OpenMP I

Diego Fabregat-Traver and Prof. Paolo Bientinesi

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

WS16/17

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`

OpenMP

- API for shared-memory parallelism
- Steered by the OpenMP ARB (industry, research)
- Supported by compilers on most platforms
- Not a programming language. Mainly annotations to the (sequential) code.
- OpenMP API consists of:
  - Compiler directives
  - Library routines
  - Environment variables
- Simple to use, high-level, incremental parallelism
- Performance oriented
- Data (and task) parallelism

(Very brief) History of OpenMP

- SC97: Group HPC experts (industry, research) presented OpenMP, to propose a unified model to program shared-memory systems.
- A company was set up to own and maintain the new standard: The openmp architecture review board (openmparb)
- People efforts on: extending the standard, developing implementations, teaching and spreading the word, cOMPunity for the interaction between vendors, researchers and users.
- Originally primarily designed to exploit concurrency in structured loop nests.

Main ideas

- User gives a high-level specification of the portions of code to be executed in parallel

```
int main( ... )
{
    ...
    #pragma omp parallel
    {
        <region executed by multiple threads>
    }
    ...
}
```

`pragma` (pragmatic): tell the compiler to use some compiler-dependent feature/extension.

Main ideas (II)

- User may provide additional information on how to parallelize
  - #pragma omp parallel num_threads(4)
  - omp_set_schedule(static | dynamic | ...);
- OpenMP takes care of the low level details of creating threads, execution, assigning work, ...
- Provides relatively easy variable scoping, synchronization and primitives to avoid data races.
- Usage:
  - #include "omp.h"
  - [gcc|icc] -fopenmp <source.c> -o <executable.x>

Hello world!

Exercise 1: Warming up

- Write an OpenMP multi-threaded program where each thread prints "Hello world!".

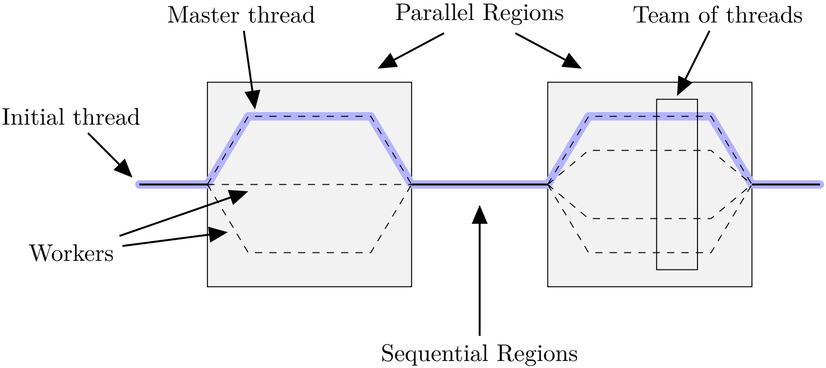
```
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
    printf("Hello world!\n");

    return 0;
}
```

Hint:

`#pragma omp parallel`

Main ideas (III)

Fork-join paradigm



Incremental parallelism

- A common approach to writing OpenMP programs:
  - Identify paralellism in your sequential code
  - Incremental parallelism: introduce directives in one portion of the code, leave the rest untoched
  - When tested, move on to next region to be parallelized until target speedup is achieved

First steps

- Directives:
  - Syntax: `#pragma omp <construct> [<clause> [<clause>]]`
  - Most constructs apply to structured blocks
  - One entry point, one exit point
- Routines (some examples):
  - `omp_set_num_threads( int nthreads );`
  - `int id = omp_get_num_threads();`
  - `int id = omp_get_thread_num();`
- Environment variables (an example):
  - `export OMP_NUM_THREADS=4; ./program.x`

Hello world! I’m thread X!

Exercise 1b

- Extend exercise 1 (below) so that 4 threads execute the parallel region and each of them prints also its thread id.

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

int main( void )
{
    #pragma omp parallel
    printf("Hello world!\n");

    return 0;
}
```

Hints:

- `#pragma omp parallel num_threads(...)`
- `omp_get_num_threads()`
- `omp_set_num_threads(...)`
- `omp_get_thread_num(...)`

Exercise 2 (axpy.c)

- Use the `#pragma omp parallel` construct 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(...)`
- `omp_set_num_threads(...)`
- `omp_get_num_threads(...)`
- `omp_get_thread_num(...)`
- Challenge: split iterations of the loop among threads

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)

Exercise 3 (pi.c)

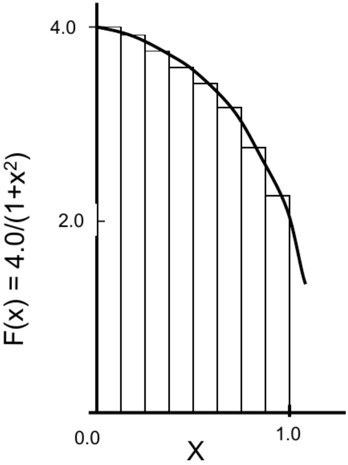
Mathematically, we know that:

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

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;
}
```

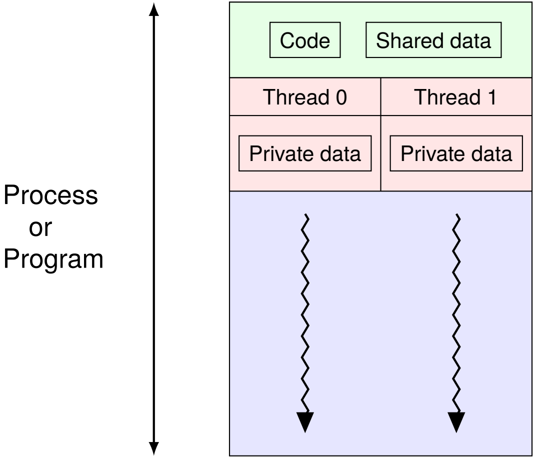
Hints:

- `#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

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`

Variable Scope



Parallel construct: Syntax in detail

- `#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`
- *list* is a list of one or more variables
- Example: `#pragma omp parallel reduction(+:mysum)`

Parallel construct

`reduction` clause

- Each reduction operator has an *initializer* and a *combiner*. For instance:
  - `+`. Initializer: 0; combiner: accum += var
  - `*`. Initializer: 1; combiner: accum *= var
- 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
- Compare `03c.pi-omp-manual-red.c` vs `03d.pi-omp-red.c`

User-defined reductions

- We can also define our own reduction operators
- The syntax is:

  `#pragma omp declare reduction (`*`reduction-identifier : typename : combiner`*` ) [initializer-clause]`

- *reduction-identifier* is the identifier we want to give to our reduction. E.g., `mymax`
- *typename* is the datatype to which the reduction applies. E.g., `int`
- (continues in next slide ...)

Parallel construct

User-defined reductions

- `#pragma omp declare reduction (`*`reduction-identifier : typename : combiner`*` ) [initializer-clause]`

- *combiner* is an expression or function that specifies how to combine the partial result of each thread with the global result. It must be expressed in terms of the variables `omp_in` and `omp_out`. E.g., `omp_out = my_max_function(omp_out, omp_in)`, where

```
int my_max_function( int omp_out, int omp_in )
{
   if ( omp_out > omp_in )
      return omp_out;
   else
      return omp_in;
}
```

  is defined somewhere in our code.

Parallel construct

User-defined reductions

- `#pragma omp declare reduction` (*reduction-identifier : typename : combiner*) `[initializer-clause]`
- *initializer-clause* is an expression or function that specifies how to initialize the private copies of each thread. It must be expressed in terms of the variable `omp_priv`. E.g., `omp_priv = INT_MIN`
- See `05b.reduction-max-userdefined.c`