Programming with Shared Memory

Part 2

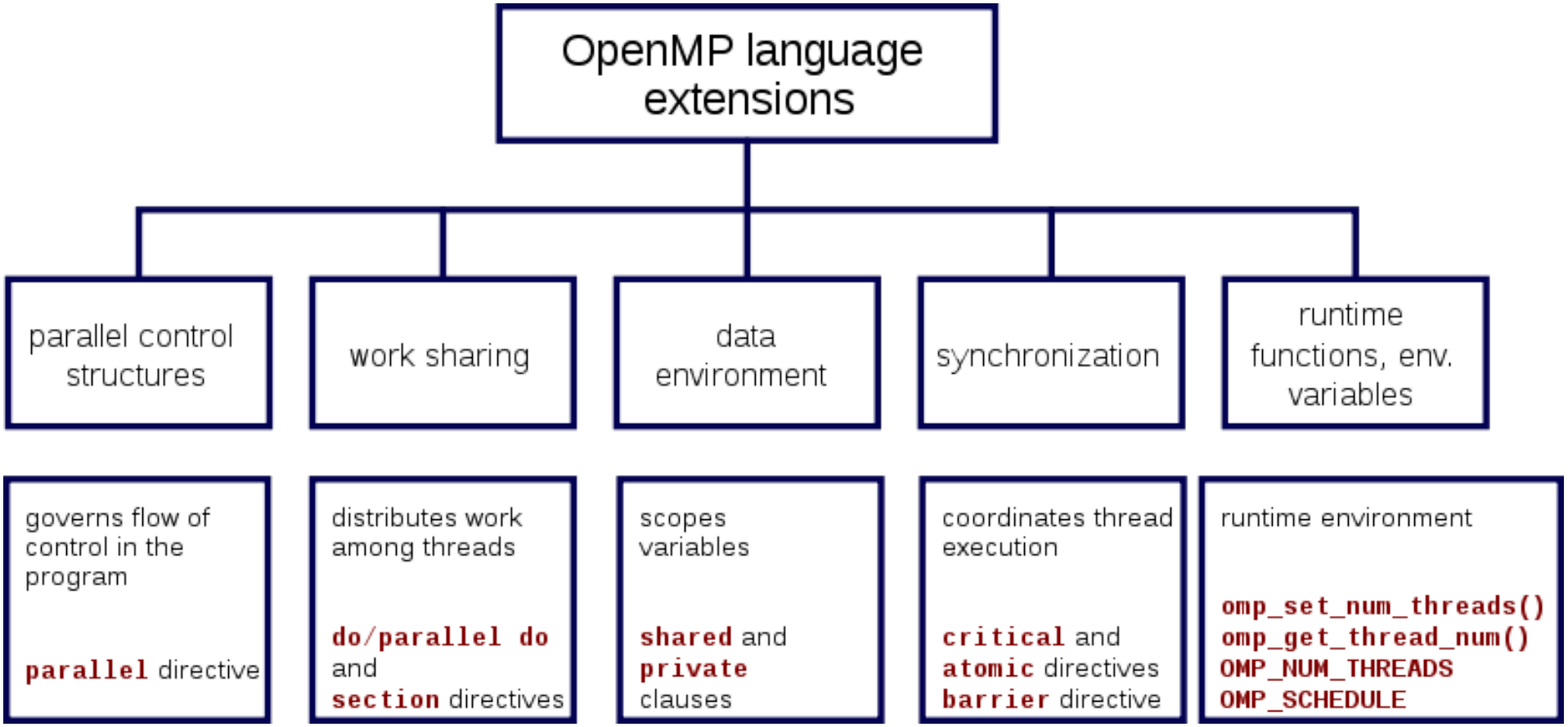
Introduction to OpenMP

OpenMP

An accepted standard developed in the late 1990s by a group of industry specialists.

Consists of a small set of compiler directives, augmented with a small set of library routines and environment variables using the base language Fortran and C/C++.

Several OpenMP compilers available.



Wikipedia OpenMP
http://en.wikipedia.org/wiki/File:OpenMP_language_extensions.svg

OpenMP

- Uses a thread-based shared memory programming model
- OpenMP programs will create multiple threads
- All threads have access to global memory
- Data can be shared among all threads or private to one thread
- Synchronization occurs but often implicit

OpenMP uses “fork-join” model but thread-based.

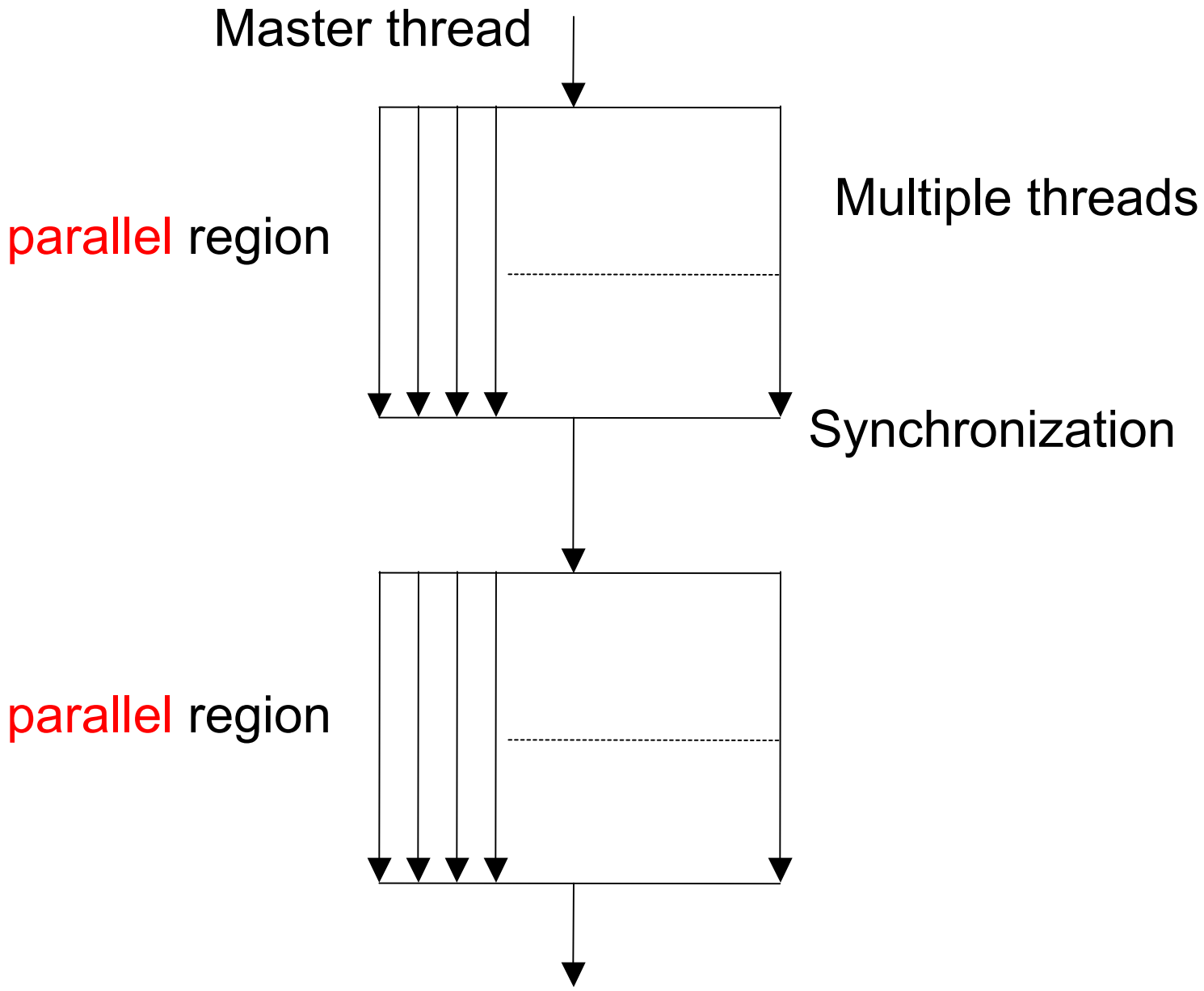
Initially, a single thread executed by a master thread.

parallel directive creates a team of threads with a specified block of code executed by the multiple threads in parallel.

The exact number of threads in the team determined by one of several ways.

Other directives used within a parallel construct to specify parallel for loops and different blocks of code for threads.

Fork/join model



For C/C++, the OpenMP directives contained in #pragma statements.
Format:

#pragma omp directive_name ...

where omp is an OpenMP keyword.

May be additional parameters (clauses) after directive name for different options.

Some directives require code to specified in a structured block that follows directive and then directive and structured block form a “construct”.

Parallel Directive

#pragma omp parallel

structured_block

creates multiple threads, each one executing the specified structured_block, (a single statement or a compound statement created with { ...} with a single entry point and a single exit point.)

Implicit barrier at end of construct.

Directive corresponds to forall construct.

Hello world example

Opening brace must on a new line

OpenMP directive for a parallel region

```
#pragma omp parallel
{

printf("Hello World from thread = %d\n", omp_get_thread_num(),
                                         omp_get_num_threads());

}
```

Output from an 8-processor/core machine:

Hello World from thread 0 of 8
Hello World from thread 4 of 8
Hello World from thread 3 of 8
Hello World from thread 2 of 8
Hello World from thread 7 of 8
Hello World from thread 1 of 8
Hello World from thread 6 of 8
Hello World from thread 5 of 8

Private and shared variables

Variables could be declared within each parallel region but OpenMP provides private clause.

Each thread has a local variable tid

```
int tid;
…
#pragma omp parallel private(tid)
{
        tid = omp_get_thread_num();
        printf("Hello World from thread = %d\n", tid);
}
```

Also a shared clause available.

Example

```
#pragma omp parallel private(x, num_threads)
{
  x = omp_get_thread_num();
  num_threads = omp_get_num_threads();
  a[x] = 10*num_threads;
}
```

Two library routines

`omp_get_num_threads()` returns number of threads that are currently being used in parallel directive

`omp_get_thread_num()` returns thread number (an integer from 0 to `omp_get_num_threads()` – 1 where thread 0 is the master thread).

Array `a[]` is a global array, and `x` and `num_threads` are declared as private to the threads.

Number of threads in a team

Established by either:

1.num_threads clause after the **parallel** directive, or

2. **omp_set_num_threads()** library routine being previously called, or

3. Environment variable **OMP_NUM_THREADS** is defined

in order given or is system dependent if none of above.

Number of threads available can also be altered dynamically to achieve best use of system resources.

Work-Sharing

Three constructs in this classification:

sections

for

single

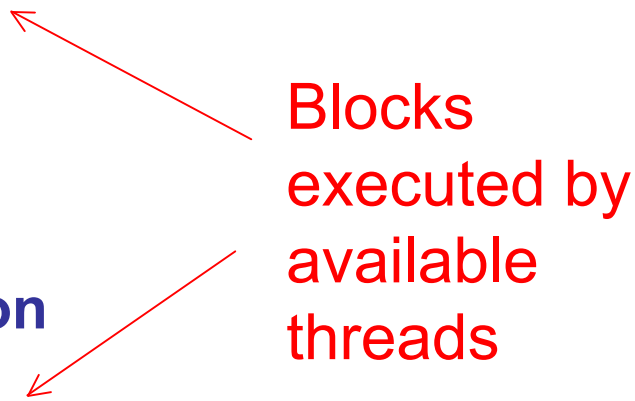
In all cases, there is an implicit barrier at end of construct unless a **nowait** clause included, which overrides the barrier.

Note: These constructs do not start a new team of threads. That done by an enclosing parallel construct.

Sections

The construct

```
#pragma omp sections
{
    #pragma omp section
        structured_block
            .
            .
            .
    #pragma omp section
        structured_block
}
```



cause structured blocks to be shared among threads in team.
The first section directive optional.

Example

```
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    #pragma omp sections nowait
    {
        #pragma omp section
        {
            printf("Thread %d doing section 1\n",tid);
            for (i=0; i<N; i++) {
                c[i] = a[i] + b[i];
                printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
            }
        }

        #pragma omp section
        {
            printf("Thread %d doing section 2\n",tid);
            for (i=0; i<N; i++) {
                d[i] = a[i] * b[i];
                printf("Thread %d: d[%d]= %f\n",tid,i,d[i]);
            }
        }
    }  /* end of sections */
}  /* end of parallel section */
```

One thread does this

Another thread does this

For Loop

#pragma omp for

for (i = 0; ….)

For loop of a simple form

causes **for** loop to be divided into parts and parts shared among threads in the team. **for** loop must be of a simple form.

Way **for** loop divided can be specified by additional “schedule” clause.

Example

schedule (static, chunk_size)

for loop divided into sizes specified by **chunk_size** and allocated to threads in a round robin fashion.

Example

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
     tid = omp_get_thread_num();
     if (tid == 0) {
          nthreads = omp_get_num_threads();
          printf("Number of threads = %d\n", nthreads);
      }
     printf("Thread %d starting...\n",tid);

 #pragma omp for schedule(dynamic,chunk)
     for (i=0; i<N; i++) {
          c[i] = a[i] + b[i];
          printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
      }

} /* end of parallel section */
```

Executed by one thread

For loop

Single

The directive

#pragma omp single

structured block

cause the structured block to be executed by one thread only.

Combined Parallel Work-sharing Constructs

If a parallel directive is followed by a single for directive, it can be combined into:

```
#pragma omp parallel for
    <for loop>
```

with similar effects.

If a parallel directive is followed by a single sections directive, it can be combined into

```
#pragma omp parallel sections
{
    #pragma omp section
        structured_block
    #pragma omp section
        structured_block
                .
                .
                .

}
```

with similar effect. (In both cases, the nowait clause is not allowed.)

Master Directive

The **master** directive:

```
#pragma omp master
structured_block
```

causes the master thread to execute the structured block.

Different to those in the work sharing group in that there is no implied barrier at the end of the construct (nor the beginning). Other threads encountering this directive will ignore it and the associated structured block, and will move on.

Loop Scheduling and Partitioning

OpenMP offers scheduling clauses to add to **for** construct:

- Static

  #pragma omp parallel for schedule (static,chunk_size)

  Partitions loop iterations into equal sized chunks specified by chunk_size. Chunks assigned to threads in round robin fashion.

- Dynamic

  #pragma omp parallel for schedule (dynamic,chunk_size)

  Uses internal work queue. Chunk-sized block of loop assigned to threads as they become available.

- Guided

  #pragma omp parallel for schedule (guided,chunk_size)

Similar to dynamic but chunk size starts large and gets smaller to reduce time threads have to go to work queue.

$$\text{chunk size} = \left\lceil \frac{\text{number of iterations remaining}}{2 * \text{number of threads}} \right\rceil$$

- Runtime

  #pragma omp parallel for schedule (runtime)

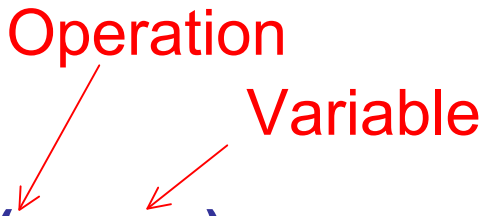
Uses OMP_SCEDULE environment variable to specify which of static, dynamic or guided should be used.

Reduction clause

Used combined the result of the iterations into a single value c.f. with MPI _Reduce().

Can be used with parallel, for, and sections,

Example




```
sum = 0
#pragma omp parallel for reduction(+:sum)
        for (k = 0; k < 100; k++ ) {
                sum = sum + funct(k);
        }
```

Private copy of sum created for each thread by complier.
Private copy will be added to sum at end.
Eliminates here the need for critical sections.

Private variables

private clause – creates private copies of variables for each thread

firstprivate clause - as private clause but initializes each copy to the values given immediately prior to parallel construct.

lastprivate clause – as private but “the value of each lastprivate variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable’s original object.”

Synchronization Constructs Critical

critical directive will only allow one thread execute the associated structured block. When one or more threads reach the critical directive:

#pragma omp critical *name*

structured_block

they will wait until no other thread is executing the same critical section (one with the same *name*), and then one thread will proceed to execute the structured block.

name is optional. All critical sections with no name map to one undefined name.

Barrier

When a thread reaches the barrier

#pragma omp barrier

it waits until all threads have reached the barrier and then they all proceed together.

There are restrictions on the placement of barrier directive in a program. In particular, all threads must be able to reach the barrier.

Atomic

The atomic directive

```
#pragma omp atomic
    expression_statement
```

implements a critical section efficiently when the critical section simply updates a variable (adds one, subtracts one, or does some other simple arithmetic operation as defined by expression_statement).

Flush

A synchronization point which causes thread to have a "consistent" view of certain or all shared variables in memory.
All current read and write operations on variables allowed to complete and values written back to memory but any memory operations in code after flush are not started.

Format:

#pragma omp flush (variable_list)

Only applied to thread executing flush, not to all threads in team.

Flush occurs automatically at entry and exit of parallel and critical directives, and at the exit of for, sections, and single (if a no-wait clause is not present).

Ordered clause

Used in conjunction with **for** and **parallel for** directives to cause an iteration to be executed in the order that it would have occurred if written as a sequential loop.

More information

Full information on OpenMP at

http://openmp.org/wp/