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 2020-21

Index

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`. Observe 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.

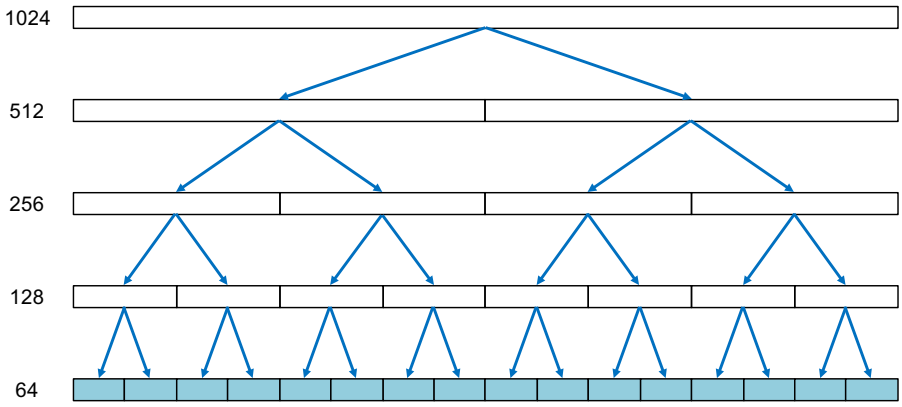

Figure 1.2: Divide–and–conquer approach for `N=1024, MIN_SIZE=64`.

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.

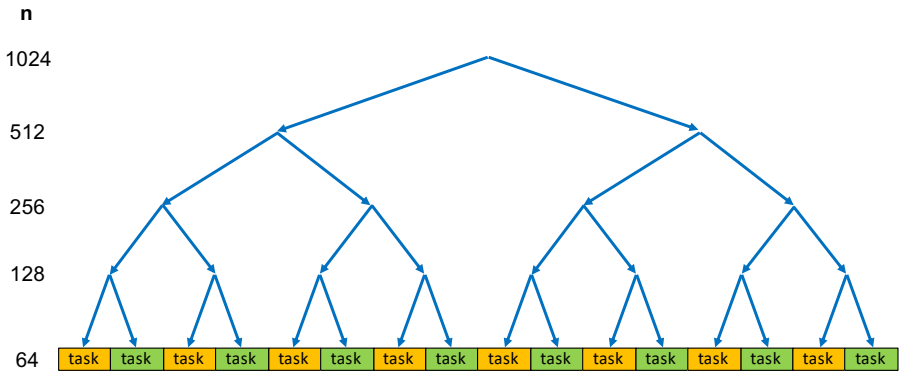

Figure 1.3: Leaf recursive task decomposition for `N=1024, MIN_SIZE=64`.

- 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 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

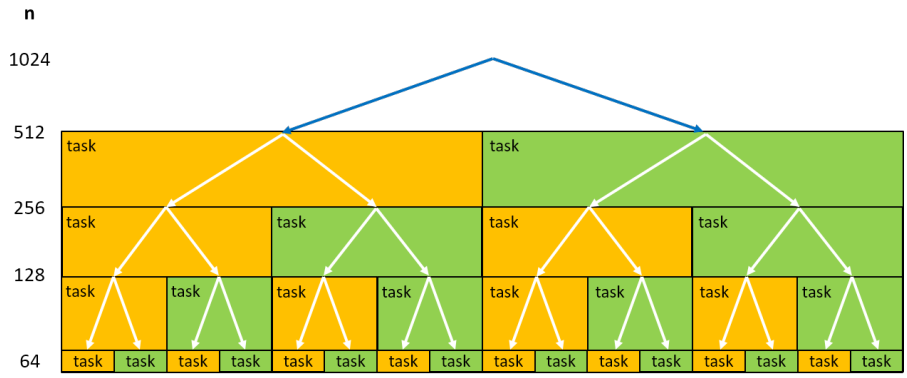

Figure 1.4: Tree recursive task decomposition for `N=1024, MIN_SIZE=64`.

(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?

Would you be able to write the parallel version in *OpenMP* for these two strategies? If your answer is "Yes!, of course", then you can skip these two paragraphs. 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?

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`[1] 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.

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

4. From the task dependence graphs that are generated for *leaf* and *tree*, 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).

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. Do you observe any differences in terms of how and when tasks doing computation are generated? You will have to zoom at the appropriate parts in the trace in order to observe these differences. Capture the necessary Paraver windows in order to support your explanations in the deliverable.

3

Shared-memory parallelisation with *OpenMP* tasks

In this second section of this laboratory assignment you will parallelise the original sequential code in `multisort.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**.

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 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 execution of the binary using the the `submit-extrae.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 you should already be familiar with 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? How many tasks simultaneously execute? ...

3.1 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 you will 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.

The *cut–off* mechanism that is 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.

5. 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.

6. Once implemented generate a trace (using the `submit-extrae.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 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.

8. For that optimum *cut–off* 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 the value for `cutoff` to the optimum value you have found in the previous 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`? 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.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.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 when doing the assignment. In the following subsections we highlight the main aspects that should be included in your document. Only PDF format for this document 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.

Also very important: 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 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.

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. Comment any difference you observe between the two recursive task decomposition strategies in terms of how and when tasks are generated. 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.

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 and 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.