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?

Solution: use reduction clause

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 work sharing construct, so it has a barrier after it

'master' is not a work sharing, has no barrier, would be the wrong choice in this example

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
- Question: interaction private and shared: initialization, persistence of values.

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;
  }
```

- `s,t` are temporaries: declare private to prevent race condition
- In Fortran, you have to do it this way; in C you can declare them in the parallel region

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

Example

```
  int x = 5;
#pragma omp parallel
  {
    x = x+1;
    printf("shared: x is %d\n",x);
  }
  printf("after: x is %d\n",x);
```

update is race condition, shared variable gets updated

Example

```
#pragma omp parallel
  {
    int x; x = 3;
    printf("local: x is %d\n",x);
  }
  printf("after: x is %d\n",x);
```

Private variable shadows shared variable;
original value after the parallel region

Example

```
#pragma omp parallel private(x)
  {
    x = x+1;
    printf("private: x is %d\n",x);
  }
  printf("after: x is %d\n",x);
```

Private variable starts with undefined variable, possibly zero
original variable is not altered

Default shared/private

C: `default(shared|none)`,
F: `default(private|shared|none)`
'none' is useful for debugging.

```
#pragma omp parallel default(none) shared(x,y,z) private(t,
{
  ..
}
```

Default forces you to specify every variable in the parallel region.

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

Example

```
#pragma omp parallel firstprivate(x)
  {
    x = x+1;
    printf("private: x is %d\n",x);
  }
  printf("after: x is %d\n",x);
```

Private variable initialized to global value;
original variable not altered

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 shared(x)
{
  // some parallel computation of x
  .....
#pragma omp barrier
#pragma omp for
  for (i=0; i<N; i++) {
    ....... x ...... // use x
  }
}
```

Without the barrier a thread could start using `x` before it's computed.

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] = ...
  }
}
```

the `nowait` clause cancels the barrier:
threads can start the second loop early

`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] ...
  }
}
```

Both loops have a static schedule:
it is safe to start the second