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.
  - Using **none** is a good practice for preventing careless bugs
- **reduction(op:x):** Creates a safe copy of a **shared x** for each thread, merges them transparently at the end using **op**. Avoids race conditions.

Example: OpenMP variable types

nowait clause

- Removes the **implicit barrier** at the end of a work sharing construct
  - Enables threads to continue working in the rest of the parallel region
- Can be added at the end of **omp for, omp sections, omp single**
- Should be added diligently; make sure that there is no dependency on the work sharing construct in the rest of the code

```
int N = …;
#pragma omp parallel default(none) num_threads(P)
shared(N)
{
#pragma omp for nowait
  for (int i = 0; i < N; i++) {
    f(i);
  } // end of for, no barrier due to nowait

  // Threads can start the second loop immediately after
  // finishing their chunk for the first loop
#pragma omp for
  for (int i = 0; i < N; i++) {
    g(i);
  } // end of for, implicit barrier
}
```

if clause

```
#pragma omp parallel num_threads(P) if(cond)
{
  // Parallel code to be executed by each thread
}
```

- **if** clause can be added when creating a parallel region or a task
- Threads/tasks are created only if the given **cond** is true/nonzero
  - Otherwise, only the mainthread executes the block of code
- Useful for preventing thread/task creation overhead for small problems
  - Example:
    - **#pragma omp parallel if (N > 128)**
    - **#pragma omp task if (N > 256)**

schedule clause for **omp for**

- Determines how iterations are distributed among threads
- **schedule(policy, chunksize)**
- **static** policy: Assign chunksize contiguous iterations to each thread in a circular order (0,1,2,3,0,1,2,3,…)
- **dynamic** policy: Assign chunksize contiguous iterations to the first available thread (0,3,2,0,1,2,1,3,2,0,…)
- **guided** policy: Start assigning big chunks dynamically, gradually reduce it to chunksize towards the end.
- **runtime** policy: Defer the decision to user in runtime (e.g. using the OMP_SCHEDULE variable)

```
int N = …;
#pragma omp parallel default(none) num_threads(P)
shared(N)
{
#pragma omp for schedule(dynamic, 32)
  for (int i = 0; i < N; i++) {
    f(i);
  } // end of for, implicit barrier
#pragma omp for schedule(static)
  for (int i = 0; i < N; i++) {
    g(i);
  } // end of for, implicit barrier
}
```

static loop scheduling

- Specified with the clause **schedule**(static, chunksize)
- **static** schedule
  - Gives chunksize contiguous iterations to threads in a round-robin manner (th0, th1, …, thP-1, th0, th1, …)
  - If no chunksize specified, it is N/P by default (single chunk per thread)
  - If no schedule is specified, it is static with chunksize=N/P by default

dynamic loop scheduling

- Specified with the clause **schedule**(schedule, chunksize)
- **dynamic** schedule
  - Gives chunksize contiguous iterations to the first available thread (th0, th1, th0, th3, th1, …)
  - If no chunksize specified, it is 1 by default (single iteration per chunk)
  - Good to make sure that no thread is idling
  - Default chunksize is 1
  - Bad for data locality per thread

guided loop scheduling

- Specified with the clause **schedule**(schedule, chunksize)
- **guided** schedule
  - Chunks are distributed dynamically to threads just like **dynamic** (th0, th1, th0, th3, th1, …)
  - Starts with big chunks, but progressively reduces the size to the given chunksize towards the end
  - Makes sure all threads have some workload on the entire iteration domain
  - Keeps good locality for the most part
  - Compromise between **static** and **dynamic**
  - Chunksize is 1 by default

Example: OpenMP loop scheduling

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**.
- You cannot put anything else other than **section** blocks after **#pragma omp parallel sections**

```
#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/N, i%N);
  }
}
```

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.
- Works for all loop schedules.
- As if two consecutive **#omp for** loops merged into one

```
// 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 are 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++;
  }
```

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 are lost
  - => **race condition**
- Solutions?
  - **atomic** directive
    - Uses hardware atomic instructions for the following instruction, has some overhead
    - Works for basic arithmetic / logical operations (+,-,*,/,min,max,&,|,…)

```
// 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++) {
#pragma omp atomic
    n++;
  }
```

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 are lost
  - => **race condition**
- Solutions?
  - **critical** directive
    - Block of code in a **critical** region is executed by **a single thread at a time**
    - Works for all code types but higher overhead than **atomic**

```
// 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++) {
#pragma omp critical
    {
      n++;
    }
  }
```

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 are lost
  - => **race condition**
- Solutions?
  - **reduction(op:x)** clause
    - Creates an invisible copy of the shared variable **x** in each thread, performs the reduction operation **op** on the private copy without atomic nor critical.
    - At the end of the loop, merges these private copies with a single atomic operation **op** on the shared **x**.
    - **op** can be +,-,*,/,min,max,&,|,&&,||,^,… .

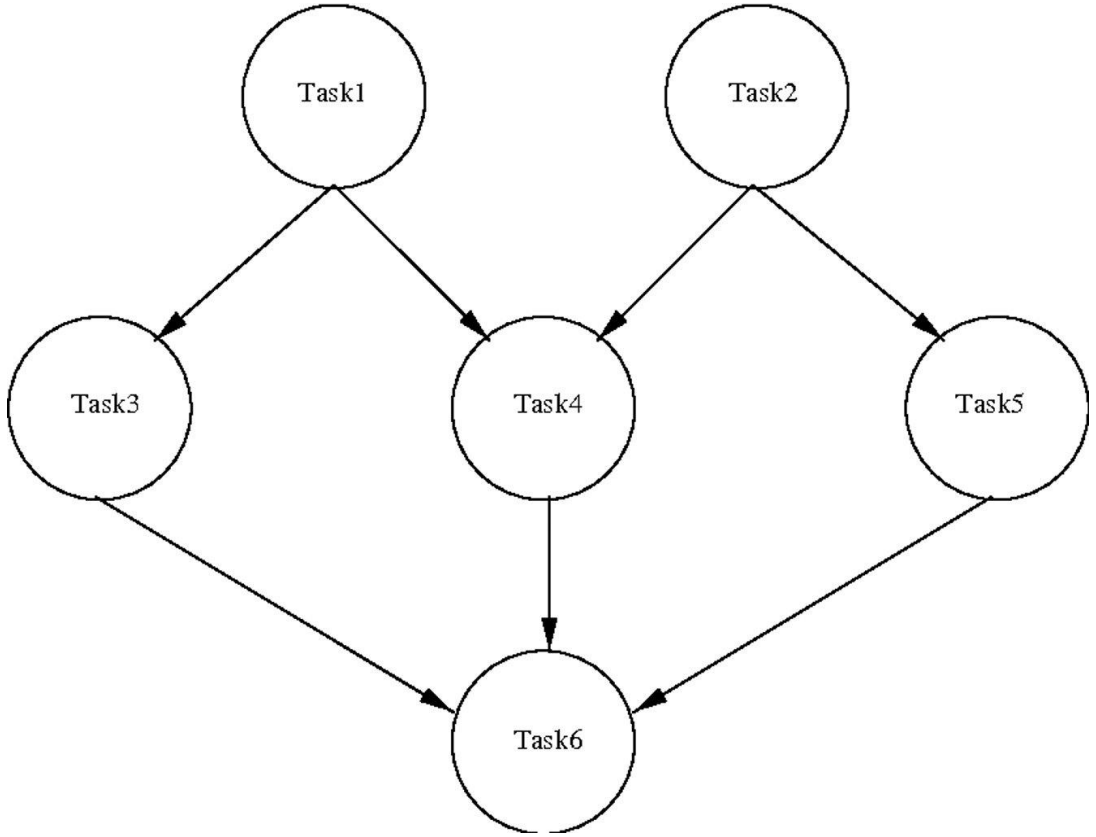
```
// We want to have result[i] = f(i) in the end
int n = 0;

#pragma omp parallel for num_threads(P) reduction(+:n)
  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**
- Dependencies can be on scalars or array variables.
- Sequential order of task creation and dependencies determine the order of execution. If t1 is created after t2, then
  - If t1 has **depend(in:x)** and t2 has **depend(in:x)**, both can be executed in any order
  - If t1 has **depend(in:x)** and t2 has **depend(out:x)**, t2 must wait for t1 to terminate
  - If t1 has **depend(out:x)** and t2 has **depend(in:x)** or **depend(out:x)**, then t2 must wait for t1 to terminate
- It is the programmer's responsibility to correctly choose task creation order and dependencies.

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

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

Example: Computing nine