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 servers `ringo.secs.oakland.edu` or `lennon.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. But `lennon` does not have a static library installed; so removing the `-static` flag works for the compilation. Other SECS servers, such as `yoko.secs.oakland.edu`, 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)
```

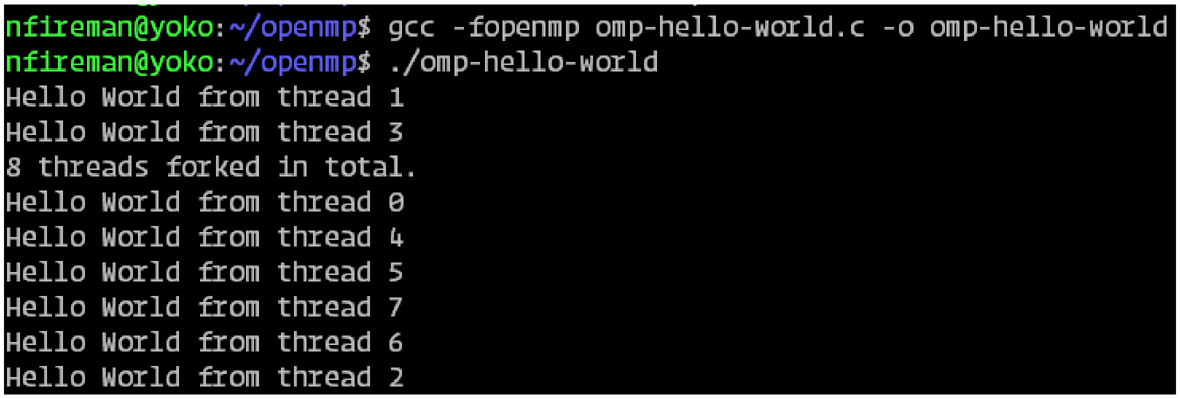
The `parallel` construct creates a number of threads, all of which immediately begin executing the structured block under the directive in parallel. There are several ways to set the number of threads created, but the two most common ways are the following:

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.

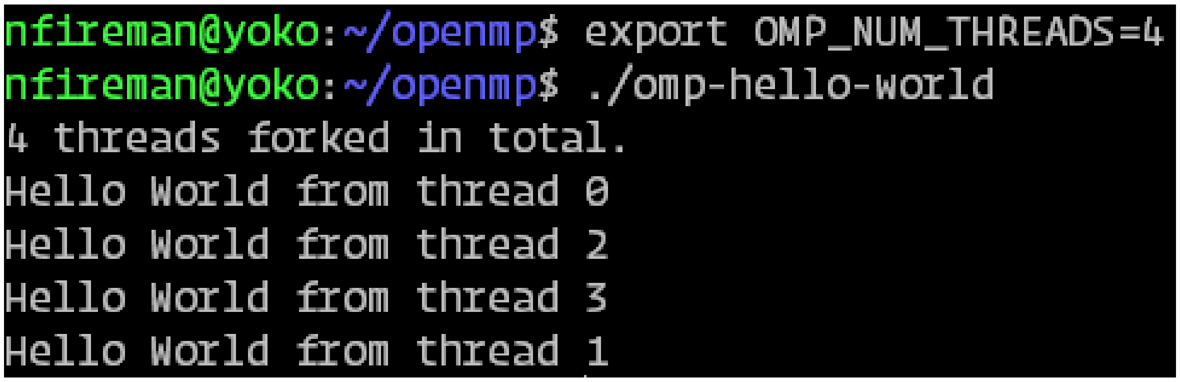
Exercise 1. Compile the `omp-hello-world.c` program using the `gcc` compiler; then execute the resulting binary file. The `-fopenmp` argument needs to be passed to `gcc` to compile an OpenMP program:



```
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 defacult value of the environment.

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
```

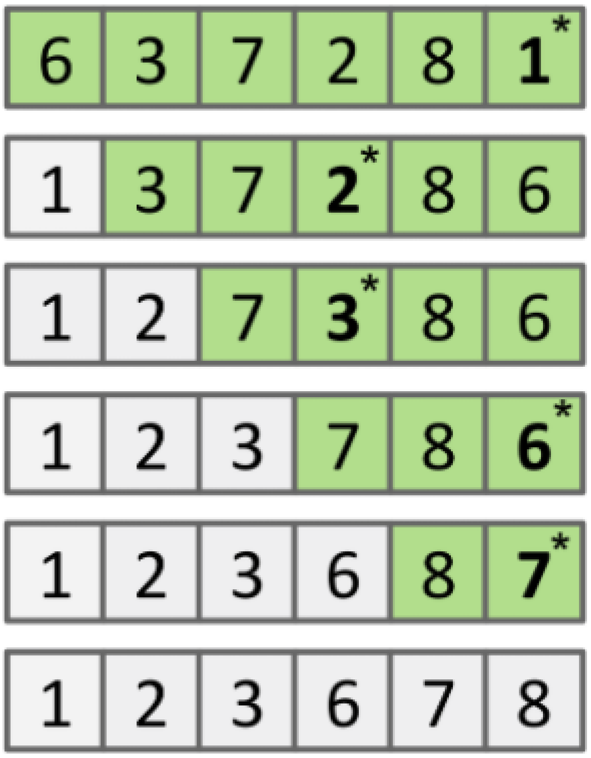


Figure 1: Selection sort. A bold number indicates the minimum unsorted value.
Source: https://courses.cs.washington.edu/courses/cse373/19au/lectures/05/reading/

Additional resources

- Slides from an introductory OpenMP presentation:

  https://www.openmp.org/wp-content/uploads/omp-hands-on-SC08.pdf

- An online OpenMP tutorial published by the Lawrence Livermore National Laboratory:

  https://hpc-tutorials.llnl.gov/openmp/

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.

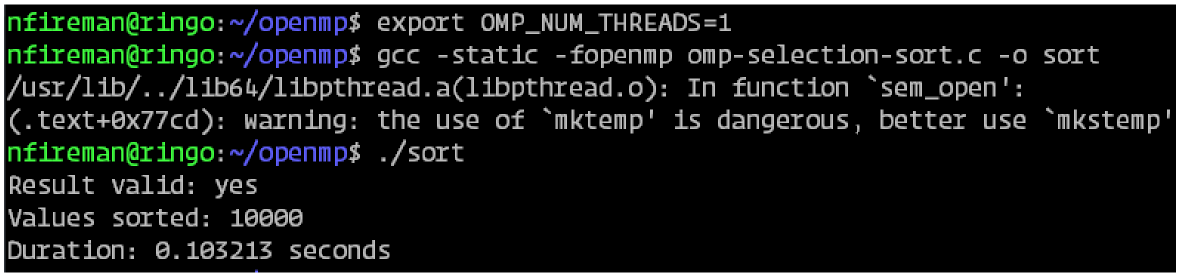
- The first argument `min_entry` is the name of the declared reduction operation.
- The second argument `entry` is the data type of the values being combined (the `struct` defined above).
- The third argument is an expression that describes how OpenMP should combine two values of the type `entry`. By 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. Write a selection sort program in C (`selection-sort.c`) without any OpenMP directive. Using varying array sizes, compile and execute both `selection-sort.c` and `omp-selection-sort.c` when only one thread is used. Draw a time vs array size graph for both programs. Discuss the results.

Exercise 5. Compile and execute the OpenMP sorting program with a fixed number of threads for different array sizes. Repeat the experiment at least two more times with different number of threads. Draw a time vs array size graph. There will be at least three line graphs. Discuss the results. In this experiment, the number of threads should be two or more.