OpenMP 2

Victor Eijkhout & Cyrus Proctor

PCSE 2015

Sections

```
#pragma omp sections
{
#pragma omp section
  // one calculation
#pragma omp section
  // another calculation
}
```

- Sections are independent
- Executed by independent or same thread

Sections example

Independent calculations: `y1 = f(a); y2 = g(b);`

```
#pragma omp sections
{
#pragma omp section
  y1 = f(a)
#pragma omp section
  y2 = g(b)
}
```

Sections example'

Largely independent: `y = f(a)+g(b)`

```
#pragma omp sections
{ double y1,y2;
#pragma omp section
  y1 = f(a)
#pragma omp section
  y2 = g(b)
y = y1+y2;
}
```

What is wrong with that last line? Can you suggest a fix?

Single and master

```
#pragma omp parallel
{
#pragma omp master
  printf("We are starting this section!\n");
  // parallel stuff
}
```

Single and master

```
#pragma omp parallel
{
  int a;
  #pragma omp single
    a = f(); // some computation
  #pragma omp sections
    // various different computations using a
}
```

'single' is a workshare, so it has a barrier after it

Fortran: workshare

Divide units of work, up to compiler

```
    SUBROUTINE WSHARE2(AA, BB, CC, DD, EE, FF, N)
    INTEGER N
    REAL AA(N,N),BB(N,N)

!$OMP  PARALLEL
!$OMP    WORKSHARE
            AA = BB
!$OMP    END WORKSHARE
!$OMP  END PARALLEL
```

Data scope

Data scope

- Data can be shared: from the master thread
- Data can be private: every thread its own copy
- How are private variables initialized?
- Private variables disappear after a parallel region

Example

```
  double x[200], s,t;
#pragma omp parallel for private(s,t) shared(x)
  for (i=0; i<200; i++) {
    s = f(i); t = g(i);
    x[i] = s+t;
  }
```

Private and shared

- `shared` is the default: sometimes dangerous
- `private`: each thread gets its own copy
  - anything declared in the region is private
  - loop variables are private
  - any ‘outside’ variable is no longer visible: private variables are initialized, after the region the outside value is restored
- C: `default(shared|none)`,
  F: `default(private|shared|none)`
  ‘none’ is useful for debugging.

Private arrays

The rules for arrays are tricky.

- Static arrays and `private` clause: really static data.
- Dynamic arrays: only a private pointer; data is shared.

Interaction private/shared

- `firstprivate` like private, but initialized to shared value
- `lastprivate` private copy of shared variable, copied out

lastprivate

```
#pragma omp parallel for \
        lastprivate(tmp)
for (i=0; i<N; i+) {
  tmp = ......
  x[i] = .... tmp ....
}
..... tmp ....
```

- `tmp` is temporary, should be private
- final value is used after the loop: use `lastprivate`
- this can also be used for the loop variable.

Synchronization

Barriers

- Let threads wait for each other in a parallel region
- No need to break up the team

```
#pragma omp parallel
{
  x = F(y)
#pragma omp for
  for (i=0; i<N; i++) {
    ....... x ......
  }
}
```

Note: barriers need to be encountered by all threads in a team; therefore can not be in a worksharing construct.

Barriers on workshare

- No barrier at the start
- Implicit barrier at the end
- No barrier with `nowait`

nowait example 1

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for nowait
  for (i=0; i<N; i++) {
    f(i)
  }
#pragma omp for schedule(dynamic,n)
  for (i=0; i<N; i++) {
    x[i] = ...
  }
}
```

`nowait` example 2

```
#pragma omp parallel
{
  x = local_computation()
#pragma omp for nowait
  for (i=0; i<N; i++) {
    x[i] = ...
  }
#pragma omp for
  for (i=0; i<N; i++) {
    y[i] = ... x[i] ...
  }
}
```

Critical / atomic

- Atomic operations: can only be executed by one thread at a time
- The `s = s+...` update of a reduction is atomic operation
- Critical section: OpenMP construct for atomic operations
- `critical` directive is general; `atomic` limited, but can use hardware support
- Critical sections can be very expensive: require operating system support

critical section

```
  double s = 0;
#pragma omp parallel for
  for (i=0; i<N; i++) {
    double t = f(i);
#pragma omp critical
    s += t;
  }
```

Critical sections can be named.

Locks

- Locks and critical sections both give exclusive execution
- Subtle difference: critical limits access to section of *code*
- lock limits access to item of *data*
- Example: writing to database

Locks

Create/destroy:

```
void omp_init_lock(omp_lock_t *lock);
void omp_destroy_lock(omp_lock_t *lock);
```

Set and release:

```
void omp_set_lock(omp_lock_t *lock);
void omp_unset_lock(omp_lock_t *lock);
```

Locks example

```
omp_lock_t lockvar;
void omp_init_lock(&lockvar);

omp_set_lock(&lockvar);
var = var+update
omp_unset_lock(&lockvar);

void omp_destroy_lock(&lockvar);
```

Not tied to parallel regions!