Data dependences

- In loops in which the computation in one iteration depends on the results of one or more previous iterations. As an example, consider the following code, which computes the first ***n***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;
  # pragmaomp parallel for num threads(thread count)
  for (i = 2; i < n; i++)
  fibo[i] = fibo[i-1] + fibo[i-2];
  ```

- The compiler will create an executable without complaint. However, if we try running it with more than one thread, we may find that the results are, at best, unpredictable.
- 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.

- It appears that the run-time system assigned the computation of ***fibo[2], fibo[3], fibo[4],*** **and*****fibo[5]*** to one thread, while ***fibo[6], fibo[7],fibo[8],*** **and*****fibo[9]*** were assigned to the other. (the loop starts withi = 2.) In some runs of the program, everything is fine because the thread that wasassigned ***fibo[2], fibo[3], fibo[4], andfibo[5]*** finishes its computations beforethe other thread starts. However, in other runs, the first thread has evidently notcomputed ***fibo[4] and fibo[5]*** when the second computes

fibo[6]. It appears thatthe system has initialized the entries in fibo to 0, and the second thread is using thevalues ***fibo[4] = 0 and fibo[5] = 0*** to compute fibo[6]. It then goes on to use***fibo[5] = 0 and fibo[6] = 0*** to compute ***fibo[7],*** and so on.

We see two important points here:
1. OpenMP compilers don't check for dependences among iterations in a loop that'sbeing 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 iterationscannot, in general, be correctly parallelized by OpenMP.

Finding loop-carried dependences

- when we're attempting to use a parallel for directive, we only need to worry about loop-carried dependences

```
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 withthe parallelization

```
1     # pragma omp parallel for numthreads(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 thesame thread.Also observe that at least one of the statements must write or update the variablein order for the statements to represent a dependence, so in order to detect a loopcarrieddependence, we should only concern ourselves with variables that are updatedby the loop body.

Estimating π :

- One way to get a numerical approximation to **π** is to use many terms in the formula

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

- 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.  piapprox = 4.0*sum;
```

- We can parallelize this with OpenMP.

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

- 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 theloop variable—is shared among the threads. So ***factor*** is shared and, for example,thread 0 might assign it the value 1, but before it can use this value in the updateto ***sum***, thread 1 could assign it the value -1. Therefore, in addition to eliminatingthe loop-carried dependence in the calculation of ***factor***, we need to insure thateach thread has its own copy of ***factor***. That is, in order to make our code correct,we need to

also insure that ***factor*** has private scope. We can do this by adding aprivate 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 anotherthread.