1 Background

This document introduces OpenMP (Open Multi-Processing), a framework for writing multithreaded parallel C programs. OpenMP has an emphasis on encouraging efficient memory sharing between threads on the same machine. Thus, the degree of parallelism available to an OpenMP program is generally limited to the number of cores on one machine.

The following assumes familiarity with using a terminal, as well as the ability to connect to the SECS Linux servers from a terminal using SSH.

Note: The SECS server `ringo.secs.oakland.edu` should be used for this assignment, which relies on a version of OpenMP that is only implemented in more recent versions of the `gcc` compiler. SECS servers other than Ringo may not have an up-to-date version of `gcc` installed.

2 Hello World in OpenMP

This section will present the simple "Hello World" program `omp-hello-world.c`, which uses the OpenMP interface. This program demonstrates two functions which are a fundamental part of the OpenMP API:

- `omp_get_thread_num()` returns the calling thread's unique thread ID.
- `omp_get_num_threads()` returns the total number of allocated threads at the time the function is called.

This program also demonstrates the use of OpenMP *compiler directives*, which usually represent most of the functionality in an OpenMP program. Compiler directives are used to parallelize *structured blocks* in an OpenMP program, where a structured block is any block of code with exactly one entry point and one exit point. An OpenMP compiler directive has the syntax:

```
#pragma omp [construct] [clauses]
```

The value of `construct` describes the behavior of the directive. Different constructs are analagous to the different functions in a traditional API. Likewise, the directive's `clauses` modify the behavior of the construct, analagous to a function's arguments.

The compiler directive used in the example program invokes the `parallel` construct with the following statement:

```
#pragma omp parallel private(tid)
```

The `parallel` construct creates a number of threads, all of which immediately begin executing the structured block under the directive in parallel. There are several ways to set the number of threads created, but the two most common ways are the following:

1. The `num_threads` clause controls the number of created threads if present. For example, the following directive would always create 4 threads: `#pragma omp parallel num_threads(4)`

2. If no `num_threads` clause is present, the terminal environment variable `OMP_NUM_THREADS` controls the number of created threads.

Since the example program doesn't have a `num_threads` clause, the `OMP_NUM_THREADS` environment variable can be used to set the number of threads.

The compiler directive in `omp-hello-world.c` also uses the `private(tid)` clause, which allocates private copies of the variable `tid` for each thread created by the directive. Without creating private copies of `tid`, the program would have a race condition, since the shared value of `tid` could be overwritten before a thread has time to print its own value. In practice, this program is small enough that this race condition is unlikely to occur, so the use of the `private` clause is mainly to demonstrate best practices: whenever each thread needs its own copy of a variable, that variable should be marked private.

Exercise 1. Compile the `omp-hello-world.c` program using the `gcc` compiler, then execute the result. The `-fopenmp` argument needs to be passed to `gcc` to compile an OpenMP program:

```
nfireman@yoko:~/openmp$ gcc -fopenmp omp-hello-world.c -o omp-hello-world
nfireman@yoko:~/openmp$ ./omp-hello-world
Hello World from thread 1
Hello World from thread 3
8 threads forked in total.
Hello World from thread 0
Hello World from thread 4
Hello World from thread 5
Hello World from thread 7
Hello World from thread 6
Hello World from thread 2
```

Note that the number of threads created isn't being explicitly set, and will vary depending on the environment default value.

Exercise 2. Explicitly set the environment variable `OMP_NUM_THREADS` to 4 using the `export` terminal command, then execute `omp-hello-world` again.

```
nfireman@yoko:~/openmp$ export OMP_NUM_THREADS=4
nfireman@yoko:~/openmp$ ./omp-hello-world
4 threads forked in total.
Hello World from thread 0
Hello World from thread 2
Hello World from thread 3
Hello World from thread 1
```

Figure 1: Selection sort. A bold number indicates the minimum unsorted value.
Source: https://courses.cs.washington.edu/courses/cse373/19au/lectures/05/reading/

Additional resources

- Slides from an introductory OpenMP presentation:
  `https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf`
  (Local copy saved to `resources/omp-hands-on-SC08.pdf`)
- An online OpenMP tutorial published by the Lawrence Livermore National Laboratory:
  `https://hpc.llnl.gov/openmp-tutorial`
  (Local copy saved to `resources/OpenMP_Tutorial.pdf`)

3 Selection sort

This section presents a selection sort algorithm that has been parallelized using OpenMP. The selection sort algorithm is demonstrated in Figure 1. At the beginning of the algorithm the entire array is assumed to be unsorted, and

selection sort proceeds to sort the array one element at a time from left to right. At each step, the unsorted section is searched to find its minimum element, and that minimum is swapped to the beginning of the unsorted section, where it will be correctly sorted. This process continues until the entire array is sorted.

The algorithm can be understood as two nested for loops, where the outer loop iterates over the elements of the array, and the inner loop finds the minimum element of the unsorted section. Assuming the input array is named `data` and has length `N`, we have the following code:

```
for (int i = 0; i < N; i++) {
    // initialize the minimum to the first unsorted element
    int min_index = i;
    int min_value = data[i];

    // search the remaining unsorted portion for its minimum
    for (int j = i + 1; j < N; j++) {
        if (data[j] < min_value) {
            min_index = j;
            min_value = data[j];
        }
    }

    // swap the minimum value to the front of the unsorted section
    swap(&data[i], &data[min_index]);
}
```

When trying to parallelize this algorithm, we first note that the outer `for` loop cannot be parallelized. This is because the array is changed at the end of each outer loop iteration (via the swap), meaning that a given outer loop iteration cannot begin until the previous iteration has ended.

The inner `for` loop, however, only searches a subarray for its minimum element, which is an operation that can be parallelized. We will assign multiple threads to search part of the subarray for that part's minimum value. Then, the minimum value of the entire subarray can be computed as the smallest among the minima found by each thread. This can be implemented as a *reduction*, an OpenMP clause used to combine one value computed by each thread into a single value.

As a brief summary of reductions, the following program computes the sum of the values in the array `data`, storing it in the variable `sum`:

```
int sum = 0;
#pragma omp parallel for reduction(+:sum)
for (int i = 0; i < N; i++) {
    sum += data[i]
}
```

The clause `reduction(+:sum)` tells OpenMP to compute the sum (indicated by `+`) of each thread's partial result stored in the `sum` variable. Notice that if the compiler directive were removed, the remaining program would still compute the sum serially. This demonstrates a very useful feature of the design of OpenMP: often, a serial program can be parallelized solely through compiler directives, without the programmer needing to modify the original program's logic.

Returning to selection sort, we could make use of the built-in `min` reduction operation, which could be used to compute the minimum unsorted value as desired:

```
#pragma omp parallel for reduction(min:min_value)
for (int j = i + 1; j < n; j++) {
  if (data[j] < min_value) {
    min_value = data[j];
  }
}
```

The issue with this approach is that we no longer record the index where `min_value` was found. Recall that the index of the minimum value is necessary, since after locating the minimum, we then need to swap it to the beginning of the unsorted section of the array.

One solution to this problem is to implement a *user-defined reduction* operation which computes the minimum value while also recording the index where that value was found. First, we define a `struct` that encodes both a value and its index in the array:

```
typedef struct entry {
  int index;
  int value;
} entry;
```

Next, we declare a reduction operation that computes the minimum of two `entry` instances by comparing their `value` fields:

```
#pragma omp declare reduction(min_entry : entry : \
  omp_out = (omp_in.value < omp_out.value) ? omp_in : omp_out)
```

The above `declare reduction` directive has three colon-separated arguments, described as follows. Note that the `\` character on the first line escapes the line break, allowing the declaration to span two lines for clarity.

- The first argument `min_entry` is the name of the declared reduction operation.
- The second argument `entry` is the data type of the values being combined (the `struct` defined above).

- The third argument is an expression that describes how OpenMP should combine two values of the type `entry`. By convention, the two values have the names `omp_in` and `omp_out`, and the combined value must be written to `omp_out`.

The expression given in the third argument compares the `value` fields of the two structs, and writes the struct with the smaller `value` field to `omp_out`. When a sequence of `entry` values are combined in this way, the final result will thus contain the minimum out of all `value` fields.

With this reduction operation defined, we can simply add a corresponding compiler directive to the inner loop discussed above, and OpenMP will parallelize it as desired:

```
entry iter_min;
#pragma omp parallel for reduction(min_entry:iter_min)
for (int j = i + 1; j < N; j++) {
  if (data[j] < iter_min.value) {
    iter_min.value = data[j];
    iter_min.index = j;
  }
}
```

Exercise 3. Compile and execute `omp-selection-sort.c` with `OMP_NUM_THREADS` set to `1`. You may need to use the `-static` argument when compiling with `gcc`:

```
nfireman@ringo:~/openmp$ export OMP_NUM_THREADS=1
nfireman@ringo:~/openmp$ gcc -static -fopenmp omp-selection-sort.c -o sort
/usr/lib/../lib64/libpthread.a(libpthread.o): In function `sem_open':
(.text+0x77cd): warning: the use of `mktemp' is dangerous, better use `mkstemp'
nfireman@ringo:~/openmp$ ./sort
Result valid: yes
Values sorted: 10000
Duration: 0.103213 seconds
```

Exercise 4. Re-execute the sorting program with 2, 3, and 4 threads. Which amount of threads yields the fastest execution?

Additional resources

- The section `omp declare reduction` from the "Intel C++ Compiler 16.0 User and Reference Guide":

  `https://scc.ustc.edu.cn/zlsc/tc4600/intel/2016.0.109/compiler_c/common/core/GUID-7312910C-D175-4544-99C5-29C12D980744.htm`

  (Local copy saved to `resources/omp-declare-reduction.pdf`)

4 OpenMP Tasks

This section will introduce the concept of *tasks* in OpenMP. At a high level, a task is just a block of code that needs to be executed by a single thread, as well as the values of variables that the thread will need to execute the task's code. For example, in a parallelized `for` loop, each iteration of the loop would be identified as a separate task. Separating each iteration into a separate task is what allows OpenMP to distribute work among many threads in parallel.

In early versions of OpenMP, tasks were only an implementation detail, and not directly accessible to OpenMP programmers. Compiler directives such as `#pragma omp parallel` would create tasks under the hood, but a single task could not be explicitly created. In OpenMP version 3.0, the `#pragma omp task` directive exposed tasks as part of the OpenMP API.

Through the explicit use of tasks, OpenMP programs are able to share work between threads in more complicated ways than by only using the `#pragma omp parallel` directive. For example, the latter does not allow the parallelization of a `while` loop or a recursive function; tasks allow both.

To demonstrate the use of tasks, the program `omp-pairwise-sum.c` has been provided, which implements a parallel version of the *pairwise summation* algorithm. This algorithm sums an array of floating-point numbers using a divide-and-conquer approach: the left and right halves of the array are summed recursively, and the sum of the two partial results is the sum of the entire array. In practice, this algorithm has applications when summing arrays of floating-point numbers, since it has been shown to be less vulnerable to round-off error than naively summing the array in order.

The program uses the `#pragma omp task` directive in the `parallel_sum` function to parallelize the recursive logic described above. Both recursive calls are placed in `#pragma omp task` directives, which allows OpenMP to schedule them in parallel with other recursive calls when possible. Note that since each task is only one statement, the curly braces defining the block can be omitted; this is analagous to when curly braces can be omitted from the body of a `for` loop.

```
// data: the array of input numbers
// count: the number of elements to be added
int parallel_sum(int *data, int count) {
  if (count == 1) return *data;

  int left_sum, right_sum;
  int middle = count / 2;

  #pragma omp task shared(left_sum)
  left_sum = parallel_sum(data, middle);

  #pragma omp task shared(right_sum)
  right_sum = parallel_sum(data + middle, count - middle);
```

```
  #pragma omp taskwait
  return left_sum + right_sum;
}
```

By default, variables in a task block are private to that block. Therefore, in order to make sure the values of `left_sum` and `right_sum` are preserved from each task block to the final line of the function, the `shared` clauses must be added to each task directive. In general, more than one variable can be shared from a task block using a comma-separated list:

```
#pragma omp task shared(var0, var1, var2)
{
  // The three variables ‘var0‘, ‘var1‘, and ‘var2‘ are shared
}
```

While the above program works correctly, its performance is poor as a result of its very high degree of parallelism. In fact, even when `count` is `2` (meaning only two numbers from `data` need to be added), the function will still spawn two OpenMP tasks and attempt to compute the sum "in parallel", which is much slower than simply adding the two numbers in a single thread.

Thus, we can vastly improve the performance of the above function by introducing a *cutoff* array length, where input arrays smaller than the cutoff length will be added serially instead of resorting to task-based recursion. This cutoff value is named `MIN_PARALLEL_COUNT` in the provided program, and serial summation is implemented in the function `serial_sum`.

Finally, we turn to how the `parallel_sum` function is called from `main`. As before, since the body of the directives is a single statement, the curly braces can be omitted, even though there are now two nested directives:

```
#pragma omp parallel
#pragma omp single
sum = parallel_sum(data, N);
```

In order to indicate to OpenMP that the `parallel_sum` function should be executed with multiple threads, it must be enclosed in a `#pragma omp parallel` directive, as we have seen before. But by default, this would execute `parallel_sum` once for each thread that is available to the directive. Since `parallel_sum` should only be executed once, we also use the `#pragma omp single` directive to guarantee that only one thread will actually execute the line of code calling `parallel_sum` (usually, the first thread to read the `single` directive).

Exercise 5. Compile and execute `omp-pairwise-sum.c` with 1, 2, 3, and 4 threads available. Record the execution times in each case. You may need to use the `-static` argument when compiling with `gcc`.

```
nfireman@ringo:~/openmp$ gcc omp-pairwise-sum.c -fopenmp -static -o omp-pairwise-sum
/usr/lib/../lib64/libpthread.a(libpthread.o): In function `sem_open':
(.text+0x77cd): warning: the use of `mktemp' is dangerous, better use `mkstemp'
nfireman@ringo:~/openmp$ export OMP_NUM_THREADS=4
nfireman@ringo:~/openmp$ ./omp-pairwise-sum
Invocations per thread number: { 57287, 56008, 35782, 55623 }
Result sum: 2002318
Duration: 0.134740 seconds (100 trials)
```

Exercise 6. Currently, the cutoff value `MIN_PARALLEL_COUNT` is 1000. With 4 threads available, record the execution durations when the cutoff value is 250 and 4000, and compare the results to Exercise 5. Try several other values in attempt to further improve the execution duration.

5 Linked list traversal

This section will demonstrate a linked list traversal algorithm, which will highlight another use case of OpenMP tasks. If the length of a linked list is not known in advance, then traversing it will require a loop with a variable number of iterations. Such a loop cannot be parallelized via OpenMP without the use of tasks.

The provided program `omp-linked-list.c` demonstrates the parallelized traversal algorithm. This program uses the following data type for the nodes in our linked list, which contains an `int` field representing that node's data, as well as a field `next` pointing to the next node in the list:

```
typedef struct node {
  int data;
  struct node* next;
} node;
```

The function `construct_linked_list` creates a linked list (serially, without the use of OpenMP) of a length given by the passed argument `length`. This function allocates memory for an appropriate number of `node` values, links the nodes together via their `next` fields, and initializes each `data` field to a random integer value. The last node in the list is made to have a `next` value of `NULL` so that it can be identified during traversal. In order to *serially* traverse such a linked list, one could use a loop such as the following:

```
node* current_node = construct_linked_list(10);
while (current_node != NULL) {
    // code to process ‘current_node‘ belongs here

    current_node = current_node->next;
}
```

When this loop processes the last node in the list, that node will have a `next` value of `NULL`, causing the loop to terminate. Given this loop, it should be noted that the actual traversal of the linked list must ultimately occur serially, since

the location of a given node can only be computed once that node's predecessor has been traversed first. Thus, this loop should be parallelized only if processing each node is time-consuming relative to the traversal itself. In this case, by packaging the processing of each node into an OpenMP task, the list traversal can be easily parallelized among the threads available to the program.

To parallelize the above loop, we first enclose it in the same `#pragma omp parallel` and `#pragma omp single` directives that were used in the previous section. As before, the `parallel` directive makes multiple threads available to work on future tasks that might be created, while the `single` directive makes sure that the loop is only executed once. Without the `single` directive, the linked list would be traversed once for each available thread. Inside the `parallel` directive, the processing of the node is enclosed in a `task` directive so that OpenMP can distribute that computation to available threads.

```
#pragma omp parallel
#pragma omp single
while (current_node != NULL) {
    #pragma omp task
    {
        // code to process ‘current_node‘ belongs here
    }

    current_node = current_node->next;
}
```

Note that in order to simulate a time-consuming computation, this program uses the `usleep` system call to force each task to wait for a fixed amount of time.

Exercise 7. Compile and execute `omp-linked-list.c` with 1 thread available to the program. You may need to use the `-static` argument when compiling with `gcc`:

```
nfireman@ringo:~/openmp$ gcc -static -fopenmp omp-linked-list.c -o oll
/usr/lib/../lib64/libpthread.a(libpthread.o): In function `sem_open':
(.text+0x77cd): warning: the use of `mktemp' is dangerous, better use `mkstemp'
nfireman@ringo:~/openmp$ export OMP_NUM_THREADS=1
nfireman@ringo:~/openmp$ ./oll
Thread 0 processed data 83
Thread 0 processed data 86
Thread 0 processed data 77
Thread 0 processed data 15
Thread 0 processed data 93
Thread 0 processed data 35
Thread 0 processed data 86
Thread 0 processed data 92
Thread 0 processed data 49
Thread 0 processed data 21
Duration: 1.001359
```

Exercise 8. Execute the program with 2 and 3 threads available, and compare

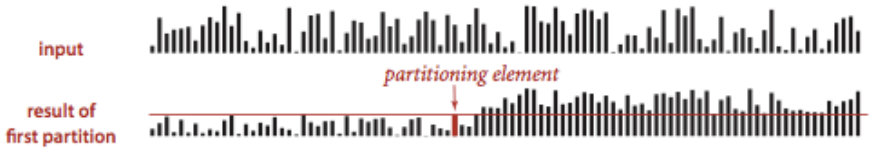

Figure 2: The effect of one partition.
Source: https://algs4.cs.princeton.edu/23quicksort

the durations to when only 1 thread was used. Predict what the execution time should be if 4 threads were used, then verify your prediction.

Exercise 9. Comment out the `usleep` call inside the `task` directive. After this change is made, processing each node will be very fast relative to the overhead of creating and scheduling tasks introduced by OpenMP, so making more threads available should yield a net decrease in performance. Execute the modified program with 1, 2, and 4 threads available, and record the execution times to confirm that this is the case.

Additional resources

- Section 3.2 of "Using OpenMP—The Next Step" by Ruud van der Pas, Eric Stotzer and Christian Terboven

  (Local copy saved to `resources/using-openmp/3-tasking.pdf`)

6 Quicksort

Quicksort is a popular sorting algorithm that is both relatively simple to implement and performant on common types of data in practice. This section will develop a parallel implementation of quicksort using OpenMP. As in the previous sections, we will see that OpenMP allows us to parallelize a serial implementation of quicksort simply by adding compiler directives, and without modifying the underlying logic of the program.

The main subroutine in quicksort is the *partition* operation, pictured in Figure 2. One element is chosen as the *partioning element* or the *pivot*, which is the value highlighted in the second row of the figure. To perform the partition, the array is reordered so that all values less than the pivot are left of the pivot, and all values greater than the pivot are to the right of the pivot. After the partition, the pivot value will be located in its correctly sorted position, and sorting the full array is reduced to recursively sorting the left and right subarrays on either side of the pivot.

The provided program `omp-quicksort.c` implements quicksort using the recursive, divide-and-conquer approach described above. The partition operation is implemented in the `partition` function, which partitions the subarray of `data` between the indices `start` and `end`:

```
int partition(int *data, int start, int end)
```

In the `partition` subroutine, the pivot is chosen (arbitrarily) as the middle element of the subarray. It is desirable to choose the pivot as near to the median value of the subarray as possible, and with prior knowledge about the state of the input data, a better choice may sometimes be possible. Note that `partition` must return the final index of the pivot after performing the partition, since that index also describes the bounds of the two subarrays that need to be recursively sorted. The sorting routine itself is implemented in the `quicksort` function:

```
void quicksort(int *data, int start, int end) {
  if (start >= end) return;

  int pivot_idx = partition(data, start, end);

  #pragma omp task
  quicksort(data, start, pivot_idx - 1);

  #pragma omp task
  quicksort(data, pivot_idx + 1, end);
}
```

The quicksort routine first partitions the subarray of `data` between the indices `start` and `end`, then recursively sorts the two subarrays on either side of the sorted position of the pivot. The recursive calls are wrapped in task directives, which parallelizes the algorithm as we have seen before.

As we saw in Section 4, using OpenMP tasks carries a certain overhead that can slow a program down when tasks are spawned for very small recursive cases. To remedy this, we can introduce a cutoff length such that calling `quicksort` on a subarray shorter than the cutoff won't spawn new tasks, and will instead sort the subarray serially. This modification will be performed in Exercise 11.

Exercise 10. Compile and execute `omp-quicksort.c` with 1, 2, and 4 threads available. You may need to use the `-static` argument when compiling with gcc:

```
nfireman@ringo:~/openmp$ gcc -static -fopenmp omp-quicksort.c -o omp-quicksort
/usr/lib/../lib64/libpthread.a(libpthread.o): In function `sem_open':
(.text+0x77cd): warning: the use of `mktemp' is dangerous, better use `mkstemp'
nfireman@ringo:~/openmp$ ./omp-quicksort
Result valid: yes
Values sorted: 200000
Duration: 0.506495 seconds
```

Exercise 11. Perform the following modifications to `omp-quicksort.c`:

- Create a new function `serial_quicksort` that has the same behavior as `quicksort`, except that it doesn't create new tasks to handle its recursive calls. It should have the same signature as `quicksort`, namely:
  `void serial_quicksort(int *data, int start, int end)`

- Modify `quicksort` to sort the input subarray by calling `serial_quicksort` when the length of the input subarray is below 1000.
- Compare the performance of the modified program to the performance obtained in Exercise 10.

Additional resources

- Section 3.3 of "Using OpenMP—The Next Step" by Ruud van der Pas, Eric Stotzer and Christian Terboven

  (Local copy saved to `resources/using-openmp/3-tasking.pdf`)

7 Polynomial root-finding

The problem of *polynomial root-finding* has wide applications in engineering, physics, and other fields. In this problem, we are given an input polynomial $f(x)$:

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0.$$

We say that $f(x)$ has *degree* n, the largest power of x appearing in $f(x)$. Our goal will be to compute the *roots* of $f(x)$, i.e. numbers r so that $f(r) = 0$. $f(x)$ will always have at most n distinct roots (a result known as the Fundmental Theorem of Algebra), but in practice may have anywhere between 0 and n roots.

Finding the exact values of every root isn't possible in general, but it is for small values of n. You may be familiar with the quadratic formula, which tells us the exact roots when $n = 2$; similar *cubic* and *quartic* formulas exist that give exact roots when n is 3 and 4, respectively, but no such general formula exists when $n \geq 5$.

Thus in general, the roots of $f(x)$ must instead be approximated. The simplest such approximation algorithm is the *bisection method*, which has a very similar main idea to a typical binary search. The crux of the bisection method is that one can efficiently identify when an interval $[a, b]$ must contain a root of $f(x)$. Namely, if $f(a)$ and $f(b)$ have opposite signs (either $f(a) > 0$ and $f(b) < 0$, or vice versa), then $f(r)$ must attain the value 0 at some point r between a and b. This is illustrated in Figure 3.

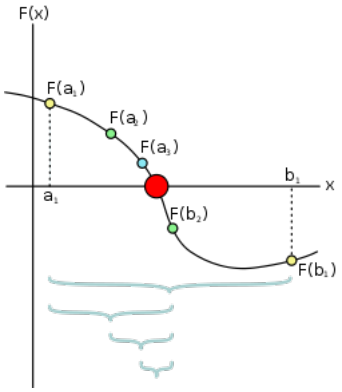

Figure 3: Illustration of the bisection method.
Source: Wikipedia

Once we know that a root must lie in the interval $[a, b]$, we compute the interval's midpoint $m = \frac{a+b}{2}$ and perform two more tests to determine whether a root lies in the subintervals $[a, m]$ or $[m, b]$. Since these two subintervals cover all of $[a, b]$, the root must lie in one of them. When we identify which subinterval contains the root, we repeat the bisection process again, and so on, until we narrow down the root to a sufficiently small interval.

Since searching one interval for a root in this way is independent of searching any other interval, the bisection method can be naturally parallelized by conducting multiple interval searches concurrently. The provided program `omp-rootfind.c` contains a parallel implementation of the bisection method, using OpenMP tasks to schedule recursive calls as we have already seen. In terms of the variables used in the program, the interval `[X_MIN, X_MAX]` is searched for roots by splitting this interval into subintervals of width `X_STEP`, and using the bisection method to concurrently search for a root in each subinterval.

Note that in general, it is possible for two (or more) roots to go undetected if they lie in the same subinterval of width `X_STEP`, since in this case it is possible for the input polynomial to have the same sign on both endpoints of the subinterval. This is a weakness of the bisection method, and the situation of two or more roots very close together can only be resolved by using more a advanced technique.

One complication with the parallel bisection method implemented in this way is that the multiple threads must be able to record each root they find in a shared-memory root list, which must be done in a thread-safe way. In this program, the root list is implemented as a global array `roots`, and the `record_root` function appends a value to this array. To safely append roots without creating a race condition, the `critical` directive is used to guarantee that only one thread may call the `record_root` function at a time:

```
#pragma omp critical (root)
record_root((min + mid) / 2);
```

The `critical` directive takes an optional *label* argument, which is `root` in the above code. The label can be used to make multiple blocks of code share the same thread-safe guarantee. Since this program calls `record_root` in two different places, the `root` label is used to prevent one thread recording a root from one call site while another thread is recording a root from the other call site.

Exercise 12. Compile and execute `omp-rootfind.c` with 1, 2, 3, and 4 OpenMP threads. Record the execution time in each case.

Exercise 13. Modify `omp-rootfind.c` to run serially by removing both `#pragma omp task` directives in the body of the `find_roots` function. Compile and execute the modified program with 1, 2, 3, and 4 OpenMP threads.

8 Data pipeline

This section will demonstrate the `depend` clause for OpenMP task directives, which allow the programmer to create explicit *dependencies* between tasks. In this context, if task B depends on task A, then task B cannot begin execution until task A has completed its own execution. By using the `depend` clause in the directives which create tasks A and B, one can guarantee that this ordering will be respected by OpenMP. The following is an example implementation of this two-task scenario using the `depend` clause:

```
int status;

#pragma omp task depend(out:status)
{
  // task A, to be executed first
}

#pragma omp task depend(in:status)
{
  // task B, to be executed after task A completes
}
```

The variable `status` is used as a marker to link the dependent tasks. Any variable can be used for this purpose, but in order to create a dependence between tasks, the same variable must be used for each task. Note that the variable itself may or may not be modified by either task; OpenMP only uses its location as the marker, so its usage in the code is independent of its usage as a marker.

The `depend` clause is used in two different ways in the above code. Modifying a task directive with `depend(out:status)` means that the modified task can have other tasks which depend on it. Likewise, `depend(in:status)` means that the modified task depends on tasks with the `status` marker, and will

be performed once a corresponding `depend(out:status)` task has completed. Thus, by combining both the `in` and `out` versions of the clause, a dependence between the two tasks is attained.

The `depend` clause can be used to parallelize a data *pipeline*, where each section of data is processed in multiple stages that must be completed in a consistent order. By creating a data dependence between each stage of the pipeline, OpenMP can be used to fully parallelize the pipeline, while also making sure to execute the pipeline in the correct order for each section of data.

The provided program `omp-pipeline.c` shows a full program which uses the `depend` clause to implement a two-stage data pipeline. In the first stage, the `read_data` function simulates an asynchronous I/O operation by sleeping for a small amount of time, then returns a pointer to an array of integers constituting the read data. The second stage `process_data` receives the array of integers from the first stage and computes the sum of its values. A task is created for each stage, and the `depend` clause is used to guarantee that the stages occur in the proper order:

```
int *data_ptrs[i];

for (int i = 0; i < N; i++) {
  #pragma omp task depend(out: data_ptrs[i])
  {
    data_ptrs[i] = read_data(DATA_SIZE);
  }

  #pragma omp task depend(in: data_ptrs[i])
  {
    results[i] = process_data(DATA_SIZE, data_ptrs[i]);
  }
}
```

Here, the elements of the array of pointers `data_ptrs` are used as markers for the `depends` clause. Each iteration of the loop uses a different element as a marker, so each iteration will induce its own two-task dependency chain.

Exercise 14. Add a third stage to the pipeline in `omp-pipeline.c` that prints the sums stored in `results`, as follows:

- Create a function `void print_result(int idx)` that prints `results[idx]` to the console for a given index `idx`.
- Add a third task directive to the body of the program's main loop which calls `print_result(i)`. Use `results[i]` as a marker to ensure that the second task (which computes `results[i]`) finishes executing before the newly added third task begins.

Additional resources

- Section 3.4 of "Using OpenMP—The Next Step" by Ruud van der Pas, Eric Stotzer and Christian Terboven

  (Local copy saved to `resources/using-openmp/3-tasking.pdf`)