SCHEDULING LOOPS:

- Most OpenMP implementations use roughly a block partitioning: if there are n iterations in the serial loop, then in the parallel loop the first n/thread_count are assigned to thread 0, the next n/thread_count are assigned to thread 1, and so on. It's not difficult to think of situations in which this assignment of iterations to threads would be less than optimal

- For example, suppose we want to parallelize the loop

```
sum = 0.0;
for (i = 0; i <= n; i++)
sum += f(i);
```

- Also suppose that the time required by the call to f is proportional to the size of theargument i. Then a block partitioning of the iterations will assign much more work to thread thread_count-1 than it will assign to thread 0. A better assignment of work to threads might be obtained with a cyclic partitioning of the iterations among the threads. In a cyclic partitioning, the iterations are assigned, one at a time, in a "round-robin" fashion to the threads. Suppose ***t = thread_ count***. Then a cyclic partitioning will assign the iterations as follows:

Thread	Iterations
0	0, n/t, 2n/t, ...
1	1, n/t+1, 2n/t+1, ...
⋮	⋮
t−1	t−1, n/t+t−1, 2n/t+t−1, ...

A program in which we defined

```
double f(int i) {
int j, start = i_(i+1)/2, finish = start + i;
double return val = 0.0;
for (j = start; j <= finish; j++) {
returnval += sin(j);
}
returnreturnval;
}   /* f */
```

- The call f (i) calls the sine function i times, and, for example, the time to execute f(2i) requires approximately twice as much time as the time to execute f (i). When we ran the program with n = 10,000 and one thread, the run-time was 3.67seconds. When we ran the program with two threads and the default assignment—iterations 0–5000 on thread 0 and iterations 5001–10,000 on thread 1—the run-timewas 2.76 seconds. This is a speedup of only 1.33.

The *schedule*clause :

- In our example, we already know how to obtain the default schedule: we just add a ***parallel*** for directive with a ***reduction*** clause:

```
sum = 0.0;
# pragmaomp parallel for num_threads(thread_count)\
reduction(+:sum);
for (i = 0; i <= n; i++)
sum += f(i);
```

- To get a cyclic schedule, we can add a schedule clause to the ***parallel*** for directive:

```
sum = 0.0;
# pragmaomp parallel for num threads(thread count) n
reduction(+:sum) schedule(static,1)
for (i = 0; i <= n; i++)
sum += f(i);
```

  In general, the schedule clause has the form
  schedule(<type> [, <chunksize>])

The type can be any one of the following:

- ***static.*** The iterations can be assigned to the threads before the loop is executed.
- ***dynamic or guided***. The iterations are assigned to the threads while the loop is executing, so after a thread completes its current set of iterations, it can request more from the run-time system.
- ***auto***. The compiler and/or the run-time system determine the schedule.
- ***runtime***. The schedule is determined at run-time.

The *static* schedule type

- For a ***static*** schedule, the system assigns chunks of ***chunksize*** iterations to eachthread in a round-robin fashion. As an example, suppose we have 12 iterations,0, 1, : : : ,11, and three threads. Then if ***schedule(static,1)*** is used in the ***parallelfor*** or ***for*** directive, we've already seen that the iterations will be assigned as

  Thread 0: 0, 3, 6,9
  Thread 1: 1, 4, 7,10
  Thread 2: 2, 5, 8,11

- If schedule(static,2) is used, then the iterations will be assigned as
  Thread 0: 0, 1, 6,7
  Thread 1: 2, 3, 8,9
  Thread 2: 4, 5, 10,11

- If schedule(static,4) is used, the iterations will be assigned as

Thread 0: 0, 1, 2,3
Thread 1: 4, 5, 6,7
Thread 2: 8, 9, 10,11

- Thus the clause ***schedule(static, total iterations/thread_count)*** is more or less equivalent to the default schedule used by most implementations of OpenMP.
  The ***chunksize*** can be omitted. If it is omitted, the ***chunksize*** is approximately
  total iterations/thread_count.

The *dynamic* and *guided* schedule types

- In a ***dynamic*** schedule, the iterations are also broken up into chunks of ***chunksize*** consecutive iterations. Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from the run-time system. This continues until all the iterations are completed. The chunksize can be omitted. When it is omitted, a ***chunksize*** of 1 is used.

- In a ***guided*** schedule, each thread also executes a chunk, and when a thread finishes a chunk, it requests another one. In a guided schedule, as chunks are completed, the size of the new chunks decreases. Ex: if we run the trapezoidal rule program with the ***parallel for*** directive and a ***schedule(guided)*** clause, then when n =10,000 and thread count = 2, the iterations are assigned as shown in Table 5.3. We see that the size of the chunk is approximately the number of iterations remaining divided by the number of threads.
- The first chunk has size 9999/2 ≈ 5000, since there are 9999 unassigned iterations. The second chunk has size 4999/2≈ 2500, and so on.
  In a ***guided*** schedule, if no chunksize is specified, the size of the chunks decreases down to 1. If chunksize is specified, it decreases down to chunksize,
  with the exception that the very last chunk can be smaller than chunksize.

The *runtime* schedule type

- To understand schedule(runtime) we need to digress for a moment and talk aboutenvironment variables. As the name suggests, environment variables are namedvalues that can be accessed by a running program. That is, they're available in theprogram's environment. Some commonly used environment variables are ***PATH, HOME***,and ***SHELL***. The ***PATH*** variable specifies which directories the shell should searchwhen it's looking for an executable. It's usually defined in both Unix and Windows.
- The ***HOME*** variable specifies the location of the user's home directory, and the ***SHELL*** variable specifies the location of the executable for the user's shell. Theseare usually defined in Unix systems. In both Unix-like systems (e.g., Linux andMac OS X) and Windows, environment variables can be

examined and specifiedon the command line. In Unix-like systems, you can use the shell’s command line.

- In Windows systems, you can use the command line in an integrated evelopment environment.
- As an example, if we’re using the bash shell, we can examine the value of an environment variable by typing

```
$ echo $PATH
```

and we can use the export command to set the value of an environment variable

```
$ export TEST VAR="hello"
```