Main Features of OpenMP

Computational Science II (CAAM 520)

Christopher Thiele

Rice University, Spring 2021

Parallel computation: an example

Suppose we want to integrate a function

$$f : [a, b] \to \mathbb{R}$$

numerically using the composite trapezoidal rule

$$\int_a^b f(x)dx \approx h \sum_{i=0}^{n-1} \frac{f(x_i) + f(x_{i+1})}{2},$$

where $x_0 = a$, $x_n = b$, and

$$x_{i+1} - x_i = h$$

for $i = 0, \ldots, n-1$.

Parallel computation: an example

Our goal is to implement a function `quad_trapezoidal()` that approximates the integral ***in parallel*** using OpenMP.

What signature should the function have?

Parallel computation: an example

Our goal is to implement a function `quad_trapezoidal()` that approximates the integral ***in parallel*** using OpenMP.

What signature should the function have?

```
double quad_trapezoidal(double (*f)(double),
                        double a,
                        double b,
                        int n);
```

→ Let us try to implement the function with our current knowledge of OpenMP.

Parallel computation: an example

As a reasonably challenging test, we will approximate

$$\int_0^{\frac{\pi}{2}} \sin(x) + x \, dx = \frac{\pi^2}{8} + 1.$$

Parallel computation: an example

Did our first attempt work?

No, we get random results.

$\longrightarrow$ There is a ***data race*** in the code!

If multiple threads modify the same variable (here: `sum`), their updates can interfere!

Shared and private variables

Variables in an OpenMP application can be ***shared*** between threads or ***private*** to each thread.

By default, variables are

- private if declared within a parallel region, and
- shared if declared before a parallel region.

```
int shared_var;

#pragma omp parallel
{
  int private_var;
}
```

Shared and private variables

Alternatively, variables can be declared as shared or private for a parallel region.

```
int shared_var, private_var;

#pragma omp parallel shared(shared_var) \
                     private(private_var)
{
  // ...
}
```

Note:

- The `shared` clause is redundant in this case.
- The `private` clause is necessary unless C99 is used.

Shared and private variables

Caution: If declared outside a parallel region, the value of a private variable is ***undefined*** inside the parallel region.

```
int private_var = 123;

#pragma omp parallel private(private_var)
{
  // Value of private_var is undefined!
}
```

→ Consider using `firstprivate` instead of `private`.

Shared and private variables

Caution: If declared outside a parallel loop, the value of a private variable is ***undefined*** after the parallel loop.

```
int private_var;

#pragma omp parallel private(private_var)
{
  // ...
}
// Value of private_var is undefined!
```

→ Consider using `lastprivate` instead of `private`.

Shared and private variables

Caution: The code below does ***not*** create a private array for each thread.

```
int *array = malloc(8*sizeof(int));

#pragma omp parallel firstprivate(array)
{
  // Each thread has its own private pointer
  // to the *same* array!
}
```

Data races

Data races can occur when a shared resource, e.g., a variable, is modified.

Data races can be hard to fix, because they can easily go unnoticed.

$\longrightarrow$ Whether the ***race condition*** occurs is random!

Data races

Example: Multiple threads write to a shared variable, causing a data race.

```
int sum = 0;

#pragma omp parallel
{
  sum += omp_get_thread_num();
}
```

$\longrightarrow$ Note that += involves both reading and writing!

Data races

To avoid data races, we must ensure that when a thread modifies a shared resource, no other thread reads from or writes to it concurrently.

$\rightarrow$ ***Mutual exclusion*** (mutex)

Locks

Mutexes are called ***locks*** in OpenMP:

```
omp_lock_t lock;
omp_init_lock(&lock);
#pragma omp parallel
{
  // ...
  omp_set_lock(&lock);
  // Only one thread can be have the lock
  // set at any given time.
  omp_unset_lock(&lock);
  // ...
}
omp_destroy_lock(&lock);
```

Deadlocks

When locks are used, ***deadlocks*** can occur!

```
void transfer(account_t a, account_t b, int amount)
{
  omp_set_lock(&a.lock);    // Lock account A.
  withdraw(a, amount);
  omp_set_lock(&b.lock);    // Lock account B.
  deposit(b, amount);
  omp_unset_lock(&b.lock); // Release account B.
  omp_unset_lock(&a.lock); // Release account A.
}
```

→ What happens if one thread calls `transfer(a, b, 100)` while another thread is processing `transfer(b, a, 50)`?

The `critical` directive

Locks/mutexes are cumbersome. Is there a simpler way?

Yes, the `critical` directive:

```
#pragma omp parallel
{
  // ...
  #pragma omp critical
  {
    // Only one thread can be inside the
    // critical block at any given time.
  }
  // ...
}
```

The `for` directive

Distributing loop iterations among threads is cumbersome.

Again, there is a simpler way to do it:

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < n; i++) {
    // The n loop iterations will be distributed
    // among threads automatically.
  }
}
```

The `for` directive

A parallel region which only contains a single for loop can be simplified as follows:

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  // The n loop iterations will be distributed
  // among threads automatically.
}
```

The `for` directive

The `for` directive is not only more convenient than manual distribution of loop iterations. It is also a generalization:

```
#pragma omp parallel for schedule(SCHEDULE)
for (int i = 0; i < n; i++) {
  // ...
}
```

Possible values for `SCHEDULE` are `static`, `dynamic`, `guided`, and `auto`.

The `for` directive

Caution: Not every loop can be parallelized, as dependencies between iterations can lead to data races.

```
#pragma omp parallel for
for (int i = 2; i < n; i++) {
  fibonacci[i] = fibonacci[i - 1]
               + fibonacci[i - 2];
}
```

→ The above code will compile without warning!

The `for` directive

Is there anything left to simplify?

Yes, results from all iterations of a parallel loop can be combined using the `reduction` clause:

```
#pragma omp parallel for reduction(OP:VAR)
for (int i = 0; i < n; i++) {
  // Each thread has a private instance of VAR.
  // At the end of the loop, all values of VAR
  // are combined using the operator OP.
}
```

Possible values for `OP` are `+`, `-`, `*`, `min`, `max`, `&`, `&&`, `|`, `||`, `^`.
Custom operators can also be defined.

The `for` directive

Example: Compute the Euclidean norm of a vector using reductions.

```
double sum;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < n; i++) {
  sum += x[i]*x[i];
}
return sqrt(sum);
```

Note: Each threads copy of the reduction variable is initialized with the neutral element of the reduction operator, e.g., zero for `+`, one for $*$, etc.

Barriers

If we need to synchronize all threads, we can use a ***barrier***:

No thread can get past the barrier before ***all*** threads have reached it.

```
#pragma omp parallel
{
  // ...

  // Wait for other threads.
  #pragma omp barrier

  // ...
}
```

Barriers

Example: Ensure that other threads have finished their work before we use their results.

```
#pragma omp parallel
{
  const int id = omp_get_thread_num();
  results[id] = do_work(id);

  #pragma omp barrier

  // Do something with results from other threads.
  do_more_work((id + 1)%omp_get_num_threads());
}
```

Barriers

The end of a parallel region or a parallel for loop is an ***implicit barrier***.

```
#pragma omp parallel
{
  #pragma omp for
  for (int i = 0; i < n; i++) {
    // ...
  } // Implicit barrier!

  // No thread gets here before all
  // threads have finished the loop.
}
```

Barriers

Barriers tend to make it easier to write correct code without data races, but they cause idling and synchronization.

$\longrightarrow$ Avoid unnecessary barriers to improve performance!

Barriers

Implicit barriers can be avoided using the `nowait` clause:

```
#pragma omp parallel for nowait
for (int i = 0; i < n; i++) {
  // ...
}

#pragma omp parallel for
for (int i = 0; i < n; i++) {
  // ...
}
```

→ ***Caution:*** This is a data race if the second loop depends on results from the first.

Barriers

If used improperly, barriers can also cause deadlocks.

```
#pragma omp parallel for
for (int i = 0; i < n; i++) {
  #pragma omp barrier
}
```

→ Causes a deadlock unless each thread performs exactly the same number of loop iterations.

The single and master directives

Use the single directive if part of a parallel region is to be executed by only one thread.

```
#pragma omp parallel
{
  #pragma omp single
  {
    shared_data = malloc(size);
  }

  #pragma omp barrier

  // ...
}
```

The `single` and `master` directives

If only the master thread should execute part of a parallel region, use the `master` directive.

```
#pragma omp parallel
{
  #pragma omp master
  {
    // Much like "single," but the single
    // thread which executes this block
    // must be the master thread.
  }
}
```

The `sections` directive

If multiple, independent blocks of code are to be executed in parallel, use sections.

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      // 1st block
    }
    #pragma omp section
    {
      // 2nd block
...
```