Advanced OpenMP

Advanced clauses, variable types, race conditions, task parallelism

OpenMP variable types

- When threads are created, an existing variable **x** in the master thread can have different status in the newly created threads, among which:
- **private(x)**: create private copy in each thread, uninitialized
- **firstprivate(x)**: create private copy in each thread, copy initial value
- **threadprivate(x)**: as if having a **global** copy of a variable per thread
- **shared(x):** x of the master is shared among all threads for read/write.
- **default(shared|none):** Variables **shared** by default, **none** disables it.
- **reduction(op:x):** Creates a safe copy for each thread, merges them transparently at the end. Avoids race conditions.

Merging **parallel** and **for/sections** directives

- If the parallel region has an **omp for** or **omp sections** inside, and **nothing else**, you can merge two directives.
- You cannot use curly braces ({ … }) after **#pragma omp parallel for**.

```
#pragma omp parallel num_threads(P)
{
#pragma omp for
  for (int i = 0; i < N; i++) {
    f(i)
  }
}
```

```
#pragma omp parallel for num_threads(P)
  for (int i = 0; i < N; i++) {
    f(i)
  }
```

```
#pragma omp parallel sections num_threads(P)
{
#pragma omp section
  {
    g();
  }
#pragma omp section
  {
    h();
  }
}
```

collapse clause for **omp for**

- **collapse** clause flattens nested loops into a single domain to provide more parallelism (e.g., when iteration domain is small).
- Iteration domain must be a fixed size for all nested loops.
- Can collapse multiple loops (>=2)

```
#pragma omp parallel num_threads(P)
{
// This loop is
#pragma omp for collapse(2)
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      f(i, j);
    }
  }


// Same as this
#pragma omp for
  for (int i = 0; i < N*N; i++) {
    f(i/10, i%10);
  }
}
```

ordered clause for **omp for**

- Specifies a block in a parallel loop that respects the sequential order of iterations in execution.
- Useful for gathering results in order at the end, after performing expensive computations in parallel.

```
// We want to have result[i] = f(i) in the end
std::vector<int> result;

#pragma omp parallel num_threads(P)
{
#pragma omp for ordered
  for (int i = 0; i < N; i++) {
    int res = f(i);  // Done in parallel, time consuming
#pragma omp ordered
    result.push_back(res); // Done in order sequentially
  }
```

Race conditions

- What is the final value of n? min/max?
- There are three hidden instructions:
  - **load** n
  - **add** n, 1
  - **store** n
- Threads simultaneously read the old value before updating n
  - Some increments lost
  - => **race condition**
- Solutions?
  - **atomic**
  - **critical**
  - **reduction**

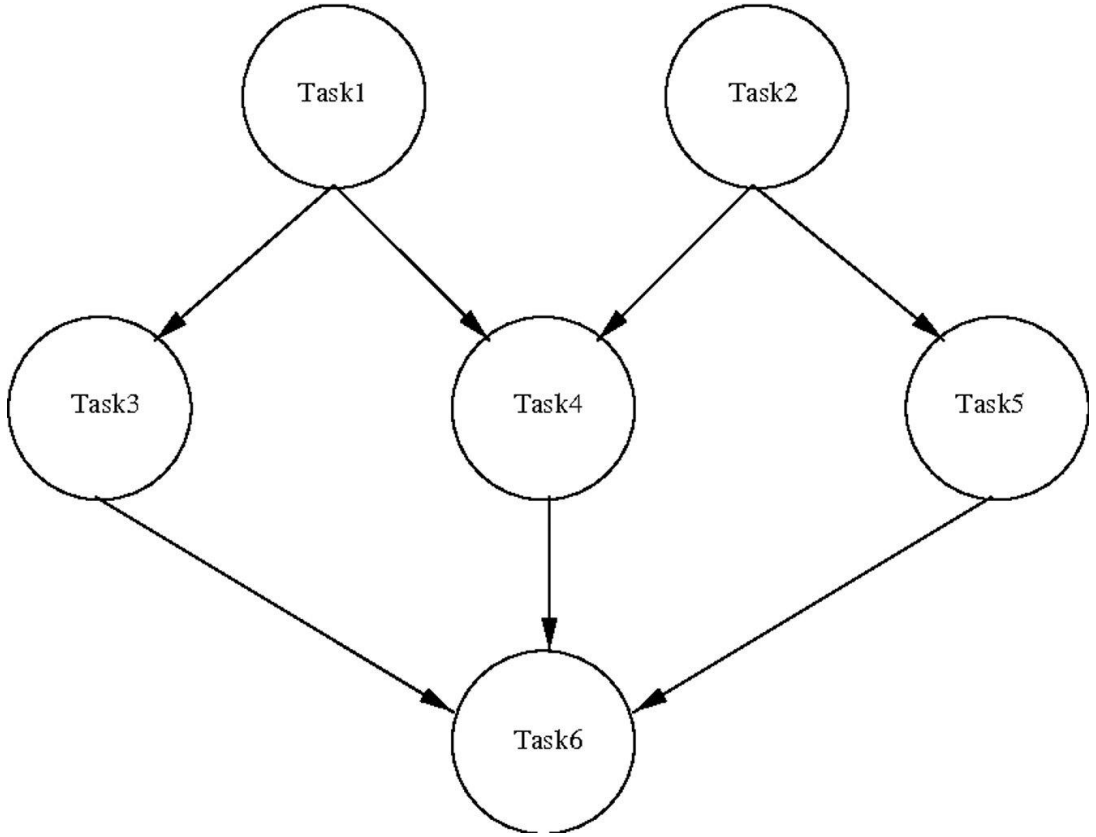
```
// We want to have result[i] = f(i) in the end
int n = 0;

#pragma omp parallel for num_threads(P)
  for (int i = 0; i < 1000000; i++)
    n++;
```

OpenMP task parallelism

Task creation, management, and dependencies

Task-based parallelism

- A task is a block of computation (block of code, function, even loop iterations)
- A task might need another task's output, which is called a **dependency**.
- Task-based parallelism expresses the computation as a bunch of tasks, and their interdependencies.
- A **runtime system** then orchestrates their execution among multiple threads.
- It is particularly useful for
  - irregular parallel computations
  - recursive functions
- Advanced task-based runtime systems enable hybrid GPU+CPU+FPGA parallelism
  - For each task, specify data dependencies
  - For each device/task, specify code/kernel
  - Runtime system orchestrates the distribution of tasks to compute units



Creating tasks in OpenMP

- When a thread encounters a **#pragma omp task** directive, it creates a task involving the following code block, and puts it into a task pool instead of executing it.
- When one of the threads in the parallel region is idle, it executes one of the **ready** tasks (whose all dependencies are satisfied).
- No particular order for task execution, aside from dependencies.
- The program on the right, is it correct?

```
#pragma omp parallel num_threads(P)
{
#pragma omp task
  f();
#pragma omp task
  g();
}
```

Creating tasks in OpenMP

- Each task must be created by a **single thread**
  - Must use a work sharing construct
- **#pragma omp taskwait** waits until all tasks created by current thread/task are done
  - It **does not** wait for all tasks globally.
- **#pragma omp barrier** implies a **taskwait**
- The program on the right, is it correct?

```
#pragma omp parallel num_threads(P)
{
#pragma omp single
  {
#pragma omp task
    f();
#pragma omp task
    g();
#pragma omp taskwait
    // After taskwait f() and g() are guaranteed to
finish.
  }
  // Here, even without taskwait f() and g() are executed.
}

#pragma omp parallel for num_threads(P)
  for (int i = 0; i < N; i++) {
#pragma omp task
    h(i);
  }
```

Task dependencies

- Tasks can have dependencies on variables.
- Dependency can be
  - **input** dependency: Needed to start the task
    - **depend(in:var)** clause after **task**
  - **output** dependency: Other tasks having an input dependency on this variable must wait for this task.
    - **depend(out:var)** clause after **task**
- It is the programmer's responsibility to correctly specify dependencies.

```
// We want to compute x = f() + g();
int x;

#pragma omp parallel num_threads(P)
{
#pragma omp single
  {
#pragma omp task depend(out:x)
    x = f();
#pragma omp task depend(in:x)
    x = x + g();
#pragma omp taskwait
    printf("f() + g() = %d\n", x);
  }
}
```

Example: Fibonacci

- Fib(0) = 0;
- Fib(1) = 1;
- Fib(N) = Fib(N – 1) + Fib(N –2);