OpenMP

Dr. William McDoniel and Prof. Paolo Bientinesi

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

WS17/18

Loop construct - Clauses

- `#pragma omp for [clause [, clause] ...]`
- The following clauses apply:
  - `private, firstprivate, lastprivate`
  - `reduction`
  - `schedule`
  - `collapse`
  - `nowait`

Loop construct - Clauses

Data-sharing attributes

- `private` and `firstprivate` as in the `parallel` construct
- Important: the iterator variable is made private by default. That is, in

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

  `i` is made private automatically

- `lastprivate (list)`: the last value of a private variable listed in this clause is available after the construct completes
- By last iteration we mean the value from the iteration that would come last in a sequential execution
- See `11.loop-lastprivate.c`

Loop construct - Clauses

reduction (see 11.loop-reduction.c)

```
double sum = alpha;
#pragma omp parallel for reduction(+:sum)
for ( int i = 0; i < N; i++ ) {
  sum += x[i] * y[i] // Accumulate to sum (no race condition)
}
```

```
double sum = alpha; // We can have any initial value here
                    // This is application specific
#pragma omp parallel
{
  double priv_sum = 0.0; // Initializer (0.0 for sum of doubles)
  #pragma omp for // Same computation on the local accumulator
  for ( int i = 0; i < N; i++ ) {
    priv_sum += x[i] * y[i] // Accumulate to your private copy of sum
  }
  #pragma omp critical // Reduce partial sums using the
  sum += priv_sum;     // Combiner function (+ in this case)
}
```

Loop construct - Scheduling

- **Scheduling**: how loop iterations are distributed among threads.
- To achieve best performance, threads must be busy most of the time, minimizing the time they remain idle, wasting resources.
- Keys:
  - Good load balance (work evenly distributed)
  - Minimum scheduling overhead
  - Minimum synchronization

Loop construct - Scheduling

- OpenMP allows us to choose among several scheduling schemes via the `schedule` clause

```
schedule(kind[, chunk_size])
```

where `kind` is one of

- `static`
- `dynamic`
- `guided`
- `auto`
- `runtime`

The value of `chunk_size` influences the divisions into chunks of the iteration space and the `kind` specifies how the chunks are distributed among threads.

Loop construct

`schedule: static`

- Divide the iterations in `NUM_THREADS` (roughly) equal chunks and give one to each of them (in order)
- If `chunk_size` is specified, divide in chunks of size `chunk_size`, and distribute them cyclically in a round robin fashion
- Example:

```
#pragma omp for schedule(static, 4)
for (i = 0; i < 20; i++)
  [...]
```

Assuming execution with 2 threads:

TH0: [0-3]	TH1: [4-7]	TH0: [8-11]	TH1: [12-15]	TH0: [16-19]

Loop construct - Clauses

`schedule: dynamic`

- Conceptually, this scheme creates a queue of chunks, from which the threads keep grabing chunks to execute, until no more chunks are left
- By default, `chunk_size` equals 1
- Example:

```
#pragma omp for schedule(dynamic, 2)
for (i = 0; i < 10; i++)
  [...]
```

Possible run assuming execution with 2 threads:

TH0: [0-1]	TH1: [2-3]	TH1: [4-5]	TH0: [6-7]	TH1: [8-9]

Loop construct

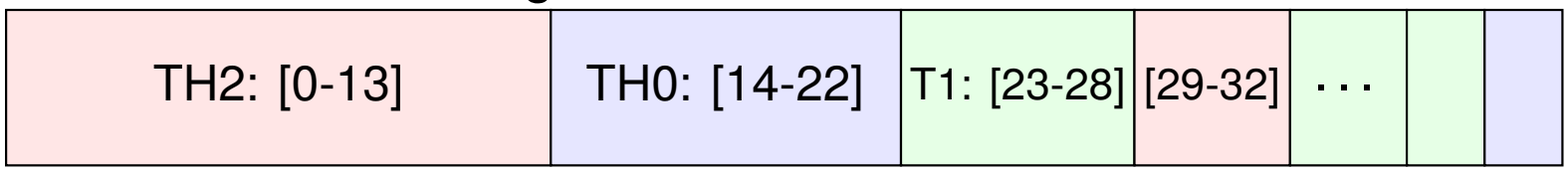
`schedule: guided`

- Similar to `dynamic`
- Difference: start with large chunk sizes, which exponentially decrease in size
- Chunks consist of at least *chunk_size* iterations (except maybe the last one)
- *chunk_size* defaults to 1

- Example:

```
#pragma omp for schedule(guided, 2)
for (i = 0; i < 40; i++)
  [...]
```

Possible run assuming execution with 3 threads:



Loop construct

`schedule: auto` and `runtime`

- `auto`
  - Decision on scheduling delegated to the compiler/runtime
  - The programmer gives freedom to the implementation
  - May simply resort to static or dynamic

- `runtime`
  - Decision on scheduling is deferred until run time
  - Schedule and chunk size are taken from internal variables
  - May be specified via a routine call or via environment variables:
    - `omp_set_schedule( kind, chunk_size )`
    - `export OMP_SCHEDULE="kind,chunk_size"`
  - Mainly used for testing (so that we do not need to edit and recompile every time)

Loop construct - Scheduling

- Most used: `static` and `dynamic`
- `static`:
  - Workload is predetermined and predictable by the programmer
  - Cyclic distribution allows good load balancing
  - Scheduling is done at compile time, adding little parallel overhead
- `dynamic`:
  - Unpredictable/variable work per iteration
  - Decisions on scheduling made at runtime, adding overhead
- See: `11.loop-schedule.c`

Loop construct - Clauses

`collapse`

- Syntax:

```
collapse(n)
```

- Indicates how many loops are associated with the loop construct
- `n` must be a constant positive integer expression
- The iterations of the `n` loops are collapsed into one larger iteration space
- The order of the collapsed space is that of the equivalent sequential execution of the iterations

Loop construct - Clauses

`collapse`

- Example:

```
#pragma omp for collapse(3)
for (i = 0; i < m; i++)
  for (j = 0; j < n; j++)
    for (k = 0; k < l; k++)
      [...]
```

The resulting iteration space is of size $m * n * l$

Loop construct - Clauses

`nowait`

- Syntax:

```
#pragma omp for nowait
```

- Removes the implicit barrier at the end of the construct
- When threads reach the end of the construct, they will proceed immediately to perform other work
- Allows fine tuning of a program's performance
- Use with care, incorrect usage will introduce bugs

Loop construct - Clauses

`nowait`

- Example

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 0; i < n; i++)
    a[i] = a[i] + i;
  #pragma omp for
  for (i = 0; i < m; i++)
    b[i] = 2 * b[i];
}
```

Loop construct - Clauses

`nowait`

- The following example shows an incorrect usage of nowait

```
#pragma omp parallel
{
  #pragma omp for nowait
  for (i = 0; i < n; i++)
    a[i] = a[i] + i;
  #pragma omp for
  for (i = 0; i < n; i++)
    b[i] = 2 * a[i];
}
```

- The code assumes that the distribution of iterations to threads is identical for both loops
- Thus, removing the barrier may lead to wrong results

Loop construct - Clauses

`nowait`

Exception:

- Loops belong to the same parallel region
- Both loops use `static` scheduling
- Both loops run for the same number of iterations
- Both specify the same `chunk_size` or do not specify it at all
- (none is associated with the SIMD construct)

If all of the above holds, then, and only then, can we assume that the same iteration numbers are assigned to the same threads in both loops.

Loop construct - Clauses

nowait

```
  // CORRECT :)
  #pragma omp parallel
  {
    #pragma omp for nowait schedule(static)
    for (i = 0; i < n; i++)
      a[i] = a[i] + i;
    #pragma omp for schedule(static)
    for (i = 0; i < n; i++)
      b[i] = 2 * a[i];
}
```

Sections

Sections

- The `sections` construct allows different threads to carry out different tasks (task parallelism)
- Non-iterative worksharing construct
- It consists of a pair of directives: `sections` and `section`
- `sections` indicates the start of the construct (enclosing the region with the multiple tasks)
- `section` marks each different section/task (a structured block)

Sections

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    computation_x()
    #pragma omp section
    computation_y()
    #pragma omp section
    computation_z()
  }
}
```

- Think of a queue of executable blocks of code (each individual `section`)
- Theads grab one block at a time from the queue and execute it, until no more blocks remain
- Each block is executed only once
- See `12.sections.c`

Exercise

- Consider the following sequence of operations. Assuming the only dependencies among the operations are given by their input and output variables, write a piece of OpenMP code that exposes as much parallelism as possible using the `sections` and `section` constructs.

```
mx = mean( x );
my = mean( y );
fout = f( mx, my );
gout = g( mx, my );
final = summary( fout, gout );
```

Hints:

- `#pragma omp parallel`
- `#pragma omp sections`
- `#pragma omp section`
- Hint:
  - You may need multiple `sections` regions.

Sections - Clauses

- `#pragma omp sections [clause [, clause] ...]`
- The following clauses apply:
  - `private, firstprivate, lastprivate`
  - `reduction`
  - `nowait`
- Worksharing construct!
- Implicit barrier at the end of `sections`!

Master and Single

Master and Single

- The `master` and `single` constructs guarantee that code in a parallel region is executed by one single thread
- Initialization of data, I/O, ...

- Both constructs apply to a structured block
- Differences:
  - The `master` construct guarantees the block is executed only by the master thread
    The `single` construct guarantees the block is executed by any one and only one thread
  - A `single` construct has an implicit barrier at the end
    A `master` construct does not
  - Efficiency may differ. Application and implementation dependent.

Master

06.master.c

```
#pragma omp parallel num_threads(2)
{
  int id = omp_get_thread_num();
  #pragma omp master
  {
    printf("[%d] Executed only by the master thread\n", id);
  }
  printf("[%d] Executed by all threads\n", id);
}
```

Single

06.single.c

```
#pragma omp parallel num_threads(2)
{
  int id = omp_get_thread_num();
  #pragma omp single
  {
    printf("[%d] Executed by only one thread\n", id);
  }
  printf("[%d] Executed by all threads\n", id);
}
```

Single - Clauses

- `#pragma omp single [clause [, clause] ...]`
- The following clauses apply:
  - `private, firstprivate`
  - `nowait`

- `single` is a worksharing construct!
- Remember: Implicit barrier at the end of `single`!

- `master` is not a worksharing construct!
- Remember: **NO** implicit barrier at the end of `master`!

Exercise

- Parallelize the body of the *while* loop in the following code. Use the `for` construct to parallelize the *for* loop. Use the `master/single` construct to serialize the portions of the code that should be executed by only one thread. Produce two versions of your code.

```
#include <stdio.h>
#include <stdlib.h>

#define N 1000

int main( void ) {
    int iter = 0;
    float buffer_in[N*N], buffer_out[N*N];
    while (1) {
        iter++;
        printf( "Timestep %d\n", iter );
        read_input( buffer_in );
        for (i = 0; i < N; i++)
           process_signal( &buffer_in[i*N], &buffer_out[i*N] );
        write_output( buffer_out );
    }
}
```

Hints:

- `#pragma omp parallel`
- `#pragma omp master`
- `#pragma omp single`
- Challenge:
  - Pay special attention to the subtle differences between `master` and `single`

Pitfalls - Deadlocks

From the OpenMP 4.0 specifications (Section 2.7):

- *Each worksharing region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region*.
- *The sequence of worksharing regions and barrier regions encountered must be the same for every thread in a team.*

Otherwise, the behavior is unspecified, leading to deadlocks or unexpected behavior.

- See `14.worksharing-order.c`

Summary

Work-sharing

- `#pragma omp parallel`
  - Indicates a portion of code to be parallelized
  - Creates team of threads
  - Replicates work: distribute manually or via worksharing constructs
  - Clauses to control data-sharing attributes, number of threads, etc.
- `#pragma omp for`
  - Work-sharing construct
  - Distributes iterations among threads
  - Clauses to control data-sharing attributes, load balancing, etc.
  - Careful with loop-carried dependencies!
- Also
  - `pragma omp sections/section`
  - `pragma omp single`
  - `pragma omp master` (not a work-sharing construct)