Directive-Based Programming with OpenMP

Shared Memory Programming

- Explicit thread creation (pthreads):

```
pthread t thread;
pthread create(&thread, &attr, some_function, (void *)
&result);
…
pthread join(thread, NULL);
```

- Tasks (C++ 11):

```
auto handle = std::async(std::launch::async, some_code, ...);
auto result = handle.get();
```

- Kernels (CUDA):

```
__global__
void someKernel(...) {
  int idx = blockIdx.x*blockDim.x + threadIdx.x
  // execute code for given index
}
```

OpenMP

- API for shared memory parallel programming targeting Fortran, C and C++
- specifications maintained by OpenMP Architecture Review Board (ARB)
- members include AMD, ARM, Cray, Intel, Fujitsu, IBM, NVIDIA
  - versions 1.0 (Fortran ’97, C ’98) - 3.1 (2011) shared memory
  - 4.0 (2013) accelerators, NUMA
  - 4.5 (2015) improved memory mapping, SIMD
  - 5.0 (2018) improved accelerator support

OpenMP

- comprises compiler directives, library routines and environment variables
  - C directives (case sensitive)
    ```
    #pragma omp directive-name [clause-list]
    ```
  - library calls begin with omp
    ```
    void omp_set_num_threads(int nthreads);
    ```
  - environment variables begin with OMP
    ```
    export OMP_NUM_THREADS=4
    ```
- requires compiler support
  - activated via `-fopenmp` (gcc/clang) or `-openmp` (icc) compiler flags

The `parallel` Directive

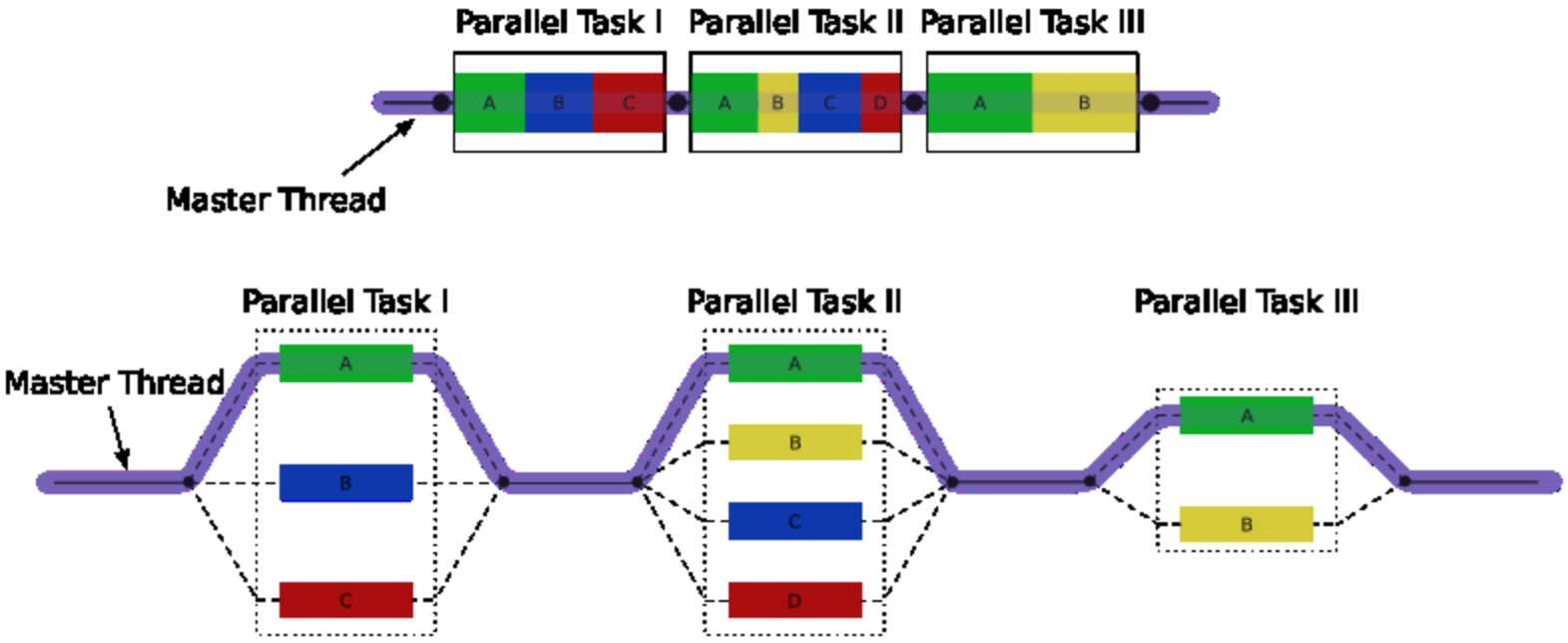
- OpenMP uses a fork/join model: programs execute serially until they encounter a `parallel` directive:
  - this creates a group of threads
  - number of threads is dependent on the `OMP_NUM_THREADS` environment variable or set via function call, e.g. `omp_set_num_threads(nthreads)`
  - main thread becomes the master thread, with thread id of 0

```
#pragma omp parallel [clause-list]
{
/*structured block*/
}
```

- each thread executes the structured block

Fork/Join in OpenMP

- Conceptually, threads are created and destroyed for each parallel region; in practice, usually implemented as a thread pool



Parallel Directive: Clauses

Clauses are used to specify:

- conditional parallelization: to determine if the parallel construct results in creation/use of threads
  `if (scalar-expression)`
- degree of concurrency: explicit specification of the number of threads created/used
  `num threads(integer-expression)`
- data handling: to indicate if specific variables are local to the thread (allocated on the thread's stack), global, or 'special'
  `private(variable-list)`
  `shared(variable-list)`
  `firstprivate(variable-list)`
  `default(shared j none)`

Compiler Translation: OpenMP to Pthreads

- OpenMP code

```
main() {
  int a, b;
  // serial segment
  # pragma omp parallel num_threads(8) private(a) shared(b)
  { /* parallel segment */ }
  // rest of serial segment
}
```

- Pthreads equivalent (structured block is *outlined*)

```
main() {
  int a, b;
  // serial segment
  for (i=0; i<8; i++) pthread_create (..... , internal_thunk ,...);
  for (i=0; i<8; i++) pthread_join (........);
  // rest of serial segment
}
void *internal_thunk(void *packaged argument) {
  int a;
  /* parallel segment */
}
```

Parallel Directive Examples

```
# pragma omp parallel if (is_parallel == 1) num_threads(8) \
   private(a) shared(b) firstprivate(c)
```

- if the value of variable `is_parallel` is one, eight threads are used
- each thread has private copy of `a` and `c`, but all share one copy of `b`
- the value of each private copy of `c` is initialized to value of `c` before the parallel region

```
# pragma omp parallel reduction(+ : sum) num_threads(8) \
   default(private)
```

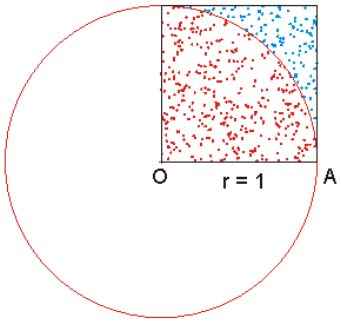
- eight threads get a copy of the variable `sum`
- when threads exit, the values of these local copies are accumulated into the `sum` variable on the master thread
  - other reduction operations include *, -, &, |, ^, &&, ||
- all variables are private unless otherwise specified

Example: Computing Pi

compute π by generating random points in square of side length 2 centred at (0,0), and counting points falling within circle of radius 1

- area of square = 4, area of circle: $\pi r^2 = \pi$
- ratio of points in circle to outside approaches π/4

```
# pragma omp parallel private(i) shared(npoints) \
   reduction(+ : sum) num_threads(8)
{ int seed = omp_get_thread_num(); // private
  num threads = omp_get_num_threads();
  sum = 0;
  for (i = 0; i < npoints / num_threads; i++) {
    rand_x = ( double ) rand_range (& seed , -1, 1);
    rand_y = ( double ) rand_range (& seed , -1, 1);
    if (( rand_x * rand_x + rand_y * rand_y ) <= 1.0)
      sum ++;
  }
}
```



Jirah, Monte-Carlo01, CC BY-SA 3.0

The `for` Work-Sharing Directive

- use with the parallel directive to partition a subsequent `for` loop

```
# pragma omp parallel shared(npoints) \
reduction (+: sum) num_threads(8)
{ int seed = omp_get_thread_num();
  sum = 0;
  # pragma omp for
  for (i = 0; i < npoints ; i++) {
    rand x = (double) rand_range(& seed, -1, 1);
    rand y = (double) rand_range(& seed, -1, 1);
    if (( rand_x * rand_x + rand_y * rand_y ) <= 1.0) sum++;
  }
}
```

  - the loop index (`i`) is assumed to be private
  - only two directives plus sequential code (code is easy to read/maintain)
- implicit synchronization at the end of the loop
  - can add a `nowait` clause to prevent this
- it is common to merge the directives: `#pragma omp parallel for ...`

Assigning Iterations to Threads

- the `schedule` clause of the `for` directive assigns iterations to threads
- `schedule(static[,`*`chunk-size`*`])`
  - splits the iteration space into chunks of size *`chunk-size`* and allocates to threads in round-robin fashion
  - if chunk size is unspecified, number of chunks equals number of threads
- `schedule(dynamic[,`*`chunk-size`*`])`
  - iteration space is split into *`chunk-size`* blocks scheduled dynamically
- `schedule(guided[,`*`chunk-size`*`])`
  - chunk size decreases exponentially with iterations to a minimum of *`chunk-size`*
- `schedule(runtime)`
  - determine scheduling based on setting of the `OMP_SCHEDULE` environment variable

Synchronization in OpenMP

- `barrier`: each thread waits until others arrive (`nowait`: skip barrier)
- `single`: executed by one thread only

```
#pragma omp parallel
{
// my part of computation
#pragma omp single
{ /* executed by one thread */ }
#pragma omp barrier
#pragma omp for nowait
  for (i=0; i<N; i++) {
    // data parallel part
  }
  // threads continue here automatically
}
```