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

Mario Acosta, Eduard Ayguadé, Rosa M. Badia (Q2), Jesus Labarta,
Josep Ramon Herrero (Q1), Daniel Jiménez-González, Pedro Martínez-Ferrer, Adrian Munera,
Jordi Tubella and Gladys Utrera

Fall 2023-24

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Index

Index **1**

1 Before starting this laboratory assignment ... **2**
1.1 Recursive task decompositions 2
1.2 Should I remember something from previous laboratory assignments? 5
1.3 So, what should I do next? 5

2 Task decomposition analysis for Mergesort **6**
2.1 "Divide and conquer" 6
2.2 Task decomposition analysis with *Tareador* 6

3 Shared-memory parallelisation with *OpenMP* tasks **8**
3.1 Leaf strategy in *OpenMP* 8
3.2 Tree strategy in *OpenMP* 9
3.3 Task granularity control: the *cut–off* mechanism 9

4 Shared-memory parallelisation with *OpenMP* task using dependencies **11**

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/lab4.tar.gz`. Uncompress it with this command line:
`"tar -zxvf ~par0/sessions/lab4.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.

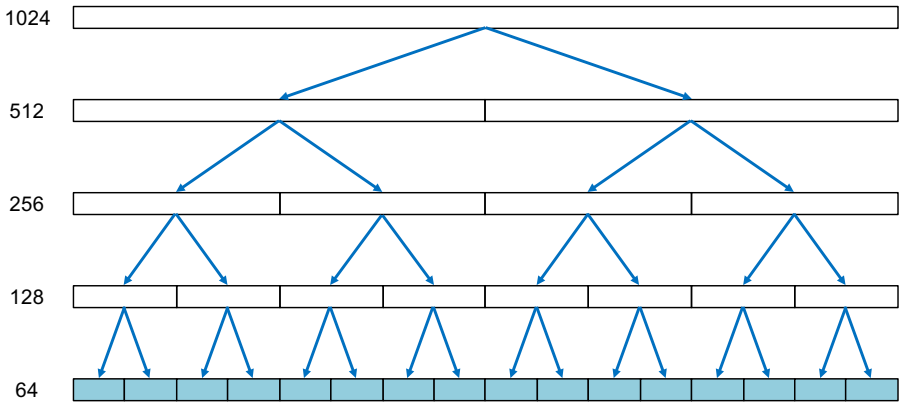

Figure 1.2: Divide–and–conquer approach for `N=1024, MIN_SIZE=64`. The initial problem is decomposed into two problems of size 512 in the first invocation of `rec_dot_product`, And then each one of these into two subproblemes of size 256. And again, into 128 and finally 64 (the value for `MIN_SIZE)`.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

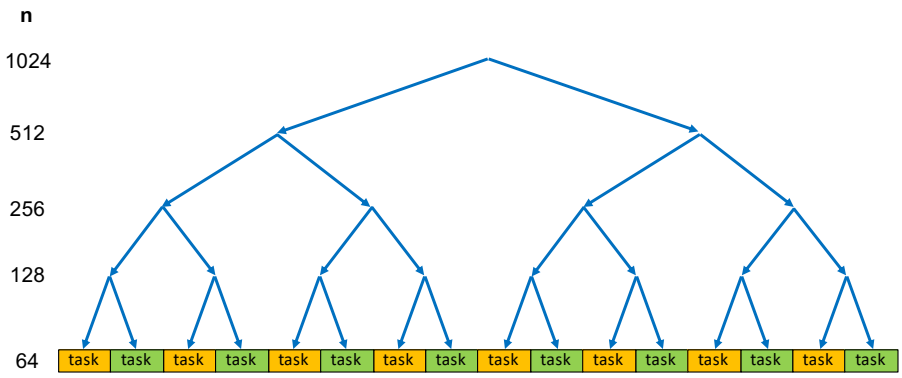

Figure 1.3: Leaf recursive task decomposition for `N=1024, MIN_SIZE=64`.

- 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.

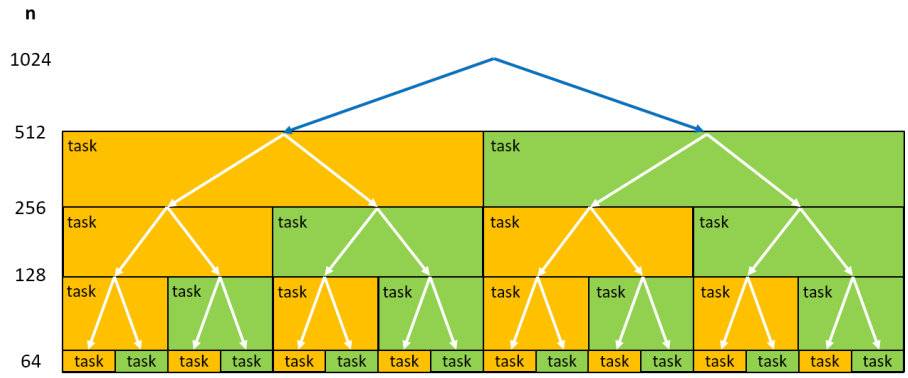

Figure 1.4: Tree recursive task decomposition for `N=1024, MIN_SIZE=64`.

Leaf vs Tree Recursive Strategies

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?

Discussion with your professor: Yes! In the case of *Leaf* strategy, tasks are sequentially generated by the master thread, that sequentially traverses the recursive tree. On the oher hand, in the case of *Tree* strategy, tasks are generated in parallel in each recursive tree level; reaching case base in $\log_2(n)$ steps. Therefore, the task granularity is different in the two parallel strategies.

Task granularity: Cut–off control

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?

Discussion with your professor: At class we have seen how to control the granularity of the two versions by controlling the recursive depth level (you should add a new parameter to the function). In the case of *Tree* strategy, we create tasks at each recursive call until the CUTOFF depth level is reached. Once this level is reached, following recursive calls are done sequentially and no more tasks are generated. In the case of *Leaf* strategy, we only create tasks when the depth level is equal to the CUTOFF depth level or the recursive base case is reached and no tasks has been created yet.

Data Sharing and Syncrhonizations

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?

Discussion with your professor: In the case of the shared variable `result` version, `atomic, critical or locks` can be used. However, these mechanisms would introduce too much overheads if you use them in each body loop iteration. We saw how to reduce that by using a temporal variable within the body loop and then doing a `atomic` after the loop. In the case of Figure 1.5, one way is to use `taskwait or taskgroup` mechanims in order to wait for tasks to be finished. Remember that you should make `shared` variables `tmp1` and `tmp2`. Another way is to use `depend` clauses to allow the runtime to take care of the dependences between tasks.

```
#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 *Paraver* to visualise the execution of your parallel *OpenMP* program and understand its performance after running *modelfactors*.

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. Run `sbatch submit-seq.sh multisort-seq` to execute multisort-seq. Script `submit-seq.sh` compiles the sequential version of the program using `make multisort-seq` and executes `./multisort-seq -n 1024 -s 256 -m 256`. The program randomly initialises the vector, sorts it and checks that the result is correct. The three command–line arguments (all of them power–of–two) indicates: 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).

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`.

   For the deliverable: Include both tareador codes (in the .zip file).

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.

   For the deliverable: Save the TDF graphs generated by *Tareador* to include it in the deliverable.

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

For the deliverable: Identify the points of the code where you should include synchronizations to guarantee the dependences for each task decomposition strategy. Which directives/annotations/clauses will you use to guarantee those dependences in the *OpenMP* implementations?

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, run `sbatch submit-omp.sh multisort-omp number_of_threads` specifying a small number of threads (for example 2 or 4) with the default input parameters. 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 (`speedup-global-*.txt` for complete application and `speedup-partial-*.txt` for multisort only) generated when submitting the `submit-strong-omp.sh` script. This script executes the binary with a number of threads in the range 1 to 20 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. If you are not convinced with the performance results you got (by the way, you should not!), let's analyze it with *modelfactors*.

Analysis with *modelfactors* and *Paraver*

In order to evaluate the overall performance of the `leaf` strategy, we ask you to:

4. Run `sbatch submit-strong-extrae.sh`, which will trace the execution of the parallel execution with 1, 4, 8, 12 and 16 threads with the same sequential input sizes (`-n 1024 -s 256 -m`

256) and execute `mfLite.py` to generate three tables that can help you to analyse the performance and scalability of your parallel program. Those execution traces and tables can be found in `multisort-omp-strong-extrae` directory as explained in `lab1`.

For the deliverable: Include the three tables generated. 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, ...

5. Open with *Paraver* the trace generated for the execution with 8 threads running:

```
wxparaver multisort-omp-strong-extrae/multisort-omp-8-boada-??-cutter.prv
```

and from the *Paraver* menu, load Hints, Useful, **instantaneous parallelism** and Hints, OpenMP tasking, **Explicit Task Functions Created and Executed** configurations in order to see how many tasks are simultaneously executed and which thread/s create/s and execute/s them. You will need to do zoom in the *Paraver* traces.

For the deliverable: Include parts (zoom in) of the *Paraver* visualisations that help you explain the lack of scalability. In particular, we think it would be good to show those parts that show examples of: 1) the amount of task executed in parallel, 2) which threads are executing tasks and 3) which thread/s is/are creating tasks?. Then, you may comment/answer the following questions: Is the program generating enough tasks to simultaneously feed all threads? How many? How many threads are creating tasks?

3.2 Tree strategy in *OpenMP*

Repeat the previous steps for the alternative tree strategy: implementation, check correctness, overall analysis with *modelfactors* and detail analysis with *Paraver* with the same Hints commented above.

For the deliverable: Include the three tables generated with *modelfactors*. Which of the following options do you think is making the parallelisation better than the leaf version? 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, ...

For the deliverable: Include sections of the *Paraver* visualisations that help you explain the lack of scalability. In particular, we think it would be good to show those parts that show examples of: 1) the amount of task executed in parallel, 2) which threads are executing tasks and 3) which thread/s is/are creating tasks?. Then, you may comment/answer the following questions: Is the program generating enough tasks to simultaneously feed all threads? How many? How many threads are creating tasks?

For the deliverable: On the other hand, the tree strategy is not showing a good parallel performance neither. Is the granurality of both (*leaf* and *tree*) strategies influencing the parallel performance (see overheads in *modelfactor* tables)? What is the number of tasks created (see *modelfactor* tables)?

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.

Implementation

1. Edit 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. Run `sbatch ./submit-omp.sh multisort-omp num_of_threads` for 1 and 8 threads to validate the correctness of the new version of the program.

3. Explore values for the *cut–off* level depending on the number of threads used. For that you can submit the `submit-cutoff-omp.sh` script specifying as argument the number of threads to use. We recommend you to use 8 threads. The script internally explores different values for the *cut–off* argument, allowing recursion to go to deeper levels. The problem input sizes by default are `-n 1024 -s 256 -m 256`. The number of threads 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`.

   For the deliverable: Include the postscript generated. Which is the best value for cut-off for this problem size and 8 threads?

Overall Analysis

In this section we propose you to analyze the application to find out the best cut–off level under the point of view of performance of the application.

4. 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.

5. Look at the information reported by the three tables of *modelfactors*: 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).

   For the deliverable: Include the tables of *modelfactors* for cut–off level equal to 4 and the information of the number of tasks generated for each of the cut–off levels used. Does it significantly change with the number of threads used? Which is the best value for cut-off?

6. Open with *Paraver* the trace generated for the execution with 8 threads (and cut–off level equal to 4) running:

   ```
   wxparaver multisort-omp-strong-extrae/multisort-omp-8-boada-??-cutter.prv
   ```

   and from the *Paraver* menu, load Hints, Useful, **instantaneous parallelism** and Hints, OpenMP tasking, **Explicit Task Functions Created and Executed** configurations in order to see how many tasks simultaneously execute and which thread/s create/s and execute/s them.

   For the deliverable: Include the *Paraver* trace you have analyzed. Is the instantaneous parallelism achieved larger than one along all the execution trace?

7. Submit the `submit-omp-strong.sh` script using the best *cut–off* you have obtain in previous point. You can change the value for the cut-off in the script itself by changing the variable `cutoff`.

For the deliverable: Include the strong scalability plot generated for the best cut–off level. Are the speedup for the complete and `multisort` function close to the speed-up ideal? Reason the difference in speedup based on *Paraver* trace you have analyzed with the instantaneous parallelism hint and the parallelization of your `multisort-omp.c` code.

4

Shared-memory parallelisation with *OpenMP* task using 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]`.

Implementation

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 threads, 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.

Analysis

3. Analyse its scalability by looking at the 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. Submit the execution of the binary using `submit-strong-extrae.sh` script, which will trace the execution of the parallel execution with 1, 4, 8, 12 and 16 threads.

5. Open with *Paraver* the trace generated for the execution with 8 threads and load Hints, Useful, **instantaneous parallelism** configuration in order to see how many tasks simultaneously execute.

For the deliverable: Include the tables of *modelfactors*, scalability plots and *Paraver* windows to support the comparison. Are they better or worse compare to *OpenMP* versions of previous chapter in terms of performance? In terms of programmability, was this new version simpler to code?