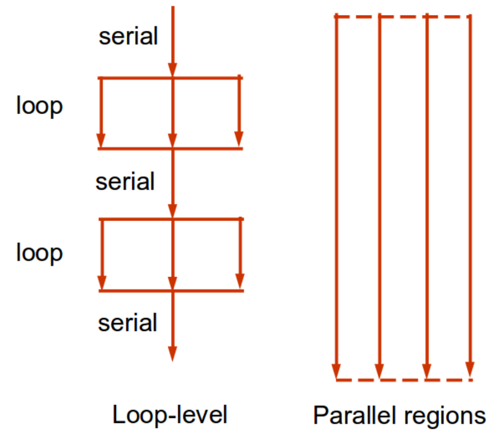
CMSC 691
High Performance Distributed Systems

OpenMP

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu

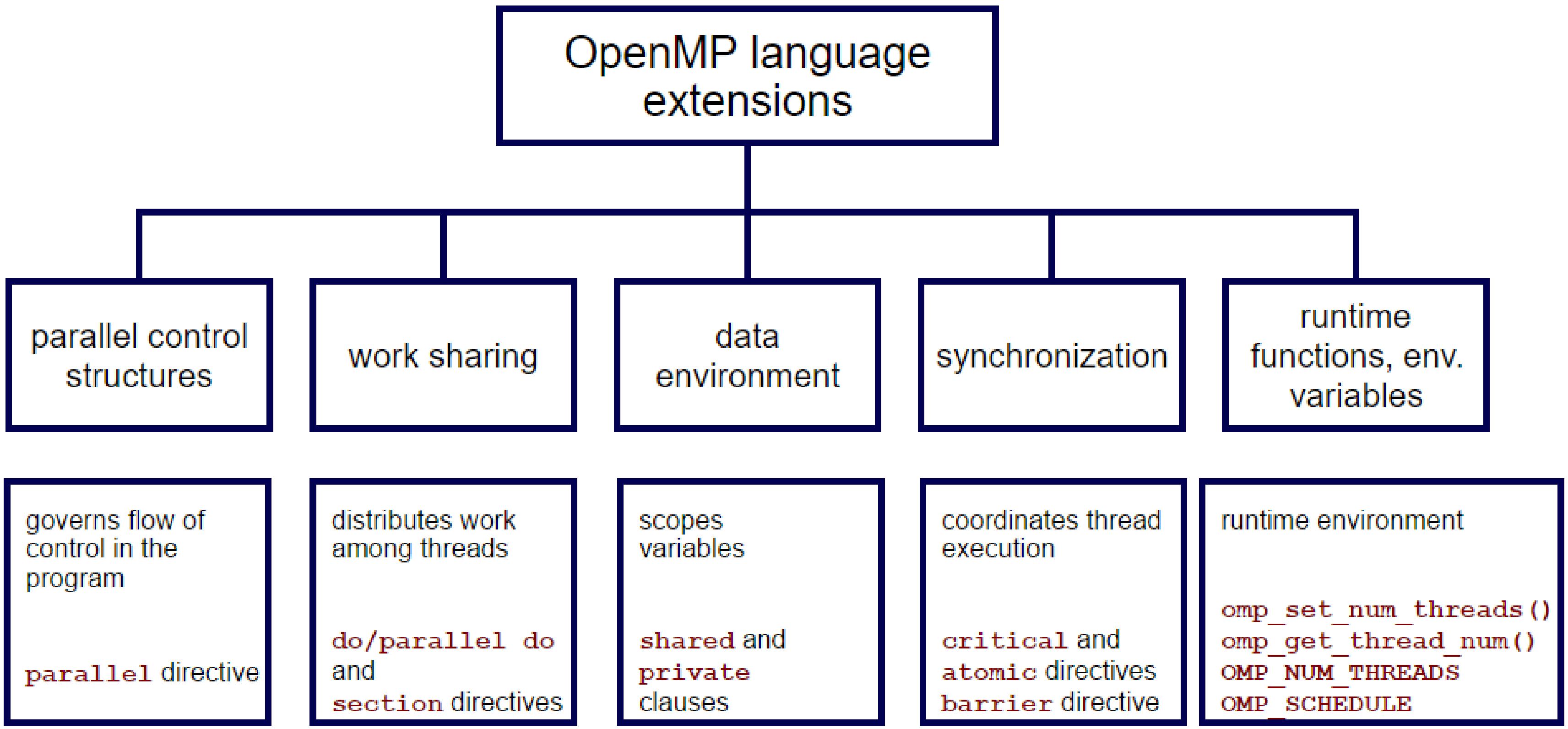
OpenMP (Open Multi-Processing)

- API for shared memory parallel programming in C/C++/Fortran
- Compiler directives and library routines for Windows / Linux / OS X
- Higher level of abstraction than POSIX threads
- Easy to use, incremental parallelization, flexible, portable
- Basic approaches:
  - Loop-level parallelism
  - Parallel regions



OpenMP (Open Multi-Processing)

- Core elements



Thread creation: Hello World!

- The pragma omp parallel is used to fork additional threads

```
void main() {
  omp_set_num_threads(8);

  #pragma omp parallel
  {
      int tid = omp_get_thread_num();
      int nthreads = omp_get_num_threads();
      printf("Hello world thread %d, total %d\n", tid, nthreads);
  }
}
```

- Compile `gcc -fopenmp hello.c -o hello`
- By default it creates as many threads as the number of cores
- omp_set_num_threads(nthreads); to override default nthreads

Loop-level parallelism

```
#pragma omp parallel for num_threads(4)
for(int i = 0; i < size; i++)
{
   c[i] = a[i] + b[i];
}
```

- The loop workload will be automatically parallelized into nthreads, each one computing (size / nthreads) sums
- Implies a barrier at the end of the loop
- Easy and transparent scalability!
- num_threads(nthreads) allows to specify the number of threads

Loop-level parallelism vs parallel region

- Equivalency of the code

```
#pragma omp parallel for
for(int i = 0; i < size; i++)
   c[i] = a[i] + b[i];
```

```
#pragma omp parallel
{
  int id, nthrds, start, end;
  id = omp_get_thread_num();
  nthrds = omp_get_num_threads();
  start = id * size / nthrds;
  end = (id+1) * size / nthrds;
  if (id == nthrds-1) end = size;
  for(int i = start; i < end; i++)
    c[i] = a[i] + b[i];
}
```

Shared and private variables

- In parallel region, the default behavior is that all variables are shared except loop index, which is private for each thread
- Ok when all threads read and write different memory locations, accessing different elements of an array. Problem if threads write same scalar or array element, example:

```
int result;
#pragma omp parallel for
for(int i = 0; i < size; i++)
{
   result = a[i] + b[i];
   c[i] = result;
}
```

Shared and private variables

- *private* clause creates a separate memory location for the specified variable for each thread

```
int result;
#pragma omp parallel for private(result)
for(int i = 0; i < size; i++)
{
   result = a[i] + b[i];
   c[i] = result;
}
```

- Let's see the behavior of the code with/without the private clause
- Data races happen among the threads!

Shared and private variables

- *default none* allows to define all variables as shared or private

```
int result;
#pragma omp parallel for default (none) \
                         shared (size,a,b,c) \
                         private(result)
for(int i = 0; i < size; i++)
{
   result = a[i] + b[i];
   c[i] = res;
}
```

- Explicit control of the behavior per shared or private variable
- Loop iterative variable is always private

Data dependencies

- Cannot assume threads will run in any specific order
- Example: Fibonacci sequence

```
seq[0] = 0;
seq[1] = 1;
for(int i = 3; i < size; i++)
{
   seq[i] = seq[i-1] + seq[i-2];
}
```

- Easy test for checking dependencies. If the serial loop is executed in reverse order, will it give the same result?
- Be careful when calling depended functions

Reduction

- Each thread performs its own reduction
- Results from all threads are automatically reduced at the end

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for(int i = 0; i < size; i++)
{
   sum += array[i];
}
```

- Operators: +, *, -, /, &, ^, |, &&, ||
- Roundoff error may be different than serial case (e.g. Kahan summation algorithm to reduce the numerical error)

Performance hints

- OpenMP will do what you tell it to do, be aware of dependencies
- Do not parallelize small loops, overhead will be > than speedup!
- Parallelize outer loops, maximize number of parallel operations
- Merge parallel sections

```
#pragma omp parallel
#pragma omp for
for(int i = 0; i < size1; i++)
    ...
#pragma omp for
for(int i = 0; i < size2; i++)
    ...
#pragma omp parallel end
```

Barriers

- Synchronization point, threads wait until all threads arrive

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp barrier
  do_many_other_things();
}
```

Master / single

- Forces a single unique thread to execute a section

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp single
  { exchange_boundaries(); }
  // implicit synchronization
  do_many_other_things();
}
```

```
#pragma omp parallel
{
  do_many_things();
  #pragma omp single
  { exchange_boundaries(); }
  // no synchronization implied
  do_many_other_things();
}
```

Critical sections

- Mutual exclusion: only one thread at a time can enter a critical section

```
int counter = 0;
#pragma omp parallel num_threads(1000)
{
  #pragma omp critical
  counter++;
}
```

- Atomic: provides mutual exclusion but only applies to the update of a memory location

```
int counter = 0;
#pragma omp parallel num_threads(1000)
{
  #pragma omp atomic
  counter++;
}
```

Forcing sequential execution

```
#pragma omp parallel for
{
  #pragma omp ordered
  printf("Thread %d\n", omp_get_thread_num());
}
```

Locks

```
omp_lock_t lock;
omp_init_lock(&lock);

#pragma omp parallel
{
  omp_set_lock(&lock);
  ... // critical section
  omp_unset_lock(&lock);
}

omp_destroy_lock(&lock);
```

CMSC 691
High Performance Distributed Systems

OpenMP

Dr. Alberto Cano
Assistant Professor
Department of Computer Science
acano@vcu.edu