CS 5220: Shared memory programming

David Bindel

2017-09-28

OpenMP: Open spec for MultiProcessing

- Standard API for multi-threaded code
  - Only a spec — multiple implementations
  - Lightweight syntax
  - C or Fortran (with appropriate compiler support)
- High level:
  - Preprocessor/compiler directives (80%)
  - Library calls (19%)
  - Environment variables (1%)
- Basic syntax: `#pragma omp` *construct* `[`*clause* `...]`
  - Usually affects structured block (one way in/out)
  - OK to have `exit()` in such a block

Last time

- Environmental inquiries with `omp_get_*` functions
- Creating parallel regions with `#pragma omp parallel`
- Annotations for variables (shared, private, reduction)
- Synchronization via critical sections, atomic ops, barriers
- Today: Work sharing, tasks, and some examples

Work sharing

Work sharing constructs split work across a team

- Parallel `for`: split by loop iterations
- `sections`: non-iterative tasks
- `single`: only one thread executes (synchronized)
- `master`: master executes, others skip (no sync)

Parallel iteration

Idea: Map **independent** iterations onto different threads

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

#pragma omp parallel
{
    // Stuff can go here...
    #pragma omp for
    for (int i = 0; i < N; ++i)
        a[i] += b[i];
}
```

Implicit barrier at end of loop (unless `nowait` clause)

Parallel iteration

The iteration can also go across a higher-dim index set

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; ++i)
  for (int j = 0; j < M; ++j)
    a[i*M+j] = foo(i,j);
```

Restrictions

- `for` loop must be in "canonical form"
  - Loop var is an integer, pointer, random access iterator (C++)
  - Test compares loop var to loop-invariant expression
  - Increment or decrement by a loop-invariant expression
  - No code between loop starts in `collapse` set
  - Needed to split iteration space (maybe in advance)
- Iterations should be independent
  - Compiler may not stop you if you screw this up!
- Iterations may be assigned out-of-order on one thread!
  - Unless the loop is declared `monotonic`

Reduction loops

How might we parallelize something like this?

```
double sum = 0;
for (int i = 0; i < N; ++i)
    sum += big_hairy_computation(i);
```

Reduction loops

How might we parallelize something like this?

```
double sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; ++i)
    sum = big_hairy_computation(i);
```

Ordered

OK, what about something like this?

```
for (int i = 0; i < N; ++i) {
    int result = big_hairy_computation(i);
    add_to_queue(q, result);
}
```

Work is *mostly* independent, but not wholly.

Ordered

Solution: `ordered` directive in loop with `ordered clause`

```
#pragma omp parallel for ordered
for (int i = 0; i < N; ++i) {
    int result = big_hairy_computation(i);
    #pragma ordered
    add_to_queue(q, result);
}
```

Ensures the `ordered` code executes in loop order.

SIMD loops

As of OpenMP 4.0:

```
#pragma omp parallel simd reduction(+:sum) aligned(a:64)
for (int i = 0; i < N; ++i) {
    a[i] = b[i] * c[i];
    sum = sum + a[i];
}
```

Can also declare vectorized functions:

```
#pragma omp declare simd
float myfunc(float a, float b, float c)
{
    return a*b + c;
}
```

Other parallel work divisions

- `sections`: like cobegin/coend
- `single`: do only in one thread (e.g. I/O)
- `master`: do only in master thread; others skip

Sections

```
#pragma omp parallel
{
    #pragma omp sections nowait
    {
        #pragma omp section
        do_something();

        #pragma omp section
        and_something_else();

        #pragma omp section
        and_this_too();
        // No implicit barrier here
    }
    // Implicit barrier here
}
```

`sections nowait` to kill barrier.

Task-based parallelism

- Work-sharing so far is rather limited
    - Work cannot be produced/consumed dynamically
    - Fine for data parallel array processing...
    - ... but what about tree walks and such?
- Alternate approach (OpenMP 3.0+): Tasks

Tasks

Task involves:

- Task construct: `task` directive plus structured block
- Task: Task construct + data

Tasks are handled by run time, complete at barriers or `taskwait`.

Example: List traversal

```
#pragma omp parallel
{
    #pragma omp single nowait
    {
        for (link_t* link = head; link; link = link->next)
            #pragma omp task firstprivate(link)
            process(link);
    }
    // Implicit barrier
}
```

One thread generates tasks, others execute them.

Example: Tree traversal

```
int tree_max(node_t* n)
{
    int lmax, rmax;
    if (n->is_leaf)
        return n->value;

    #pragma omp task shared(lmax)
        lmax = tree_max(n->l);
    #pragma omp task shared(rmax)
        rmax = tree_max(n->l);
    #pragma omp taskwait

    return max(lmax, rmax);
}
```

The `taskwait` waits for all child tasks.

Task dependencies

What happens if one task produces what another needs?

```
#pragma omp task depend(out:x)
x = foo();
#pragma omp task depend(in:x)
y = bar(x);
```

Topics not addressed

- Low-level synchronization (locks, `flush`)
- OpenMP 4.x constructs for accelerator interaction
- A variety of more specialized clauses

See `http://www.openmp.org/`

Some examples (at board)

What are different ways to organize these:

- Dot product?
- Monte Carlo computation with adaptive termination?
- Wave equation time stepper?