Assignments P1.6 - Part 2

General remarks

All assignments are to be programmed in *portable* Fortran 2008 except where explicitly noted. If an assignment requires a modification of a main program from a previous section, a copy under a new name should be made. Prefix main program sources by the section number. The assignments in this part depend on completion of at least sections 01 to 04 from part 1.

09 Simple sort

Integrate the provided files *09_simplesort.f90* and *sorting.f90* into your build system from part 1. *09_sorting.f90* provides a framework for testing subroutines that sort arrays for real numbers, and *sorting.f90* provides a module containing two implementations of sorting algorithms, a very simple one and a quick sort implementation. Now take the swap subroutine and move it into your list_tools module instead. Furthermore add calls to is_sorted() from the same module after each sort is completed and print out a warning, if an array is not sorted. Since you validated the is_sorted() function with the data files in part 1, you can now validate sorting algorithms with it. Run the resulting executable and extract the timing information for the various problem set sizes.

10 Bubblesort

Implement a bubblesort algorithm into the sorting module and create a new main program based on the executable from section 09 that now calls bubblesort instead. Again, record benchmark data provided by the executable.

11 Insertion sort

Now Implement an insertion sort into the sorting module and set up an executable for testing and benchmarking it like in the previous sections and collect the performance data.

12 Quicksort

Use the provided quicksort implementation to build a corresponding test and benchmark executable for quicksort. As noted in the lecture, the choice of the pivot element is arbitrary as far as the algorithm as such is concerned, but it can have a significant impact on the performance of quicksort. In the *sorted.f90* file is a commented out alternative strategy for choosing the pivot element. Use this as well and collect benchmark data for both variants of quicksort. Since this implementation of quicksort uses recursions, it may use large amounts of stack space, thus you may need to increase the available stack size using ‘ulimit -s unlimited’ to avoid crashes.

13 Merge sort

Now implement a merge sort algorithm. Implement a “bottom up” version, so you can most easily avoid recursions and conveniently implement section 14. As before, collect benchmark data for later analysis and discussion.

14 Hybrid sort

Well scaling sorting algorithms often have significant overhead and thus are less efficient for small arrays. Thus, the fastest sorting implementations are often hybrids that combine multiple algorithms. Implement such a hybrid sort from insertion sort and merge sort: rather than starting the merge with lists of length 1, do a loop over the data in chunks of 32 elements and sort each of them with insertion sort; then continue with merge sort on these pre-sorted sublists. One more time, collect timings for different data set sizes and degree of previous sorting.

15 Benchmarks and discussion

Try to classify the various sorting algorithms by performance for different problem sizes and properties of the data to be sorted. Try to extract information from “perf stat” looking at performance counters like clock cycles, instructions, branch instructions, branch misses, cache references and cache misses to find explanations for the relative performance. Also discuss why algorithms with the same big O scaling behavior have different performance. Finally, observe and describe how the performance of the algorithms is affected by compiler optimization. Write this down in a brief report and Include suitable graphs to support your arguments.