Scheduling loops

When we first encountered the parallel for directive, we saw that the exact assignment of loop iterations to threads is system dependent.

- Most openMP implementations use a block partitioning to divide the amount of iterations given to each thread
  - 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.
  - This way of dividing up the work to each thread isnt always the best/most optimal way of diving up the iterations
  - It could be that the first thread will have less work to do than the last thread (aka the last thread does more work than any other thread)
- A better assignment of work to threads might be obtained with a cyclic partitioning of the iterations among the threads
  - Here by default, we give each thread an iteration one at a time until there are no more iterations to give
  - There are many situatiuons where a cyclic partition can be significantly better performance wise than block

The schedule clause

- OpenMP provides us with a schedule clause that can help us control how we want the work to be partitioned among the threads
- we already know how to obtain the default schedule: we just add a parallel for directive with a reduction clause:

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

- To get a cyclic schedule we would do:

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

```
      sum = 0.0;
#     pragma omp parallel for num_threads(thread_count) \
         reduction(+:sum) schedule(static,1)
      for (i = 0; i <= n; i++)
         sum += f(i);
```

In general, the `schedule` clause has the form

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 based on an environment variable (more on this later).
- The chunk size is a positive integer. In OpenMP parlance, a chunk of iterations is a block of iterations that would be executed consecutively in the serial loop.
  - The number of iterations in the block is the chunksize.
  - Only static, dynamic, and guided schedules can have a chunksize.

The static schedule type

For a `static` schedule, the system assigns chunks of `chunksize` iterations to each thread in a round-robin fashion. As an example, suppose we have 12 iterations, $0, 1, \ldots, 11$, and three threads. Then if `schedule(`**static**`, 1)` is used, in the `parallel` **for** 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

The default schedule is defined by your particular implementation of OpenMP, but in most cases it is equivalent to the clause

```
schedule(static, total_iterations / thread_count)
```

It is also worth noting that the `chunksize` can be omitted. If omitted, the `chunksize` is approximately `total_iterations / thread_count`.

The `static` schedule is a good choice when each loop iteration takes roughly the same amount of time to compute. It also has the advantage that threads in subsequent loops with the same number of iterations will be assigned to the same ranges; this can improve the speed of memory accesses, particularly on NUMA systems (see Chapter 2).

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.

The primary difference between `static` and `dynamic` schedules is that the `dynamic` schedule assigns ranges to threads on a first-come, first-served basis. This can be advantageous if loop iterations do not take a uniform amount of time to compute (some

Table 5.4 Assignment of trapezoidal rule iterations 1–9999 using a guided schedule with two threads.

Thread	Chunk	Size of Chunk	Remaining Iterations
0	1 – 5000	5000	4999
1	5001 – 7500	2500	2499
1	7501 – 8750	1250	1249
1	8751 – 9375	625	624
0	9376 – 9687	312	312
1	9688 – 9843	156	156
0	9844 – 9921	78	78
1	9922 – 9960	39	39
1	9961 – 9980	20	19
1	9981 – 9990	10	9
1	9991 – 9995	5	4
0	9996 – 9997	2	2
1	9998 – 9998	1	1
0	9999 – 9999	1	0

algorithms are more compute-intensive in later iterations, for instance). However, since the ranges are not allocated ahead of time, there is some overhead associated with assigning them dynamically at run-time. Increasing the chunk size strikes a balance between the performance characteristics of `static` and `dynamic` scheduling; with larger chunk sizes, fewer dynamic assignments will be made.

The `guided` schedule is similar to `dynamic` in that each thread also executes a chunk and requests another one when it's finished. However, in a `guided` schedule, as chunks are completed, the size of the new chunks decreases. For example, on one of our systems, 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 5.4. 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 $9999/2 \approx 5000$, since there are 9999 unassigned iterations. The second chunk has size $4999/2 \approx 2500$, 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 `guided` schedule can improve the balance of load across threads when later iterations are more compute-intensive.

The runtime schedule type

To understand `schedule(runtime)`, we need to digress for a moment and talk about **environment variables**. As the name suggests, environment variables are named values that can be accessed by a running program. That is, they're available in the program's *environment*. Some commonly used environment variables are `PATH`, `HOME`, and `SHELL`. The `PATH` variable specifies which directories the shell should search when it's looking for an executable and is 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. These are usually defined in Unix systems. In both Unix-like systems (e.g., Linux and macOS) and Windows, environment variables can be examined and specified on 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 development environment.

As an example, if we're using the bash shell (one of the most common Unix shells), 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"
```

Note:
This is a really good way to schedule your program if you want to test schedule types and what not to see which one performs the best through the command line instead of changing your program each time and having to compile and run to test

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

```
$ export TEST_VAR="hello"
```

These commands also work on `ksh`, `sh`, and `zsh`. For details about how to examine and set environment variables for your particular system, check the `man` pages for your shell, or consult with your system administrator or local expert.

When `schedule(runtime)` is specified, the system uses the environment variable `OMP_SCHEDULE` to determine at run-time how to schedule the loop. The `OMP_SCHEDULE` environment variable can take on any of the values that can be used for a static, dynamic, or guided schedule. For example, suppose we have a `parallel` **for** directive in a program and it has been modified by `schedule(runtime)`. Then if we use the bash shell, we can get a cyclic assignment of iterations to threads by executing the command

```
$ export OMP_SCHEDULE="static,1"
```

Now, when we start executing our program, the system will schedule the iterations of the **for** loop as if we had the clause `schedule(static,1)` modifying the `parallel` **for** directive. This can be very useful for testing a variety of scheduling configurations.

The following bash shell script demonstrates how one might take advantage of this environment variable to test a range of schedules and chunk sizes. It runs a matrix-vector multiplication program that has a `parallel` **for** directive with the `schedule(runtime)` clause.

```
#!/usr/bin/env bash

declare -a schedules=("static" "dynamic" "guided")
declare -a chunk_sizes=("" 1000 100 10 1)

for schedule in "${schedules[@]}"; do
   echo "Schedule: ${schedule}"
   for chunk_size in "${chunk_sizes[@]}"; do
       echo "  Chunk Size: ${chunk_size}"
       sched_param="${schedule}"

       if [[ "${chunk_size}" != "" ]]; then
           # A blank string indicates we want
           # the default chunk size
           sched_param="${schedule},${chunk_size}"
       fi

       # Run the program with OMP_SCHEDULE set:
       OMP_SCHEDULE="${sched_param}" ./omp_mat_vect 4 2500 2500
   done
   echo
done
```

Which schedule is best?

If we have a **for** loop that we're able to parallelize, how do we decide which type of schedule we should use and what the `chunksize` should be? As you may have guessed, there *is* some overhead associated with the use of a `schedule` clause. Furthermore, the overhead is greater for `dynamic` schedules than `static` schedules, and the overhead associated with `guided` schedules is the greatest of the three. Thus if we're getting satisfactory performance without a `schedule` clause, we should go no further. However, if we suspect that the performance of the default schedule can be substantially improved, we should probably experiment with some different schedules.

In the example at the beginning of this section, when we switched from the default schedule to `schedule(`**static** `,1)`, the speedup of the two-threaded execution of the program increased from 1.33 to 1.99. Since it's *extremely* unlikely that we'll get speedups that are significantly better than 1.99, we can just stop here, at least if we're only going to use two threads with 10,000 iterations. If we're going to be using varying numbers of threads and varying numbers of iterations, we need to do more experimentation, and it's entirely possible that we'll find that the optimal schedule depends on both the number of threads and the number of iterations.

It can also happen that we'll decide that the performance of the default schedule isn't very good, and we'll proceed to search through a large array of schedules and iteration counts only to conclude that our loop doesn't parallelize very well and *no* schedule is going to give us much improved performance. For an example of this, see Programming Assignment 5.4.

There are some situations in which it's a good idea to explore some schedules before others:

- If each iteration of the loop requires roughly the same amount of computation, then it's likely that the default distribution will give the best performance.
- If the cost of the iterations decreases (or increases) linearly as the loop executes, then a `static` schedule with small chunksizes will probably give the best performance.
- If the cost of each iteration can't be determined in advance, then it may make sense to explore a variety of scheduling options. The `schedule(runtime)` clause can be used here, and the different options can be explored by running the program with different assignments to the environment variable `OMP_SCHEDULE`.