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

E. Ayguadé, J. R. Herrero, P. Martínez-Ferrer,
J. Morillo, J. Tubella and G. Utrera

Spring 2021-22

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Note:

- Each of chapters 2, 3 and 4 in this document roughly corresponds with a laboratory session (2 hours each). However, you can start the work in subsequent chapters once you have completed the preceeding chapters.
- All files necessary to do this laboratory assignment are available in a compressed tar file available from the following location: `/scratch/nas/1/par0/sessions/lab4modelfactors.tar.gz`. Uncompress it with this command line:
`"tar -zxvf ~par0/sessions/lab4modelfactors.tar.gz"`.

1

Before starting this laboratory assignment ...

Before going to the labroom to start this laboratory assignment, we strongly recommend that you take a look at this section and try to solve the simple questions we propose to you. This will help to better face your second programming assignment in OpenMP: the Merge Sort problem.

1.1 Recursive task decompositions

In this laboratory assignment we explore the use of parallelism in recursive programs. **Recursive task decompositions** are parallelisation strategies that try to exploit this parallelism. For example, consider the simple recursive program in Figure 1.1 computing the dot product of vectors `A` and `B`. Notice that the original problem of size `N` is solved by calling the recursive function `rec_dot_product`; this function recursively breaks down the problem (of size `n`) into two sub-problems of approximately the same size (`n/2`), until these become short enough to be solved directly by using the iterative function `dot_product`, which contributes to the result in shared variable `result`. Figure 1.2 shows the recursive "divide–and–conquer" solution that is done for `N=1024` and `MIN_SIZE=64`.

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

void dot_product(int *A, int *B, int n) {
   for (int i=0; i< n; i++) result += A[i] * B[i];
}

void rec_dot_product(int *A, int *B, int n) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       rec_dot_product(A, B, n2);
       rec_dot_product(A+n2, B+n2, n-n2);
   }
   else dot_product(A, B, n);
}

void main() {
   rec_dot_product(a, b, N);
}
```

Figure 1.1: Program performing dot product of vectors `a` and `b` using a recursive "divide–and–conquer" strategy.

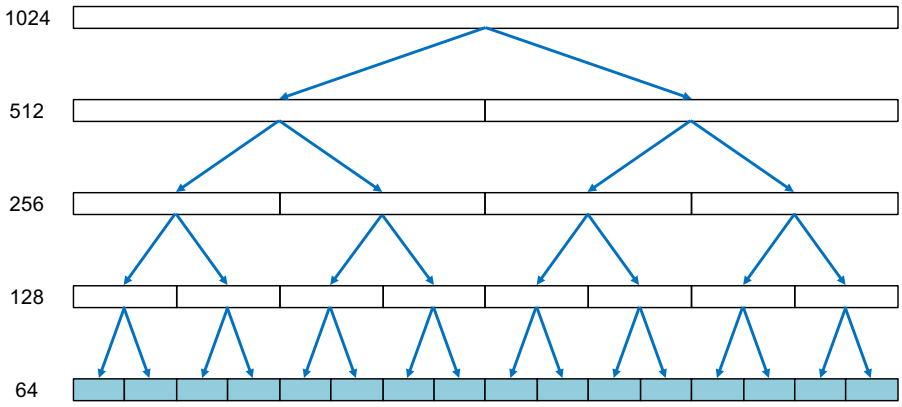

Figure 1.2: Divide–and–conquer approach for `N=1024, MIN_SIZE=64`. The initial problem is decomposed into two problems of size 512 in the first invocation of `rec_dot_product`, And then each one of these into two subproblemes of size 256. And again, into 128 and finally 64 (the value for `MIN_SIZE`). At this point the divide and conquer strategy is stopped.

How can we decompose a recursive problem like the one shown in Figure 1.2 in tasks? In a **recursive task decomposition** tasks correspond with the execution of one or more leaves of the recursion tree. The granularity of the task decomposition would be determined by the number of leaves executed per task. For this first part of the assignment, let's consider by now granularity one (later we will look at mechanisms to insert granularity control). Depending on how do we generate the tasks, we differentiate two different generation strategies: *Leaf* and *Tree* recursive task decomposition.

- In a *leaf recursive task decomposition* a new task is generated every time the recursion base case is reached (i.e. a leaf in the recursion tree is reached). In the example shown in Figure 1.1 this happens every time the condition `n>MIN_SIZE` is not true and the execution flows into the execution of function `dot_product`. Figure 1.3 shows the tasks that would be generated for the specific case of `N=1024, MIN_SIZE=64` shown in Figure 1.2.

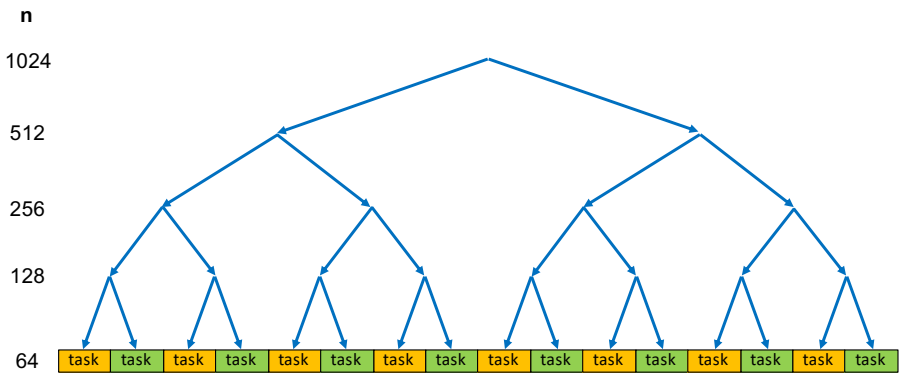

Figure 1.3: Leaf recursive task decomposition for `N=1024, MIN_SIZE=64`.

- In a *tree recursive task decomposition* a new task is generated every time a recursive call is performed (i.e. at every internal branch in the recursion tree). In the example shown in Figure 1.1 this happens every time the condition `n>MIN_SIZE` is true and the execution flows into the execution of the two recursive calls to function `rec_dot_product`; two tasks would be generated, one for each recursive call. When a leaf in the recursion tree is reached, it will be executed by the task itself. Figure 1.4 shows the tasks that would be generated for the specific case of `N=1024, MIN_SIZE=64` shown in Figure 1.2.

Make sure that you understand how tasks are generated in each recursive task decomposition. Is there a big difference in how the tasks are generated? Which is this difference? In other words, after how many

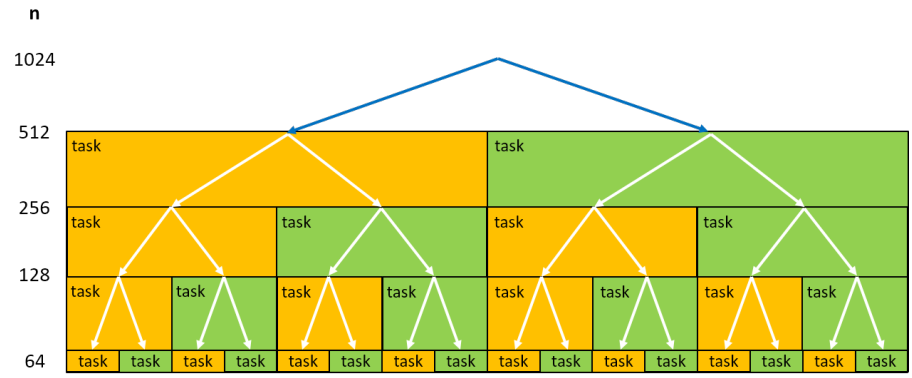

Figure 1.4: Tree recursive task decomposition for `N=1024, MIN_SIZE=64`.

steps is the last task generated in each strategy? Is the granularity for tasks executing invocations to `dot_product` the same in both strategies? How could you control the number of leaves a task reaches (executes)? For example, in a *Tree* recursive task strategy, how would you force a task to execute 2 leaves in the example in Figure 1.1? And 4? And in a *Leaf* recursive task strategy?

Finally, a recursive task decomposition is named "embarrassingly parallel" if the execution of all tasks can be performed totally in parallel without the need of satisfying data sharing and/or task ordering constraints. Is this the case for the dot product example shown in Figure 1.1? Of course not, some sort of synchronisation is required in order to avoid the possible data races caused by the access to variable `result`. But what if the code would have written as shown in Figure 1.5? Would you use the same kind of synchronisation?

```
#define N 1024
#define MIN_SIZE 64
int result = 0;

int dot_product(int *A, int *B, int n) {
   int result = 0;
   for (int i=0; i< n; i++) result += A[i] * B[i];
   return(result);
}

int rec_dot_product(int *A, int *B, int n) {
   int tmp1=0, tmp2=0;
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       tmp1 = rec_dot_product(A, B, n2);
       tmp2 = rec_dot_product(A+n2, B+n2, n-n2);
   }
   else tmp1 = dot_product(A, B, n);
   return(tmp1+tmp2);
}

void main() {
   int result = rec_dot_product(a, b, N);
}
```

Figure 1.5: Alternative version for the program performing dot product of vectors `a` and `b` using a recursive "divide–and–conquer" strategy.

1.2 Should I remember something from previous laboratory assignments?

In this assignment you will continue using the basic elements in *OpenMP* to express explicit tasks (mainly with `task` in this assignment), with the appropriate thread creation (`parallel` and `single`) and how to enforce task order with task barriers (`taskwait`, `taskgroup`) and task dependences (`depend` clause).

In addition you should remember from the first assignment how to use the *Tareador* API and GUI to understand the potential parallelism available in a sequential code, as well as the causes that limit this parallelism. You will visualise nested tasks, something that you have not practised before. And also the use of *Extrae* and *Paraver* to visualise the execution of your parallel *OpenMP* program and understand its performance.

1.3 So, what should I do next?

Would you be able to write the parallel version in *OpenMP* for these two strategies? We hope your answer is "Yes!, of course". Simply we ask you to think (better if you try to write in paper) the complete parallel versions for the two recursive task decompositions (*leaf* and *tree*) for the codes shown in Figures 1.1 and 1.5. You don't need to deliver them, but we will comment your different solutions in the lab session.

2

Task decomposition analysis for Mergesort

2.1 "Divide and conquer"

Mergesort is a sort algorithm which combines 1) a "divide and conquer" sort strategy, which divides the initial list (positive numbers randomly initialised) into multiple sublists recursively; 2) a sequential *quicksort*, which is applied when the size of these sublists is sufficiently small; and 3) a merge of the sublists back into a single sorted list. To start with, you should understand how the code `multisort.c` that we provide implements the "divide and conquer" strategy, recursively invoking functions `multisort` and `merge`.

1. Compile the sequential version of the program using `make multisort-seq` 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-seq -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.

Take note of the times reported for the sequential execution and use them as reference times to check the scalability of the parallel versions in *OpenMP* you will develop.

2.2 Task decomposition analysis with *Tareador*

Next you will investigate, using the *Tareador* tool, potential task decomposition strategies and their implications in terms of parallelism and task interactions required.

2. `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`.
3. 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 2048 -m 2048`) to generate a small task dependence graph in a reasonable amount of time.
4. From the task dependence graphs that are generated for *leaf* and *tree*, do you observe any major differences in terms of structure, types, number and granularity of tasks, ...

5. Continue the analysis of the task graphs generated in order to identify the task ordering constraints that appear in each case and the causes for them, and the different kind of synchronisations that could be used to enforce them.

6. Do simulated executions with 16 processors for each task decomposition. From the *Paraver* windows, do you notice any differences in terms of how and when tasks doing computation are generated? You will have to zoom at the appropriate parts of the trace in order to observe these differences.

Important: Please, refer to section 5.1 to make sure you have everything you need to include in the deliverable for this part of the laboratory assignment (task dependence graphs for both strategies and other information provided by *Tareador*, *Paraver* windows to support your explanations, ...).

3

Shared-memory parallelisation with *OpenMP* tasks

In this second section of this laboratory assignment you will parallelise the original sequential code in `multisort-omp.c` using OpenMP (**not** the `multisort-tareador.c` version), having in mind all the conclusions you gathered from your analysis with *Tareador*.

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**.

3.1 Leaf strategy in *OpenMP*

1. Insert the necessary *OpenMP* `task` for task creation with granularity one 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 increase (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, submit the execution with `sbatch` specifying 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 either looking at the two speed–up plots (complete application and multisort only) or the text files (`speedup1.txt` for complete application and `speedup2.txt` for 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 execution of the binary using `submit-strong-extrae.sh` script, which will trace the execution of the parallel execution with 1, 2, 4 and 8 processors with a much smaller input (`-n 1024 -s 256 -m 256`) and execute `modelfactors.py`. Which of the factors do you think is making the parallelisation efficiency so low? Several options to think about: parallel fraction, in-execution efficiency (related with the overheads of sync and sched reported in the third table), number of tasks and their execution time, load balancing, ... You could open with *Paraver* the trace generated for the execution with 8 processors. Is the program generating enough tasks to simultaneously feed all processors? How many tasks simultaneously execute?

3.2 Tree strategy in *OpenMP*

Repeat the previous steps for the alternative tree strategy. Compare the scalability results and traces generated for both strategies and draw the appropriate conclusions.

Important: Please, refer to section 5.2 to make sure you have everything you need to include in the deliverable for this part of the laboratory assignment (scalability plots for both strategies, *Paraver* windows to support your explanations, ...).

3.3 Task granularity control: the *cut–off* mechanism

Up to this point, in both strategies that you have implemented there is no control on the number of leaves in the recursion tree that are executed by computational tasks. Next we propose you include a *cut–off* mechanism in your OpenMP implementation for the *tree* recursive task decomposition in order to increase task granularity; the *cut–off* mechanism should allow you to control the maximum recursion level for task generation. Observe that the *cut–off* mechanism used to control task granularity should not be confused with the mechanism already included in the original sequential code to control the maximum recursion level (and controlled with the `-s` and `-m` optional flags in the execution command line for `multisort-omp`) that determines when the program branches to the recursion base case.

1. Modify the parallel code implementing the *tree* strategy to include a *cut–off* mechanism based on recursion level. To control the *cut–off* level from the execution command line, an optional flag (`-c value`) has been included, so that a `value` for the recursion level that stops the generation of tasks in the tree decomposition strategy can be provided at execution time. We recommend you use the `final` clause for `task` and `omp_in_final` intrinsic to implement your *cut–off* mechanism.

2. Once implemented submit again the `submit-strong-extrae.sh` script to just check the number of tasks generated when using different values for the cut–off level: 0, 1, 2, 4 and 8. You can change the value for the cut-off in the script itself by changing the variable `cutoff`. For the values of 0 and 1 check that the number of tasks generated is what you expected to be. You could open the trace generated for example with 8 processors to see the tasks generated and executed, although it may not be necessary. Look also at the information reported by `modelfactors.py`, looking at how the execution time (first table), the number of tasks and their execution time and associated overheads change with the value of the cutoff (third table), as well al the load balancing and in–execution metrics (second table).

3. Finally 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 specifying as argument the number of processors to use. The script internally explores different values for the *cut–off* argument, allowing recursion to go to deeper levels (`-n 32768 -s 128 -m 128`). The number of processors is the only argument required by the script. Once executed you can check how the execution time varies with cut-off, by either by displaying the PostScript file generated or by checking the values in `elapsed.txt`. Which is the best value for cut-off? Does it change with the number of processors used?

Important: Please, refer to section 5.3 to make sure you have everything you need to include in the deliverable for this part of the laboratory assignment (execution time and scalability plots, *Paraver* windows to support your explanations, ...).

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`? Set the maximum number of cores to be used (variable `np_NMAX`) by editing the `submit-strong-omp.sh` script in order to do the complete analysis.

4

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, with the level of cut-off that you consider appropriate from the previous section. 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.

Important: Please, refer to section 5.4 to make sure you have everything you need to include in the deliverable for this part of the laboratory assignment (scalability plots, *Paraver* windows to support your explanations, ...).

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.

5

Deliverable

You have to write a short document that presents the main results and conclusions that you have obtained when doing this assignment. Only PDF format for this document will be accepted. In the following subsections we highlight the aspects that you should included in the document. Your explanations should be based on the relevant parts of the output provided by `modelfactors.py`, execution timelines and other *Paraver* windows, as well as scalability plots generated by the execution scripts.

- The document should have an appropriate structure, including, at least, the following sections: Introduction, Parallelisation strategies, Performance evaluation and some Final 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.
- **Important:** You DON'T have to include in the document all the steps you have followed during the laboratory sessions, ONLY the main results and conclusions derived from them.
- 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.

Your professor will open the assignment in *Atenea* and set the appropriate dates for the delivery. You will have to deliver TWO files, one with the report in PDF format and one file compressed file (`tgz`, `.gz` or `.zip`) with the requested C source codes.

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.

5.1 Analysis with *Tareador*

Briefly describe the structure of the *Task Decomposition Graphs (TDG)*, and the types, number and granularity of the tasks for each of the two recursive task decomposition strategies considered. You should also highlight any difference you notice between the two recursive task decomposition strategies in terms of how and when tasks are generated. And finally, comment on 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.

5.2 Parallelisation and performance analysis with tasks

For the the *leaf* and *tree* strategies with no *cut–off* mechanism, include the speed–up (strong scalability) plots/tables that have been obtained for the different numbers of processors. Reason about the performance differences that you detect between *leaf* and *tree* implementations, including tables, or relevant parts of tables, generated by `modelfactors.py` that help you to explain and justify the differences between them. Explain what is causing the different scalability that is obtained for the whole program and for the `multisort` function only.

5.3 Controlling task granularities: cut–off mechanism

Include the execution time plots/tables 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 and analyse the scalability of your parallelisation with this value. Include any *Paraver* window or exit from `modelfactors.py` that helps you to explain how your *cut–off* mechanism is working.

5.4 Parallelisation and performance analysis with task dependencies

Regarding programmability, was the parallel version using task dependencies easier to think and/or simpler to code? Reason your answer. 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 or output from `modelfactors.py` to justify your explanation.

5.5 Optional

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 and output from `modelfactors.py`.