PMC Lecture 14

Gianmaria Romano

2 December 2025

Chapter 5

Shared memory systems

5.1 Parallel loops

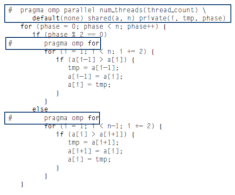
Using a dedicated pragma, OpenMP programs are able to parallelize sequential loops by distributing iterations among the threads of a team.
However, since the runtime system needs to know the exact number of iterations in advance, this approach only works with for loops as they can provide a clear iteration count.
Most particularly, the control variables should remain fixed, with the index variable changing only with the increment specified in the loop's header.

Legal forms for parallelizable for statements

```
                                  index++
                                  ++index
                    index < end   index--
                    index <= end  --index
for  index = start  ; index >= end ; index += incr
                    index > end   index -= incr
                                  index = index + incr
                                  index = incr + index
                                  index = index - incr
```

Why? It allows the runtime system to determine the number of iterations prior to the execution of the loop

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 using an odd phase, which compares and, if needed, swaps adjacent pairs where the first element has an odd index, and an even phase, which compares and, if needed, swaps adjacent pairs where the first element has an even index.
To achieve better efficiency, it is possible to implement this algorithm using parallel loops, as shown in the following code:



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.

5.1.2 Handling loop scheduling

Suppose you want to parallelize a loop by distributing n iterations among t threads.
The default OpenMP standard applies static block distribution, although, for some programs, it might be more convenient to opt for a cyclic distribution.

Thread	Iterations (Default)
0	$0, \ldots, \frac{n}{t} - 1$
1	$\frac{n}{t}, \ldots, \frac{2n}{t} - 1$
$\ldots$	$\ldots$
$t-1$	$\frac{n(t-1)}{t}, \ldots, n-1$

Thread	Iterations (Cyclic)
0	$0, \frac{n}{t}, \frac{2n}{t}, \ldots$
1	$1, \frac{n}{t}+1, \frac{2n}{t}+1, \ldots$
$\ldots$	$\ldots$
$t-1$	$t-1, \frac{n}{t}+t-1, \frac{2n}{t}+t-1, \ldots$

Generally speaking, it is possible to handle loop scheduling using the "schedule" clause, which specifies both the scheduling type and the number of loops that are assigned to each thread.
In particular, the scheduling type can be chosen among the following options:

- **Static:** the iterations are assigned to threads before the loop is executed.
- **Dynamic:** the iterations are assigned to threads as the loop is executed. In particular, each thread executes an iteration chunk on demand and, when done, it can request another one from the run-time system, allowing to improve workload balance.
- **Guided:** this scheduling type is a variant of dynamic scheduling in which the chunk size decreases as more iteration chunks are completed.

- **Auto:** the scheduling type is decided by the compiler or by the run-time system.
- **Runtime:** the scheduling type is defined by the programmer at run-time through a dedicated environmental variable.
  Generally speaking, static scheduling is suitable for homogeneous iterations, whereas dynamic scheduling is more convenient for varying execution costs.

5.1.3 Handling data dependencies

One of the main issues of parallel loops lies in the risk of having (loop-carried) data dependencies among iterations, which do not allow to correctly parallelize some parts of the loop.
Generally speaking, a parallel loop can give rise to four main types of dependencies:

- **Read after write:** an iteration needs to read a value that was produced by a previous iteration, setting a limit to loop parallelization.
  This type of dependence can be handled in one of the following ways:
  - **Variable fix:** identify variables whose values predictably change across iterations and transform them into reduction/induction variables to safely parallelize the loop.
  - **Loop skewing:** rearrange the loop statements in order to make sure that they use values that prevent or limit data dependencies.
- **Write after read:** an iteration writes to a memory location that a previous iteration still needs to read.
- **Write after write:** two different iterations write to the same memory location, meaning that the final result will depend on which operation is carried out last.
- **Read after read:** multiple iterations read the same location without modifying it.
  Compared to other data dependencies, this dependence is not really problematic as it does not prevent parallelization.

5.2 Synchronization constructs

Generally speaking, a synchronization construct is a mechanism that coordinates the execution of multiple threads working in parallel.
In particular, OpenMP provides the following synchronization directives:

- **Master/Single:** these directives force the execution of a structured block by a single thread.
  Note that "single" also implies that there is a barrier immediately after the structured block.

- **Barrier:** this directive guarantees that the execution of a code can go on if and only if all threads reach the barrier, providing a useful synchronization point.
- **Section:** this directive divides a program in different sections that will be executed by different threads in parallel.
- **Ordered:** this directive is used inside a parallel for loop to execute some blocks sequentially in their original order.