PAR Laboratory Assignment
Lab 3: Divide and Conquer parallelism with OpenMP: Sorting

Ll. Àlvarez and E. Ayguadé, J. R. Herrero, J. Morillo, J. Tubella, G. Utrera

Fall 2018-19

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

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" mergesort strategy that divides the initial list (positive numbers randomly initialized) 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 Kiloelements of the vectors that breaks the recursions during the sort and merge phases (`-s` and `-m`, respectively). For example `./multisort -n 32768 -s 32 -m 32`, which actually are the default values when unspecified. The program randomly initializes 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 the "divide and conquer" strategy provides when applied to the sort and merge phases. Once modified, compile the code using the `multisort-tareador` target in the `Makefile`. Execute the binary generated using `run-tareador.sh multisort-tareador` script. This script uses a very small case to generate a reasonable task graph. Analyze the task graph generated, the dependences, their causes and the task synchronizations that are needed to enforce them.

2. 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 and complete the table requested in the Deliverables section.

[1] Copy file `lab3.tar.gz` from `/scratch/nas/1/par0/sessions`.

Session 2

Shared-memory parallelization with *OpenMP* tasks

In this second section of the laboratory session you will parallelize the original sequential code using OpenMP, following the task decomposition analysis that you have conducted in the previous section. Two different parallel versions will be explored: *Leaf* and *Tree*.

- In *Leaf* you should define a task for the invocations of `basicsort` and `basicmerge` once the recursive divide–and–conquer decomposition stops.
- In *Tree* you should define tasks during the recursive decomposition, i.e. when invoking `multisort` and `merge`.

Implement these two parallel *OpenMP* versions using `task` for task creation and the appropriate `taskwait` and/or `taskgroup` to guarantee the appropriate task ordering constraints. **Important:** Do not include in this implementation any *cut–off* mechanism to control the granularity of the tasks generated. We suggest that you start with the *Leaf* version and do the 4 steps below to compile, execute and instrument. After that, implement the *Tree* version and repeat all the steps.

1. Edit the `multisort-omp.c` (**NOT the Tareador instrumented version**) and insert the necessary *OpenMP* `pragmas` to implement each parallel version, one at a time.
2. Compile using the `multisort-omp` target in the `Makefile` to generate the executable file and submit it using the `submit-omp.sh` (specifying 8 processors for the parallel execution). Take a look at the script file to understand what it does and the name of the file where the result of the execution is stored. **Important: make sure that the program verifies the result of the sort process and does not throw errors about unordered positions**.
3. Submit the `submit-omp-i.sh` script to trace the execution of your *OpenMP* program. In order to understand the behaviour of the parallel execution, please 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.
4. Once you understand the behaviour using *Paraver*, analyze the scalability of your parallel program by looking at the two speed–up plots (complete application and multisort only) generated when submitting the `submit-strong-omp.sh` script. Reason about the factors that limit the scalability of this parallel version, according to the understanding got with *Paraver*.

2.1 Task cut–off mechanism

Finally modify the parallelization of the *Tree* version in order to include a *cut–off* mechanism that controls the maximum recursion level for task generation, based on the use of the *OpenMP* `final` and `mergeable` clauses, to control the number of tasks generated (and their granularity). The mechanism should be

independent from the mechanism already included in the code to control the maximum recursion level. The `multisort-omp.c` file already accepts an optional command line argument (`-c cutoff`) thay you can use to externally provide from the command line a value for the recursion level that stops the generation of tasks. Once implemented generate a trace (using the `submit-omp-i.sh` script 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. level 0) and visualize the trace with *Paraver*. Try to understand what is visualized. Once you understand and are sure that the cut–off mechanism works, you can submit the `submit-cutoff-omp.sh` script to explore different values for the cut–off argument and observe if there is an optimum value for it. For that optimum value, analyze the scalability by looking at the two speed–up plots generated when submitting the `submit-strong-omp.sh` script.

Optional 1: Complete your scalability analysis for the *Tree* version on the other node types in boada. Remember that the number of cores is different in `boada-1 to 4`, `boada-5` and `boada-6 to 8` as well as the enablement of the hyperthreading capability. 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.

Optional 2: Complete the parallelization of the *Tree* version by parallelizing the two functions that initialize the `data` and `tmp` vectors[1]. Analyze 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 initialized 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.

Session 3

Using *OpenMP* task dependencies

Finally you will change the *Tree* parallelization in the previous chapter in order to express dependencies among tasks and avoid some of the `taskwait/taskgroup` synchronizations 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 the *Tree* version in `multisort-omp.c` to replace `taskwait/taskgroup` synchronizations by point–to–point task dependencies. Probably not all previous task synchronizations will need to be removed, only those that are redundant after the specification of dependencies among tasks.

2. Compile using the usual `multisort-omp` target in the `Makefile` to generate the executable file and submit it using the usual `submit-omp.sh` (executed using 8 processors). Make sure that the program verifies the result of the sort process and does not throw errors about unordered positions.

3. Once the parallel verifies, analyze its scalability by looking at the two speed–up plots (complete application and multisort only) generated when submitting the `submit-strong-omp.sh` script. Compare the results with the ones obtained in the previous chapter. Are they better or worse? Submit the `submit-omp-i.sh` script to trace its execution and use the appropriate configuration file to visualize how the parallel execution was done and to understand the performance achieved.

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, Parallelization 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, ...) 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 parallellization strategies and their implementation (i.e. for Tareador instrumentation and for all the OpenMP parallelization strategies).
- You also have to deliver the complete C source codes for Tareador instrumentation and all the OpenMP parallelization strategies that you have done. 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 Raco 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.

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. Comment also about the task graph generated and the causes of the dependences that appear.
2. Write a table with the execution time and speed-up predicted by *Tareador* (for 1, 2, 4, 8, 16, 32 and 64 processors) for the task decomposition specified with Tareador. Are the results close to the ideal case? Reason about your answer.

Parallelization and performance analysis with tasks

1. Include the relevant portion of the codes that implement the two versions (*Leaf* and *Tree*), commenting whatever necessary.
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 that is observed, including captures of *Paraver* windows to justify your explanations.
3. Show the changes you have done in the code in order to include a cut-off mechanism based on recursion level. Is there a value for the cut-off argument that improves the overall performance? Analyze the scalability of your parallelization with this value.

Parallelization and performance analysis with dependent tasks

1. Include the relevant portion of the code that implements the *Tree* version with task dependencies, commenting whatever necessary.

2. Reason about the performance that is observed, including the speed–up plots that have been obtained for different numbers of processors and with captures of *Paraver* windows to justify your reasoning.

Optional

1. If you have done any of the optional parts in this laboratory assignment, please include and comment in your report the relevant portions of the code and performance plots that have been obtained.