OpenMP - II

Diego Fabregat-Traver and Prof. Paolo Bientinesi

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

WS15/16

OpenMP

References

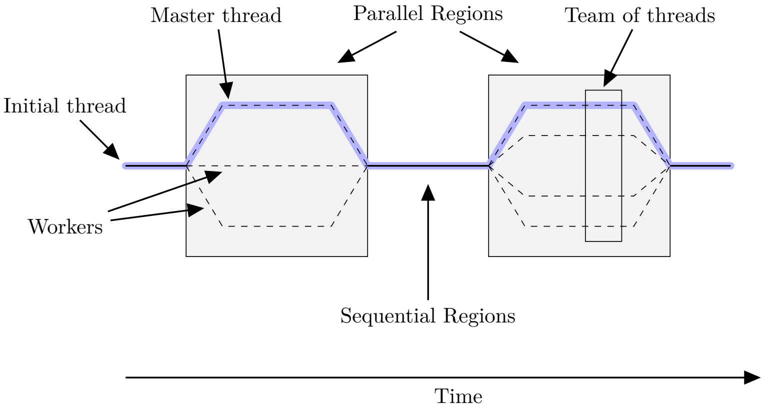
- Using OpenMP: Portable Shared Memory Parallel Programming. The MIT Press, 2007.

  B. Chapman, G. Jost , R. van der Pas.
- Introduction to OpenMP by *Tim Mattson*.

  `http://openmp.org/mp-documents/Intro_To_OpenMP_Mattson.pdf`

Previously on... Parallel Programming

- OpenMP: Fork-join programming model



Previously on... Parallel Programming

- OpenMP API consists of:
  - Compiler directives
  - Library routines
  - Environment variables
- Header: `#include <omp.h>`
- Compilation: `[gcc|icc] -fopenmp <source.c> -o <executable.x>`

Previously on... Parallel Programming

The parallel construct and the SPMD approach

- The most important construct:

```
#pragma omp parallel [clause [, clause] ...]
```

- Creates a team of threads to execute the parallel region (fork)
- Has an implicit barrier at the end of the region (join)
- It does not distribute the work
- So far: we distributed the work based on thread id and number of threads (SPMD or *à la* MPI)

Previously on... Parallel Programming

Shared vs private variables

- Shared: single instance that every thread can read/write
- Private: each thread has its own copy and others cannot read/write them (unless a pointer to them is given)

- So far: shared or private depending on where they were declared
- See, for instance, `02b.axpy-omp.c`

Exercise 3 (pi.c)

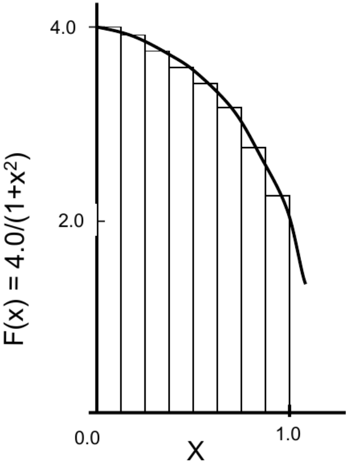
Mathematically, we know that:

$$\pi = \int_0^1 \frac{4}{1+x^2}\,dx$$

Numerically, we can approximate the integral as the sum of rectangles:

$$\pi \approx \sum_{i=0}^{N} \frac{4}{1+x_i^2}\,\Delta x$$

where each rectangle has width Δx and height $F(x_i)$ at the middle of the interval i.



Source: Timothy Mattson, Intel.

Exercise 3 (pi.c)

- Use the `#pragma omp parallel` construct to parallelize the code below so that 4 threads collaborate in the computation of π. Pay attention to shared vs private variables!

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_STEPS 10000

int main( void )
{
  int i;
  double sum = 0.0, pi, x_i;
  double step = 1.0/NUM_STEPS;

  for ( i = 0; i < NUM_STEPS; i++ ) {
    x_i = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x_i * x_i);
  }
  pi = sum * step;
  printf("Pi: %.15e\n", pi);
  return 0;
}
```

Hints:

- `#pragma omp parallel num_threads(...)`
- `omp_set_num_threads(...)`
- `omp_get_num_threads(...)`
- `omp_get_thread_num(...)`
- Challenges:
  - split iterations of the loop among threads
  - create an accumulator for each thread to hold partial sums, which can later be combined to generate the global sum

Parallel construct

- `#pragma omp parallel [clause [, clause] ...]`
  structured-block

- The following clauses apply:
  - `if`
  - `num_threads`
  - `shared, private, firstprivate, default`
  - `reduction`

  - `copyin`
  - `proc_bind`

Parallel construct

`if` clause

- Conditional parallel execution
- Avoid parallelization overhead if little work to be parallelized
- Syntax: “`if (`*scalar-logical-expression*`)`”
- If the logical expression evaluates to *true*: execute in parallel
- Example:

```
int main( ... )
{
    [...]
    #pragma omp parallel if (n > 1000)
    {
        [...]
    }
    [...]
}
```

Parallel construct

num_threads clause

- Specifies how many threads should execute the region
- The runtime may decide to use less threads than specified (never more)
- Syntax: "`num_threads (`*scalar-logical-expression*`)`"
- Example:

```
int main( ... )
{
    [...]
    #pragma omp parallel num_threads (nths)
    {
        [...]
    }
    [...]
}
```

Parallel construct

Data-sharing attributes

- Shared-memory programming model
- Variables are shared by default
- Shared:
  - All variables visible upon entry of the construct
  - Static variables
- Private:
  - Variables declared within a parallel region
  - (Stack) variables in functions called from within a parallel region

Parallel construct

```
int N = 10;
int main( void )
{
  double array[N];
  #pragma omp parallel
  {
    int i, myid;
    double thread_array[N];
    [...]
    for ( i = 0; i < N; i++ )
      thread_array[i] = myid * array[i];
    function( thread_array );
  }
}
double function( double arg )
{
    static int cnt;
    double local_array[N];
    [...]
}
```

Within parallel region:

- Shared: `array`, `N`, `cnt`
- Private: `i`, `myid`, `thread_array`, `local_array`

Note:

- Lexical extent vs dynamic/runtime extent

Parallel construct

General rules for data-sharing clauses

- Clauses `default`, `private`, `shared`, `firstprivate` allow changing the default behavior
- The clauses consist of the keyword and a comma-separated list of variables in parenthesis. For instance: `private(a,b)`
- Variables must be visible in the lexical extent of the directive
- A variable can only appear in one clause
    - Exception: a variable can appear in both `firstprivate` and `lastprivate` (coming later)

Parallel construct

`shared` clause

- Syntax: "`shared` (*item-list*)"
- Specifies that variables in the comma-separated list are shared among threads
- One single instance, each thread can freely read and modify its value
- When the parallel region finishes, the final values reside in the shared space where the master thread will be able to access it
- CAREFUL: Every thread can access it, race conditions may occur. Synchronize/order the access when needed (e.g., critical construct)

Parallel construct

`private` clause

- Syntax: "`private (`*item-list*`)`"
- Specifies data that will be replicated so that each thread has a local copy
- Changes made to this data by one thread are not visible to the other threads
- Values are undefined upon entry to and exit from the construct
- The storage lasts until the block in which it is created exists

Parallel construct

`firstprivate` clause

- Syntax: "`firstprivate` (*item-list*)"
- Variables in the list are private
- Variables are also initialized with the value the corresponding original variable had when the construct was encountered

Parallel construct

`default` clause

- Syntax: "`default ( shared | none )`"
- `default(shared)` causes all variables to be shared by default
- `default(none)` requires that each variable must have its data-sharing attribute explicitly listed in a data-sharing clause
- Only one single default clause may be specified
- It is considered a **good programming practice** to *always* use `default(none)` to enforce the explicit listing of data-sharing attributes

Exercise on data sharing: Think about it!

- Given the following sample code

```
int A=1, B=1, C=1;
#pragma omp parallel private(B) firstprivate(C)
{
  [...]
}
```

- Are A, B, and C shared or private to each thread inside the parallel region?
- What are the initial values inside the region?
- What are the values after the parallel region?

Parallel construct

`reduction` clause

- Specifies a reduction operation
- Syntax: "`reduction (`*operator:list*`)`"
- Predefined operators: `+, *, -, &, |, ^, &&, ||, max, min`
- Each reduction operator has an *initializer* and a *combiner*. For instance:
  - `+`. Initializer: 0; combiner: accum += var
  - `*`. Initializer: 1; combiner: accum *= var
- Arrays are not allowed in a reduction clause!

Parallel construct

`reduction` clause

- For each list item, each thread gets a private copy
- The private copy is initialized with the initializer value
- At the end of the region, the original item is updated with the values of the private copies using the combiner
- See `04.parallel-reduction.c`

Exercise 3 (pi.c) - `reduction` clause

- This time use the `parallel` construct in combination with the `reduction` clause to compute π.

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_STEPS 10000

int main( void )
{
  int i;
  double sum = 0.0, pi, x_i;
  double step = 1.0/NUM_STEPS;

  for ( i = 0; i < NUM_STEPS; i++ ) {
    x_i = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x_i * x_i);
  }
  pi = sum * step;
  printf("Pi: %.15e\n", pi);
  return 0;
}
```

Hints:

- `#pragma omp parallel reduction(...)`
- `omp_get_num_threads(...)`
- `omp_get_thread_num(...)`
- Challenges:
  - split iterations of the loop among threads
  - create an accumulator for each thread to hold partial sums, which can later be combined to generate the global sum. Let `reduction` take care of this!

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

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

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:

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

Exercise 2 (axpy.c) - revisited

- Use the `parallel` and `for` constructs to parallelize the code below so that 4 threads collaborate in the computation of z.

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

int main(int argc, char *argv[])
{
    int i, N = 10;
    double x[N], y[N], z[N], alpha = 5.0;

    for (i = 0; i < N; i++) {
        x[i] = i;
        y[i] = 2.0*i;
    }

    for (i = 0; i < N; i++)
        z[i] = alpha * x[i] + y[i];
    // Print results. Should output [0, 7, 14, 21, ...]
    return 0;
}
```

Hints:

- `#pragma omp parallel num_threads(...)`
- `#pragma omp for`

Exercise 3 (pi.c) - `reduction + for`

- Use the `parallel` and `for` constructs to parallelize the code below. Use also the `reduction` clause.

```
#include <stdio.h>
#include <stdlib.h>
#define NUM_STEPS 10000

int main( void )
{
  int i;
  double sum = 0.0, pi, x_i;
  double step = 1.0/NUM_STEPS;

  for ( i = 0; i < NUM_STEPS; i++ ) {
    x_i = (i + 0.5) * step;
    sum = sum + 4.0 / (1.0 + x_i * x_i);
  }
  pi = sum * step;
  printf("Pi: %.15e\n", pi);
  return 0;
}
```

Hints:

- `#pragma omp parallel reduction(...)`
- `#pragma omp for`

Loop construct - Clauses

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