PAR Laboratory Assignment
Lab 2: Divide and Conquer parallelism with OpenMP: Sorting

E. Ayguadé, J. Corbalán, D. Jiménez,
J. I. Navarro, J. Tubella and G. Utrera

Fall 2015-16

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

1

”Divide and conquer”

Multisort is a sort algorithm which combines a ”divide and conquer” mergesort strategy that divides the initial list 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. This will report you the three parameters that you need to provide to sort a list of positive numbers randomly initialized: size of the list in Kiloelements and size in Kiloelements of the vectors that breaks the recursions during the sort and merge phases. For example `./multisort 32768 32 512`. The program randomly initializes the vector, sorts it and checks that the result is correct.

2. `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.

3. 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 `lab2.tar.gz` from `/scratch/nas/1/par0/sessions`.

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. 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. Copy the original sequential `multisort.c` (**NOT the Tareador instrumented version**) into `multisort-omp.c` 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` (executed using 8 processors). 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. Once the parallel version verifies its result, 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. Reason about the factors that may limit the scalability of this parallel version.
4. In order to verify your assumptions in the previous point, submit the `submit-omp-i.sh` script to trace the execution of your `OpenMP` program. We suggest that you make use of the following configuration files, available inside the `OpenMP/OMP_tasks` directory inside `cfgs`, to understand the behavior:

OpenMP events	**Timeline showing ...**
task_instantiation	when tasks are created
task_execution	when tasks are executed
useful_task_execution	useful time during task execution
taskid_instantiation	task identifier at instatiation time
taskid_execution	task identifier at execution time
taskwait	when taskwait synchronization constructs are executed
in_taskgroup	taskgroup regions
taskgroup	when taskgroup synchronization is executed

To finish with the analysis of the *Tree* version (not necessary for the *Leaf* version), submit the `submit-depth-omp.sh` script which performs a number of executions changing the recursion depth (size in Kiloelements that breaks the recursion) using 8 threads and generates a plot showing how the execution time changes with that parameter. Reasoning about the behavior observed in that plot.

Optional 1: 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.

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 synchronizations that you had to introduce in order to enforce task dependencies. 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 synchronizations by point–to–point task dependencies. Not all taskwait will 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.

Optional 2: Explore the best possible values for the sort_size and merge_size arguments used in the execution of the program. For that you can use the submit-depth-omp.sh script, modified to first explore the influence of one of the two arguments, select the best value for it, and then explore the other argument. Once you have these two values, modify the submit-strong-omp.sh script to obtain the new scalability plots.

4

Deliverables

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 name, the identifier of the group, date, ...) and, If necessary, include references to other documents and/or sources of information.

You also have to deliver the C source codes for Tareador instrumentation and all the OpenMP parallelization strategies for the *Leaf* and *Tree* strategies. 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 generic competence "Tercera llengua". Deliver all 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.

4.1 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.

4.2 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. Analyze the influence of the recursivity depth in the *Tree* version, including the execution time plot, when changing the recursion depth and using 8 threads. Reason about the behavior observed. Is there an optimal value?

4.3 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 different numbers of processors and with captures of Paraver windows to justify your reasoning.

4.4 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.