An Approach to Task-based Parallel Programming for Undergraduate Students

Eduard Ayguadé[a,b,*], Daniel Jiménez-González[a,b]

[a] *Computer Architecture Department, Universitat Politècnica de Catalunya (UPC)*
[b] *Computer Sciences Department, Barcelona Supercomputing Center (BSC-CNS)*

Abstract

This paper presents the description of a compulsory parallel programming course in the bachelor degree in Informatics Engineering at the Barcelona School of Informatics, Universitat Politècnica de Catalunya UPC–BarcelonaTech. The main focus of the course is on the shared-memory programming paradigm, which facilitates the presentation of fundamental aspects and notions of parallel computing. Unlike the “traditional” loop-based approach, which is the focus of parallel programming courses in other universities, this course presents the parallel programming concepts using a task-based approach. Tasking allows students to explore a broader set of parallel decomposition strategies, including linear, iterative and recursive strategies, and their implementation using the current version of OpenMP (OpenMP 4.5), which offers mechanisms (pragmas and intrinsic functions) to easily map these strategies into parallel programs. Simple models to understand the benefits of a task decomposition and the trade-offs introduced by different kinds of overheads are included in the course, together with the use of tools that allow an easy exploration of different task decomposition strategies and their potential parallelism (*Tareador*) and instrumentation and analysis of task parallel executions on real machines (*Extrae* and *Paraver*).

Keywords: Task decomposition strategies and programming, OpenMP tasking model, Performance models and tools

*Corresponding author
Email address: `eduard@ac.upc.edu` (Eduard Ayguadé)

1. Introduction

For decades, single-core processors were steadily improving in performance thanks to advances in integration technologies (bringing more transistors and ever-increasing clock speeds) and micro-architectural innovations (providing higher potential instruction-level parallelism, or ILP). The target's ILP could be satisfactorily exploited by the compiler, and sequential programming was the dominant paradigm. Programming courses for undergraduate students were based on this sequential paradigm, without the need for programmers to learn to consider parallelism. Concurrency was mainly presented in operating system (OS) courses as a way to express the concurrent execution of multiple activities, such as processes and/or threads, inside the OS. Parallel computing was a subject mainly considered in courses at the most advanced levels of computer science and engineering curricula.

This sequential paradigm was challenged by the move towards multicore architectures, caused by the power wall (due to ever-increasing clock frequencies) and increasing difficulties in exploiting the available ILP. Today, from mobile to desktops to laptops to servers, multicore processors and multiprocessor systems are commonplace. In order to utilise the increasing number of available cores, it is necessary to parallelise existing sequential applications. Unfortunately, neither hardware nor current compilers can automatically detect and exploit the levels of parallelism required to feed current parallel architectures.

Due to the increasing demand in the IT sector for parallel programming expertise, efforts have been made to introduce parallel programming to undergraduate students. In most cases the design of these parallel programming courses stayed rooted in "traditional" regular loop-level parallelisation strategies, not allowing parallelism to be exploited in more irregular applications, such as those traversing dynamically-allocated data structures (lists, trees, etc.) and making use of other control structures, such as recursion. In addition, it has been proven, both by the research community and through the evolution of parallel programming standards, that this "traditional" approach is not sufficient to pave the path towards exploiting the potential scalability of future processor generations and architectures. To provide an alternative to the loop-based approach, some programming models and standards (such as OpenMP) evolved to include the tasking model. The task-based approach offers a means to express irregular parallelism, in a top down manner, that scales to large numbers of processors.

In this paper we present the proposed syllabus and framework for teaching parallel programming to "fresh" students in *Parallelism*, a third-year compulsory subject in the Bachelor Degree in Informatics Engineering at the Barcelona School of Informatics (FIB) of the Universitat Politècnica de Catalunya (UPC–BarcelonaTech). This subject has been our first opportunity to teach parallelism at the undergraduate level. The tasking model in OpenMP [1] (currently version 4.5 for C/C++) was chosen as the vertebral axis in the design of this course, providing support for tasks (including task dependences) in addition to traditional loop-level parallelism, which is considered to be a particular case of the generic tasking model. The course also includes models and tools to understand the potential of task decomposition strategies (*Tareador* [2]) as well as to understand their actual behaviour when expressed in OpenMP and executed on a real parallel architecture (*Extrae*, a dynamic tracing package, and *Paraver*, a trace visualisation and analysis tool [3]). The complete framework motivates the learning process, improves the understanding of the proposed task decompositions and significantly reduces the time to develop parallel implementations of the original sequential codes.

The paper is organised as follows: Section 2 presents the context for the subject presented in this paper. Then, Sections 3, 4, 5 and 6 describe the main units in the subject, in terms of concepts and methodology. Finally, Section 7 concludes the paper by analysing how the proposed subject covers the main topics identified in the *NSF/IEEE-TCPP Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates*, and how the gradual evolution from a traditional loop-based course has improved the students' results.

2. Course description and context

The bachelor degree in Informatics Engineering at the Barcelona School of Informatics of the Universitat Politècnica de Catalunya is designed to be completed in seven terms (two terms per academic year) plus one term for a final project. The four initial terms cover subjects that are mandatory for all students, while the three final terms comprise mandatory and elective courses within one specialisation (computer engineering, networks, computer sciences and software engineering).

Parallelism (PAR) is the first subject in the above-mentioned degree that teaches parallelism, and it is the one described in detail in this paper. It is a compulsory subject, in the fifth term, that covers parallel programming and parallel computer architecture fundamentals—basic tools to take advantage of the multi-core architectures that constitute today's computers. The subject follows a series of subjects on computer organisation and architecture, operating systems, programming and data structures, all of which are focussed on uniprocessor architectures and sequential programming.

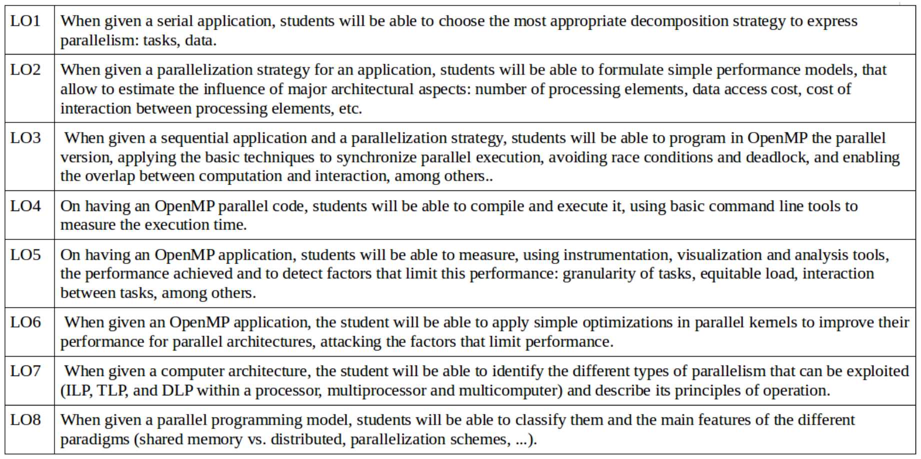
2.1. Learning objectives and student learning outcomes

The three main learning objectives of PAR are the following: (1) to design, implement and analyse parallel programs for shared-memory parallel architectures; (2) to write simple models to evaluate different parallelisation strategies and understand the trade-off between parallelism and the overheads of parallelism; and (3) to gain an understanding of the architectural support for parallel programming models (data sharing and synchronisation).

The expected student learning outcomes for PAR are summarised in Figure 1; these learning outcomes are related to the different theory/laboratory sessions shown in Table 1 and described in the next subsection.

2.2. Complementary courses

Two elective subjects in the specialisation of Computer Engineering follow PAR. First, *Parallel Architectures and Programming* (PAP) extends the



LO1	When given a serial application, students will be able to choose the most appropriate decomposition strategy to express parallelism: tasks, data.
LO2	When given a parallelization strategy for an application, students will be able to formulate simple performance models, that allow to estimate the influence of major architectural aspects: number of processing elements, data access cost, cost of interaction between processing elements, etc.
LO3	When given a sequential application and a parallelization strategy, students will be able to program in OpenMP the parallel version, applying the basic techniques to synchronize parallel execution, avoiding race conditions and deadlock, and enabling the overlap between computation and interaction, among others..
LO4	On having an OpenMP parallel code, students will be able to compile and execute it, using basic command line tools to measure the execution time.
LO5	On having an OpenMP application, students will be able to measure, using instrumentation, visualization and analysis tools, the performance achieved and to detect factors that limit this performance: granularity of tasks, equitable load, interaction between tasks, among others.
LO6	When given an OpenMP application, the student will be able to apply simple optimizations in parallel kemels to improve their performance for parallel architectures, attacking the factors that limit performance.
LO7	When given a computer architecture, the student will be able to identify the different types of parallelism that can be exploited (ILP, TLP, and DLP within a processor, multiprocessor and multicomputer) and describe its principles of operation.
LO8	When given a parallel programming model, students will be able to classify them and the main features of the different paradigms (shared memory vs. distributed, parallelization schemes, ...).

Figure 1: Student's Learning Outcomes (LO) for PAR.

concepts and methodologies introduced in PAR, by focussing on the low-level aspects of implementing a programming model such as OpenMP, making use of low-level threading (*Pthreads*); the subject also covers cluster architectures and how to program them using MPI. Second, *Graphical Units and Accelerators* (TGA) explores the use of accelerators, with an emphasis on GPUs, to exploit data-level parallelism.

PAR, PAP and TGA are complemented by a compulsory course in the Computer Engineering specialisation, *Multiprocessor Architectures*, in which the architecture of (mainly shared-memory) multiprocessor architectures is covered in detail. Another elective subject in the same specialisation, *Architecture-aware Programming* (PCA), mainly covers programming techniques for reducing the execution time of sequential applications, including through SIMD vectorisation and FPGA acceleration.

Week	Theory/problem solving		Laboratory		Learning Outcomes (LO)
	Topic	**Session (2h)**	**Topic**	**Session (2h)**	
1	Fundamentals	Motivation. Serial, multiprogrammed, concurrent and parallel execution	Environment	Compilation and execution of programs	LO1,4
2		Abstract program representation (TDG). Simple performance models and overheads.		Tools: Tareador	LO1
3		Amdahl's law. Strong vs. weak scalability		Tools: Paraver and Extrae	LO2,5,6
4		Wrap-up and exercises	OpenMP tutorial	Parallel and work–sharing	LO1,2
5	Task decomposition	Linear, iterative and recursive. Task granularities.		Tasking execution model	LO3
6		Task ordering vs. data sharing constraints	Model analysis	Evaluation of overheads	LO3,6
7		Wrap-up and exercises	Embarrassingly Parallel	Design	LO1,3
8	More advanced exercises covering decomposition strategies and task ordering / data sharing constraints			Implementation and analysis	LO1-6
9	1st Midterm Evaluation				
10	Architecture support for shared memory programming	How data is shared among processors?	Divide and conquer	Design	LO1,3,7,8
11		How are processors able to synchronise?		Implementation	LO3
12		Wrap-up and exercises		Analysis	LO1,3,7,8
13	Data decomposition	Strategies to improve data locality: think about data. Owner-computes rule	Geometric decomposition	Design	LO1,3
14		Why sharing data? Distributed memory and MPI		Implementation	LO8
15		Wrap-up and exercises		Analysis	LO1,3,7,8
	2nd Midterm Evaluation				

Table 1: Weekly course outline and student learning outcomes.

2.3. Course outline

Each term effectively lasts for 15 weeks. In PAR there are four contact hours per week: two hours devoted to theory and problems (with a maximum of 60 students per class) and two hours for laboratory sessions (with a maximum of 15 students per class). Students are expected to invest about six additional hours per week to complete homework and for personal study (over these 15 weeks). Thus, the total effort devoted to the subject is six ECTS credits.[1]

Table 1 shows the main contents of PAR and their weekly distribution in theory/problem and laboratory sessions. After an introductory unit motivating the course and presenting the differences between sequential, multiprogrammed, concurrent and parallel execution, PAR continues with four units that cover the objectives of the course: fundamentals of parallelism (described in Section 3), task decomposition strategies (described in Section 4), introduction to parallel computer architectures (described in Section 5) and data decomposition strategies (described in Section 6).

Theory/problem contact classes follow the flipped classroom methodology: before class students complete one or more interactive learning modules that include videos explaining the main concepts, and during the class students apply the key concepts and extend them to more complex concepts. Finally, after class, students check their understanding and extend their learning to more complex tasks. In addition, there are several wrap-up sessions to help the learning process, and there are two midterm exams. As shown in Table 1, several laboratory sessions are coordinated with the theory and problem contact classes.

The structure of the course and some of its main concepts are based on two books: *Patterns for Parallel Programming* [4] and *Introduction to Parallel Computing* [5]. The latest OpenMP specification [1] is also used as reference

[1]The European Credit Transfer System (ECTS) is a unit of appraisal of the academic activity of the student. It takes into account student attendance at lectures and time of personal study, exercises, labs and assignments, together with the time needed to do examinations. One ECTS credit is equivalent to 25–30 hours of student work.

material. Finally, *Computer Architecture: a Quantitative Approach* [6] is recommended as complementary material.

3. The fundamentals

After introducing the differences between serial, multiprogrammed, concurrent and parallel execution, the subject starts by presenting an abstract representation for task-based parallelisation strategies: the *task dependence graph* (TDG), which allows an analysis of the parallelism of a particular decomposition into tasks. The TDG is a directed acyclic graph in which each node represents a task, which is an arbitrary sequential computation, and each directed edge represents a data dependence relationship between the predecessor and successor tasks. The weight of a node represents the amount of work to be done in the task. For illustration purposes, the left part of Figure 2 shows a simple TDG.

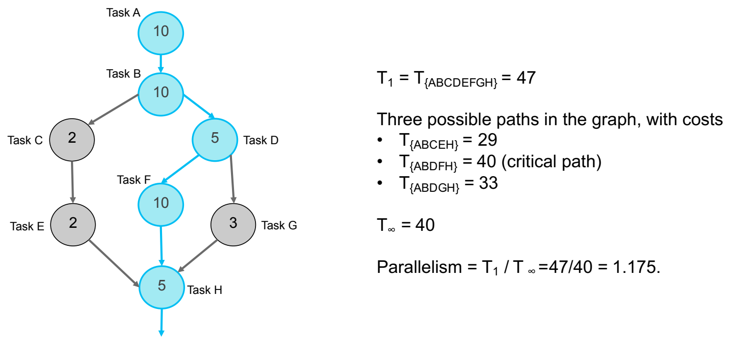

Task A
10
Task B
10
Task C
2
5
Task D
Task F
10
Task E
2
3
Task G
5
Task H
$T_1 = T_{\{ABCDEFGH\}} = 47$
Three possible paths in the graph, with costs
• $T_{\{ABCEH\}} = 29$
• $T_{\{ABDFH\}} = 40$ (critical path)
• $T_{\{ABDGH\}} = 33$
$T_\infty = 40$
Parallelism = T_1 / T_∞ =47/40 = 1.175.

Figure 2: Left: Task Dependence Graph (TDG) example, with nodes annotated with task execution cost (in blue nodes that compose the critical path in the TDG). Right: computation of T_1, T_∞ and *Parallelism* metrics for the TDG on the left.

With this abstraction of the task decomposition and a simplified machine abstraction that assumes identical processors, each processor executing one task at a time, the student is presented with the *parallelism* metric, defined as the quotient between T_1, the time to execute all the nodes in the TDG on a single processor and T_∞, the time to execute the critical path in the TDG with infinite processors and resources:

- $T_1 = \sum_{i=1}^{nodes}(work\_node_i)$
- $T_\infty = \sum_{i \in criticalpath}(work\_node_i)$
- $Parallelism = T_1/T_\infty$

The right part of Figure 2 shows the computation of these metrics: (a) T_1, defined above, (b) $T_{\{list\}}$, the execution time of each path *list* from the top node to the bottom node, (c) T_∞, which equals the execution time of the largest path $T_{\{ABDFH\}}$, and (d) the *parallelism* metric. The *parallelism* metric of 1.175 indicates that a parallel execution of this task decomposition can execute up to 1.175 times faster than sequential if sufficient (e.g. infinite) resources are made available.

In order to perform the aforementioned TDG analysis, the student is presented with the question of how to define the scope of a task, how to figure out the dependences among tasks, and the *granularity* concept (size of each node in the TDG). This is done using simple codes. For example, Figure 3 shows a simple *Jacobi* relaxation computation code in C (top) and different task granularities to be considered (bottom). In this case, any task definition leads to a fully independent set of tasks, since there are no data dependencies among computations in different iterations of the innermost loop. By analysing T_∞ and the *Parallelism* metrics, the student can understand the concept of granularity and extract a first (premature) conclusion that could lead to an interesting discussion: finer-grain tasks are able to attain more parallelism.

The previous conclusion favouring fine-grain tasks (at the top) is dramatically changed once overheads are brought into consideration. The students are introduced to the three main sources of overhead: task creation, task synchronisation and data sharing.

3.1. Task granularity vs. task creation overhead

At this point, it is appropriate to introduce the effect of the task creation overhead, resulting in a trade-off between the granularity of the tasks and the parallelism that can be obtained when those overheads are considered. For

```
void compute(int n, double *u, double *utmp) {
int i, j;
double tmp;

for (i = 1; i < n-1; i++)
  for (j = 1; j < n-1; j++) {
    tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
          u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
          4 * u[n*i + j];                    // element u[i][j]
    utmp[n*i + j] = tmp/4;                   // element utmp[i][j]
  }
}
```

Task is … (granularity)	T_1	T_∞	**Parallelism**	**Task creation ovh**
All iterations of i and j loops	$n^2 \cdot t_{body}$	$n^2 \cdot t_{body}$	1	t_{create}
Each iteration of i loop	$n^2 \cdot t_{body}$	$n \cdot t_{body}$	n	$n \cdot t_{create}$
Each iteration of j loop	$n^2 \cdot t_{body}$	t_{body}	n^2	$n^2 \cdot t_{create}$
r consecutive iterations of I loop	$n^2 \cdot t_{body}$	$n \cdot r \cdot t_{body}$	$n \div r$	$(n \div r) \cdot t_{create}$
c consecutive iterations of j loop	$n^2 \cdot t_{body}$	$c \cdot t_{body}$	$n^2 \div c$	$(n^2 \div c) \cdot t_{create}$
A block of r x c iterations of i and j, respectively	$n^2 \cdot t_{body}$	$r \cdot c \cdot t_{body}$	$n^2 \div (r \cdot c)$	$(n^2 \div (r \cdot c)) \cdot t_{create}$

Figure 3: Jacobi relaxation example (top) and different task granularities to be explored (bottom). The number of iterations of the loops on i and j is approximated by n in order to make the analysis simple and simplify the expressions for the different metrics.

example in the *Jacobi* relaxation example we could consider the effect of the task creation overhead (last column in Figure 3), assuming that one of the infinitely-many processors is devoted to linearly creating all the tasks and creating each task requires the same overhead of t_{create}. Adding this overhead to the initial value of T_∞ already shows that making the tasks smaller will decrease the per-task execution time and increase the total overhead: the execution time decreases with r and c while the overall overhead increases.

3.2. Task ordering constraints and synchronisation overhead

The simple *Jacobi* relaxation example is evolved in order to introduce data dependences between tasks. Figure 4 shows the main loop body for a simplified *Gauss–Seidel* relaxation (top) and the TDG (bottom left) when a block task decomposition strategy is applied (r times c consecutive iterations of the i and j loops, respectively, per task). The concept of *true* (Read-After-Write, or RAW) and *false* (Write-After-Read, or WAR, and Write-After-Write, or WAW) data dependences is introduced. For different reasons, these true and false data dependences will imply task synchronisation and, as will be seen later, they incur data sharing actions. The TDG in that figure shows in green one of the possible critical paths and the expression for the corresponding value of T_∞, in which two components are included: the computation time, which depends only on the number of tasks in the critical path, and the synchronisation overhead introduced by the arrows between consecutive tasks in the critical path, each taking an overhead of t_{synch}. In this case, to simplify the analysis, the task creation overhead is not considered. Again, the student is presented with the trade-off between these two components when exploring different possible granularities for the task. Plotting this expression as a function of c and r certainly helps to understand the trade-off.

```
void compute(int n, double *u, double *utmp) {
int i, j;
double tmp;

for (i = 1; i < n-1; i++)
  for (j = 1; j < n-1; j++) {
    tmp = u[n*(i+1) + j] + u[n*(i-1) + j] +  // elements u[i+1][j] and u[i-1][j]
          u[n*i + (j+1)] + u[n*i + (j-1)] -  // elements u[i][j+1] and u[i][j-1]
          4 * u[n*i + j];                    // element u[i][j]
    u[n*i + j] = tmp/4;                      // element u[i][j]
  }
}
```

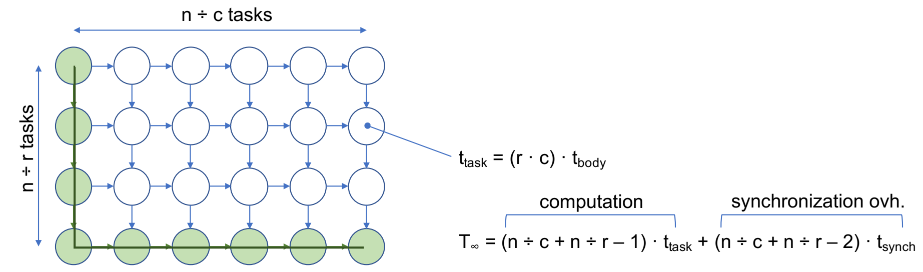
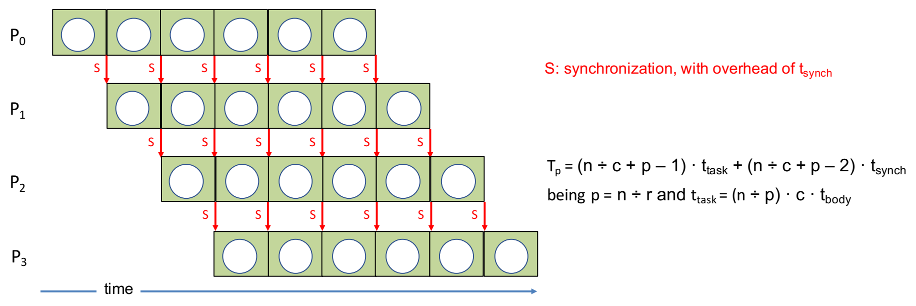

n ÷ c tasks
n ÷ r tasks
$t_{task} = (r \cdot c) \cdot t_{body}$
computation
synchronization ovh.
$T_\infty = \underbrace{(n \div c + n \div r - 1) \cdot t_{task}}_{\text{computation}} + \underbrace{(n \div c + n \div r - 2) \cdot t_{synch}}_{\text{synchronization ovh.}}$

Figure 4: Gauss–Seidel relaxation example and resulting TDG when each task is a block of $r \times c$ consecutive iterations of the i and j loops, respectively. Green nodes compose one of the possible critical paths in the TDG. Computation of T_∞ taking into account synchronisation overheads, t_{synch}.

3.3. Mapping tasks to processors

Once these ideas are clear, students are presented with the need to map the tasks in the TDG to a particular number of processors P in the machine. With this mapping, the students can compute T_p, the execution time of the tasks of the program when using P processors, and the speed-up metric, defined as the quotient $S_p = T_1/T_p$. The speed-up metric, S_p, gives the relative reduction in the execution time when using P processors, with respect to sequential. The efficiency metric, Eff_p, given by $\mathit{Eff}_p = S_p/P$, measures the fraction of time for which the processors are usefully employed. In addition, the notions of strong scaling and weak scaling are introduced in a natural way during the analysis of the dependence of S_p on the number of processors, P.

For the previous example in Figure 4, if we assume strong scaling and $p = n/r$, then T_p would have the same value as T_∞, assuming the same synchronisation overhead. This can be derived from the timeline shown in Figure 5. In fact, only those dependences that are not internalised in the same processor (i.e. that are between tasks mapped to different processors) need to be considered in the computation of T_p.



P_0
P_1
P_2
P_3
S
S: synchronization, with overhead of t_{synch}
$T_p = (n \div c + p - 1) \cdot t_{task} + (n \div c + p - 2) \cdot t_{synch}$
being $p = n \div r$ and $t_{task} = (n \div p) \cdot c \cdot t_{body}$
time

Figure 5: Timeline for the execution of tasks in the Gauss–Seidel relaxation example, assuming that $p = n/r$ processors are used.

3.4. Data sharing overhead

Next, the students are presented with the last source of overhead that we consider: data sharing overheads. The initial simplified machine abstraction used to compute the basic metrics is now leveraged in order to consider that each processor has its own memory and processors are interconnected through an interconnection network. Processors access local data (in their own memory) using regular load/store instructions, with zero overhead. Processors can also access remote data (computed by other processors and stored in their memories) using remote access instructions in the form of messages. To model the overhead caused by these remote accesses we consider an overhead of the form $T_{\text{access}} = t_{\text{s}}+m\times t_{\text{w}}$, where t_{s} is the start-up time spent in preparing the remote access and t_{w} is the time spent in transferring each element from the remote location, which is multiplied by the number of elements to access, m. Additional assumptions are made to simplify the model, such as that a processor P_i can only execute one remote memory access at a time and only serve one remote memory access from another processor P_j at a time, but both can happen simultaneously. Later in the course, students will see that these messages could be cache lines in a shared-memory architecture or messages in a distributed-memory architecture with message passing.

The easy-to-understand *owner-computes rule* can be stated at this point to map data to processors. For example, for the code in Figure 4 one could say that each processor will store in its local memory all those $r \times c$ elements of matrix u that are computed by the tasks assigned to it. This would result in the assignment of data to processors shown in the left part of Figure 6. But in order to execute each assigned task, the processor will have to access the upper, lower, left and right boundary elements, which are computed by other tasks (shown with different colours for one of the tasks in the same figure). Some of these elements are local to processor P_i (left and right boundaries in yellow and green colours, respectively) but some others are stored in the memory of neighbour processors P_{i-1} and P_{i+1} (upper and lower boundaries in blue and orange colours, respectively). It is important to differentiate between true and false data dependencies. True dependences force a task to wait for the availability of data, which is what happens for the elements coloured in blue (remote access happens once the producing task finishes). False dependencies mean that the task has to access the data before the task that owns it starts computation (elements coloured orange) because it overwrites the data due to reuse.

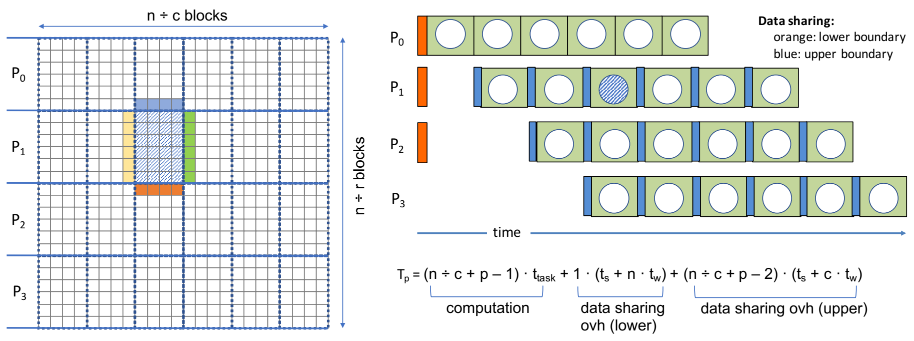

n ÷ c blocks
P_0
P_1
P_2
P_3
n ÷ r blocks
Data sharing:
orange: lower boundary
blue: upper boundary
time
$T_p = \underbrace{(n \div c + p - 1) \cdot t_{task}}_{\text{computation}} + \underbrace{1 \cdot (t_s + n \cdot t_w)}_{\text{data sharing ovh (lower)}} + \underbrace{(n \div c + p - 2) \cdot (t_s + c \cdot t_w)}_{\text{data sharing ovh (upper)}}$

Figure 6: Data mapping (left) and execution timeline (right), including data sharing overheads, for the mapping of tasks to processors for the Gauss–Seidel relaxation example.

Temporal diagrams, such as the one shown in the right part of Figure 6, are very useful at this point to understand where remote accesses should be performed (guaranteeing that when a task is ready to be executed all data that is needed is available), with the possibility of reducing the number of messages due to the effect of t_s, which is usually much larger than t_w). For example, remote accesses involved in the false data dependence could be done as soon as possible, at once for all tasks mapped to the same processor, before the parallel execution starts, as shown in the timeline and considered in the expression of T_p. Again, an analysis of the trade-off introduced by the reduction of the execution time when using more processors and the data sharing overheads allows students to extract interesting conclusions, having the possibilities of plotting the expression for T_p that is obtained and discussing how it changes with the parameters t_s and t_w, or even applying differentiation to see that there exists an optimum task granularity. Note that, for reasons of simplicity, at this point the task creation and synchronisation overheads are not explicitly considered in this analysis.

This unit finishes with the formulation of *Amdahl's law*, allowing students to understand the need for the program to have the highest possible parallel fraction to parallelise. The effect of the overheads previously addressed in the expression of *Amdahl's law* is also considered.

3.5. Methodology and support tools

This part of the course takes about three theory sessions (two hours each) and three laboratory sessions (also two hours each) in which the students access a shared-memory architecture (small cluster with nodes of 16 cores). For this part of the course we also offer video material and online quizzes that cover the fundamental concepts. This material is used by some professors to implement a *flipped–classroom* methodology and offered by other professors simply as study material for the students to consolidate the ideas presented in class. Finally a collection of exercises is made available, some of which are solved in class in order to assess the understanding of these fundamental concepts and metrics.

In the laboratory sessions, students take simple parallel examples written in OpenMP, learning how to compile and execute them. At this point they do not need to fully understand how the parallelism is expressed in OpenMP, but they are able to easily capture the idea of the pragma-based parallel programming approach. How to measure execution time is introduced, allowing students to plot scalability as a function of the number of processors, observing how easily the behaviour deviates from the ideal case. Students are presented with *Tareador* (described in detail in [2]), a tool specifically developed to explore the potential of different task decomposition strategies, visualise the TDG and simulate its parallel execution.

Students are also presented with two tools, *Extrae* and *Paraver*, which instrument and visualise the actual parallel execution and visualise some of the overheads explained in class. One session is devoted to measuring those overheads, observing that these overheads are non-negligible in comparison to the time needed for the processor to execute an arithmetic instruction.

4. Task Decomposition Strategies

Once the fundamentals have been understood, students are faced with the need to express the tasks that appear in the TDG of a sequential program, which we call its *task decomposition.* In the proposed design, we present the various task decomposition strategies for shared-memory architectures using the OpenMP programming model, in particular, the OpenMP tasking model.

The unit starts by presenting three strategies for task decomposition: linear, iterative and recursive. In linear decompositions, a task is simply a code block or procedure invocation. In iterative decompositions, tasks are originated from the body of iterative constructs, such as countable or uncountable loops. Finally, in recursive decompositions, tasks are originated from recursive procedure invocations, for example in divide-and-conquer and branch-and-bound problems.

Three constructs from the OpenMP specification are introduced at this point: `parallel single`, `task` and `taskloop`. The `parallel single` construct simply creates a team of threads and its data context to execute tasks. In fact, `parallel single` is the direct concatenation of two constructs in OpenMP: `parallel`, which creates the team of threads, and `single`, which assigns to one of these threads the execution of an implicit task that contains the body of the `parallel` region in which explicit tasks will be created using the two other constructs. The `single` construct could be avoided, resulting in all threads executing an instance of the implicit task that corresponds to the body of the `parallel` region, replicating its execution as many times as the number of threads that were created. In order to effectively perform work in parallel, the programmer will have to use intrinsic functions (to know which thread is executing the task instance) to manually decompose the work. This way of expressing decompositions will be covered in a different unit, as a way to express the tasks bearing in mind an explicit data decomposition strategy.

The `task` construct is presented to students as the key component for specifying an explicit child task, whose execution will be (possibly) delegated to one of the threads that are part of the team of threads. `Task` constructs can be nested, allowing a rich set of possibilities to express parallelisation strategies. The *task pool* is the main concept in the OpenMP tasking model, in which explicit tasks are created for asynchronous deferred dynamic execution. For this reason, it is important to understand how the child task's data environment is defined, partially regarding variables whose value is captured when the task is created (`firstprivate` clause), variables that are shared with the parent task (`shared` clause) and per-task private copies of variables (`private` clause).

The `taskloop` construct is presented to handle the specification of explicit tasks in loops, which is in fact one of the most important sources of parallelism. The `taskloop` construct includes two clauses to manage task granularity: `grainsize` (used to define the number of consecutive loop iterations that constitute each task generated from the loop) and `num_tasks` (used to define the number of tasks to be generated).

4.1. Linear and iterative task decompositions

Figure 7 shows the simple vector addition example that is used in this unit to illustrate the different linear and iterative task decomposition strategies and how to express them using OpenMP constructs and clauses.

Tasking also allows the expression of iterative decompositions when the number of iterations is unknown (uncountable), such as in problems traversing dynamic data structures such as lists and trees. The list traversal in Figure 8 is one of the simplest examples, showing the importance of capturing the whole scope (basically the list element pointed by `p`) that needed by the task processing each list element when executed in a deferred way (possibly) by another thread.

The dynamic nature of the tasking execution model does not assume any static mapping of chunks of iterations (i.e. tasks) to threads, which may have an important effect on data locality. These static mappings are considered later in the course when covering data decomposition strategies, making use of the so-called work-sharing constructs in OpenMP. We propose to present them once students have been presented with the architectural support for data sharing and the overheads that memory coherence may introduce when data locality is not taken into account.

4.2. Recursive task decomposition

Once iterative decomposition strategies are well-understood, students are faced with the necessity of expressing parallelism in recursive problems, and in

```
void main() {
   ....
   #pragma omp parallel
   #pragma omp single
   vector_add(a, b, c, N);
   ...
}
```

(a) Team of threads creation for task execution

```
void vector_add(int *A, int *B, int *C, int n) {
   #pragma omp task private(i) shared(A, B, C)
   for (int i=0; i< n/2; i++)
        C[i] = A[i] + B[i];
   #pragma omp task private(i) shared(A, B, C)
   for (int i=n/2; i< n; i++)
        C[i] = A[i] + B[i];
}
```

(b) Linear task decomposition, task granularity of *n/2* iterations

```
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++)
       #pragma omp task firstprivate(i) shared(A, B, C)
       C[i] = A[i] + B[i];
}
```

(c) Iterative task decomposition with `task`, task granularity of 1 iteration

```
void vector_add(int *A, int *B, int *C, int n) {
   #pragma omp taskloop shared(A, B, C) grainsize(BS)
   for (int i=0; i< n; i++)
        C[i] = A[i] + B[i];
}
```

(d) Iterative task decomposition with `taskloop`, task granularity of *BS* iterations

Figure 7: Different alternatives in OpenMP to express iterative task decompositions in a vector addition example.

```
int main() {
struct node *p;
p = init_list(n);
#pragma omp parallel
#pragma omp single
while (p != NULL) {
    #pragma omp task firstprivate(p)
    process_work(p);
    p = p->next;
    }
}
```

Figure 8: Using OpenMP to express an iterative task decomposition with unknown loop limits.

particular the two basic questions: "what should be a task?" and "how can I control task granularities?" The first question is simply addressed by analysing a recursive implementation of the vector addition example previously commented, which is shown in Figure 9. Two possible decomposition strategies are presented: 1) the *leaf strategy*, in which a task corresponds to the code that is executed once the recursion finishes (in the example, this is each invocation of `vector_add`); and 2) the *tree strategy*, in which a task corresponds to each invocation of the recursive function (`rec_vector_add` in the example). Figure 10 shows the leaf and tree parallel implementations of the code in Figure 9. Figure 11 shows the tasks that would be generated in both cases. The main difference between the two approaches is that in the *leaf* approach tasks are sequentially generated by the thread that entered the `single` region; however, in the *tree* approach tasks also become task generators, so that the tasks that execute the work in the base case are created in parallel.

4.3. Controlling task granularities

Once students have analysed the tasks generated in both cases, they are faced with the second question, which is related to the control of task granularity. With the simple observation that the task granularity depends on the depth of recursion to reach the base case, students can propose different alternatives to control the number of tasks generated and/or the granularity, which we call *cut-off* control mechanisms. We usually discuss three different alternatives: stopping task generation (a) after a certain number of recursive calls (static control), (b) when the size of the vector is too small (static control), or (c) when the number of tasks generated or pending to be executed is too large (dynamic control). For example, the code in Figure 12 shows how depth-based cut-off control could be implemented with the leaf strategy, either using conditional statements (top) or using the `final` and `mergeable` clauses available on the OpenMP `task` construct (bottom). It is important to differentiate the base case from the cut-off mechanism since they have different functionalities.

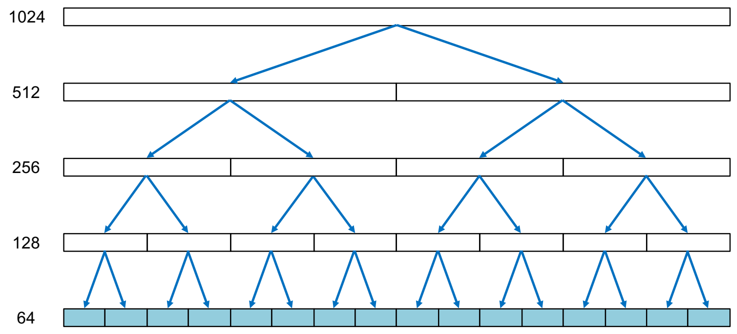
Other cases in which recursive task decomposition could be applied include

```
#define N 1024
#define BASE_SIZE 64
void vector_add(int *A, int *B, int *C, int n) {
   for (int i=0; i< n; i++) C[i] = A[i] + B[i];
}

void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>BASE_SIZE) {
       int n2 = n / 2;
       rec_vector_add(A, B, C, n2);
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   } else
       vector_add(A, B, C, n);
}

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

(a) Sequential code



(b) Divide-and-conquer division of the vectors `A`, `B` and `C` originated after recursive invocations to function `rec_vector_add`.

Figure 9: Sequential recursive version for the vector addition example in Figure 7 and the resulting recursion tree.

branch-and-bound problems, for example the problem of placing n non-attacking queens on a chess board or the travelling salesman problem. These together with other examples based on divide-and-conquer are left to the student as problems to be resolved and discussed in class.

4.4. Task ordering constraints

Once the students know the basic mechanisms available in OpenMP to express different kinds of task decomposition strategies, together with the mechanisms to control task granularity, they are faced with the necessity of expressing

```
void main() {
   #pragma omp parallel
   #pragma omp single
   rec_vector_add(a, b, c, N);
}
```

(a) Main program

```
void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>BASE_SIZE) {
       int n2 = n / 2;
       rec_vector_add(A, B, C, n2);
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   } else
       #pragma omp task
       vector_add(A, B, C, n);
}
```

(b) *Leaf* decomposition

```
void rec_vector_add(int *A, int *B, int *C, int n) {
   if (n>BASE_SIZE) {
       int n2 = n / 2;
       #pragma omp task
       rec_vector_add(A, B, C, n2);
       #pragma omp task
       rec_vector_add(A+n2, B+n2, C+n2, n-n2);
   } else
       vector_add(A, B, C, n);
}
```

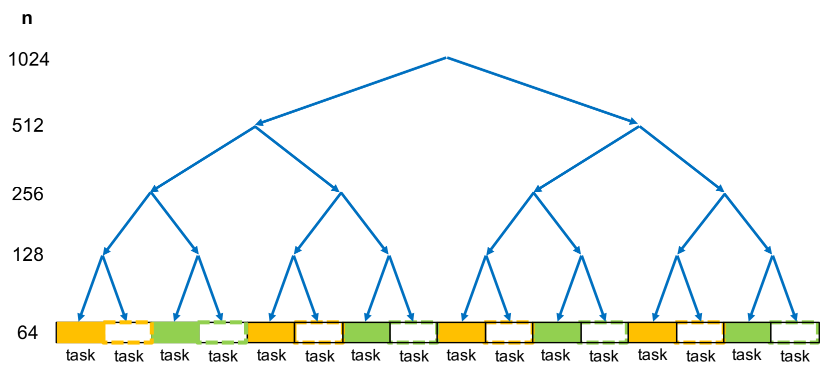
(c) *Tree* decomposition

Figure 10: Leaf and tree recursive task decomposition strategies applied to the vector addition example in Figure 9.

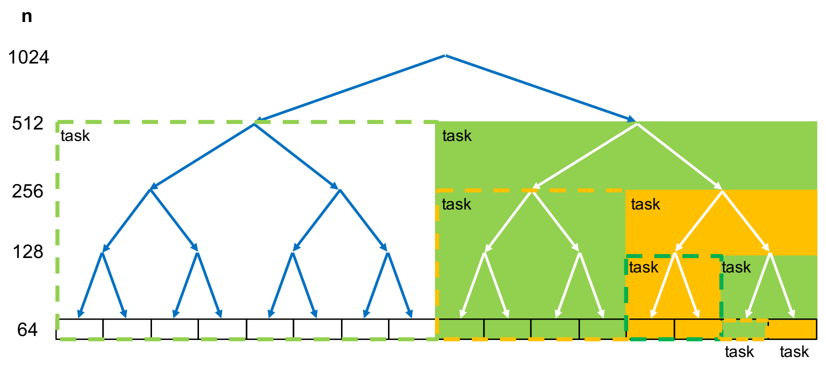
task ordering and data sharing constraints. Task ordering constraints enforce the execution of (groups of) tasks in a required order while data sharing constraints force data accesses to fulfil certain properties (write-after-read, exclusive, commutative, etc.).

Task ordering constraints can be due to control dependences (e.g. the creation of a task depends on the outcome of one or more previous tasks) or data dependences (e.g. the execution of a task cannot start until one or more previous tasks have computed some data). These constraints can be easily imposed by sequentially composing dependent tasks, by inserting (global) task barrier synchronisations, which avoid the creation of tasks until the tasks that introduce the control/data dependency finish, or by expressing task dependencies.

The two different mechanisms available in OpenMP to express task barriers


n
1024
512
256
128
64
task task task task task task task task task task task task task task task task

(a) *Leaf* decomposition



(b) *Tree* decomposition

Figure 11: Tasks generated for the leaf and tree recursive task decomposition strategies in Figure 10.

are presented to students: `taskwait`, which suspends the current task at a certain point waiting for all child tasks to finish, and `taskgroup`, which suspends the current task (at the end of the structured block it defines) waiting on the completion of all its child tasks and their descendent tasks. Figure 13 shows a simple example that is used in class to explain these constructs. In the top-left corner we have a simple TDG, showing task durations, and a trace of an ideal execution of these tasks. Task barriers enforce dependences by not generating tasks that depend on previously generated tasks. This causes extra delays, as shown in the top-center and top-right codes and execution timelines that make

```
#define CUTOFF 3
void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
    if (n>MIN_SIZE) {
        int n2 = n / 2;
        if (depth < CUTOFF) {
            #pragma omp task
            rec_vector_add(A, B, C, n2, depth+1);
            #pragma omp task
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        } else {
            rec_vector_add(A, B, C, n2, depth+1);
            rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
        }
    } else vector_add(A, B, C, n);
}
```

(a) Using conditional statements to control task generation}

```
#define CUTOFF 3
void rec_vector_add(int *A, int *B, int *C, int n, int depth) {
   if (n>MIN_SIZE) {
       int n2 = n / 2;
       #pragma omp task final(depth >= CUTOFF) mergeable
       rec_vector_add(A, B, C, n2, depth+1);
       #pragma omp task final(depth >= CUTOFF) mergeable
       rec_vector_add(A+n2, B+n2, C+n2, n-n2, depth+1);
   } else vector_add(A, B, C, n);
}
```

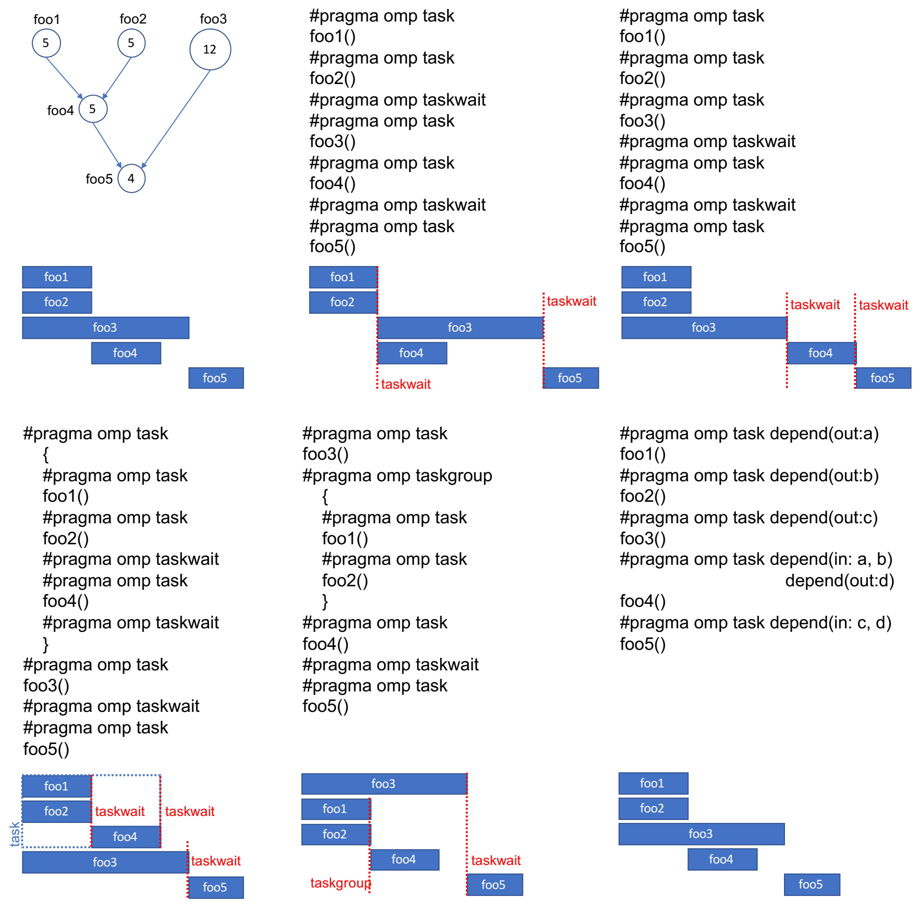
(b) Using `task` clauses to control task generation

Figure 12: Depth-based cut-off control for the tree recursive task decomposition strategy.

use of `taskwait`. The two solutions at the bottom-left and bottom-center make use of task nesting and combined use of `taskgroup` and `taskwait` constructs to achieve the expected behaviour, using task control mechanisms to express data dependencies, but requiring "global thinking" in an unnatural way.

The bottom-right code and execution timeline in Figure 13 show the use of task dependences in OpenMP to express the TDG in a more natural "local thinking" way (having in mind only what a task requires in order to be executed and what it produces after being executed, independently of the task that produces or uses the data). Task dependences among sibling tasks (i.e. from the same parent task) are derived at runtime from the information provided through directionality clauses, expressing which of the data used by the task is read, written or both.

Task dependences are derived from the items in the `in`, `out` and `inout` variable lists. These lists may include array sections. Figure 14 shows another example that could be used to get a better understanding of how these direc-



```
#pragma omp task
foo1()
#pragma omp task
foo2()
#pragma omp taskwait
#pragma omp task
foo3()
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```

```
#pragma omp task
foo1()
#pragma omp task
foo2()
#pragma omp task
foo3()
#pragma omp taskwait
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```

```
#pragma omp task
   {
   #pragma omp task
   foo1()
   #pragma omp task
   foo2()
   #pragma omp taskwait
   #pragma omp task
   foo4()
   #pragma omp taskwait
   }
#pragma omp task
foo3()
#pragma omp taskwait
#pragma omp task
foo5()
```

```
#pragma omp task
foo3()
#pragma omp taskgroup
   {
   #pragma omp task
   foo1()
   #pragma omp task
   foo2()
   }
#pragma omp task
foo4()
#pragma omp taskwait
#pragma omp task
foo5()
```

```
#pragma omp task depend(out:a)
foo1()
#pragma omp task depend(out:b)
foo2()
#pragma omp task depend(out:c)
foo3()
#pragma omp task depend(in: a, b)
                          depend(out:d)
foo4()
#pragma omp task depend(in: c, d)
foo5()
```

Figure 13: Different alternatives to ensure the dependences in a simple TDG using mechanisms available in OpenMP.

tionality clauses are used. The dependences cause a wavefront execution of the tasks, similar to that studied in the previous unit (Figures 4 and 5).

4.5. Data sharing constraints

Finally in this unit the student is presented with mechanisms that allow the concurrent execution of tasks if exclusive access to certain variables or parts of them can be guaranteed. This implies that the execution of tasks is commutative in terms of their execution order, eliminating task ordering constraints. Two basic mechanisms are presented: *atomic accesses*, which guarantee atom-

```
#pragma omp parallel private(i, j)
#pragma omp single
{
   for (i=1; i<n i++) {
        for (j=1; j<n;j++) {
           #pragma omp task // firstprivate(i, j) by default
                            depend(in : block[i-1][j], block[i][j-1])
                            depend(out: block[i][j])
           foo(i,j);
         }
    }
}
```

Figure 14: Task dependences example, simplified *Gauss–Seidel* code.

icity for load/store instruction pairs, and *mutual exclusion*, which ensures that only one task at a time can execute the code within the critical section or access certain memory locations. The three specific mechanisms in OpenMP related to tasks are presented: `atomic` (which includes atomic updates, reads and writes), `critical` (with and without a name) and `locks`, including the intrinsic functions for acquiring and releasing locks. Understanding the differences among the three mechanisms is key, and examples are used to ensure that students achieve a good understanding. Code excerpts based on the use of lists, hash tables, etc., are excellent examples to illustrate the differences among these mechanisms.

4.6. Methodology

This part of the course typically requires about four theory sessions (two hours each) and five laboratory sessions (also two hours each). During the five laboratory sessions, students receive two different assignments. Some examples for these assignments are:

- Two-dimensional *Mandelbrot Set* computation. This is an embarrassingly parallel iterative task decomposition in which students can experiment with different task granularities, expressed using `task` and/or `taskloop` with different values for the `grainsize`. Tasks are totally independent unless the result is displayed on the screen while the set is computed, in which case mutual exclusion is required to plot on the screen.

- *Sieve of Eratosthenes.* The program finds (and counts) all prime numbers up to a certain given *lastNumber* and it is well suited for an iterative task decomposition, using either `task` or `taskloop` to have a better control of granularity. In order to improve locality, the computation of the prime numbers is done in a range between `from` and `to` and then the program uses an outer loop that sieves blocks of a certain block size in order to cover the full range between 1 and *lastNumber*.

- *Multisort*, using a divide-and-conquer recursive task decomposition strategy. The divide-and-conquer strategy recursively splits the vector to sort into four parts, which are sorted with four independent invocations of sort. Once these sort tasks end, two merge tasks follow, each one joining the results of two sort tasks. Their results are merged again with a final merge call. In this code `task`, task barriers (`taskwait` and `taskgroup`) and task dependences are the main ingredients to effectively parallelise the sequential code.

- *Sudoku*, using branch-and-bound recursive task decomposition. The code is useful to show the need of data replication to enable exploratory parallelisation strategies and the need to control task generation based on recursion depth or the number of tasks to avoid excessive overheads.

For each assignment, students first use *Tareador* to explore possible task decompositions, analyse the resulting dependences between tasks and identify the variables that cause dependencies. The students try to understand the reasons for the dependencies and decide how to enforce them in OpenMP. Given the potential parallelism of the explored task decompositions, students start coding different versions using OpenMP. As mentioned in the previous unit, *Extrae* and *Paraver* are used to visualise and analyse the behaviour and performance of their parallelisation strategies. An analysis of overheads and strong scalability concludes each assignment, which also offers some optional parts to further explore the possibilities of OpenMP and/or potential parallelisation strategies.

For this part of the course we also offer video material that covers the basic task decomposition strategies and online quizzes to understand how and when tasks are created and executed. As in the previous unit, this material is used by some professors to implement a *flipped–classroom* methodology and by other professors it is simply offered as study material for the students to consolidate the ideas presented in class.

As mentioned before, students have a collection of exercises available, some of which are solved in class. These exercises are an important component of the course methodology to assess the understanding of different task decomposition strategies and how to specify them in OpenMP via the available constructs for specifying tasks, guaranteeing task ordering and sharing data.

5. Architecture support to shared-memory programming

While students practise the concepts and strategies explained in the previous unit in the laboratory sessions, they are exposed to the basics of parallel architectures, with a clear focus on understanding the support that different organisations provide for two fundamental aspects covered in the previous units: *how is data shared among processors?* and *how are processors able to synchronise?* Figure 15 lists the three presented architectures.

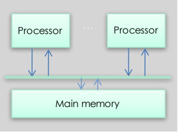
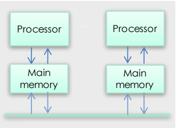
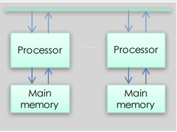
Memory architecture	Address space(s)	Connection	Model for data sharing	Names
(Centralized) Shared-memory architecture	Single shared address space, uniform access time		Load/store instructions from processors. Snoopy-based coherence	• SMP (Symmetric Multi-Processor) architecture • UMA (Uniform Memory Access) architecture
Distributed-memory architecture	Single shared address space, non-uniform access time		Load/store instructions from processors. Directory-based coherence	• DSM (Distributed-Shared Memory architecture • NUMA (Non-Uniform Memory Access) architecture
	Multiple separate address spaces		Explicit messages through network interface card	• Message-passing multiprocessor • Cluster Architecture • Multicomputer

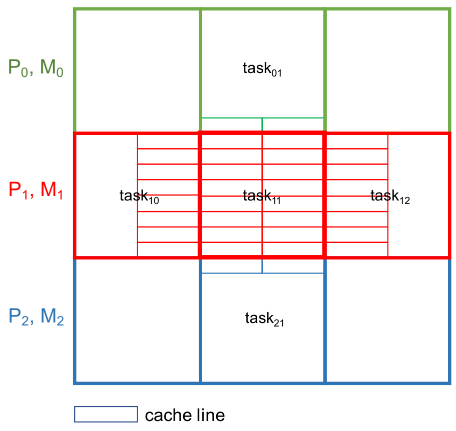
Figure 15: Classification of multiprocessor architectures.

5.1. How data is shared between processors?

Starting from the initial cache hierarchy for single-processor architectures that they already know, the students try to evolve the system to accommodate more than one processor, with the objective of sharing the access to memory. Private vs. shared cache hierarchies easily enter the discussion and the cache coherence problem is presented. The two usual solutions (write-update vs. write-invalidate coherence protocols) are described and their pros and cons are analysed. Snoopy-based coherence mechanisms are presented first, based on: 1) the fact that every cache that has a copy of a block from main memory keeps its sharing status (*status distributed*); and 2) the existence of a *broadcast* medium (e.g. a bus) that makes all transactions visible to all caches and defines an *ordering*. The unit then focusses on understanding the basic MSI and MESI write-invalidate snooping protocols, with their states and the state transitions triggered by CPU events and bus transactions. The students' curiosity and interest easily reveal the need for more advanced protocols, such as MOESI and MESIF, in order to minimise the intervention of main memory.

Students are questioned about the scalability of a mechanism based on a broadcast medium and are helped to evolve it to a distributed solution in which the sharing status of each block in memory is kept in just one location (the directory). The need to physically distribute main memory across different nodes while keeping cache coherence has a price: non-uniformity in terms of access time to memory (*NUMA* architectures). The structure of the directory is presented (the need for a sharers list in addition to the status bits) together with a simplified coherence protocol and the coherence commands that are exchanged between nodes (local generating the request, owner of the line in main memory and remote with clean/dirty copies).

At this point it is important to go back to a parallel program in OpenMP (such as the well known *Gauss–Seidel* relaxation code) and analyse how the memory accesses performed by one of the tasks trigger different coherence actions and cause changes in the state of memory/cache lines. Figure 16 shows the example that is used to motivate the discussion. The example assumes that 1) the blocks of the matrix are distributed in the main memories of three NUMA nodes (M_{0-2}) by rows and 2) the tasks computing the blocks in each node are executed by the processor in that node (P_{0-2}, respectively). Based on that, and the dependences that order the execution of tasks, the evolution of the lines shown in the figure is analysed based on the coherence commands issued from the processors in each NUMA node. Students are asked to think about what would happen if tasks were dynamically assigned to processors, as actually happens in the OpenMP tasking model, and use this as a motivation for the next unit in the subject (data decomposition strategies described in the next section).



Access pattern:
u[i][j] = f(u[i-1][j], u[i+1][j], u[i][j-1], u[i][j+1])

Dependences:
task_{11} can only be computed when P_0 finishes with task_{01} and the same processor (P_1) finishes with task_{10}

Questions for student discussion:
Assuming uncached status for all lines at the beginning of the execution …

1. Which will be the contents of the directory for lines accessed by task_{01} , task_{10} , task_{12} and task_{21} when task_{11} is ready for execution? In which caches there exist copies of those lines? (if cached)
2. And for the lines accessed by task_{11}?
3. Repeat questions 1 and 2 above when P_1 is finishing the execution of task_{11}.

Figure 16: Example based on the *Gauss-Seidel* computation that is used to understand the coherence traffic generated.

This is also a good point to see one of the problems that occur in cache-based parallel architectures: *false sharing* in contrast to *true sharing*, and ways to address it when defining shared data structures (e.g. use of padding).

5.2. How are processors able to synchronise?

Once students understand the key role of the memory system in providing the shared-memory abstraction that OpenMP is based on, they are presented with the need for low-level mechanisms to guarantee safety for accesses to shared-memory locations (e.g. mutual exclusion and atomicity) or to signal certain events (e.g. task barriers and dependences). After motivating the impossibility of guaranteeing them at a higher level, the professor introduces the first mechanism based on atomic (indivisible) instructions to fetch and update memory on top of which other user-level synchronisation operations can be implemented: test&set (read the value at a location and replace it by the value one), atomic exchange (interchange of a value in a register with a value in memory) and fetch&op (read the value at a location and replace it with the result of a simple arithmetic operation, usually add, increment, subtract or decrement). Students are also presented with the other mechanism currently available based on Load-linked Store-conditional instruction sequences (ll-sc), working through some examples to see how to conditionally re-execute them in order to simulate atomicity.

The basic mechanisms are used to code simple high-level synchronisation patterns; after that the discussion goes back to memory coherence, analysing how these synchronisation mechanisms increase coherence traffic and the interest of using test-test&set or load-ll-sc whenever possible in order to avoid writing to memory and invalidating other copies of the synchronisation variable.

This part finishes with an example in which, apparently, there is no need to use any of the synchronisation mechanisms presented before to synchronise the execution of tasks. The kind of example is shown on the left side of Figure 17. In this code two tasks synchronise their execution through a shared variable `next`; the second task always goes one iteration behind the first task, doing a busy–wait while loop to ensure this. This example introduces the discussion about memory consistency and the relaxed consistency model used in OpenMP. The same code on the right side of Figure 17 solves the problem by using `#pragma omp flush` to explicitly force consistency.

5.3. Scaling through the distributed-memory paradigm

Finally students are questioned about the need to actually share memory and presented with the third paradigm in Figure 15: multiple separate address spaces. However, the simplicity of the distributed-memory paradigm in terms

```
int next = 0;
#pragma omp parallel
#pragma omp single
{
   #pragma omp task
   for (int end = 0; end == 0; ) {
      …
      next++;
      #pragma omp flush(next)
      if (next==N) end=1;
   }
```

```
#pragma omp task
   {
      int mynext = 0;
      for (int end = 0; end == 0; ) {
         while (next <= mynext) {
               #pragma omp flush(next)
               ; }
         …
         mynext++;
         if (mynext==N) end=1;
      }
   }
}
```

Figure 17: Synchronisation through a shared variable and the use of `flush` to enforce consistency.

of hardware comes at the cost of programmability. The key point to understand here is that since each processor has its own address space, a processor cannot access data resident in the memory of other processors and any interaction with them has to be done through the network interface card and interconnection network. With the knowledge that students have about computer networks the message passing paradigm flows very naturally. The basic primitives for data exchange are presented, both in the form of point-to-point communication (basic send and receive) and in the form of collectives (basic broadcast, scatter, gather and reduction).

5.4. Methodology

This part of the course takes about three theory sessions (two hours each), with no laboratory sessions. We offer to the students video material that covers cache coherence for both bus and directory-based shared-memory architectures and for distributed-memory architectures together with online quizzes to understand the main concepts. As in the previous unit, this material is used by some professors to implement a *flipped-classroom* methodology and simply offered by other professors as study material for the students to consolidate the ideas presented in class. However, the video material used in this unit belong to the course High Performance Computer Architecture from Georgia Tech University by Profs. Milos Prvulovic and Catherine Gamboa, which is available on Udacity.

6. Data decomposition strategies

Once students understand the NUMA aspect of shared-memory architectures and the lack of data sharing in distributed-memory architectures, they are presented with an alternative approach to task decomposition. The new approach is based on extracting parallelism from the multiplicity of data (e.g. elements in vectors, rows/columns/slices in matrices, elements in a list, subtrees in a tree, and so on).

Data decomposition is first motivated by the excessive level of implicit data movement that may be introduced in NUMA architectures by a task decomposition that is unaware of how data is accessed by tasks. The dynamic assignment of tasks to processors does not favour the data locality that would be required to minimise the negative effect of accessing remote data. This is motivated by the conclusions drawn from the analysis of the *Gauss–Seidel* example in Figure 16 and by the new synthetic example shown in Figure 18, consisting of a sequence of loops in which the tasks originate from a `taskloop` construct that executes chunks of consecutive iterations. Observe also the use of the `nowait` clause to avoid the implicit barrier at the end of each `for` construct: data dependences between tasks are internalised within the execution of each implicit task.

```
#define n 100
#pragma omp parallel
#pragma omp single
for (iter=0; i<num_iters; iter++) {
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<n; i++)
        b[i] = foo1(a[i]);
    #pragma omp taskwait
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<n; i++)
        c[i] = foo2(b[i]);
    #pragma omp taskwait
    #pragma omp taskloop num_tasks(4)
    for (int i=0; i<n; i++)
        a[i] = foo3(c[i]);
    #pragma omp taskwait
}
```

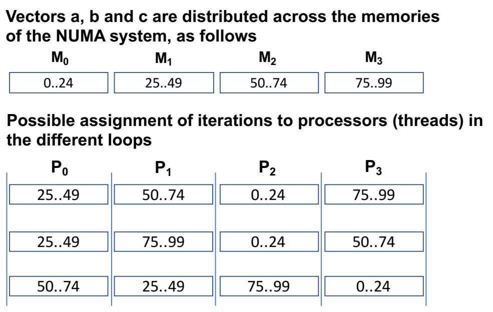

Vectors a, b and c are distributed across the memories of the NUMA system, as follows
M0
M1
M2
M3
0..24
25..49
50..74
75..99
Possible assignment of iterations to processors (threads) in the different loops
P0
P1
P2
P3
25..49
50..74
0..24
75..99
25..49
75..99
0..24
50..74
50..74
25..49
75..99
0..24

Figure 18: Example used to illustrate the implicit data movement when task decomposition is applied. Tasks are dynamically executed by processors, as shown on the right for a possible assignment of tasks to processors. This dynamic assignment imply penalties in the access time to data accessed by the tasks.

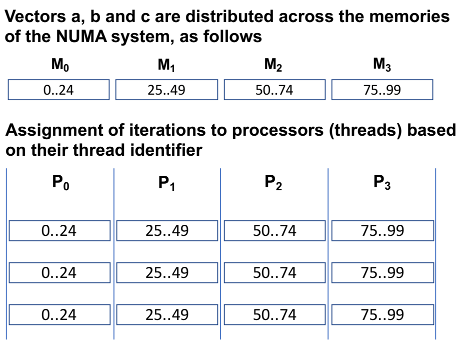
It should be clear at this point that data locality could be easily improved if the programmer takes into account the data that is accessed by each task and controls the assignment of tasks to processors. The proposed parallel code in the upper part in Figure 19 makes use of the implicit tasks that are generated in `parallel` constructs in OpenMP: one implicit task per thread executing the parallel region. As can be seen in the example, each implicit task queries the identifier of the thread executing it (call to `omp_get_thread_num` intrinsic function in OpenMP) and the number of threads that participate in the parallel region (call to `omp_get_num_threads` intrinsic function in OpenMP). With this information each implicit task decides on a range of iterations to execute, which can be the same for all the loops in the sequence in order to improve data locality. In this example, in addition, the use of task barriers (`taskwait` in Figure 18) can be avoided because data dependences between tasks are internalised within the execution of each implicit task.

6.1. Loop vs. task-based approaches

This is a good point to explain the `#pragma omp for` directive in OpenMP, which clearly represents the "traditional" loop-based approach to teach parallelism in a large body of parallel programming courses. As shown in the lower part in Figure 19 the `for` work-sharing construct and `schedule(static [, chunk])` clause in OpenMP allow the programmer to statically assign groups of consecutive iterations (each group of size `chunk`) to consecutive threads in a round-robin way; if `chunk` is omitted, then the compiler simply generates as many groups of consecutive iterations as threads in the parallel region.

The *doacross* model introduced in the most recent OpenMP specification is also presented as the mechanism available to define ordering constraints between loop iterations. The `ordered` clause in the `for` work-sharing construct is used to indicate the *doacross* execution, and the `depend` clauses in the `ordered` construct are used to indicate the `source` and `sink` of the dependence relationships between iterations, as shown in the two examples in Figure 20.

```
// Solution based on thread identifiers
#pragma omp parallel
    {
    whoamI = omp_get_thread_num();
    howmany = omp_get_num_threads();
    chunk = n / howmany;
    lower = whoamI * chunk;
    upper = (whoamI == (howmany-1) ?
                    n :  lower+chunk);
    for (iter=0; i<num_iters; iter++) {
        for (int i=lower; i<upper; i++)
            b[i] = foo1(a[i]);
        for (int i=lower; i<upper; i++)
            c[i] = foo2(b[i]);
        for (int i=lower; i<upper; i++)
            a[i] = foo3(c[i]);
    }
}
```


Vectors a, b and c are distributed across the memories of the NUMA system, as follows
M_0
M_1
M_2
M_3
0..24
25..49
50..74
75..99
Assignment of iterations to processors (threads) based on their thread identifier
P_0
P_1
P_2
P_3

```
// Solution based on for work-sharing
#pragma omp parallel
for (iter=0; i<num_iters; iter++) {
    #pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
        b[i] = foo1(a[i]);
    #pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
        c[i] = foo2(b[i]);
    #pragma omp for schedule(static) nowait
    for (int i=0; i<n; i++)
        a[i] = foo3(c[i]);
}
```

Figure 19: Continuation of the example in Figure 18 to illustrate the use of implicit tasks in OpenMP (one implicit task per thread, each implicit task executing the body of the parallel region) to control the assignment of iterations (in chunks) to processors. The code on the top makes use of intrinsic functions in OpenMP to determine the identifier of the thread executing the implicit task and the total number of threads. The code on the bottom makes use of `#pragma omp for` to achieve the same assignment of iterations to threads.

6.2. Geometric and recursive data decompositions

The idea behind data decomposition is 1) to identify the data used and/or produced in the computations, which can be output data, input data or both; 2) logically partition this data across various tasks, with two possible strategies considered in this lesson (geometric decomposition and recursive decomposition) or consider the necessity of data replication; and 3) obtain a computational partitioning that corresponds to the data partitioning, following the owner-computes rule. For distributed-memory architectures, one more step will be required in order to add the necessary data allocation and movement actions.

With output data decomposition, the programmer selects data structures that are produced by the tasks and decides how to partition them; input data

```
#pragma omp for ordered(1)
for ( i = 1; i < N; i++ ) {
    A[i] = foo (i);
    #pragma omp ordered depend(sink: i-1)
    B[i] = goo( A[i], B[i-1] );
    #pragma omp ordered depend(source)
    C[i] = too( B[i] );
    }
```

```
#pragma omp for ordered(2)
for (i = 1; i < N; i++) {
  for (j = 1; j < M; j++) {
    A[i][j] = foo(i, j);
    #pragma omp ordered depend(source)
    B[i][j] = alpha * A[i][j];
    #pragma omp ordered depend(sink: i-1,j)
                        depend(sink: i,j-1)
    C[i][j] = 0.2 * (A[i-1][j] + A[i][j-1]);
  }
}
```

Figure 20: Example making use of the *doacross* loop execution mode in OpenMP.

structures may follow the same decomposition or require replication in order to avoid task interactions, or they may incur implicit data movement. With input data decomposition, the programmer selects data structures that are read by the tasks and decides how to partition them; output data may or may not follow the same decomposition, and require combining partial results in order to generate the output data structures. Input and output data decomposition could be combined. In both cases, the so-called *Owner Computes Rule* defines who is responsible for performing the computations. In the case of output data decomposition, the owner-computes rule implies that the output is computed by the task to which the output data is assigned; in the case of input data decomposition, the owner-computes rule implies that all computations that use the input data are performed by the task to which the input is assigned.

Once the basic idea is captured, students are presented with different basic alternatives for logically decomposing the data structures, which are shown in Figure 21 for a two-dimensional matrix and Figure 22 for a recursive quad-tree data structure, representing for example the particles in an n-body problem. The code generation strategies that correspond to these different decompositions are discussed in class and/or left as exercises. Recursive data decomposition strategies are clearly more difficult to understand and implement, but they represent a good opportunity for students to think about possibilities.

The granularity associated to the tasks generated out of a data decomposition strategy is clearly defined by the owner-computes rule, which determines the amount of data assigned to each task. For example, the number of consecutive rows in a geometric block decomposition or the size of the subtree in a

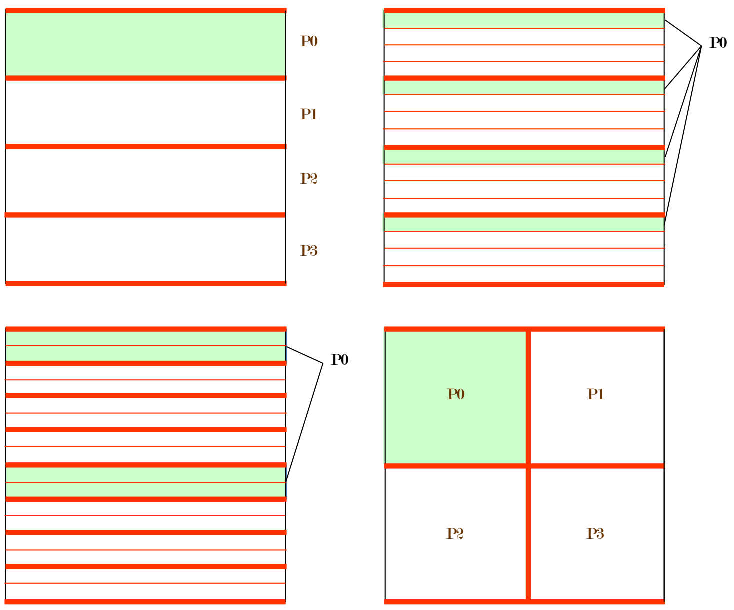


Figure 21: Simple geometric data decomposition strategies for a 2D matrix: per row in a block, cyclic or block-cyclic way and per blocks.

recursive one. Different options are discussed to obtain a good load balancing.

6.3. Task interactions in distributed-memory architectures

For shared-memory architectures students already know the mechanisms that can be used to guarantee task ordering and data sharing constraints; the most appropriate ones for implicit tasks are reviewed: `barrier`, `atomic`, `critical` and lock primitives.

The previous unit finished with an overview of distributed-memory architectures and the mechanisms available to move and exchange data among processors that have disjoint address spaces, i.e. when a processor cannot directly access data stored in the memory of another processor. Now is the time to show students how these mechanisms could be used to ensure that the data needed to perform the computation is available, without entering into much detail since this is a topic to be studied in detail (using MPI) in the PAP subject later in the Computer Engineering specialisation. A simple matrix multiplication

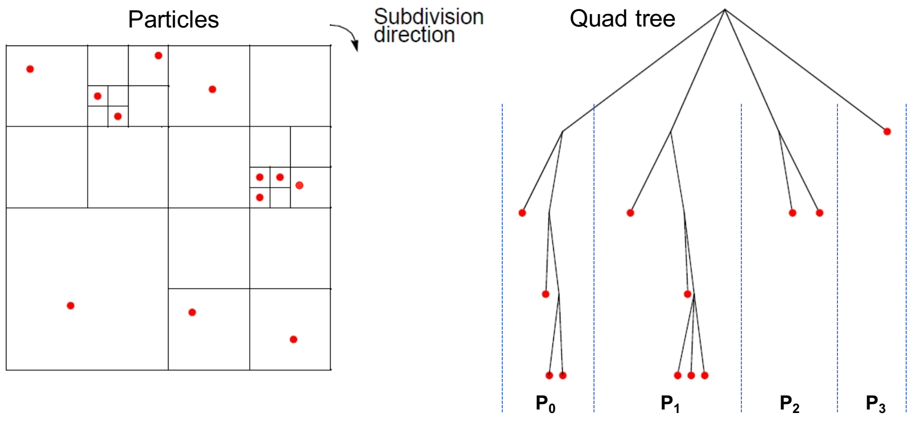


Figure 22: Recursive data decomposition strategy for a quad-tree representing the particles in an n-body problem.

code is used to glue the ideas and see how the different communication mechanisms can be used to broadcast and reduce data, scatter and gather data, or to exchange data point-to-point.

6.4. Methodology

This part of the course takes about three theory sessions (two hours each) and three laboratory sessions (also two hours each). During these three laboratory sessions, the students receive a single assignment to understand the benefits of using a data decomposition.

One of the possible assignments for this unit is the computation of the well-known *heat equation*. Two different solvers are used: *Jacobi* and *Gauss–Seidel*, which students already know because they have been used in theory classes. The program solving the heat equation makes use of a two-dimensional data structure iteratively traversed using loop nests. Although *Jacobi* results in an embarrassingly parallel task decomposition, it is important to guarantee data locality for the matrices that are accessed. The *Jacobi* solver is invoked iteratively in a sequential time-step loop, returning at each iteration a *residual* value that is used to determine convergence and the termination of the iterative loop. The iterative loop also finishes if convergence is not reached after a certain number of iterations. Ensuring that the processors always work with the same blocks of data is necessary to improve locality and reach a good scalability. For the *Gauss–Seidel* solver, the same idea applies, but in this case the task decomposition has dependences among tasks, as already commented. The use of dependences between tasks allows students to express these data dependence constraints albeit at the cost of worse data locality. The use of the *doacross* model for the OpenMP `for` work-sharing construct is recommended at this point as the way to enforce the dependences and ensure data locality.

7. Final remarks

This paper presented the design of a compulsory parallel programming course (*Parallelism* – PAR) for undergraduate students, using the tasking execution model as the backbone for presenting the main concepts and models. The tasking model is identified as more appropriate for this introductory parallel processing course instead of the usual loop-based approach used by many courses that teach parallel processing and OpenMP programming. In this section we show how the proposed design covers the main topics contained in the *Curriculum Initiative on Parallel and Distributed Computing - Core Topics for Undergraduates* [7]. Table 7 shows the organisation of those main topics on Parallel and Distributed Computing throughout the four main units in PAR.

Architecture topics are explained throughout the course and cover levels of parallelism on single cores, multicores and SMP architectures, memory coherence and writing-policy protocols, true/false sharing concepts, memory consistency, synchronisation support, and performance metrics. Floating point representation and precision issues are not studied in this course, since they have already been presented in previous basic computer organisation courses.

Programming topics correspond to concepts and practices related to performance, correctness and semantics, and paradigms and notations. Regarding correctness and semantics, the main concurrency issues are presented in the introductory unit for the course, warning students about the potential problems that may appear in concurrent and parallel programs. Performance metrics, including speed-up, Amdahl's law and efficiency, among others, are presented to students in the Fundamentals unit: performance issues due to task granularities, synchronisation overheads and load balance are well covered in the unit, but also kept in mind during the rest of the course. Once the students have assimilated the above concepts, the main paradigms and notations are presented. OpenMP, the standard shared-memory programming model, is used throughout the course, both in theory sessions as well as in laboratory assignments. MPI is briefly presented as the de facto standard for distributed-memory programming in the Data Decomposition unit. SIMD instructions for data level parallelism are not covered in depth in this course.

Algorithm Topics such as parallel and distributed models and complexity are important concepts that are covered in this course. The directed task dependency graph (TDG) is presented to students as a mechanism to model the potential parallelism of a parallel strategy, based on the abstraction of infinite resources for computation and communication. Afterwards, divide-and-conquer, linear and iterative implementation strategies are analysed in the Task Decomposition unit, where students begin to enjoy parallel programming. Different explicit communications, as point-to-point and collective communications, are presented in the Data Decomposition unit using a simple example: an MPI implementation of matrix multiplication.

Cross-cutting and Advanced Topics are covered along the whole course. In particular, we focus on data locality exploitation in some programming practices by measuring the impact of memory access, and by doing exercises focussed on concurrency issues and performance modelling to achieve correctness and efficiency.

Table 7 also shows the main examples and practices used in the aforementioned topics. Practices are developed in a cluster of shared-memory nodes with 16 cores (two sockets) per node, with the support of different parallel programming tools mentioned in the paper: *Tareador* for the exploration of task decomposition strategies, *Extrae* for the instrumentation of parallel programs and *Paraver* to visualise the behaviour of the parallel execution and understand performance bottlenecks and inefficiencies.

Finally, although the scope of this paper is the description of a compulsory parallel programming course in the bachelor degree in Informatics Engineering, we include in this final section a brief analysis of the evolution of the subject for six academic years, considering: the percentage of students that pass the subject, their level of satisfaction and the average grade obtained by the students. We observed that the new methodology and course organisation have contributed to improving the percentage of students that pass the course, being currently over 80% with an average grade over 6.5 (out of 10). Results prior to using the proposed course organisation showed average grades around 5.5 and a percentage around 70% of students that pass the course, revealing a clear improvement in the student learning process. We also consider these results to be very successful for a fifth-term mandatory course that includes all the bachelor students of the degree (more than 150 per semester). The satisfaction of the students expressed in the quality survey is superior to the rest of mandatory subjects in the same term, and in general the comments received from the students are very positive. The video lessons and quizzes made available through a *moodle* platform for flipped-classroom and/or self-study is also considered by the students to be a great addition to the classical written material (slides, problems and laboratory assignments).

Acknowledgments

This work has been supported by the grant SEV-2015-0493 of the Severo Ochoa Program, awarded by the Spanish Government, by the Spanish Ministry of Science and Innovation (contract TIN2015-65316-P) and by Generalitat de

Catalunya (contracts 2014-MOOC-00057 and 2014-SGR-1051). We also thank the anonymous reviewers and editor for their comments during the review process, other professors that have been involved in the implementation of the PAR course and Paul Carpenter at BSC for his corrections and suggestions to improve the text.

References

[1] OpenMP application program interface: version 4.5. OpenMP Specification [online] (2015). http://www.openmp.org/.

[2] E. Ayguadé, R. M. Badia, D. Jiménez, J. R. Herrero, J. Labarta, V. Subotic, G. Utrera, Tareador: A tool to unveil parallelization strategies at undergraduate level, in: Proceedings of the Workshop on Computer Architecture Education, WCAE '15, 2015, pp. 1:1–1:8. doi:10.1145/2795122.2795123.

[3] Barcelona Supercomputing Center. BSC Performance Tools [online]. https://www.bsc.es/computer-sciences/performance-tools.

[4] T. Mattson, B. Sanders, B. Massingill, Patterns for Parallel Programming, 1st Edition, Addison-Wesley Professional, 2004.

[5] V. Kumar, Introduction to Parallel Computing, 2nd Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[6] J. L. Hennessy, D. A. Patterson, Computer Architecture, Fifth Edition: A Quantitative Approach, 5th Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2011.

[7] S. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. Le Blanc, A. Lumsdaine, D. Padua, M. Parashar, V. Prasanna, Y. Robert, A. Rosenberg, S. Sahni, B. Shirazi, A. Sussman, C. Weems, J. Wu. Nsf/ieee-tcpp curriculum initiative on parallel and distributed computing - core topics for undergraduates, version I [online] (2012). http://www.cs.gsu.edu/ tcpp/curriculum/.

Unit	**Parallel and Distributed Computing Topics**				**Codes**
	Architecture Topics	*Programming Topics*	*Algorithm Topics*	*Crosscutting and Advanced Topics*	
Fundamentals	Performance Metrics	Paradigms and Notations Performance Metrics and Issues Correctness and semantics	Parallel/Distributed models and computing	Locality, Concurrency and Performance Modeling	Jacobi and Gauss-Seidel Relaxation
Task Decomposition	Performance Metric Usage	Paradigms and Notations for Shared Memory, Correctness and Semantics	Algorithm Paradigms		Mandelbrot Set, Eratosthenes Sieve, Multisort, Sudoku
Parallel Architectures	Architecture Classes, Memory Hierarchy, Performance Metrics	Performance Issues			Exercises and Overhead Measurements
Data Decomposition	Memory Hierarchy, Performance Metric Usage	Paradigms and Notations, Distributed Memory, Performance issues	Algorithm Problem		Jacobi and Gauss-Seidel Relaxation

Table 2: Coverage in PAR of the Core Topics in the Curriculum on Parallel and Distributed Computing.