init_lock(&m);
#pragma omp parallel private(t,iproc)
{
  iproc=omp_get_thread_num();
  omp_set_lock(&m);
  for(t=-1;t<iproc;++t) write(1,“.”,1);
  write(1,”\n”,1);
  omp_unset_lock(&m);
}
```

Parallel Sections

- Can embed MIMD code in a parallel region without specifying who does each section

```
#pragma omp parallel sections
{
#pragma omp section
/* Thing 1 */ ...
#pragma omp section
/* Thing 2 */ ...
}
```

Parallel Loops

- A for loop can be made parallel:

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

- Loop can't have loop-carried dependences
- The loop index (`i` above) is replaced by a local copy (even without `private(i)` clause)
- `N` doesn't have to match `nproc`

Loop-Carried Dependences

- Accessing a value from another iteration:

```
for (i=0; i<N; ++i) a[i]=(++j);
for (i=0; i<N; ++i) sum+=a[i];
for (i=1; i<N; ++i) a[i]=i+a[i-1];
for (i=0; i<N-1; ++i) a[i]=i+a[i+1];
```

- Various ways to fix:

```
for (i=0; i<N; ++i) a[i]=(j+1+i);
#pragma omp parallel for reduction(+:sum)
for (i=0; i<N; ++i) sum+=a[i];
```

Reductions

- Associative operations that reduce dimension:
  - Sum is +
  - Product is *
  - Bitwise AND &, OR |, XOR ^
  - Logical AND (all) &&, OR (any) ||

- Reduction order can vary across runs
- Note that sum isn't really associative for floats, but this is one of the most common reductions

Sequential Ordered Reductions

- Since sum isn't really associative for floats, can force reduction part of loop to be ordered

```
#pragma omp parallel private(t)
#pragma omp for ordered reduction(+:sum)
for (i=0; i<N; ++i) {
  t=evil_computation(i);
#pragma omp ordered
  sum+=t;
}
```

Loop Scheduling

- Many options, effects overhead & load balance
- Assign work equally in fixed-size chunks: **`schedule(static)`**
- Assign work from a queue in fixed-size chunks: **`schedule(dynamic)`**
- Assign work from a queue in decreasing size chunks: **`schedule(guided)`**
- Optional min chunk size as second argument

Barrier Synchronization

- Can be explicitly invoked:

```
#pragma omp barrier
```

- Implied at end of each parallel construct, but can be overridden to allow overlap

```
#pragma omp parallel for nowait
for (…) { … }
```

Sequential Code

- It's all master only outside of `parallel`
- Can be explicitly invoked inside `parallel`:

```
#pragma omp master
{ /* only master does this */ ... }
```

- Can also explicitly say it's any one process:

```
#pragma omp single
{ /* only one does this */ ... }
```

Shared Memory Model

- `shared` by default:
  - Global and `static` variables
  - Heap memory, e.g., `malloc()`
- `private` by default:
  - `auto` and `register` variables
- `firstprivate` initializes with shared value
- `lastprivate` sets final value into shared
- `threadprivate`, `copyin`, `copyout`
- `volatile` isn't used; can explicitly `flush(`*v*`)`

Conclusion

- **That's everything about OpenMP?**
- **Nope.** We've ignored:
  - Nested parallelism
  - Tasks
  - SIMD
  - ~11 pages of that 16-page reference card….
    We will discuss at least the SIMD stuff later