MYCHP203 — TOP : Lab 2-3

Gabriel Dos Santos
gabriel.dossantos@cea.fr
gabriel.dos-santos@uvsq.fr

Hugo Taboada
hugo.taboada@cea.fr

April 8th, 2024

Principle of locality, data structure packing,
Structure of Arrays vs Arrays of Structures, false sharing
& OpenMP tasks.

I Cache blocking

1. Profile the `cross-stencil` code using Linux Perf. Specifically, look at cache misses in the L1 and L3 (LLC) caches.

2. Implement a cache-blocked version of the `cross_stencil::run()` function to improve the spatial and temporal locality of the loop.

3. Assert that your implementation computes the correct result. Update the `CMakeLists.txt` file accordingly.

4. Tune the block size to maximize the performance of your cache-blocked version. Use Perf to assert the improvements in spatial and temporal locality.

II Data structure packing

5. Given the alignment rules imposed on structure by the ISO C standard, determine how many bytes are used by the `cell_t` structure in the `mesh` code.

6. Rearrange the layout of the `cell_t` structure to minimize its memory footprint. Compute how much memory is saved using the optimized layout.

7. Using Perf, check if there is a change in the number of cache misses between the two structure implementations.

III AoS vs SoA

8. Using MAQAO's CQA module, determine if the innermost loop in the `mesh_compute_velocity()` function is correctly vectorized.

9. Measure the strong and weak scalabilty of the current code. Write a small script (can be in bash, Python, R, ...) to plot the results.

10. Update the code in `mesh` to use Structure of Arrays (SoA) instead of Array of Structures (AoS).

11. Use CQA to assert that the change has helped the compiler vectorize the code.

12. Measure the strong and weak scalabilty of the new code.

IV False sharing

Consider the parallelization of an integer sum reduction, as the sample code in `integer-sum-redux`.

13. Implement this benchmark in `integer-sum-redux/sequential.cpp` sequentially. Measure the execution time using the `getticks()` function in the `"cycle.h"` header. Give the result as a « cycles/elements » metric.

14. Try measuring with both `-O0` and `-O3` compiler optimizations passes. What do you see?

15. Parallelize the program using OpenMP. For this lab, we will not use OpenMP's reduction directives and implement the solution using only `#pragma omp parallel` and `#pragma omp for`.
Are your results correct? Why?

16. To fix the problem, we propose using the `#pragma omp atomic`. This directive ensures that a shared variable can only be read and written one thread at a time. Insert it in the parallel code section. Is it sufficient to get the correct result? Why?

17. Update the code to use `#pragma omp critical` instead of `atomic`. A critical region can be seen as a generalization of the `atomic` directive. Threads execute this region in a non-deterministic order but still keep the « one-at-a-time » semantic using a lock/unlock system.

18. Update the previous code (the one using `atomic`) but reverse the terms of the equation so that the `sum` variable is read instead of being written. What do you notice?

19. Without using any OpenMP directive, implement the sum reduction using a temporary array indexed by the thread ID. Measure the performance. What do you see?

20. Add some padding between the elements of the temporary array, so that each element is on its own cache line (generally 64 bytes on x86-64/amd64). Try again with a smaller amount of padding. What do you notice? How can you explain it? What can we conclude about the precautions we need to take when defining data structures in parallel programs?

21. Try out the OpenMP `#pragma omp reduction` directive and fill out the following table:

Implementation	**Sequential**	**Parallel**	**Speedup (to baseline)**
Sequential `-O0` (baseline)		N/A	N/A
Sequential `-O3`		N/A	
`#pragma omp parallel for`			
`#pragma omp critical`			
`#pragma omp atomic`			
`#pragma omp atomic` (read)			
Temporary array			
Temporary array & padding			
`#pragma omp reduction`			

22. What are the limits of our final implementation if we plan on using it on a many-core architecture (e.g., a GPU)?
Propose alternatives (do not implement them).

V OpenMP Tasks

An OpenMP task is an independent block of work that will be executed by one thread of a given team. The OMP thread that encounters the task first will put it in a queue to be scheduled later, either by itself or by another thread. When a thread reaches a barrier, it executes the tasks currently in the queue.

We are now going to parallelize a program using OpenMP tasks. An example of a parallel Fibonacci series is available in the `exercises/omp-task/fibonacci` directory.

23. Parallelize the code in the `nqueens.cpp` file using OpenMP tasks. Is the performance satisfactory?

24. Find a way to optimize this code.