A Learning Experience Toward the Understanding of Abstraction-Level Interactions in Parallel Applications

Alejandro Valero[1], Rubén Gran-Tejero, Darío Suárez-Gracia, Emanuel A. Georgescu, Joaquín Ezpeleta, Pedro Álvarez, Adolfo Muñoz, Luis M. Ramos, and Pablo Ibáñez

Department of Computer Science and Systems Engineering
Universidad de Zaragoza
Spain

Abstract

In the curriculum of a Computer Engineering program, concepts like parallelism, concurrency, consistency, or atomicity are usually addressed in separate courses due to their thoroughness and extension. Isolating such concepts in courses helps students not only to focus on specific aspects, but also to experience the reality of working with modern computer systems, where those concepts are often detached in different abstraction levels. However, due to such an isolation, it exists a risk of inducing to the students an absence of interactions between these concepts, and, by extension, between the different abstraction levels of a system.

This paper proposes a learning experience showcasing the interactions between abstraction levels addressed in laboratory sessions of different courses. The driving example is a parallel ray tracer. In the different courses, students implement and assemble components of this application from the algorithmic level of the tracer to the assembly instructions required to guarantee atomicity. Each lab focuses on a single abstraction level, but shows students the interactions with the rest of the levels. Technical results and student learning outcomes through the analysis of surveys validate the proposed experience and confirm the students learning improvement with a more integrated view of the system.

Keywords: Ray tracing, task queue, semaphore, futex, assembly instructions.

1. Introduction

The development of a Computer Engineering (CE) program must catch up with the fast evolution of the field. Since the end of the 2000s decade, technological limitations have led to an increase in the number of execution contexts running in parallel on a computer system. This requires the ability to not only implement algorithms that expose as much parallelism as possible, but also make an efficient use of hardware mechanisms to guarantee safe parallel execution. In this sense, following the recommendations of both the NSF/IEEE-TCPP Curriculum Committee [1] and the ACM/IEEE Joint Task Force on Computing Curricula [2], numerous approaches have made an effort to increase the presence or to reinforce Parallel and Distributed Computing (PDC) in CE programs. Recent work distribute PDC topics across different courses through the integration of modules into existing courses [3, 4, 5], introducing parallel programming in lower-level courses [6, 7, 8], the proposal of research-oriented teaching methodologies [9, 10, 11], or the creation of new courses [12, 13, 14].

A common approach to design and explain a computer system is to split the complexity of the whole system into self-contained levels. Since such levels relate to each other, each level provides a working interface to the remaining levels. These interfaces model a simplified abstraction of the underlying complexity and establish clear boundaries across the different parts of a system [15].

In most CE programs, each course typically resorts to abstractions in order to design and explain computer systems. Abstractions help to strengthen the learning process, since they make students fo-

[1]Corresponding author
Email address: `alvabre@unizar.es` (A. Valero)

cus on specific aspects. However, in our experience, students often lose the overall view of a computer system with such an approach. This may lead students to the conclusion that some courses are self-contained and do not relate to each other. Particularly, many of them forget the hardware implications underlying high-level abstractions, in terms of performance and power.

Previous work have proposed to teach PDC topics from the perspective of either high-level abstractions to ease both algorithm and software designs [16, 17], or low-level abstractions such as assembly programming to understand what is required to support parallel execution [18]. Unlike these approaches, this paper reinforces PDC topics from the highest to the lowest level of abstraction that underlie complex parallel applications in a computer system [19]. More precisely, this work exposes to the students how the Instruction Set Architecture (ISA) and the operating system provide the required support to high-level synchronization operations, which in turn help strengthen the knowledge on how the essential concepts of parallelism, concurrency, consistency, and atomicity entangle among them and with the hardware [20, 21, 22].

To better understand the relations among the aforementioned concepts, this paper proposes to develop multiple components that at the end build a fully parallel ray tracing application. Ray tracing has been used in the past as a cross teaching experience to integrate two upper-level courses referring to high-level abstractions such as CUDA programming and advanced rendering concepts [23]. On the contrary, we present a learning experience involving multiple laboratory sessions of lower-level and upper-level courses of a CE program. The ray tracer serves as a motivating example that uses a concurrent queue to assign tasks to different execution threads. The queue is accessed in mutual exclusion to preserve data integrity. With this purpose, the access to the queue is managed according to each abstraction level, with *mutexes* implemented with library functions, system calls, or directly in assembly language. This way, the proposed learning experience covers four abstraction levels: Application, Library, Operating System, and ISA. Each abstraction level implicates a different course.

Each proposed lab is mainly tied to the interaction of two specific levels of abstraction, and purposely endowed with a context referring to the rest of the levels, contributing this way to integrate the different abstraction levels. In this work, we introduce the main guidelines, objectives, and results of the proposed experience, which allow to implement other experiences reinforcing inter-course learning.

Prior work has proved the suitability of a single-board computer for teaching parallel computing over mobile devices, student laptops, virtual machines, or remote multicore servers [24, 25, 26]. We build upon these studies by using a common hardware board in all the proposed labs, which contributes to consolidate an integrated view of the system. To this end, we analyze several boards and conclude that Raspberry Pi meets the vast majority of the hardware and software requirements of an inter-course learning experience.

The presented experience is the result of a project carried out in a CE program during the current and the past two academic years, in which assessment studies of the proposal have been already carried out thanks to a set of volunteer students. For the current 2020/2021 academic year, all the proposed labs are fully deployed and all the enrolled students in the involved courses are taking part in the proposed experience. This paper discusses experimental results for all the proposed labs, including both the technical details of the lab assignments and the students learning outcomes using pre/post surveys. These surveys expose that students effectively demand a deeper understanding of the interactions between the abstraction levels, and such demands are fulfilled after the completion of the labs.

The remainder of this paper is organized as follows. Section 2 introduces the context of the CE program and specific courses in which the proposed experience is established. Section 3 describes in depth the learning experience. Section 4 discusses the requirements to implement inter-course learning and the suitability of the selected boards. Section 5 shows the technical results. Section 6 presents a qualitative assessment of the applied learning method and students learning outcomes. Finally, Section 7 summarizes the paper.

2. Context

This section describes the organization of the CE program, including a brief description of the types of courses in each academic year. In addition, the syllabus of the involved courses in the proposed learning experience are described in more detail.

2.1. CE Program

The proposed experience is planned to be fully integrated in the CE program at the *Universidad de Zaragoza* (UNIZAR). This program consists of four academic years, 240 ECTS[2] credits in total[3]. The first two and a half years are common for all students. The core courses in this period mostly focus on the knowledge that any CE graduate should learn: algebra, calculus, discrete mathematics, programming theory, data structures and algorithms, computer architecture and organization, operating systems, physics and electronics, computer networks, databases, distributed systems, software engineering, artificial intelligence, and human-computer interaction. Afterward, students reinforce their knowledge in the major that most interests them within five available options: Computing, Computer Engineering, Information Systems, Information Technology, and Software Engineering. Each major consists of eight compulsory courses. In addition, students select two optional courses from any other major, as well as two core courses that are studied regardless of the chosen major. Finally, the students achieve the program by undertaking an undergraduate dissertation of 12 ECTS.

The CE program focuses on the application of theoretical knowledge in real-life problems, including the development of labs and projects. This approach is ideal to help students with the assimilation of the concepts studied in the different courses. Each course offers a well-though out lab sessions tailored to reinforce the theoretical contents. At best, they are coordinated with other courses that belong to the same area of knowledge. As mentioned above, this can lead students to perceive a course, or a group of courses, as isolated islands, which makes it difficult for them to apply the knowledge acquired in each course in their professional career. In fact, these divisions are purely organizational, and all the courses have many interactions with each other. According to the Computer Engineering Curricula [20], students should learn the development of a *whole computer* in the lab experiments that include exposure to hardware and operating systems in the context of a relevant application, which is, in our case, the ray tracing algorithm.

[2]ECTS refers to European Credit Transfer and accumulation System: `http://ec.europa.eu/education/resources-and-tools/european-credit-transfer-and-accumulation-system-ects_en`

[3]`https://estudios.unizar.es/estudio/ver?id=148`

2.2. Involved Courses

The proposed experience implicates four different courses within the program to jointly face the problem. These courses are Computer Graphics, Distributed and Concurrent Systems Programming, Operating Systems, and Multiprocessors, which are related to the Application, Library, Operating System, and ISA abstraction levels, respectively.

Computer Graphics (CG) is a core course of the Computing major, and an optional course in other majors. CG focuses on mathematical models and algorithms that generate synthetic images (or videos) in which performance is a must. Students learn the underlying mathematical and physical concepts that define appearance and, as practical assignments, develop algorithms like ray tracing that output images from such concepts. Parallelization is key to the performance of such algorithms, and different parallelization strategies (static/dynamic, with different high level structures and partitions) must be explored.

Distributed and Concurrent Systems Programming (DCSP) is a core course that concentrates on the fundamentals of programming such classes of systems. In the case of concurrency, the lectures focus on the explanation of the problems that arise when a set of processes have to share data and resources, and the way such a problem has been solved, from the main mutual exclusion algorithms based on shared variables to higher level structures such as semaphores and monitors. In the case of distributed systems, students learn how to coordinate processes by means of synchronous and asynchronous message passing as well as by means of the use of a shared tuple space. Besides studying the concepts from a conceptual point of view, students work in a set of laboratory sessions and a final team project in which they have to develop some programs where the studied concepts are a crucial part.

Operating Systems (OS) is a core course that presents in a comprehensive way the structure and functions of an operating system. The operating system is presented as a resource manager and as a service provider at the system call and command interpreter levels. At each level, the student acquires concepts and skills related to the management and the use of the main system resources such as the processor, memory, and input/output devices. In relation to the topic presented in this paper, the course presents the synchronization primitives offered in the pthread library and studies the keys to

Table 1: Relations among the abstraction levels, courses, activities, academic years, semesters, and chronological order.

Abstraction level	***Course***	***Activity***	***Academic year***	***Semester***	***Chronological order***
Application	CG	Ray tracing	4th	Fall	4th
Library	DCSP	Concurrent task queue	2nd	Fall	1st
Operating System	OS	Futex system calls	2nd	Fall	2nd
ISA	MP	Futexes with assembly code	3rd	Spring	3rd

their implementation. First, it analyzes the implementation of spinning primitives with the support offered by the processor in the form of atomic memory access instructions and then it motivates the need of the operating system support to implement sleeping primitives.

Multiprocessors (MP) is a core course of the Computer Engineering major, and an optional course in other majors. MP focuses on the mechanisms that support the parallel execution of tasks in a computer system from the point of view of the architecture and the organization of a computer. More precisely, this course focuses on parallel processors with shared memory, which are basic elements of current complex digital systems. The covered topics include performance analysis, performance modeling, on-chip networks, atomicity, consistency, and coherence in the memory hierarchy of a parallel processor. In the laboratory sessions of the course, OpenMP is presented and used as a tool for parallel programming of computers, as well as tools for performance measurement.

3. Proposed Learning Experience

This section presents the proposed experience that helps students to accomplish an integrated view of a computer system. The lab materials and resources for each abstraction level consist of a description of the work to be done, code snippets, and a series of milestones, where each one builds on top of the previous one. The experience involves a total of eight hours, since each lab session comprises two hours in the course associated with the level. Interested readers may refer to the following repository with the source code of every lab: `https://github.com/universidad-zaragoza/learning-experience-ray-tracing`.

3.1. Overview

The proposed learning experience allows students to consolidate the concepts of parallelism, concurrency, consistency, and atomicity exploitable in current multicore computers. We focus on ray tracing, an appealing application which can be efficiently parallelized by learning and using the above concepts. Table 1 shows the involved four levels of abstraction and the associated courses. The table also shows, for each course, the academic year, semester, and chronological order in which the activities take place.

According to the chronological order, students start the experience in the second academic year. The first lab, which belongs to the DCSP core course, focuses on the library level. This lab deals with the implementation and management of a task queue with concurrent access by multiple threads. Synchronization aspects must be considered in order to avoid race conditions. To do so, students use a semaphore library for handling such synchronization questions.

The subsequent lab takes place shortly, during the same academic year and semester, and focuses on the Operating System level. In this core lab, a mutex is implemented with a futex (fast userspace mutex) mechanism through atomic primitives and operating system calls that are only invoked when the mutex is contested [27]. This mutex is then used to implement a new semaphore library, which replaces the one used in the previous lab.

The following academic year covers the third lab, that is, the Assembly level, which is developed in the optional MP course. In this lab, assembly instructions are used to implement the mutex/futex, which have the potential to achieve a greater efficiency in energy consumption and performance compared to library functions and system calls.

Finally, in the fourth year, the students focus on the Application level by implementing a ray tracer in a lab of the CG optional course. In this activity, the rendering of an image is parallelized by dividing the image into regions. These regions are assigned to different threads by using the concurrent task queue. At this moment, the students fully evaluate and state the differences of protecting concurrent accesses to the task queue by using library functions, system calls, or assembly instructions.

Note that the development of the presented experience is subject to certain risks; e.g., students transferring from one institution to another, or students failing a course or simply not choosing the involved optional courses would not complete the full experience. To mitigate such risks, all the labs include two parts. The first one, which is self-contained, includes the material for the actual lab, and the second part links the lab with the others. Therefore, if a student does not complete a preceding or following lab assignment, the faculty can provide a solution, so that students can accomplish the second part of the lab and establish the links between the abstraction levels.

3.2. Abstraction Levels

The application under study is presented in the next sections following the chronological order that students will experience.

3.2.1. ***Concurrent Task Queue***

The aim of this lab is the implementation of one of the most common concurrent data structures: a queue. Queues, whose sequential approach has already been studied in a previous course of Data Structures and Algorithms, are a very suitable mechanism for the collaborative work of a set of processes. In the considered case, producers and consumers can, in a natural way, use one or more queues to share information and to synchronize [28]. As in any shared data structure, in order to preserve data integrity, the concurrent access to the shared data requires the use of some synchronization mechanisms.

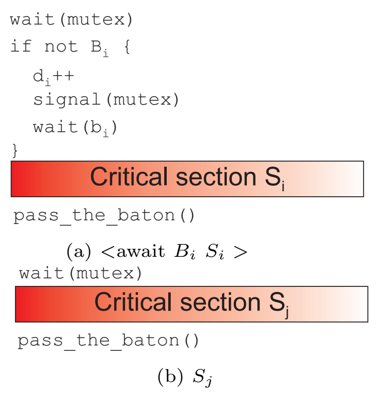
In this context, the main objectives of this lab work are as follows: i) to implement a concurrent bounded queue for generic data types, ii) to get familiarity with semaphores as a mean for solving synchronization problems, and iii) to use a general and powerful approach to solve general synchronization problems using semaphores.

Controlling the concurrent access to a queue requires to consider not only mutual exclusion access to some queue components, but also condition synchronization (no first element exists in an empty queue, or no new element can be inserted when the queue is full). According to the focus proposed for the DCSP course, as a first assignment, students have to design the concurrent access to the queue using the coarse-grain atomic statement `<await B S>`, where `B` is a boolean guard, usually concerning shared data, and `S` is a block of sequential statements. The semantics of the statement ensures that `S` starts its execution being `B` true, and the whole statement is atomically executed. The high-level point of view of such a statement makes easier the task of designing correct concurrent programs, which is one of the aims of the course. This will be done in the generic class `ConcurrentBoundedQueue` sketched in Listing 1, which includes a `BoundedQueue` as one of its attributes. Students have to code the complete data structure, including both enqueue and dequeue operations using semaphores for synchronization.

Listing 1: `ConcurrentBoundedQueue` generic class.

```
template <class T>
class ConcurrentBoundedQueue {
public:
    void enqueue(const T d);
    //<await this->bq->length()<N
    //     this->bq->enqueue(d)
    //>

    void dequeue();
    //<await this->bq->length()>0
    //     this->bq->dequeue()
    //>
    ...
private:
    int N; //size of the bounded queue
    BoundedQueue<T> *bq; //data storage
    ...
};
```



```
wait(mutex)
if not B_i {
  d_i++
  signal(mutex)
  wait(b_i)
}
```

```
pass_the_baton()
```

(a) <await B_i S_i >

```
wait(mutex)
```

```
pass_the_baton()
```

(b) S_j

Figure 1: Implementation of critical sections using (binary) semaphores.

In previous lectures, students have seen the *pass the baton* technique [29] to implement `<await ...>` statements using (binary) semaphores. This technique works as follows. Let us consider all the `<await` B_i S_i`>` and `<`S_j`>` sections that must be synchronized together. A mutex semaphore ensures exclusive access to such code areas. For each different B_i, a (binary) semaphore b_i with an initial count equal to 0 is added to block a process in the `<await` B_i S_i`>` statement when B_i is false. In addition, a counter d_i with an initial value set to 0 is included to store the number of processes blocked at B_i. Figure 1 illustrates an implementation of both `<await` B_i S_i`>` and `<`S_j`>` sections. The *pass the baton* implementation is sketched in Listing 2.

Listing 2: `pass_the_baton` function.

```
void pass_the_baton() {
    switch {
        ...
        Bi and di>0:
            di--
            signal(bi)
        ...
        otherwise:
            signal(mutex)
    }
}
```

In the case of the concurrent bounded queue, there are only two conditions: non-empty and non-full. Therefore, three semaphores and two counters will be used. The private part of the `ConcurrentBoundedQueue` class can be extended as shown in Listing 3.

Listing 3: `ConcurrentBoundedQueue` class including a synchronization mechanism.

```
template <class T>
class ConcurrentBoundedQueue {
public:
    ...
private:
    int N; //size of the bounded queue
    BoundedQueue<T> *bq; //data storage
    Semaphore *mutex; //initial count will
//be 1
    Semaphore *b_not_full; //to block until
//queue is not full
    Semaphore *b_not_empty; //to block
//until queue is not empty
    int d_not_full; //# of b_not_full-
//blocked processes
    int d_not_empty; //# of b_not_emtpy-
//blocked processes
    void pass_the_baton(); //to pass the
//baton
};
```

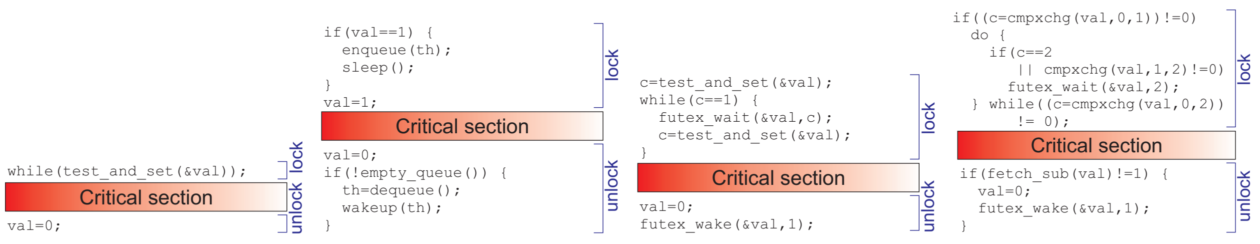
Since we want students to finally implement general semaphores, the presented technique will be developed using general semaphores instead of binary semaphores. Anyway, notice that the way the technique is implemented, every semaphore count is always 1 or 0, having a behavior equivalent to binary semaphores. At this level, the (binary) semaphores are the lowest abstraction construct to manage synchronization, considered as an abstract data type. The students know two possible semantics, equivalent from a safeness point of view, but with possible different liveness properties. However, they still do not know how a semaphore can be implemented and how operates internally, whether it makes a process spin until the access is granted or the process goes to sleep controlled by the operating system. These issues are outlined in the lab, and students will find out the answers by implementing the semaphore abstract data type in the two following labs.

As a final and optional lab assignment, students are asked to implement a second approach of the concurrent queue. In the first one, each operation on the queue is executed in mutual exclusion. In the second one, students have to adapt the readers-writers approach so as to allow multiple access to *reading* operations (operations with no side effects on the queue) while preserving mutual exclusion access for *writing* operations, giving priority to writers in case of conflict. During the lectures, students have already designed a solution based on the `<await ...>` instruction, whereas this assignment deals with the implementation.

After completing this lab, students will have reinforced their knowledge about the main concepts related to semaphore-based synchronization. In addition, the proposed assignments also deal with the use of design techniques focusing on the synthesis of correct concurrent programs.

3.2.2. ***Task Queue Protection with Futex System Calls***

This lab is intended to present the mechanisms required by the operating system to provide synchronization in concurrent algorithms. The main objectives of this lab are: i) show the operating system as a service provider for the user through system calls, ii) learn an efficient use of the futex system calls and the primitives of atomic instructions provided by the operating system and the C standard library, iii) understand the necessary mechanisms to provide execution in mutual exclusion with futexes and atomic instructions, and iv) show and use self-



```
while(test_and_set(&val));
Critical section
val=0;
```

(a) Spin-Lock (SL)

```
if(val==1) {
  enqueue(th);
  sleep();
}
val=1;
Critical section
val=0;
if(!empty_queue()) {
  th=dequeue();
  wakeup(th);
}
```

(b) Naive-sleep

```
c=test_and_set(&val);
while(c==1) {
  futex_wait(&val,c);
  c=test_and_set(&val);
}
Critical section
val=0;
futex_wake(&val,1);
```

(c) Basic-Sleep (BS)

```
if((c=cmpxchg(val,0,1))!=0)
  do {
    if(c==2
        || cmpxchg(val,1,2)!=0)
      futex_wait(&val,2);
  } while((c=cmpxchg(val,0,2))
        != 0);
Critical section
if(fetch_sub(val)!=1) {
  val=0;
  futex_wake(&val,1);
}
```

(d) Advanced-Sleep (AS)

Figure 2: Lock and unlock procedures of spin-lock and sleep mutexes. The sleep implementations include operating system calls to change the thread status.

implemented lock and unlock primitives of a mutex abstraction to manage the access to the concurrent task queue implemented in the previous activity.

The lab material firstly describes the C11 atomic instructions from `stdatomic.h` and solicits the students to implement a mutex with spin-lock based on atomic instructions. Next, the *sleep* approach of a mutex is motivated, introducing the mandatory intervention of the operating system to change the thread status, and providing a naive approach of the sleep mutex using hypothetical `sleep` and `wakeup` system calls as well as management operations on a system queue. The limitations of this approach are used to motivate the futex system calls. Then, the syntax and use of the parameters of the `futex_wait` and `futex_wake` system calls are described. By using these calls, the students are guided to implement an intuitive and straightforward version of the sleep mutex referred to as basic implementation. Finally, the pseudo-code algorithm of a more efficient mutex is offered as a guideline to code an advanced implementation. This approach is based on the mutex implementation proposed by U. Drepper [30], which is integrated into the Linux kernel [31].

Figure 2(a) shows the lock and unlock procedures of a Spin-Lock (SL) mutex protecting a critical section. The value of the userspace `val` variable represents the two states of the mutex: not taken (`val=0`) and taken (`val=1`). The `test_and_set` atomic instruction changes the mutex state[4]. More precisely, this instruction sets `val` to 1 and loads its previous value into `c` without the overhead of a system call. Then, a thread enters into the critical section if the lock is uncontested (`c=0`). Otherwise, the thread keeps spinning in the lock. In the unlock procedure, the thread simply sets `val` to `0` to release the mutex. Since the SL mutex leaves all the waiter threads in the lock awake, it may suffer system performance losses when the mutex is contested.

Figure 2(b) illustrates the naive-sleep approach of a mutex. These procedures are similar to other versions offered in textbooks of operating system concepts such as [32, 33], and [34]. This code is only correct if both procedures are executed atomically. However, assuming a non-atomic execution presents several problems that are listed in the lab material and should be understood by the students, specifically: i) the reading and writing operations of `val` are not atomically performed, which can lead to multiple threads reading the lock as not taken, ii) the reading of the lock and the insertion of the thread in the queue are neither atomic, which can lead to an indefinitely suspended thread if the lock is freed between the reading and insertion operations, and iii) after waking up from the `sleep` call, a thread has no guarantee of obtaining the lock in mutual exclusion since another thread can enter into the critical section before the former takes the lock.

Figure 2(c) shows the Basic-Sleep (BS) implementation addressing all the incorrect behaviors stated above. In the lock function, the atomic operation changes the state of the mutex. If the lock is uncontested, the kernel is not invoked and the thread enters into the critical section. Otherwise, the `futex_wait` system call is invoked. It suspends the calling thread in a system queue if the lock is still taken (`val=1`), or it returns immediately if the lock has been released in the meantime (`val=0`). In the first case, the thread remains suspended until another thread wakes it up. Notice too that every time `futex_wait` returns, the thread tries to acquire the lock again.

The unlock procedure sets `val` to `0` and calls `futex_wake`. This system call wakes up a number

[4]For the sake of brevity, we have shortened the original `stdatomic.h` function names; e.g., `test_and_set` corresponds to `atomic_flag_test_and_set` and the assignment operator for `val` refers to `atomic_store`.

of threads stated in the second argument (`1` in the example as only a single thread is allowed to enter into the critical section) from those suspended in the system queue. Notice that such a call is invoked regardless of the lock is uncontested or not, which may impact on the system performance.

The Advanced-Sleep (AS) implementation shown in Figure 2(d) addresses the performance problem of the basic approach. In this case, there are three mutex states: not taken (`val=0`), taken and no waiter threads (`val=1`), and taken and at least one waiter thread (`val=2`). In the lock procedure, `test_and_set` is no longer useful since `val` takes three values. Instead, the atomic `cmpxchg` primitive is used, in which a `1` (*desired* third argument) is loaded into `val` on a successful comparison between `val` and `0` (*expected* second argument). Regardless of the result of the comparison, the original value of `val` is loaded into `c`. If `c==0`, the calling thread updates the state of the mutex as taken and no waiters, and then enters into the critical section. Otherwise, the thread is suspended in the system queue by calling `futex_wait`. Previously, the second `cmpxchg` sets `val` to `2` if necessary, updating the state of the mutex as taken and at least one waiter. Note that, if the lock is freed between the first and second `cmpxchg`, the latter returns 0 and the thread is not suspended. The third `cmpxchg` ensures that a thread takes the mutex only if a 0 is returned. In such a case, `val` is set to `2` because there is no certainty of the number of waiters.

The unlock method subtracts 1 to `val` with the atomic `fetch_sub`, which returns the previous value of the argument. The `futex_wake` call is invoked just in the case of a suspended thread in the lock, avoiding such costly system calls when there are no waiter threads. The reader is referred to [30] for further details about the AS mutex implementation.

Once the different mutexes have been coded and understood, the students use them to support a complex abstraction, that is, the concurrent task queue implemented in the previous lab. Listing 4 shows an implementation alternative of the `Semaphore` class introduced in Listing 1, referred to as `Library`. This approach uses standard library mutexes (`std::unique_lock <mutex>`) to ensure mutual exclusion in `Semaphore` class methods. On the other hand, Listing 5 shows a different implementation of the same class, referred to as `Thread-suspension`, in which the `lock` and `unlock` methods are replaced by the procedures of each mutex version (see Figure 2).

Listing 4: **`Library`** implementation alternative of the `Semaphore` class.

```
class Semaphore {
private:
    std::mutex mtx;
    std::condition_variable_any cv;
    int count=0;
public:
 ...

    void Semaphore::signal() {
        std::unique_lock<mutex> lck(mtx);
        count=1;
        cv.notify_all();
    }

    void Semaphore::wait() {
        std::unique_lock<mutex> lck(mtx);
        while(count == 0)
            cv.wait(lck);
        count=0;
    }
};
```

Listing 5: `Thread-suspension` implementation alternative of the `Semaphore` class. Mutex `lock` and `unlock` procedures refer to the different approaches from Figure 2.

```
class Semaphore {
private:
    std::mutex mtx;
    int count=0;
public:
 ...

    void Semaphore::suspend(int ve) {
        syscall(__NR_futex, &(count),
FUTEX_WAIT, ve, NULL, 0, 0);
    } //suspends thread iff ve!=count

    void Semaphore::wakeup() {
        syscall(__NR_futex, &(count),
FUTEX_WAKE, INT_MAX, NULL, 0, 0);
    } //wake up all suspended threads

    void Semaphore::signal() {
        mtx.lock();
        count=1;
        wakeup();
        mtx.unlock();
    }

    void Semaphore::wait() {
        mtx.lock();
        while(count == 0) {
            int vr = count;
            mtx.unlock();
            suspend(vr);
            mtx.lock();
        }
        count=0;
        mtx.unlock();
    }
};
```

For both implementation alternatives, the `wait` method consists of a loop over a `count` variable, but, if `count` equals to 0, the current thread suspends its execution. On the other hand, the `signal` method wakes up all suspended threads after freeing the semaphore. In order to implement such a functionality for `Thread-suspension`, the `suspend` and `wakeup` methods are coded by the students using futex system calls. On the contrary, the `Library` approach relies on condition variables to suspend/wake up threads. Using both alternatives, the students assess the suitability of not only their coded spin-lock and sleep mutexes, but also a library mutex by measuring the execution time and the chip temperature under different contention scenarios (see Section 5.2).

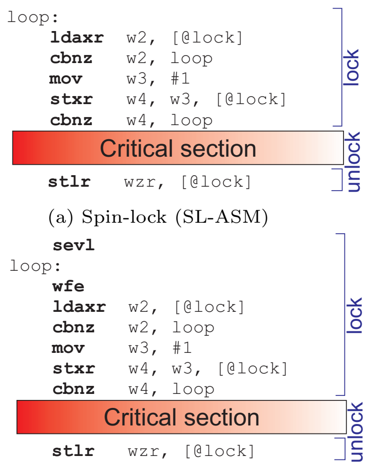
Overall, the students will be able to use futex system calls and atomic instructions to implement spin-lock and sleep versions of a basic synchronization abstraction such as a mutex, incorporate such approaches to protect the concurrent task queue, and experimentally state the performance differences among them.

3.2.3. Futexes with Assembly Code

The main purpose of this lab is to help students understand the support provided by the ISA level to implement fast and reliable mutual exclusion, in terms of consistency and atomicity. The ARM processors include load-link/store-conditional instructions and memory barriers, providing the foundation for higher level structures such as mutexes and futexes. In addition, these instructions do not require any privilege level for being executed, so programmers can directly exploit them to improve efficiency and reduce the overhead of systems calls.

By the end of this lab, students will have accomplished the following goals: i) understand how atomic instructions operate at the ISA level for the ARMv8 processors, ii) know why data memory barriers are often required when writing atomic instructions, and iii) learn the performance and energy implications of the different mutex implementations.

The assignments of this lab are designed to help students to engage with complex code enhancing their low-level programming skills, especially concerning performance and energy efficiency. In addition, they show how important is for an ISA to provide support for complex high-level constructors such as the mutexes used by operating systems, libraries, and applications. Finally, students gain knowledge on the relation between the C/C++11 memory model and the corresponding consistency models at the ISA level.



```
loop:
    ldaxr   w2, [@lock]
    cbnz    w2, loop
    mov     w3, #1
    stxr    w4, w3, [@lock]
    cbnz    w4, loop
    stlr    wzr, [@lock]
```

(a) Spin-lock (SL-ASM)

```
    sevl
loop:
    wfe
    ldaxr   w2, [@lock]
    cbnz    w2, loop
    mov     w3, #1
    stxr    w4, w3, [@lock]
    cbnz    w4, loop
    stlr    wzr, [@lock]
```

(b) Low-power state (LPS-ASM)

Figure 3: Lock and unlock procedures with ARMv8 assembly code.

The lab material of this session is organized in two parts. In the first part, the students are asked to generate a race condition with the writing of a multi-threaded program that reduces an array by adding all the elements without synchronization primitives. Then, the students code a *fetch and add* primitive with ARMv8's *load-link* (`ldaxr`) and *store-conditional* (`stlxr`) instructions [35]. The implemented *fetch and add* is included in the previous program to verify that the code is now free of race conditions.

The second part comprises two assignments. The first one proposes a basic implementation of lock and unlock mutex functions based on `ldaxr/stlxr` instructions as plotted in Figure 3(a). Threads in the lock function spin until they acquire the lock. This mutex approach is referred to as SL-ASM. The spinning can occur at the two `cbnz` instructions. Either if the lock is already taken (first `cbnz`) or the `stxr` instruction fails the attempt to take the lock (second `cbnz`), the branch instructions return the flow to the beginning of the loop. Notice too that, likewise the SL and BS implementations from the OS level, just two mutex states, taken and not taken, are considered in the assembly level.

The second assignment proposes an advanced implementation of the lock function by replacing the power-hungry spin-lock with a `wfe` instruction. This instruction puts the core into a low-power state without returning the control to the operating system.

Figure 3(b) shows such an energy-efficient implementation, referred to as LPS-ASM, also with the two mutex states taken and not taken. The student will learn how the operating system considers that the program is running, while it is actually waiting for the lock to be released, and how the thread can regain the lock without a system call. In particular, the `stlr` instruction, located in the unlock function, performs a store with a release barrier and wakes up any core that could be in a low-power state after executing a `wfe` instruction. To guarantee progress, the cores also leave the low-power state after an interruption occurs (e.g., a context switch).

With both SL-ASM and LPS-ASM implementations, students will carry out a quick comparison between them in terms of performance and energy consumption. Since Raspberry does not provide energy hardware counters, as an indirect measurement, we periodically measure the temperature provided by the chip itself, which is mapped by the OS in the filesystem[5].

3.2.4. ***Parallel Ray Tracing***

The CG course proposes a practical assignment involving the implementation of a ray tracing algorithm [36], which is parallelized by assigning different tasks (partitions or regions of the expected synthesized image) to different threads. The main objectives of this lab are: i) find and understand the computational bottlenecks of the algorithm, ii) devise parallelization strategies that affect performance without any accuracy loss, and iii) test, explore, and analyze the impact (and overhead) of the combination of different parallelization strategies, including partitioning structures and thread assignment methodologies, on performance.

The contents of this lab include a description of the ray tracer, and an introduction on how to parallelize it. In particular, this assignment makes use of the minimalist C++ *smallpt* ray tracer by K. Beason[6]. This algorithm generates a 2D image from a 3D representation of a virtual scene, including geometry and optical properties of the objects and physical characterizations of sensors (cameras) and light sources. In practice, the algorithm simulates light transport paths across the virtual scene in order to obtain the final color that reaches each of the pixels of the image. Paths are generated from

[5]Temperature can be found in the path `/sys/devices/virtual/thermal/thermal_zone0/temp`

[6]http://www.kevinbeason.com/smallpt/

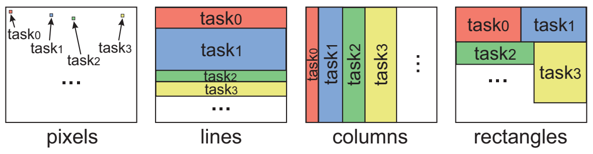


Figure 4: Diagrams of a 2D image split into render tasks with different kinds of image regions and sizes.

the camera and traverse each pixel independently. Since the computation associated to each pixel is independent, the algorithm is highly parallelizable. Moreover, such a parallelization is worthwhile because the algorithm is computational intensive and takes quite a long time to converge (about 1 or 2 hours for a good quality result for a simple virtual scene, and even days in the case of more complex scenes).

A common ray tracing parallelization strategy is to subdivide the image into different regions, converting the computation of each of the regions into a render task to be assigned to an execution thread. The students are required to explore different parallelization strategies in different dimensions as illustrated in Figure 4:

- Different kinds of image regions: pixels, lines, columns, or rectangles.
- Different region sizes: smaller or larger rectangles and line or column batches.

Depending on the geometry and other properties of the virtual scene, and the different implementation details of the algorithm, the computational load can vary greatly from one region to another [37]. For this reason, we need a safe mechanism to distribute tasks among threads. This assignment can be static (pre-assigned per thread) or dynamic (using a concurrent task queue).

Listing 6 shows the implementation of a static parallelization assignment, distinguishing between the generation of the regions and the rendering process. In this strategy, prefixed indices are computed as a vector of regions and threads access to such indices during the rendering process.

Since it is impossible to estimate the computational load of each task beforehand, a dynamic assignment is likely to be more efficient. Listing 7 shows this implementation, where the vector of regions is replaced by the `ConcurrentBoundedQueue` class from the former lab, including the enqueue and dequeue operations in the generation and rendering processes, respectively. Figure 5 depicts a diagram of such a concurrent task queue where a main thread generates and enqueues tasks, whereas multiple worker threads dequeue tasks and perform the rendering process in parallel. Finally, notice too that both static and dynamic strategies are orthogonal to the region distributions depicted in Figure 4.

Listing 6: `Static` parallelization strategy for computer graphics algorithms.

```
struct Region {
    int row0, col0, row1, col1, spp;
};

//Producer
void generate() {
    Image image(width,height);
    std::vector<Region> regions;
    for (region in <regions according to
strategy>)
        regions.push_back(region);
    std::vector<std::thread> threads;
    for (int i=0; i<n_th; ++i)
        threads.push_back(std::thread(
render,i,n_th,regions,image));
    for(auto &t : threads)
        t.join();
}

//Consumer
void render(unsigned int thread, unsigned
int nthreads, std::vector<Region>& parts,
Image& image) {
    for (int p=thread; p<parts.size();
p+=nthreads)
        for (int row=parts[p].row0; row<
parts[p].row1; ++row)
            for (int col=parts[p].col0;
row<parts[p].col1; ++col)
                for (int s=0; s<spp; ++s)
//rays per pixel
                    image(row,col) +=
calculatepixel(row,col);
}
```

Listing 7: `Dynamic` parallelization strategy for computer graphics algorithms.

```
//Producer (main thread)
void generate() {
    Image image(width,height);
    std::vector<std::thread> threads;
    ConcurrentBoundedQueue<Region> regions;
    for (int i=0; i<n_th; ++i)
        threads.push_back(std::thread(
render,i,n_th,regions,image));
    for (region in <regions according to
strategy>)
        regions.enqueue(region);
    for(auto &t : threads)
        t.join();
}

//Consumer (worker threads)
void render(unsigned int thread, unsigned
int nthreads, ConcurrentBoundedQueue<Region
>& parts, Image& image) {
    while (!parts.done()) {
        Region part = parts.dequeue();
        for (int row=part.row0; row<part.
row1; ++row)
            for (int col=part.col0; row<
part.col1; ++col)
                for (int s=0; s<spp; ++s)
//rays per pixel
                    image(row,col) +=
calculatepixel(row,col);
    }
}
```

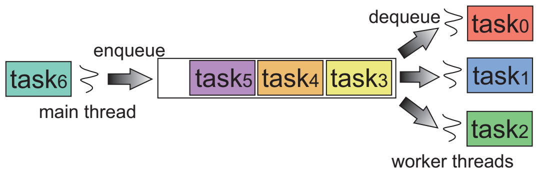


Figure 5: Concurrent thread-safe task queue to assign tasks to different worker threads.

The students should identify the different pros and cons of each of the approaches, analyzing and justifying their impact on performance. For instance, the students should answer questions such as *which is the optimal region size? Which of the mutex approaches of the task queue work best and under which circumstances?*
The final part of the lab notes that the proposed approach relies on *low-level* programming constructs that are helpful to showcase the interactions with the rest of the abstraction levels. However, to boost productivity and get the most of heterogeneous systems, which are standard nowadays, students should be advised to opt for higher level approaches [38, 39, 40].
Overall, the implementation and parallelization of the path-tracing algorithm together with the performance evaluation of each mutex will help students understand and analyze the effect of low-level mechanisms, decisions, and implementation details with high-level applications and algorithms, which will reinforce the integrated view of a computer system.

4. Experimental Environment

To consolidate the overall view of the presented computer system, we propose to use the same single-board computer in all the labs. To this end, we have analyzed a subset of commonly used boards that fulfilled two key restrictions: low-cost (price below 50 $) and multiprocessing (parallelism experiments cannot be run in single-core boards).
The selected boards are Raspberry Pi 3 Model B [41],

Table 2: Hardware (H) and Software (S) requirements evaluated for the following boards: Raspberry Pi 3 Model B (RP), ClockworkPi (CP), Rock64 (RC), Le Potato (LP), Orange Pi zero plus (OP), NanoPi M1 Plus (NP), and Pine A64-LTS (PA).

Type	***Description***	RP	CP	RC	LP	OP	NP	PA
H	**JTAG**	✗	✗	✗	✗	✗	✗	✗
H	**Ethernet**	✓	✓	✓	✓	✓	✓	✓
H	**WiFi**	✓	✓	✗	✗	✓	✓	✗
H	**Camera**	✓	✗	✗	✗	✓	✓	✗
H	**Virtualization support**	✓	✓	✓	✓	✓	✓	✓
H	**I/O Extensions (screen, buttons...)**	✓	✓	✓	✓	✓	✓	✓
H	**GPU**	✓	✓	✓	✓	✓	✓	✓
S	**Development Framework options**	✓	✗	✓	✗	✓	✓	✓
S	**SDK and runtime GPU support**	✓	✗	✓	✗	✓	✓	✓
S	**High-level/Standard OS support**	✓	✗	✓	✓	✓	✓	✓
H&S	**Bare metal (no OS) support**	✓	✓	✓	✓	✓	✓	✓

ClockworkPi[7], Rock64[8], AML-S905X-CC (Le Potato)[9], Orange Pi Zero Plus [42], NanoPi M1 Plus[10], and Pine A64-LTS[11].
Table 2 summarizes the most relevant hardware and software requirements for the development of this experience, and which of them are met by the selected boards. Of course, there are other boards offering better performance or more functionality but at a higher cost such as DragonBoard 410C [43], HiKey 960[12], or BeagleBoard X-15 [44]. Such highly-priced boards are not considered in this study.
The list of requirements is mainly focused on the subset of courses taking part in the presented experience. Nevertheless, it is desirable to choose a base board that allows future expansions by adding more courses to the experience. Therefore, we consider a broader range of requirements that would facilitate the use of the selected board for additional courses, such as Computer Architecture and Organization, Systems Administration, Computer Networks, Security, Artificial Intelligence, Machine Learning, Embedded Systems, Robotics, Video-games, or Computer Vision, among others.
Considering the results from our study of boards, requirements, and potential courses that could use them, Raspberry Pi, Orange Pi, and NanoPi turn out good choices to be used in our experience, since they meet all the requirements but the JTAG support. However, we finally chose Raspberry Pi primarily due to its broader usage and large amounts of open source and available materials [45].

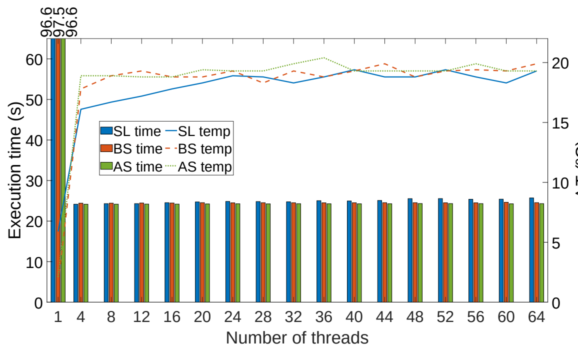
5. Technical Results

This section presents the main technical results and conclusions that should be obtained by students from the proposed lab assignments. More precisely, the impact on the system performance and chip temperature obtained for every mutex implementation is analyzed under different contention scenarios.
All the experiments are run in a Raspberry Pi 3 Model B, which includes a quad-core processor where each core is single-thread. We assumed a fixed CPU frequency of 1.4 GHz (*performance* governor) for all the experiments to guarantee reproducibility. The OS is an Ubuntu 18.04.3 LTS release with a gcc 7.5.0 compiler. Before running each experiment, we wait until the chip temperature decreases to a defined threshold of 60 ºC (the reported experiments were carried out during summer and the board does not include any heat dissipator nor fans). Once an experiment finishes, we calculate the execution time and the chip temperature increase and wait for the CPU to cool down before running the next experiment. All the presented experiments are run from 1 to 64 threads. Taking into account the quad-core processor and the fact that the studied applications are CPU-bound, the thread oversubscription (i.e., a number of threads higher than the number of physical cores) penalizes performance, which is a key insight for students. In addition, another important key insight from thread oversubscription is that the implementation of each mutex affects performance differently.

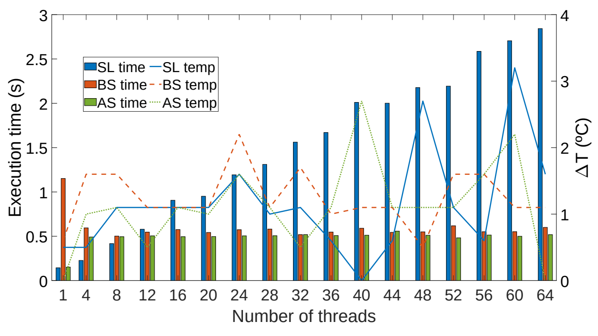
5.1. OS Level

The first lab assignment deals with the implementation of the concurrent bounded queues and the experiments are limited to ensure the correctness of such queues (Section 3.2.1). In the second lab, the first assignment refers

[7]`https://wiki.clockworkpi.com/index.php/Main_Page`
[8]`https://wiki.pine64.org/index.php?title=ROCK64`
[9]`https://libre.computer/products/boards/aml-s905x-cc/`
[10]`http://wiki.friendlyarm.com/wiki/index.php/NanoPi_M1_Plus`
[11]`https://wiki.pine64.org/index.php?title=Main_Page`
[12]`https://www.96boards.org/documentation/consumer/hikey960/hardware-docs/hardware-user-manual.md.html`



(a) Real contention



(b) Synthetic contention

Figure 6: Performance and temperature of the OS mutex implementations varying the number of threads.

to the implementation of the SL, BS, and AS mutexes using futexes (Section 3.2.2). This section evaluates such implementations.

Bars and lines in Figure 6 show respectively the execution time in seconds (s) and the chip temperature increase in Celsius degrees (ºC). Threads access a shared variable protected with a mutex. In particular, each thread acquiring the lock increments by one the value of this variable and then releases the lock. The execution finishes when the shared variable reaches a value of a million.

Two different contention scenarios are considered, referred to as real and synthetic. In the real scenario, a thread releasing the lock computes some private work consisting of a series of trigonometric functions with the shared variable as input. On the other hand, in the synthetic scenario, after releasing the lock, a thread immediately competes for the lock without performing any private work. The latter scenario covers an extreme case where students observe how a change in the amount of private work leads to unexpected conclusions.

In the real scenario, all the mutexes obtain a very similar performance for a given number of threads greater or equal than 4. For a low number of threads between 1 and 12, the overhead of always invoking a futex system call in the unlock method and the subsequent context switch in BS leads to a slight increase of the execution time compared to SL and AS. However, as the number of threads increases, SL progressively enlarges a bit the execution, since threads spin in a more disputed lock. On the other hand, the sleep mutexes maintain the same execution time as the number of threads increases. With this experiment students realize that, for this kind of application, the thread oversubscription does not improve performance but, on the contrary, depending on the mutex implementation, performance can be hurt. The increase in the chip temperature is quite steady for a number of threads greater than one, and both the private work and the chip temperature limit (around 80 ºC) prevents from obtaining significant CPU temperature differences among the different mutexes. However, SL reaches a lower temperature in most cases, which suggests that invoking futex systems calls has a greater thermal signature.

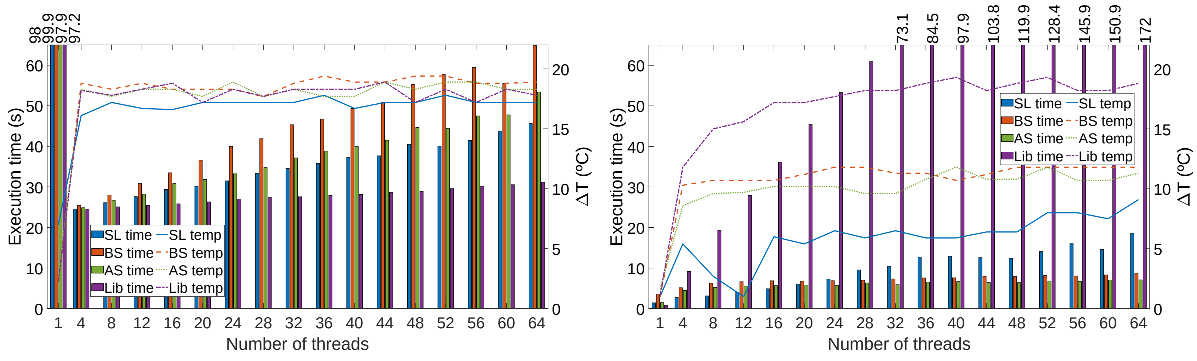
There is no parallelism to be exploited in the synthetic scenario, meaning that single-threaded executions should exhibit the best performance for a given mutex implementation. This is the case for both SL and AS mutexes. For the studied mutexes, SL obtains the lowest execution time both in the single-threaded execution and when the number of threads coincides with the number of physical cores. This confirms the overhead of the futex system calls in both BS and AS approaches. On the other hand, similarly to the previous scenario, under the thread oversubscription, spinning largely increases the execution time over the sleep approaches, whereas such mutexes maintain roughly the same performance.

An interesting observation is that, for a single thread, BS has a much larger execution time not only compared to the other mutexes but also compared to itself when multiple threads are considered. This confirms the overhead of always invoking costly `futex_wake` system calls in the unlock function even when there are no waiter threads in the lock. Regarding the chip temperature, the execution time is not sufficiently large to observe significant temperature differences before and after the execution.

5.2. Library Level

This section refers to the evaluation of the OS second lab assignment, where the previous mutexes are used to support the concurrent task queue. Figure 7 shows the results. A library mutex from the DCSP lab is included for comparison purposes. The length of the queue is a million of elements and its initial state is full. Threads within the critical section dequeue values from the queue until it is empty. The private work of each thread is the same as described in the previous section.

The real scenario confirms that the best performing number of threads is 4, where both SL and the library



(a) Real contention

(b) Synthetic contention

Figure 7: Performance and temperature of the OS mutex implementations protecting a concurrent bounded queue. A library mutex is included for comparison purposes.

mutex obtain the best performance. This suggests that the library mutex implements a spin-lock mechanism for such a number of threads. The thread oversubscription leads to larger execution times, especially for SL, BS, and AS. The library mutex clearly outperforms all the other mutexes for any number of threads greater than 4, suggesting that, differently from the spin-lock and OS-based solutions, the library implementation possibly throttles unnecessary threads according to the number of processor cores and/or exploits an internal userspace sleep-queue to minimize thread context switches. Contrary to the previous experiments, SL always performs better than AS and BS mutexes. This is mainly due to the additional overhead of the futex system calls from the new suspend and wakeup methods in the Semaphore class. As expected, AS largely reduces the execution time with respect to BS due to the additional cost of always invoking `futex_wake` system calls. Similarly to the previous study, SL shows a slightly lower increase in the CPU temperature compared to the other mutexes, including the library mutex.

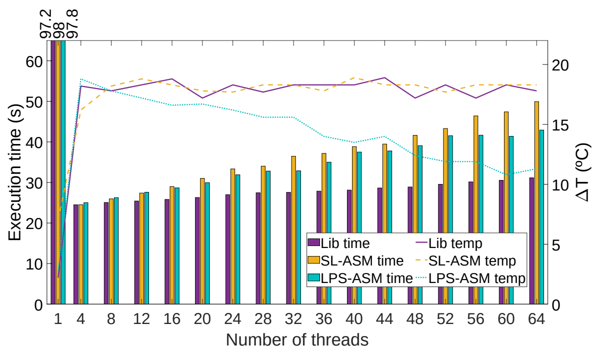
Surprisingly, the synthetic scenario greatly affects the library mutex with the number of threads, which points out that such an extreme contention scenario has not been considered in the library mutex development. The higher contention also affects SL, although in a lesser way. In this scenario, with a sufficient number of threads, the overhead of the system calls and subsequent context switches compensates the distribution of CPU time among all the active threads spinning in the lock. Removing the private work allows to see significant temperature differences. The results confirm the lower thermal signature of SL, even in those cases where SL enlarges the execution time with respect to the other approaches. On the other hand, the library mutex shows the highest temperature increase, most likely due to the extended execution times.

Overall, the library mutex is a convenient choice in scenarios where there is a relatively high amount of private work to be done. In this case, the library likely exploits a hybrid management by combining spin-lock, thread throttling, and/or a sleep-queue to provide an adaptive mechanism to the most frequent case. However, in the synthetic scenario, the library heuristics fail to adapt to such an atypical case, making the OS-based mutexes the preferable choice.

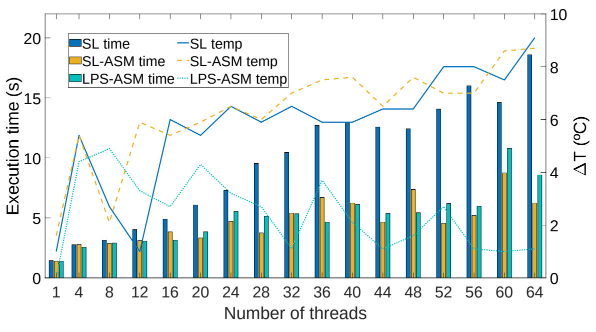
5.3. Assembly Level

This study evaluates the SL-ASM and LPS-ASM mutexes protecting the concurrent task queue (see Section 3.2.3). The initial state of the queue and the amount of private work is the same as in the previous study. Figure 8 depicts the results. For illustrative purposes, just the best performing mutexes with 4 threads from the previous study (library and SL for real and synthetic contention, respectively) are shown.

Similarly to the previous studies, the spin-lock solution increases the execution time with the thread oversubscription in the real contention scenario. Compared to SL-ASM, LPS-ASM has a lower execution time since putting cores in a low-power state until a context switch is triggered reduces the lock contention. However, this does not prevent the library mutex to obtain better performance thanks to an enhanced management of the lock. Differently from previous approaches, the LPS-ASM alternative shows a temperature reduction with the thread oversubscription. This is mainly due to, with a higher number of threads, the chance to put cores in a low-power state increases. On the contrary, the steady temperature of the library mutex suggests that such an implementation does not change the state of the cores. In the synthetic scenario, both spin-lock alternatives progressively increase the execution time with the number of threads. However, implementing the spin-lock in a higher abstraction level with respect to the ISA level introduces a performance overhead according to the timing differences between SL and SL-ASM. Finally,



(a) Real contention



(b) Synthetic contention

Figure 8: Performance and temperature of assembly mutexes protecting a concurrent bounded queue. The best performing solutions from the previous study are also shown.

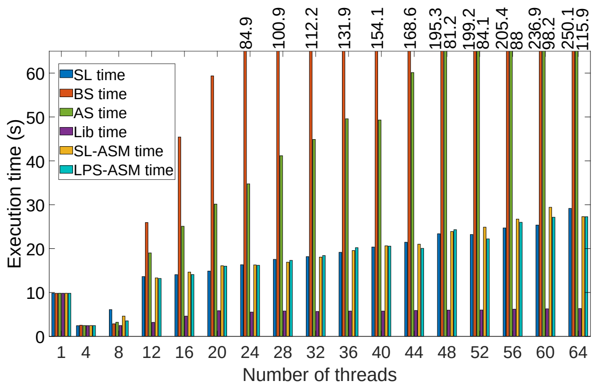


Figure 9: Performance of all the mutex implementations at the path-tracing application level assuming a dynamic parallelization strategy.

the LPS-ASM solution clearly reduces the temperature with respect to both spin-lock approaches, which show very similar temperature numbers.

5.4. Application Level

This section evaluates all the mutexes in the ray tracing application using the dynamic parallelization strategy with the concurrent task queue (see Section 3.2.4). Figure 9 plots the results, which are restricted to performance for illustrative purposes. The rendered scene in all the experiments is *forest*[13]. For simplification purposes, the image is partitioned in fixed-size rectangular regions.

As expected from the real scenarios of the previous studies, the best performing number of threads is 4 according to the number of physical cores, where all the analyzed mutexes exhibit a very similar performance. In addition, the performance of the ray tracer scales with the number of threads, since the execution time with 4 threads reduces by 4× with respect to the single-threaded performance. Results under the thread oversubscription corroborate the previous findings from lower abstraction levels: the OS overhead largely penalizes the AS mutex and especially the BS mutex, whereas the OS-free solutions like SL and the assembly mutexes show similar results. Finally, the library mutex is scarcely affected by the thread oversubscription.

6. Experience Assessment

This section provides a qualitative assessment of all the proposed labs. The attendance to such labs was voluntary (37 students participated in each lab of the DCSP and OS courses, and 12 students participated in each lab of the MP and CG courses). All labs were scheduled after the completion of the courses, giving the students the opportunity to compare between the current lab assignments (i.e., no direct interactions with any other lab) and the proposed lab assignments. Notice too that this experience has been carried out for three consecutive academic years, meaning that those students that have participated in the latest CG lab have also participated in the remaining ones. Assessment results collected for the DCSP and OS labs correspond to 2 years: current and previous, whereas results for the MP and CG labs correspond to the previous and current year, respectively. In the first year, the first two labs were run without collecting assessment results.

Two different surveys were designed for each lab, referred to as pre-survey and post-survey. Students filled out the surveys before and after completing the proposed lab sessions. Every pre-survey consists of two types of questions: a series of questions for measuring the perception of students about the interactions among the courses of the CE program in general and particularly among the involved courses in the experience, and 22 theoretical

[13]http://www.kevinbeason.com/smallpt/extraScenes.txt

Table 3: Correct answer rates of the common questions for pre and post surveys.

(a) DCSP

	Q1	**Q2**	**Q3**	**Q4**
Pre	100	72.9	27.1	56.7
Post	100	97.3	43.2	59.5

(b) OS

Q5	**Q6**	**Q7**	**Q8**	**Q9**	**Q10**
97.3	81.1	48.7	67.6	81.1	10.8
97.3	97.3	91.9	100	100	100

(c) MP

Q11	**Q12**	**Q13**	**Q14**	**Q15**
75	41.7	41.7	41.7	33.3
100	91.7	83.3	100	75

(d) CG

Q16	**Q17**	**Q18**	**Q19**	**Q20**	**Q21**	**Q22**
75.0	83.3	33.3	41.7	33.3	50.0	58.8
91.7	100	66.7	83.3	100	83.3	83.3

and practical questions for assessing the knowledge of students about the covered PDC topics.

Every post-survey is structured in four parts: a set of questions for stressing in the interactions among the courses, a series of questions for valuing the proposed lab experiences, other questions referring to the obtained technical results, and the same 22 theoretical and practical questions from the pre-surveys. The main conclusions extracted from these surveys are summarized next.

Before the lab sessions, the students considered that all the courses of the program are somehow related (8.0 points out of 10, where the edge scores 0 and 10 indicate no relation at all and totally related, respectively), but that the faculty should make an effort to make this relation more explicit (the effort was rated 7.9 out of 10). Regarding the courses involved in this learning experience, the students were very conclusive. All of them considered that the four courses are strongly related to other courses of the program, including between themselves. The responses of the post-surveys strengthened the previous results, since the students conveyed that the courses are more related to each other (8.7 out of 10).

Table 3 compares the correct answer rates of the 22 theoretical and practical questions that were included in both kind of surveys. Questions Q1-Q4 were proposed by the faculty of the DCSP course and mainly focus on the concept of mutual exclusion, the semantics of semaphores, and the use of semaphores to solve certain critical section problems. Questions Q5-Q10 refer to the OS session and are about the low-level requirements to implement different types of semaphores. Questions Q11-Q15 refer to the MP course and are about the role of atomicity and consistency in thread synchronization. Questions Q16-Q22 correspond to the CG course and deal with the impact of the different abstraction levels in the parallel ray tracing application. The reader is referred to Appendix A for further details about these questions, including the possible and the correct/desired answers, highlighted in boldface.

In general, the post-survey rates improve the correct answer rates obtained in the pre-surveys. Overall, the results support the interest of the experience and the usefulness from the students point of view. On average, the correct answer rate of the post-surveys is 88.0% versus 57.3% obtained before the beginning of the sessions. One remarkable aspect to be noticed is the rather low rate of correct answers to questions *Q3* and *Q4*. In our opinion, the main reason is that the conceptual study of the different semantics of the semaphore abstract data type has been presented and discussed two months before the experience has been carried out. For a set of reasons, we are currently thinking about the possibility of changing the order in which concepts are introduced at the OS course, so as to be able to study the semaphore abstraction and implementation closer in time. Maybe this fact will help us to get more information to understand the reason for such a low rate.

When comparing year-on-year results of the surveys, we have shown some striking facts. This is the case of questions Q16 and Q17 of the CG lab, which are the same as questions Q8 and Q9 of the OS lab. In the survey after the OS lab, 100% of the students answered them correctly. However, two years later in the previous survey of the CG lab, these same questions were answered correctly by *only* 75% and 83.3% of the students, respectively. These results demonstrate the effect of time on students' memory. Fortunately, the CG lab session helps to refresh students' memory by reaching 91.7% and 100% on the post-survey.

The post-surveys revealed that all the lab sessions were well received. All the students completed the lab assignments, and they gave an overall score of 8.8 out of 10 to the quality of the lab designs, the materials and resources, and the faculty assistance. When asked about their opinion in the CG lab, one student mentioned that he/she *"liked a lot the idea of unifying several courses in a single learning experience, and this should be done more often to consolidate the learning"*. Another student pointed out that *"this is a very positive initiative to put together everything we have seen in multiple courses"*.

In general, after completing the labs, the students have reached a broader view of the interactions among operating systems, computer architecture, and parallel and distributed computing. As learning outcomes, students discerned among the different mutex implementation alternatives and clearly identified the programmability, execution time, and efficiency trade-offs at each abstraction level.

7. Conclusions

The current structure of the Computer Engineering (CE) program, arranged in isolated courses, causes students to lose sight of the overall view of a typical computer system organized in abstraction levels. This paper has presented a learning experience that aims to reinforce this vision as a whole.
The presented experience covers the abstraction levels of Application, Library, Operating System, and Instruction Set Architecture, and consists in the implementation of a parallel ray tracing algorithm that uses a concurrent queue to assign tasks to different execution threads. The accesses in mutual exclusion to this queue are managed by mutexes implemented with either library functions, system calls, or assembly instructions.
The aforementioned abstraction levels have been introduced and related to each other in a subset of laboratories from different courses of a CE program, allowing students to consolidate the concepts of parallelism, concurrency, atomicity, and consistency. This paper has presented the structure and contents of each proposed lab, as well as the interactions with the remaining labs. In addition, a detailed study of the hardware and software requirements and the consequent choice of Raspberry Pi as the common hardware development platform is also discussed.
Experimental results consisted of a technical evaluation and an assessment study of the proposed learning experience. The technical results referred to an evaluation and discussion of the performance and temperature differences of the implemented mutexes in each lab. The experience assessment consisted of a series of pre/post surveys. Most students pointed out an enhancement in the design of the labs and a greater exposure to the relations between courses. In addition, the students showed an enhancement in the integrated perception of the addressed concepts and the acquisition of the knowledge, since the correct responses to the technical questions from the surveys improved by 30.7% after the experience.

Appendix A. Surveys

Appendix A.1. DCSP survey

Q1 Do you understand the importance of mutual exclusion?

- **Yes**
- No
- I have a rough idea

Q2 When a `P` process is said to use an `S` semaphore with busy-wait semantics, which of the following operations refers to `wait(S)`?:

- Option 1:

```
< if S.V>0
    S.V := S.V−1
  else
    S.L := S.L ∪ {P}
    P.state := blocked
  end
>
```

- **Option 2:**

```
< await S.V>0
    S.V := S.V−1
>
```

- To anyone, both are equivalent

Q3 If the semantics for semaphores in the first two answers from the previous question are taken into account, which of the following statements can be considered as true?

- Since they are equivalent, there are no differences in terms of liveliness and safety properties
- **There are no differences in terms of safety properties, but there are differences in terms of liveliness**
- There are no differences in terms of liveliness properties, but there are differences in terms of safety
- Although they are not equivalent, there are no differences in terms of liveliness and safety properties

Q4 Consider the following scheme to solve the problem of the critical section, `S` being a semaphore with busy-wait semantics. Which of the following statements is correct?

```
Semaphore S:=1
Process P                    Process Q
  loop forever                 loop forever
    SNC                          SNC
    wait (S)                     wait (S)
    SC                           SC
    signal (S)                   signal (S)
  end                          end
end                          end
```

- **The solution can generate fairness problems**
- There are no fairness issues
- There will be fairness issues depending on the first process that gets access to the semaphore

Appendix A.2. OS survey

Q5 Do you know the advantages and shortcomings of a mutex with/without active waiting?

- **Yes**

- No
- There are no differences

Q6 Would you be able to implement a mutex with busy waiting?

- **Yes**
- No
- I am not sure

Q7 Would you be able to implement a mutex without busy waiting?

- **Yes**
- No
- I am not sure

Q8 What do you consider essential to implement a mutex with busy waiting (multiple choices can be selected)?

- **Atomic memory reading and writing instructions**
- OS support
- **A shared memory space**
- Nothing, any system supports it by default

Q9 What do you consider essential to implement a mutex without busy waiting (multiple choices can be selected)?

- **Atomic memory reading and writing instructions**
- **OS support**
- **A shared memory space**
- Nothing, any system supports it by default

Q10 Do you know what a futex is?

- **Yes**
- No
- I have a rough idea

Appendix A.3. MP survey

Q11 The consistency model defined by ARMv8 is...

- Sequential
- **Relaxed**
- None of them

Q12 Which of the following instructions puts the processor on a low power state?

- sevl
- dbm
- **wfe**
- All of the above
- None of the above

Q13 For a *fetch and add*, would you use any of the following ARMv8 instructions?

- wfe, sevl
- DMB, DSB, ISB
- **ldaxr, stlr**
- None of the above

Q14 What of the following synchronization alternatives would you use in a low-contention scenario for the next program?

```
// run concurrently by several threads
long long my_add = 0;
while (work_index < n_elements) {

  // how do you protect the next code
//lines?
  int my_work_index_ini = work_index;
  work_index = work_index + chunk;
  int my_work_index_end = work_index;

  for (int i = my_work_index_ini; i <
my_work_index_end; i++) {
    my_add = my_add + v_elems[i];
  }
}
```

- ***Fetch and add***
- Mutex
- Energy-efficient mutex

Q15 What of the following synchronization alternatives would you use in a high-contention scenario for the same program as in the previous question?

- *Fetch and add*
- Mutex
- **Energy-efficient mutex**

Appendix A.4. CG survey

Q16 What are the minimal requirements to implement a non busy-wait mutex (several choices can be valid)?

- **Atomic instructions**
- **OS support**
- **A shared memory space**
- Nothing, all systems support non busy-wait mutexes

Q17 What are the minimal requirements to implement a busy-wait mutex (several choices can be valid)?

- **Atomic instructions**
- OS support
- **A shared memory space**
- Nothing, all systems support non busy-wait mutexes

Q18 For a parallel application such as a ray tracer, which implementation would be faster in a scenario where the number of logical threads is *lower* than the number of physical execution contexts of the processor?

- OS mutex
- Library mutex
- Assembly mutex
- **All three should have a similar behavior**

Q19 For a parallel application such as a ray tracer, which implementation would be faster in a scenario where the number of logical threads is *equal* to the number of physical execution contexts of the processor?

- OS mutex
- Library mutex
- Assembly mutex
- **All three should have a similar behavior**

Q20 For a parallel application such as a ray tracer, which implementation would be faster in a scenario where the number of logical threads is *higher* than the number of physical execution contexts of the processor?

- OS mutex
- **Library mutex**
- Assembly mutex
- All three should have a similar behavior

Q21 Many algorithms, such as ray tracing, whose output is a matrix of pixels (or an image), are easily parallelizable. Which is the best parallelization strategy for such algorithms?

- Static: pixels are pre-assigned to a worker thread by rows, columns, or regions. In this approach, workers perform their tasks without synchronizing with other workers
- Dynamic-per-pixel: each worker continuously fetches single-pixel tasks until completion. It requires synchronization among workers
- **Dynamic-per-region: each worker continuously fetches region tasks until completion. It requires synchronization among workers**
- Hard to guess: the best strategy depends on the variability of the execution time of each task

Q22 In a parallel ray tracing algorithm, where each pixel can be independently generated, assuming a dynamic-per-region strategy, under which circumstance reducing the region size can improve performance?

- When there is variability in the pixel-generation time
- When there are image areas where is known beforehand that the pixel-generation time is smaller than that of other areas
- **When the region task delay is significantly larger than the synchronization delay produced by the region**
- When the resulting image has many color disturbances

Acknowledgments

All authors acknowledge support from grants (1) PIIDUZ_18_246 from *Universidad de Zaragoza*, (2) PID2019-105660RB-C21 / AEI / 10.13039/501100011033 from *Agencia Estatal de Investigación* (AEI) and European Regional Development Fund (ERDF), (3) gaZ: T58_20R research group from Aragon Government and European Social Fund (ESF), (4) 2014-2020 "*Construyendo Europa desde Aragón*" from ERDF, (5) TIN2017-84796-C2-2-R from Spanish *Ministerio de Economía y Competitividad*, (6) DisCo-T21-20R from Aragon Government, (7) PID2019-105004GB-I00 from Spanish Ministry of Science and Innovation, and (8) CHAMELEON, grant No 682080 from European Research Council (ERC) under the EU's Horizon 2020 research and innovation programme.

References

[1] S. K. Prasad, A. Chtchelkanova, F. Dehne, M. Gouda, A. Gupta, J. Jaja, K. Kant, A. La Salle, R. LeBlanc, 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, `http://tcpp.cs.gsu.edu/curriculum/` (2012).

[2] ACM/IEEE, Computer Science Curricula 2013: Curriculum Guidelines for Undergraduate Degree Programs in Computer Science (2013).

[3] D. J. John, S. J. Thomas, Parallel and Distributed Computing across the Computer Science Curriculum, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, 2014, pp. 1085–1090.

[4] M. Burtscher, W. Peng, A. Qasem, H. Shi, D. Tamir, H. Thiry, A Module-Based Approach to Adopting the 2013 ACM Curricular Recommendations on Parallel Computing, in: Proceedings of the 46th ACM Technical Symposium on Computer Science Education, 2015, pp. 36–41.

[5] S. Srivastava, M. Smith, A. Ghimire, S. Gao, Assessing the Integration of Parallel and Distributed Computing in Early Undergraduate Computer Science Curriculum using Unplugged Activities, in: IEEE/ACM Workshop

on Education for High-Performance Computing, 2019, pp. 17–24.

[6] L. B. A. Vasconcelos, F. A. L. Soares, P. H. M. M. Penna, M. V. Machado, L. F. W. Góes, C. A. P. S. Martins, H. C. Freitas, Teaching Parallel Programming to Freshmen in an Undergraduate Computer Science Program, in: IEEE Frontiers in Education Conference, 2019, pp. 1–8.

[7] M. Grossman, M. Aziz, H. Chi, A. Tibrewal, S. Imam, V. Sarkar, Pedagogy and tools for teaching parallel computing at the sophomore undergraduate level, Elsevier Journal of Parallel and Distributed Computing 105 (2017) 18–30.

[8] S. A. Bogaerts, One step at a time: Parallelism in an introductory programming course, Elsevier Journal of Parallel and Distributed Computing 105 (2017) 4–17.

[9] S. Petit, J. Sahuquillo, M. E. Gómez, V. Selfa, A research-oriented course on Advanced Multicore Architecture: Contents and active learning methodologies, Elsevier Journal of Parallel and Distributed Computing 105 (2017) 63–72.

[10] S. Kumar, Research-oriented teaching of PDC topics in integration with other undergraduate courses at multiple levels: A multi-year report, Elsevier Journal of Parallel and Distributed Computing 105 (2017) 92–104.

[11] N. Giacaman, O. Sinnen, EA: Research-Infused Teaching of Parallel Programming Concepts for Undergraduate Software Engineering Students, in: IEEE International Parallel Distributed Processing Symposium Workshops, 2014, pp. 1099–1105.

[12] T. Newhall, A. Danner, K. C. Webb, Pervasive parallel and distributed computing in a liberal arts college curriculum, Elsevier Journal of Parallel and Distributed Computing 105 (2017) 53–62.

[13] J. Eckroth, A Course on Big Data Analytics, Elsevier Journal of Parallel and Distributed Computing 118 (P1) (2018) 166–176.

[14] E. Saule, Experiences on Teaching Parallel and Distributed Computing for Undergraduates, in: IEEE International Parallel and Distributed Processing Symposium Workshops, 2018, pp. 361–368.

[15] J. Kramer, Is Abstraction the Key to Computing?, Communications of the ACM 50 (4) (2007) 36–42.

[16] D. Ginat, Y. Blau, Multiple Levels of Abstraction in Algorithmic Problem Solving, in: Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education, 2017, pp. 237–242.

[17] C. Ferner, B. Wilkinson, B. Heath, Toward Using Higher-Level Abstractions to Teach Parallel Computing, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, 2013, pp. 1291–1296.

[18] B. Levandowski, D. Perouli, D. Brylow, Using Embedded Xinu and the Raspberry Pi 3 to Teach Parallel Computing in Assembly Programming, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, 2019, pp. 334–341.

[19] J. Cuenca, D. Giménez, A Parallel Programming Course Based on an Execution Time-Energy Consumption Optimization Problem, in: Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops, 2016, pp. 996–1003.

[20] ACM/IEEE, Computer Engineering Curricula 2016: Curriculum Guidelines for Undergraduate Degree Programs in Computer Engineering (2016).

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

[22] D. J. Sorin, M. D. Hill, D. A. Wood, A Primer on Memory Consistency and Cache Coherence, 1st Edition, Morgan & Claypool Publishers, 2011.

[23] C. Lupo, Z. J. Wood, C. Victorino, Cross Teaching Parallelism and Ray Tracing: A Project-based Approach to Teaching Applied Parallel Computing, in: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, 2012, pp. 523–528.

[24] S. J. Matthews, Teaching with Parallella: A First Look in an Undergraduate Parallel Computing Course, Journal of Computing Sciences in Colleges 31 (3) (2016) 18–27.

[25] S. J. Matthews, J. C. Adams, R. A. Brown, E. Shoop, Portable Parallel Computing with the Raspberry Pi, in: Proceedings of the ACM SIGCSE Technical Symposium on Computer Science Education, 2018, pp. 92–97.

[26] A. A. Younis, R. Sunderraman, M. Metzler, A. G. Bourgeois, Case Study: Using Project Based Learning to Develop Parallel Programing and Soft Skills, in: IEEE International Parallel and Distributed Processing Symposium Workshops, 2019, pp. 304–311.

[27] H. Franke, R. Russell, M. Kirkwood, Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux, in: Proceedings of the Ottawa Linux Symposium, 2002, pp. 479–495.

[28] A. Williams, C++ Concurrency in Action, Manning Publications, 2012.

[29] G. R. Andrews, Concurrent Programming. Principles and Practice, 1st Edition, The Benjamin/Cummings Publishing Company, Inc., 1991.

[30] U. Drepper, Futexes Are Tricky, `http://people.redhat.com/drepper/futex.pdf` (2011).

[31] U. Drepper, I. Molnar, The Native POSIX Thread Library for Linux, `https://akkadia.org/drepper/nptl-design.pdf` (2005).

[32] A. Silberschatz, P. B. Galvin, G. Gagne, Operating System Concepts, 9th Edition, Wiley Publishing, 2012.

[33] W. Stallings, Operating Systems: Internals and Design Principles, 6th Edition, Prentice Hall Press, 2008.

[34] R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, Operating Systems: Three Easy Pieces, 1st Edition, Arpaci-Dusseau Books, LLC, 2018.

[35] ARM, ARM DS-5 Development Studio Examples (2018).

[36] E. Veach, Robust Monte Carlo Methods for Light Transport Simulation, Ph.D. thesis, Stanford University (1998).

[37] M. Pharr, W. Jakob, G. Humphreys, Physically Based Rendering: From Theory to Implementation, 3rd Edition, Morgan Kaufmann Publishers Inc., 2017.

[38] M. Haidl, S. Gorlatch, PACXX: Towards a Unified Programming Model for Programming Accelerators Using C++14, in: Proceedings of the LLVM Compiler Infrastructure in HPC, 2014, pp. 1–11.

[39] B. Gaster, L. Howes, D. R. Kaeli, P. Mistry, D. Schaa, Heterogeneous Computing with OpenCL, 1st Edition, Morgan Kaufmann Publishers Inc., 2011.

[40] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, X. Tian, Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL, Springer Nature, 2021.

[41] E. Upton, G. Halfacree, Raspberry Pi User Guide, John Wiley & Sons Ltd., 2014.

[42] Orange Pi Zero H2 User Manual, version 0.9.1, Shenzhen Xunlong Software Co., Ltd.

[43] Qualcomm, DragonBoard™410c based on Qualcomm®Snapdragon™410E processor. Peripherals Programming Guide Linux Android, Qualcomm Technologies, Inc., 2016.

[44] G. Coley, BeagleBoard X15 System Reference Manual, BeagleBoard.org, 2016.

[45] E. Upton, J. Duntemann, R. Roberts, T. Mamtora, B. Everard, Learning Computer Architecture with Raspberry Pi, 1st Edition, Wiley Publishing, 2016.