PMC Lecture 11

Gianmaria Romano

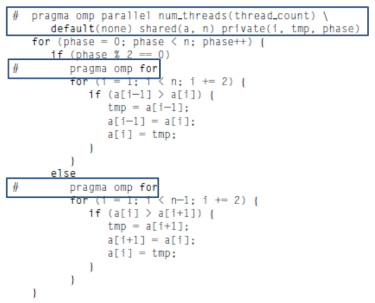
12 November 2025

Chapter 5

Shared memory systems

5.1 Parallel loops

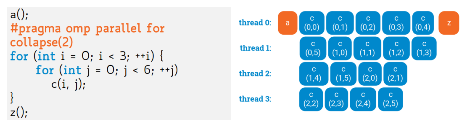
Using a dedicated pragma, OpenMP programs are able to parallelize sequential loops by distributing iterations among the threads of a team.
However, since the runtime system needs to know the exact number of iterations in advance, this approach only works with for loops as they can provide a clear iteration count.
Most particularly, the control variables should remain fixed, with the index variable changing only with the increment specified in the loop's header.
In addition, loops containing a break/return statement cannot be parallelized as the system cannot know when (or if) the condition to exit the loop will be met, unless the exit() call, which terminates the entire program, is used.
Example: The odd-even sorting algorithm sorts an array by performing an odd phase, which consists in comparing and, if needed, swapping adjacent pairs where the first element has an odd index, and an even phase, which consists in comparing and, if needed, swapping adjacent pairs where the first element has an even index.
To achieve better efficiency, it is also possible to implement this algorithm using parallel loops, as shown in the following code:



```
#  pragma omp parallel num_threads(thread_count) \
      default(none) shared(a, n) private(i, tmp, phase)
   for (phase = 0; phase < n; phase++) {
      if (phase % 2 == 0)
#        pragma omp for
         for (i = 1; i < n; i += 2) {
            if (a[i-1] > a[i]) {
               tmp = a[i-1];
               a[i-1] = a[i];
               a[i] = tmp;
            }
         }
      else
#        pragma omp for
         for (i = 1; i < n-1; i += 2) {
            if (a[i] > a[i+1]) {
               tmp = a[i+1];
               a[i+1] = a[i];
               a[i] = tmp;
            }
         }
   }
```

5.1.1 Parallelizing nested loop structures

When a program tries to parallelize a nested loop structure, it is often sufficient to parallelize the outermost loop while keeping the inner loop unchanged.
Sometimes, however, the outer loop runs for too few iterations to fully use the available threads, resulting in a lesser performance improvement: whenever this happens, it is possible to use the "collapse" clause to rewrite the structure as a single loop that is executed in parallel.



```
a();
#pragma omp parallel for
collapse(2)
for (int i = 0; i < 3; ++i) {
     for (int j = 0; j < 6; ++j)
         c(i, j);
}
z();
```

5.2 Implementing atomic operations in OpenMP

Using a dedicated pragma, it is possible to efficiently perform some atomic operations without the need for a longer critical section.
Alternatively, when dealing with longer critical sections, the OpenMP API allows to assign a name to each critical directive so that the corresponding constructs can be performed in parallel, provided different threads enter different named critical sections.
Keep in mind, however, that the compiler always needs to know the names of the critical sections beforehand before being able to parallelize them.

5.2.1 Choosing the best approach

Generally speaking, the atomic directive represents the fastest method for obtaining mutual exclusion, although, in some cases, it might be more appropriate to enforce mutual exclusion using other directives.
On the other hand, locks should mainly be used for handling critical sections involving data structures rather than on codes.
In particular, it is a good practice to avoid mixing or nesting different mutual exclusion mechanisms as, similarly for the PThreads library, there is no guarantee of fairness in the mutual exclusion constructs.