PAR Laboratory Assignment
Lab 4: Divide and Conquer parallelism with OpenMP: Sorting

E. Ayguadé, R. M. Badia, J. R. Herrero, J. Morillo, J. Tubella and G. Utrera

Spring 2019-20

Index

Note: Each chapter in this document corresponds to a laboratory session (2 hours).

Session 1

Task decomposition analysis for Mergesort

1.1 "Divide and conquer"

Mergesort is a sort algorithm which combines a "divide and conquer" sort strategy that divides the initial list (positive numbers randomly initialised) into multiple sublists recursively, a sequential *quicksort* that is applied when the size of these sublists is sufficiently small, and a merge of the sublists back into a single sorted list.

In this first section of the laboratory session you should understand how the code `multisort.c`[1] implements the "divide and conquer" strategy, recursively invoking functions `multisort` and `merge`. You will also investigate, using the *Tareador* tool, potential task decomposition strategies and their implications in terms of parallelism and task interactions required.

1. Compile the sequential version of the program using `make multisort` and execute the binary. You can provide three optional command–line arguments (all of them power–of–two): size of the list in Kiloelements (`-n`) and size in elements of the vectors that breaks the recursions during the sort and merge phases (`-s` and `-m`, respectively). For example `./multisort -n 32768 -s 1024 -m 1024`, which actually are the default values when unspecified. The program randomly initialises the vector, sorts it and checks that the result is correct.

1.2 Task decomposition analysis with *Tareador*

1. `multisort-tareador.c` is already prepared to insert *Tareador* instrumentation. Complete the instrumentation to understand the potential parallelism that each one of the two recursive task decomposition strategies (*leaf* and *tree*) provide when applied to the sort and merge phases:
   - In the *leaf* strategy you should define a task for the invocations of `basicsort` and `basicmerge` once the recursive divide–and–conquer decomposition stops.
   - In the *tree* strategy you should define tasks during the recursive decomposition, i.e. when invoking `multisort` and `merge`.
2. Use the `multisort-tareador` target in the `Makefile` to compile the instrumented code and the `run-tareador.sh` script to execute the binary generated. This script uses a very small case (`-n 32 -s 1024 -m 1024`) to generate a small task dependence graph in a reasonable amount of time.
3. From the task dependence graphs that are generated, draw up a table showing the number of tasks doing computation (i.e. those actually executing `basicsort` and `basicmerge`) and the number of internal tasks (i.e. those that only create new tasks, basically executing invocations to `multisort` and `merge`) that are generated at each recursion level for each task decomposition strategy (having a clear explanation for the numbers on that table).

[1] Copy file `lab4.tar.gz` from `/scratch/nas/1/par0/sessions`.

4. Continue the analysis of the task graph generated for each task decomposition strategy (*leaf* and *tree*) by identifying the task ordering constraints that appear and the causes for them, and the different kind of synchronisations that could be used to enforce them.

5. In order to predict the parallel performance and scalability with different number of processors, simulate in *Tareador* the parallel execution using 1, 2, 4, 8, 16, 32 and 64 processors. Draw up a table with the execution time and speed–up simulated by *Tareador*. Any observable differences between both recursive task decomposition strategies (*leaf* and *tree*)? In terms of execution time? You can zoom at the very beginning of the traces and see if there are differences on how tasks are generated. Why the scalability is limited to 16 processors for the considered input data set?

Session 2

Shared-memory parallelisation with *OpenMP* tasks

In this second section of the laboratory session you will parallelise the original sequential code using OpenMP, following the task decomposition analysis that you have conducted in the previous section.

As in the previous section, two different parallel versions will be explored: *leaf* and *tree*. We suggest that you start with the implementation and analysis of the *leaf* strategy and then proceed to the alternative *tree* strategy.

1. Insert the necessary *OpenMP* `task` for task creation and the appropriate `taskwait` or `taskgroup` to guarantee the appropriate task ordering constraints. **Important:** Do not include in this implementation neither a *cut–off* mechanism to control the granularity of the tasks generated nor use task dependences to enforce task ordering constrains; both things are considered later in this laboratory assignment.

2. Once compiled using the appropriate `Makefile` entry, interactively execute using a small number of processors (for example 2 or 4) with the default input parameters to make sure that the parallel execution of the program verifies the result of the sort process and does not throw errors about unordered positions. Execute several times to make sure your parallelisation is correct.

3. Once correctness is checked, analyze the scalability of your parallel implementation by looking at the two speed–up plots (complete application and multisort only) generated when submitting the `submit-strong-omp.sh` script. This script executes the binary with a number of processors in the range 1 to 12 by default (but you can change this range if you want to explore a different range). Be patient! This script may take a while to execute. Is the speed–up achieved reasonable? Look at the output of the executed script to check your execution for all number of threads has no errors.

4. If you are not convinced with the performance results you got (by the way, you should not!), submit the binary to the *execution* queue using the `submit-omp-i.sh` script for 8 processors, which will trace the execution of the parallel execution with a much smaller input (`-n 128 -s 128 -m 128`). Open the trace generated with *Paraver* and make use of the configuration files already listed in the document for the second laboratory assignment and available inside the `cfgs/OpenMP/OMP_tasks` directory to do the analysis and understand what is going on. Try to conclude why your parallel implementation is not scaling as you would expect. For example, is there a big sequential portion in your parallel execution? is the program generating enough tasks to simultaneously feed all processors? ...

2.1 Task *cut–off* mechanism

Next you will include a *cut–off* mechanism in your OpenMP implementation for the *tree* recursive task decomposition; the *cut–off* mechanism should allow you to control the maximum recursion level for task generation. The mechanism should be independent from the mechanism already included in the code to

control the maximum recursion level (and controlled with the `-s` and `-m` optional flags in the execution command line for `multisort-omp`). To also control the *cut–off* level from the execution command line, an optional flag (`-c value`) has been included to provide a `value` for the recursion level that stops the generation of tasks.

5. Modify the parallel code implementing the *tree* strategy to include the proposed *cut–off* mechanism, making use of the *OpenMP* `final` clause and `omp_in_final` intrinsic.

6. Once implemented generate a trace (using the `submit-omp-i.sh` script with 8 processors and using the optional argument that specifies the *cut–off* value) for the case in which you only allow task generation at the outermost level (i.e. `-c 0`). Visualise the resulting trace with *Paraver* and try to understand what is visualised. Repeat the execution using a level 1 *cut–off* (i.e. `-c 1`) and observe/understand the differences.

7. Once you understand and are sure that the *cut–off* mechanism works, you will explore values for the *cut–off* level depending on the number of processors used. For that you can submit the `submit-cutoff-omp.sh` script to explore different values for the *cut–off* argument; for that exploration, we allow recursion to go deeper levels (`-n 32768 -s 128 -m 128`). The number of processors is the only argument required by the script.

8. For that optimum value, analyse the scalability by looking at the two speed–up plots generated when submitting the `submit-strong-omp.sh` script. Change in that script the values for `sort_size` and `merge_size` to 128 and for `cut--off` the optimum value you found in your exploration.

Optional 1: Have you explored the scalability of your *tree* implementation with *cut–off* when using up to 24 threads? Why is performance still growing when using more than the 12 physical cores available in `boada-1 to 4`? Also complete your scalability analysis on the other node types in boada: `boada-5` and `boada-6 to 8`. Set the maximum number of cores to be used (variable `np_NMAX`) by editing `submit-strong-omp.sh` in order to do the complete analysis. HINT: consider the number of physical/logical cores in each node type and set `np_NMAX` accordingly.

Session 3

Using *OpenMP* task dependencies

Finally you will change the *tree* parallelisation in the previous chapter in order to express dependencies among tasks and avoid some of the `taskwait/taskgroup` synchronisations that you had to introduce in order to enforce task dependences. For example, in the following task definition

```
#pragma omp task depend(in: data[0], data[n/4L]) depend(out: tmp[0])
merge(n/4L, &data[0], &data[n/4L], &tmp[0], 0, n/2L);
```

the programmer is specifying that the task can not be executed until the sibling task (i.e. a task at its same level) that generates both `data[0]` and `data[n/4L]` finishes. Also when the task finishes it will signal other tasks waiting for `tmp[0]`.

1. Edit your *tree* recursive task decomposition implementation, including *cut–off*, in `multisort-omp.c` to replace `taskwait/taskgroup` synchronisations by point–to–point task dependencies. Probably not all previous task synchronisations will need to be removed, only those that are redundant after the specification of dependencies among tasks.

2. Compile and submit for parallel execution using 8 processors. Make sure that the program verifies the result of the sort process and does not throw errors about unordered positions.

3. Analyse its scalability by looking at the two strong scalability plots and compare the results with the ones obtained in the previous chapter. Are they better or worse in terms of performance? In terms of programmability, was this new version simpler to code?

4. Trace the parallel execution with 8 processors and use the appropriate configuration files to visualise how the parallel execution was done and to understand the performance achieved.

Optional 2: Complete your parallel implementation of the `multisort-omp.c` by parallelising the two functions that initialise the `data` and `tmp` vectors[1]. Analyse the scalability of the new parallel code by looking at the two speed–up plots generated when submitting the `submit-strong-omp.sh` script. Reason about the new performance obtained with support of *Paraver* timelines.

[1]The `data` vector generated by the sequential and the parallel versions does not need to be initialised with the same values, i.e. in both cases, the `data` vector has to be randomly generated with positive numbers but not necessarily in the same way.

Deliverables

Important:

- Deliver a document that describes the results and conclusions that you have obtained (only PDF format will be accepted).
- The document should have an appropriate structure, including, at least, the following sections: Introduction, Parallelisation strategies, Performance evaluation and Conclusions. The document should also include a front cover (assignment title, course, semester, students names, the identifier of the group, date, ...), index or table of contents, and if necessary, include references to other documents and/or sources of information.
- Include in the document, at the appropriate sections, relevant fragments of the C source codes that are necessary to understand the parallelisation strategies and their implementation (i.e. for *Tareador* instrumentation and for all the OpenMP parallelisation strategies).
- You also have to deliver the complete C source codes for *Tareador* instrumentation and all the OpenMP parallelisation strategies that you have done. Your professor should be able to re-execute the parallel codes based on the files you deliver. Include both the PDF and source codes in a single compressed tar file (`GZ` or `ZIP`). Only one file has to be submitted per group through the *Racó* website.

As you know, this course contributes to the **transversal competence "Tercera llengua"**. Deliver your material in English if you want this competence to be evaluated. Please refer to the "Rubrics for the third language competence evaluation" document to know the *R*ubric that will be used.

IMPORTANT: due to the COVID-19 exceptionality that caused a change in the evaluation methodology, this report will contribute to the *Continuous Assessment* mark that is used to decide if a student has to do the final exam. We encourage you to spend effort in writing this document, doing clear and complete explanations about the code changes that were introduced and the results (by means of tables, graphs, plots, ...) that are reported, as well as writing relevant concluding remarks about the outcomes of the laboratory assignment.

Analysis with *Tareador*

1. Include the relevant parts of the modified `multisort-tareador.c` code and comment where the calls to the *Tareador* API have been placed for each one of the two recursive task decomposition strategies considered. Comment also about the tasks graphs that are generated, including a table with the number of tasks that are generated at each recursion level for each task decomposition strategy. You should also comment the task ordering constraints that have been identified and the causes for them, and the different kind of synchronisations that could be used to enforce them.
2. For the analysis of scalability, include a table with the execution time and speed-up as predicted by *Tareador* (for 1, 2, 4, 8, 16, 32 and 64 processors). Are the results close to the ideal case? Any observable differences between both recursive task decomposition strategies (*leaf* and *tree*)? Which is (are) the factor(s) that limit the scalability in this case.

Parallelisation and performance analysis with tasks

1. Include the relevant portion of the codes that implement the two versions (*leaf* and *tree*), commenting whatever necessary in terms of task creation and synchronisation.

2. For the the *leaf* and *tree* strategies, include the speed–up (strong scalability) plots that have been obtained for the different numbers of processors. Reason about the performance differences that you observe between *leaf* and *tree* implementations, including captures of *Paraver* windows that help you to explain and justify the differences between them. Explain what is causing the different scalability that is observed for the whole program and for the `multisort` function only.

3. Show the changes you have done in the code in order to include a *cut–off* mechanism based on recursion level. Include the execution time plots for different numbers of processors and *cut–off* levels. Conclude if there is a value for the *cut–off* argument that improves the overall performance? Analyse the scalability of your parallelisation with this value. Include any *Paraver* window that helps you to explain how your *cut–off* mechanism is working.

Parallelisation and performance analysis with dependent tasks

1. Include the relevant portion of the code that implements the *tree* version with task dependencies, including *cut–off*, commenting whatever necessary in terms of synchronisation. Regarding programmability, was this parallel version simpler to code?

2. Although more powerful in terms of synchronisation semantics, are `depend` clauses introducing a noticeable overhead to your parallel execution? Comment the performance plots that you obtained, including captures of *Paraver* windows to justify your explanation.

Optional

1. If you have done any of the two optional parts in this laboratory assignment, please include and comment in your report the relevant portions of the code and/or scripts that you modified. Include all the necessary plots that contribute to understand the performance that is obtained, reasoning about the results that are shown and conclusions that you extract, well supported with relevant captures of *Paraver* windows.