OpenMP - III

Diego Fabregat-Traver and Prof. Paolo Bientinesi

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

WS15/16

OpenMP

References

- Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, 2007.

  B. Chapman, G. Jost , R. van der Pas.

Worksharing constructs

Distribute the execution of code among the team members

- 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++)
    [...]
}
```

- Canonical form:

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

Loop construct

- 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

- 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];
}
```

Loop construct

- 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]
```

- When run in my system with $n = 10$ and 2 threads:
  - Sometimes: [1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
  - Others: [1, 1, 2, 3, 5, 0, 0, 0, 0, 0]

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;
```

and execute each loop in parallel

- Not a dependence:

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

Exercise - Loop-carried dependencies

- Try to parallelize the different pieces of code below.

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

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

```
t = 1;
for ( i = 0; i < n-1; i++ ) {
  a[i] = a[i+1] + b[i] * c[i];
  t = t * a[i];
}
```

Scheduling

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`

and the iteration space is split in chunks of `chunk_size` consecutive iterations.

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

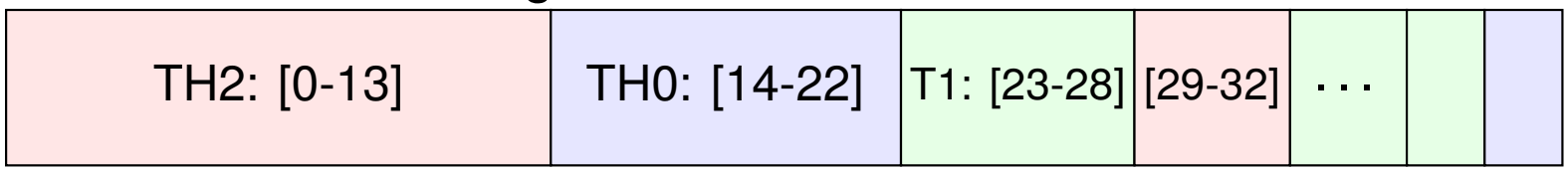
`schedule: guided`

- 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: `05.loop-schedule.c`

A few more clauses

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

`ordered`

- Used in combination with the `ordered` construct
- Specifies a structured block (portion of code) in a loop region to be executed in sequential order.
- Sequentializes and orders the execution within an `ordered` region while allowing code outside to be run in parallel

Loop construct - Clauses

`ordered`

- Example

```
#pragma omp parallel
{
  #pragma omp for ordered
  for (i = 0; i < n; i++)
  {
    int id = omp_get_thread_num();
    printf("Thread %d updates a[%d]\n", id, i);
    a[i] += i;

    #pragma omp ordered
    printf("Thread %d prints value of a[%d] = %d\n",
           id, i, a[i]);
  }
}
```

Synchronization

Synchronization

OpenMP:

- Shared-memory programming model
- Unintended sharing of data causes race conditions
- Protect data conflicts with synchronization

Main constructs/tools:

- Barriers: `barrier`
- Critical sections:
  - `critical`
  - `atomic`
- Locks (low level)

Synchronization - Barriers

- Syntax:

```
#pragma omp barrier
```

- Synchronizes all threads in the enclosing parallel region
- Ensures that all the code before the barrier has been executed by all threads before proceeding with the code beyond the barrier

Synchronization - Barriers

```
#pragma omp parallel
{
    int id = omp_get_thread_num();
    A[id] = lot_of_work_inside( id );
    #pragma omp barrier
    B[id] = more_work( A, id );
}
```

Synchronization - Critical sections

- Syntax:

```
#pragma omp critical
```

- When?
  - Every thread must execute a section of the code
  - They can execute it in any order
  - Mutual exclusion is required

Synchronization - Critical sections

Don't do this at home!

```
double sum = 0.0, pi;
double step = 1.0/NUM_STEPS;
double x_i;
int i;

#pragma omp parallel for private(x_i)
for ( i = 0; i < NUM_STEPS; i++ )
{
    x_i = (i + 0.5) * step;
    #pragma omp critical
    sum = sum + 4.0 / (1.0 + x_i * x_i);
}
pi = sum * step;
```

Synchronization - Critical sections

To keep in mind...

- Minimize the size of critical sections
  - Refactor and pull heavy work outside the region
  - Have local copies, then reduce in a critical region
- Make sure you don’t hit the critical section too often (like every ms)
- The overhead will kill your performance

Synchronization - Atomic

- Syntax:

```
#pragma omp atomic
```

- The `atomic` construct is a very restricted form or `critical`
- Often, hardware provides support to perform quick updates of memory locations
- If those hardware instructions are available, `atomic` tells the compiler to use them
- Otherwise, acts as a critical region

Synchronization - Atomic

- Restricted critical section, applied to a single statement. Valid statements:
  - `x++`, `x-`, `++x`, `-x`
  - `x binop= expr`
  - `x = x binop expr`
  - `x = expr binop x`

  where
  - `x` is of scalar type
  - `expr` is an expression of scalar type (which does not include `x`
  - `binop` is one of `+`, `-`, `*`, `/`, `&`, `^`, `|`, `«`, `»`

Synchronization

# threads	Sequential	`critical`	`atomic`	`reduction`
1	0.027 secs	0.031	0.026	0.030
2	0.027 secs	0.295	0.108	0.015
4	0.027 secs	0.572	0.117	0.008