PMC Lecture 14

Gianmaria Romano

3 December 2025

Chapter 5

Shared memory systems

5.1 Parallel loops

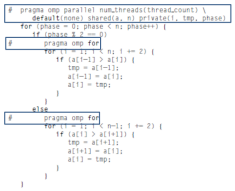
Using a dedicated pragma, OpenMP programs are able to parallelize sequential loops by distributing iterations among the threads of a team.
However, since the runtime system needs to know the exact number of iterations in advance, this approach only works with for loops as they can provide a clear iteration 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

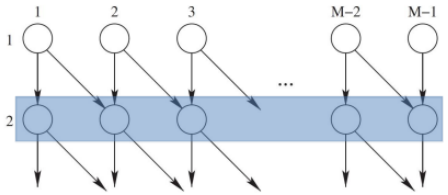
In addition, loops containing a break/return statement cannot be parallelized as the system cannot know when (or if) the condition to exit the loop will be met, unless the exit() call, which terminates the entire program, is used.
Example: The odd-even sorting algorithm sorts an array using an odd phase, 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 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 in a way that prevents, or at least limits, dependencies.
  - **Partial parallelization:** introduce an iteration space dependency graph, whose nodes indicate loop iterations and whose edges represent dependencies, and use it to find dependence-free loops that can be safely parallelized.



  - **Refactoring:** rewrite the loop(s) in order to spot parallelizable iterations.
  - **Fissioning:** split the loop into a sequential part and a parallel part.
  - **Algorithm change:** if none of the above methods is effective, it is typically more convenient to opt for a different algorithm.

- **Write after read:** an iteration writes to a memory location that a previous iteration still needs to read.
  Generally speaking, it is possible to handle this dependence by making a copy of a value before actually modifying it.

- **Write after write:** two different iterations write to the same memory location, meaning that the final result will depend on the order of execution.
  Generally speaking, it is possible to handle this dependence by refactoring the loop so that different writing operations write into different memory locations, therefore eliminating the risk of conflicts between operations.

- **Read after read:** multiple iterations read the same location without modifying it.
  Compared to other data dependencies, this dependence is not really problematic and it does not prevent parallelization.