OpenMP

Dr. William McDoniel and Prof. Paolo Bientinesi

HPAC, RWTH Aachen
mcdoniel@aices.rwth-aachen.de

WS17/18

Worksharing constructs

- To date:
  - `#pragma omp parallel` created a team of threads
  - We distributed the work manually `id, nths, ...`
- OpenMP also provides directives for automatic distribution of work:
  - Loop construct
  - Sections construct
  - Single construct

Loop construct

- Syntax:

```
#pragma omp for [clause [, clause] ...]
   for-loop
```

- Example:

```
#pragma omp parallel
{
  #pragma omp for
  for (i = 0; i < n; i++)
    [...]
}
```

- The iterations of the associated loop are executed in parallel by the team of threads that encounter it

- See, for instance, `11.loop-worksharing.c`

Loop construct - Canonical form

- Restrictions in the form of the loop to simplify the compiler optimizations
- Only *for* loops
- Number of iterations can be counted: integer counter which is incremented (decremented) until some specified upper (lower) bound is reached
- Remember: one entry point, one exit point
- No `break` statement
- `continue` and `exit` allowed.

Loop construct - Canonical form

- Loops must have the canonical form

```
for (init-expr ; var relop b ; incr-expr)
```

  where:

  - `init-expr`: initializes the loop counter *var* via an integer expression
  - `relop` is one of: `<`, `<=`, `>`, `>=`
  - `b` is also an integer expression
  - `incr-expr`: increments or decrements *var* by an integer amount:
    - `++var`, `var++`, `--var`, `var--`
    - `var += incr`, `var -= incr`
    - `var = var + incr`, `var = var - incr`

- Example (see also `11.canonical-loop.c`):

```
for (i = 0; i < n; i += 4)
```

Combined `parallel for`

- Sequential:

```
int i;
for (i = 0; i < N; i++)
  z[i] = alpha * x[i] + y[i];
```

- Parallel region:

```
#pragma omp parallel
{
  int i, id, nths, istart, iend;
  id = omp_get_thread_num();
  nths = omp_get_num_threads();
  istart = id * N / nths;
  iend = (id+1) * N / nths;
  if (id == nths-1) iend = N;
  for (i = istart; i < iend; i++)
    z[i] = alpha * x[i] + y[i];
}
```

Combined `parallel for`

- Sequential:

```
int i;
for (i = 0; i < N; i++)
  z[i] = alpha * x[i] + y[i];
```

- Parallel region:

```
#pragma omp parallel
{
  int i, id, nths, istart, iend;
  id = omp_get_thread_num();
  nths = omp_get_num_threads();
  istart = id * N / nths;
  iend = (id+1) * N / nths;
  if (id == nths-1) iend = N;
  for (i = istart; i < iend; i++)
    z[i] = alpha * x[i] + y[i];
}
```

- Loop construct:

```
#pragma omp parallel
{
  int i;
  #pragma omp for
  for (i = 0; i < N; i++)
    z[i] = alpha * x[i] + y[i];
}
```

Combined `parallel for`

- Sequential:

```
int i;
for (i = 0; i < N; i++)
  z[i] = alpha * x[i] + y[i];
```

- Parallel region:

```
#pragma omp parallel
{
  int i, id, nths, istart, iend;
  id = omp_get_thread_num();
  nths = omp_get_num_threads();
  istart = id * N / nths;
  iend = (id+1) * N / nths;
  if (id == nths-1) iend = N;
  for (i = istart; i < iend; i++)
    z[i] = alpha * x[i] + y[i];
}
```

- Loop construct:

```
#pragma omp parallel
{
  int i;
  #pragma omp for
  for (i = 0; i < N; i++)
    z[i] = alpha * x[i] + y[i];
}
```

- Shortcut:

```
int i;
#pragma omp parallel for
for (i = 0; i < N; i++)
  z[i] = alpha * x[i] + y[i];
```

Loop construct

Guidelines:

- Identify the compute intensive loops
- Can the iterations be run independently?
- If needed/possible remove dependencies
- Add the `for` directive
- Consider alternative schedulings if bad load balancing

Loop construct - Dependencies

- So far, trivially parallelizable loops:

```
for (i = 0; i < n; i++) {
  z[i] = alpha * x[i] + y[i];
  w[i] = z[i] * z[i]
}
```

- What if we run the following code in parallel?

```
fib[0] = fib[1] = 1;
for (i = 2; i < n; i++)
  fib[i] = fib[i-1] + fib[i-2]
```

- Possible outputs with $n = 10$ and 2 threads:
  - a) [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
  - b) [1, 1, 2, 3, 5, 0, 0, 0, 0, 0]

- Think about it: Why?

Loop construct - Dependencies

- Loop-carried dependencies
  - A memory location is written in one iteration, and
  - it is also read or written in another iteration

```
for (i = 1; i < n; i++)
  a[i] = a[i] + a[i-1]
```

  - Race condition: results depends on the order in which operations are performed

- Classification of dependencies:
  - Flow/True dependencies
  - Anti dependencies
  - Output dependencies

Loop construct - Dependencies

- Flow dependence:
  - Iteration *i* writes to a location
  - Iteration *i* + 1 reads from the location
- We can remove some loop-carried dependencies. For instance:
  - Reductions: `x = x + a[i];`
  - Induction variables:

```
j = 5;
for (i = 1; i < n; i++) {
  j += 2;
  a[i] = f(j);
}
```

Rewrite "`j += 2;`" as "`j = 5 + 2*i;`"

Loop construct - Dependencies

- Anti dependence:
  - Iteration *i* reads to a location
  - Iteration *i* + 1 writes from the location
- Example

```
for (i = 0; i < n-1; i++)
  a[i] = a[i+1] + f(i);
```

- Split the loop, and make a copy of array `a`

```
#pragma omp parallel for
for (i = 0; i < n-1; i++)
  a2[i] = a[i+1];
#pragma omp parallel for
for (i = 0; i < n-1; i++)
  a[i] = a2[i] + f(i);
```

Loop construct - Dependencies

- Output dependence:
  - Iteration *i* writes to a location
  - Iteration *i* + 1 writes from the location
- Example

```
for (i = 0; i < n; i++) {
  tmp = A[i];
  A[i] = B[i];
  B[i] = tmp;
}
```

- Scalar expansion of `tmp` to array unnecessarily expensive.
- Make `tmp` private.

```
#pragma omp parallel for private(tmp)
for (i = 0; i < n; i++) {
  tmp = A[i];
  A[i] = B[i];
  B[i] = tmp;
}
```

Loop construct - Dependencies

Other situations

- Even/odd parallelization

```
for (i = 2; i < n; i++)
  a[i] = a[i-2] + x;
```

can be rewritten as

```
for (i = 2; i < n; i+=2)
  a[i] = a[i-2] + x;
for (i = 3; i < n; i+=2)
  a[i] = a[i-2] + x;
```

Even though both loops still present a loop-carried dependence, they are independent from one another and we can, for instance, use task parallelism to run them as two independent tasks.

- Not a dependence:

```
for (i = 0; i < n; i++)
  a[i] = a[i+n] + x;
```