The parallel for directive

OpenMp provides the parallel for directive.

- We can use it to parallize something like this:

```
h = (b-a)/n;
approx = (f(a) + f(b))/2.0;
for (i = 1; i <= n-1; i++)
   approx += f(a + i*h);
approx = h*approx;
```

by simply placing a directive immediately before the **for** loop:

```
   h = (b-a)/n;
   approx = (f(a) + f(b))/2.0;
#  pragma omp parallel for num_threads(thread_count) \
      reduction(+: approx)
   for (i = 1; i <= n-1; i++)
      approx += f(a + i*h);
   approx = h*approx;
```

  - We highlight the for loop
- Like the parallel directive, the parallel for directive forks a team of threads to execute the following structured block.
  - However, the structured block following the parallel for directive must be a for loop
  - with the parallel for directive the system parallelizes the for loop by dividing the iterations of the loop among the threads
    - We can say that the <u>parallel for</u> directive is slightly different than the <u>parallel</u> directive because the parallel for directive splits up the iterations of the for loop to the threads.
- How is the work split up?

> In a **for** loop that has been parallelized with a `parallel` **for** directive, the default partitioning of the iterations among the threads is up to the system. However, most systems use roughly a block partitioning, that is, if there are m iterations, then roughly the first $m/$`thread_count` are assigned to thread 0, the next $m/$`thread_count` are assigned to thread 1, and so on.

- How does variable scope work in this directive handled by openMP

> However, speaking of scope, the default scope for all variables in a `parallel` directive is shared, but in our `parallel` **for** if the loop variable `i` were shared, the variable update, `i++`, would also be an unprotected critical section. Hence, in a loop that is parallelized with a `parallel` **for** directive the default scope of the loop variable is *private*; in our code, each thread in the team has its own copy of `i`.

Can we use a parallel for directive on every for loop? NO.

- First, OpenMP will only parallelize for loops—it won't parallelize while loops or do—while loops directly.
  - This may not seem to be too much of a limitation, since any code that uses a while loop or a do—while loop can be converted to equivalent code that uses a for loop instead.
  - However, OpenMP will only parallelize for loops for which the number of iterations can be determined:
    - • from the for statement itself (that is, the code for (. . . ; . . . ; . . .)), and
    - • prior to execution of the loop.
    - These are the only scenerios in which we can parallize a for loop
  - These wont work:

For example, the "infinite loop"

```
for ( ; ; ) {
   . . .
}
```

Infinite loop cannot be parallelized, since

```
for ( ; ; ) {
   . . .
}
```

cannot be parallelized. Similarly, the loop

```
for (i = 0; i < n; i++) {
   if ( . . . ) break;
   . . .
}
```

Infinite loop cannot be parallelized, since the number of iterations can't be determined from the for statement alone.

This for loop is also not a structured block, since the break adds another point of exit from the loop.

- openMP will only parallize for loops that are in canonical form, which take on one of the forms:

```
for ( index = start ;  index < end      ;  index++
                       index <= end        ++index
                       index >= end        index-
                       index > end         -index
                                           index += incr
                                           index -= incr
                                           index = index + incr
                                           index = incr + index
                                           index = index - incr )
```

Program 5.3: Legal forms for parallelizable **for** statements.

- The variable `index` must have integer or pointer type (e.g., it can't be a **float**).
- The expressions `start`, `end`, and `incr` must have a compatible type. For example, if `index` is a pointer, then `incr` must have integer type.
- The expressions `start`, `end`, and `incr` must not change during execution of the loop.
- During execution of the loop, the variable `index` can only be modified by the "increment expression" in the **for** statement.

These restrictions allow the run-time system to determine the number of iterations prior to execution of the loop. The sole exception to the rule that the run-time system must be able to determine the number of iterations prior to execution is that there can be a call to exit in the body of the loop

Data dependences

- If a for loop fails to satisfy one of the rules outlined in the preceding section, the compiler will simply reject it.
  - For example:

```
1    int Linear_search(int key, int A[], int n) {
2       int i;
3       /* thread_count is global */
4    #  pragma omp parallel for num_threads(thread_count)
5       for (i = 0; i < n; i++)
6          if (A[i] == key) return i;
7       return -1;  /* key not in list */
8    }
```

The `gcc` compiler reports:

```
Line 6: error: invalid exit from OpenMP structured block
```

- Another problem occurs in loops in which the computation in one iteration depends on the results of one or more previous iterations.
  - For example, Fibonacci numbers:

```
fibo[0] = fibo[1] = 1;
for (i = 2; i < n; i++)
   fibo[i] = fibo[i-1] + fibo[i-2];
```

Although we may be suspicious that something isn't quite right, let's try parallelizing the **for** loop with a `parallel` **for** directive:

```
   fibo[0] = fibo[1] = 1;
#  pragma omp parallel for num_threads(thread_count)
   for (i = 2; i < n; i++)
      fibo[i] = fibo[i-1] + fibo[i-2];
```

  - The compiler will execute without complaint
  - However, we will see that the results are unpredictable and not what we want for the most part

For example, on one of our systems (if we try using two threads to compute the first 10 Fibonacci numbers), we sometimes get

1 1 2 3 5 8 13 21 34 55,

which is correct. However, we also occasionally get

1 1 2 3 5 8 0 0 0 0.

- This issue occurs because of the fact the threads depend on previous threads for data and that is not good because of race conditions that can occur

- The takeaways

We see two important points here:

1. OpenMP compilers don't check for dependences among iterations in a loop that's being parallelized with a `parallel` **for** directive. It's up to us, the programmers, to identify these dependences.
2. A loop in which the results of one or more iterations depend on other iterations *cannot*, in general, be correctly parallelized by OpenMP without using features such as the Tasking API. (See Section 5.10).

Finding loop carried dependence

5.5.3 Finding loop-carried dependences

Perhaps the first thing to observe is that when we're attempting to use a `parallel` **for** directive, we only need to worry about loop-carried dependences. We don't need to worry about more general data dependences. For example, in the loop

```
1     for (i = 0; i < n; i++) {
2        x[i] = a + i*h;
3        y[i] = exp(x[i]);
4     }
```

there is a data dependence between Lines 2 and 3. However, there is no problem with the parallelization

```
1  #  pragma omp parallel for num_threads(thread_count)
2     for (i = 0; i < n; i++) {
3        x[i] = a + i*h;
4        y[i] = exp(x[i]);
5     }
```

since the computation of `x[i]` and its subsequent use will always be assigned to the same thread.

Also observe that at least one of the statements must write or update the variable in order for the statements to represent a dependence, so to detect a loop-carried dependence, we should only concern ourselves with variables that are updated by the loop body. That is, we should look for variables that are read or written in one iteration, and written in another. Let's look at a couple of examples.

Estimating π example showcasing the private directive

One way to get a numerical approximation to π is to use many terms in the formula[4]

$$\pi = 4\left[1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right] = 4\sum_{k=0}^{\infty}\frac{(-1)^k}{2k+1}.$$

We can implement this formula in serial code with

```
1        double factor = 1.0;
2        double sum = 0.0;
3        for (k = 0; k < n; k++) {
4           sum += factor/(2*k+1);
5           factor = -factor;
6        }
7        pi_approx = 4.0*sum;
```

- How can we parallelize this with OpenMP?
  - We might try to do:

```
1         double factor = 1.0;
2         double sum = 0.0;
3  #      pragma omp parallel for num_threads(thread_count) \
4             reduction(+:sum)
5         for (k = 0; k < n; k++) {
6            sum += factor/(2*k+1);
7            factor = -factor;
8         }
9         pi_approx = 4.0*sum;
```

- This would be wrong because line 6 and 7 would depend on if subsequent threads hold both k and k+1 and if they don't then we would see errors

- We can fix this by doing

$$\sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1}.$$

We see that in iteration k, the value of `factor` should be $(-1)^k$, which is $+1$ if k is even and -1 if k is odd, so if we replace the code

```
1           sum += factor/(2*k+1);
2           factor = -factor;
```

by

```
1           if (k % 2 == 0)
2              factor = 1.0;
3           else
4              factor = -1.0;
5           sum += factor/(2*k+1);
```

or, if you prefer the ?: operator,

```
1           factor = (k % 2 == 0) ? 1.0 : -1.0;
2           sum += factor/(2*k+1);
```

we will eliminate the loop dependence.

- However there is still something missing
  - Recall that in a block that has been parallelized by a parallel for directive, by default any variable declared before the loop—with the sole exception of the loop variable—is shared among the threads
  - Factor is a shared variable, there could be race conditions happening
  - We need to ensure each thread has its own copy of factor
  - We can do this by adding a private clause to the parallel for directive.

```
1         double sum = 0.0;
2  #      pragma omp parallel for num_threads(thread_count) \
3             reduction(+:sum) private(factor)
4         for (k = 0; k < n; k++) {
5            if (k % 2 == 0)
6               factor = 1.0;
7            else
8               factor = -1.0;
9            sum += factor/(2*k+1);
10        }
```

- The private clause specifies that for each variable listed inside the parentheses, a private copy is to be created for each thread
  - Thus, in our example, each of the thread_count threads will have its own copy of the variable factor, and hence the updates of one thread to factor won't affect the value of factor in another thread.

It's important to remember that the value of a variable with private scope is unspecified at the beginning of a `parallel` block or a `parallel` **for** block. Its value is also unspecified after completion of a `parallel` or `parallel` **for** block. So, for example, the output of the first `printf` statement in the following code is unspecified, since it prints the private variable `x` before it's explicitly initialized. Similarly, the output of the final `printf` is unspecified, since it prints `x` after the completion of the `parallel` block.

```
1     int x = 5;
2  #  pragma omp parallel num_threads(thread_count) \
3        private(x)
4     {
5        int my_rank = omp_get_thread_num();
6        printf("Thread %d > before initialization, x = %d\n",
7              my_rank, x);
8        x = 2*my_rank + 2;
```



```
9        printf("Thread %d > after initialization, x = %d\n",
10             my_rank, x);
11    }
12    printf("After parallel block, x = %d\n", x);
```

More on variable scope

5.5.5 More on scope

Our problem with the variable `factor` is a common one. We usually need to think about the scope of each variable in a `parallel` block or a `parallel` **for** block. Therefore, rather than letting OpenMP decide on the scope of each variable, it's a very good practice for us as programmers to specify the scope of each variable in a block. In fact, OpenMP provides a clause that will explicitly require us to do this: the **default** clause. If we add the clause

```
default(none)
```

to our `parallel` or `parallel` **for** directive, then the compiler will require that we specify the scope of each variable we use in the block and that has been declared outside the block. (Variables that are declared within the block are always private, since they are allocated on the thread's stack.)

For example, using a **default**(none) clause, our calculation of π could be written as follows:

```
        double sum = 0.0;
#       pragma omp parallel for num_threads(thread_count) \
           default(none) reduction(+:sum) private(k, factor) \
           shared(n)
        for (k = 0; k < n; k++) {
           if (k % 2 == 0)
              factor = 1.0;
           else
              factor = -1.0;
           sum += factor/(2*k+1);
        }
```

In this example, we use four variables in the **for** loop. With the default clause, we need to specify the scope of each. As we've already noted, `sum` is a reduction variable (which has properties of both private and shared scope). We've also already noted that `factor` and the loop variable `k` should have private scope. Variables that are never updated in the `parallel` or `parallel` **for** block, such as `n` in this example, can be safely shared. Recall that unlike `private` variables, `shared` variables have the same value in the `parallel` or `parallel` **for** block that they had before the block, and their value after the block is the same as their last value in the block. Thus if `n` were initialized before the block to 1000, it would retain this value in the `parallel` **for** statement, and since the value isn't changed in the **for** loop, it would retain this value after the loop has completed.

- The takeaways here are..
    - We can dd the <u>default clause</u> to make it so that we have to explicitly tell openMP the scope of our variables or else we get an error thrown at us