Computació paral·lela en iOS per experts en MPI

Marc Català i Montaner

Director: Jordi Fornés
Especialització: Tecnologies de la informació

Universitat Politècnica de Catalunya
2019

Resum

Cada dia apareixen noves aplicacions relacionades amb el món de la tecnologia. Com és lògic, hi ha molts camps que han estat objectius d'una gran quantitat d'estudis, mentre que hi ha altres que passen més desapercebuts. Dintre d'aquests camps que passen més desapercebuts, hi ha alguns que són força interessants i mereixen ser estudiats més en profunditat. Un camp que pot ser molt important en el futur és el de la computació paral·lela en dispositius iOS, ja que els dispositius mòbils cada cop són més potents i van adquirint noves funcionalitats. És en aquest àmbit en el qual es centra aquest Treball de Fi de Grau.

Una de les tècniques aplicades més habitualment per realitzar paral·lelisme és mitjançant MPI. Actualment hi ha una gran quantitat de programadors experts en MPI, mentre que es troben molt pocs estudis de programació paral·lela en dispositius iOS.

Aquest projecte té com a objectiu l'anàlisi de l'estat actual de la computació paral·lela i distribuïda en dispositius mòbils iOS. Aquest projecte vol proporcionar una visió general de les característiques d'algunes llibreries primitives de MPI, i de les eines de suport que ens dóna Apple, per tal d'aconseguir l'òptima paral·lelització en la programació de dispositius iOS.

Per tant, durant el projecte s'ha desenvolupat una aplicació per dispositius iOS que permet aplicar paral·lelisme. A més d'aquesta aplicació, s'ha explicat detalladament totes les diferències amb una aplicació desenvolupada amb MPI, per facilitar a programadors experts en MPI poder programar amb facilitat i agilitat en l'entorn de les aplicacions per dispositius iOS.

Per implementar aquesta aplicació s'ha utilitzat l'entorn de desenvolupament Xcode, el *framework* Network i la tecnologia de GCD, tot realitzat amb el llenguatge de programació Swift destinat al desenvolupament d'aplicacions iOS.

Resumen

Cada día aparecen nuevas aplicaciones relacionadas con el mundo de la tecnología. Como es lógico, hay muchos campos que han sido objetivos de una gran cantidad de estudios, mientras que hay otros que pasan más desapercibidos. Dentro de estos campos que pasan más desapercibidos, hay algunos que son muy interesantes y merecen ser estudiados más en profundidad. Un campo que puede ser muy importante en el futuro es el de la computación paralela en dispositivos iOS, ya que los dispositivos móviles son cada vez más potentes y van adquiriendo nuevas funcionalidades. Es en este ámbito en el que se centra este Trabajo de Fin de Grado.

Una de las técnicas aplicadas más habitualmente para realizar paralelismo es mediante MPI. Actualmente hay una gran cantidad de programadores expertos en MPI, mientras que se encuentran muy pocos estudios de programación paralela en dispositivos iOS.

Este proyecto tiene como objetivo el análisis del estado actual de la computación paralela y distribuida en dispositivos móviles iOS. Este proyecto quiere proporcionar una visión general de las características de algunas librerías primitivas de MPI, y de las herramientas de apoyo que nos da Apple, para conseguir la óptima paralelización en la programación de dispositivos iOS.

Por lo tanto, durante el proyecto se ha desarrollado una aplicación para dispositivos iOS que permite aplicar paralelismo. Además de esta aplicación, se ha explicado detalladamente todas las diferencias con una aplicación desarrollada con MPI, para facilitar a programadores expertos en MPI poder programar con facilidad y agilidad en el entorno de las aplicaciones para dispositivos iOS.

Para implementar esta aplicación se ha utilizado el entorno de desarrollo Xcode, el framework Network y la tecnología de GCD, todo realizado con el lenguaje de programación Swift destinado al desarrollo de aplicaciones iOS.

Abstract

Every day there are new applications related to the world of technology. Obviously, there are many fields that have been the targets of a large number of studies, while others are more unnoticed. Within these fields that go unnoticed, there are some that are quite interesting and deserve to be studied more in depth. A field that can be very important in the future is that of parallel computing in iOS devices, as mobile devices are becoming more powerful and are acquiring new features. It is in this area where this End-of-Degree Project focuses.

One of the most commonly applied techniques to perform parallelism is through MPI. There is currently a large number of expert programmers in MPI, while there are very few parallel programming studies on iOS devices.

This project aims to analyze the current state of parallel computing and distributed on iOS mobile devices. This project aims to provide an overview of the characteristics of some primitive MPI libraries, and Apple's support tools, in order to achieve the optimal parallelization of iOS device programming.

Therefore, during the project an application for iOS devices has been developed that allows parallelism to be applied. In addition to this application, it has been explained in detail all the differences with an application developed with MPI, facilitating programmers experts in MPI to be able to program with facility and agility in the surroundings of the applications for iOS devices.

To implement this application, the Xcode development environment, the Network framework and the GCD technology have been used, all done with the Swift programming language for the development of iOS applications.

Índex

Índex de figures

Índex de taules

1 Context

1.1 Introducció

La tecnologia cada cop és més important en el dia a dia, sigui per cobrir necessitats o per gust propi. El camp dels dispositius mòbils ha anat variant molt durant els últims anys amb l'aparició dels telèfons intel·ligents. Els telèfons mòbils van adquirir una gran varietat de funcions que no tenien els dispositius anteriors i que els aproximava bastant a ser ordinadors portables. Cada cop es van millorant les característiques dels nous telèfons mòbils i s'assemblen més a ordinadors que els anteriors. Tot i això, encara hi ha diferències entre ordinadors i mòbils, sigui en el *hardware* o en el *software*. A causa de les millores que es produeixen constantment en els telèfons mòbils, alguns queden antiquats i deixen de tenir utilitat. A més, una gran quantitat de gent decideix canviar-se de mòbil de forma bastant freqüent, encara que el seu dispositiu funcioni correctament. A partir d'aquí et pots plantejar que fer amb aquests dispositius mòbils. La majoria de gent els ven en llocs especialitzats, els llença a la brossa o els guarda sense cap motiu especial. La idea d'aquest projecte és donar una utilitat diferent als dispositius iOS, explorant les capacitats *High-performance computing* (HPC) d'aquests dispositius, per tal de presentar-les a experts en *Message Passing Interface* (MPI). Estarà destinat a totes les versions disponibles actualment dels dispositius iOS (iPhone Operating System), siguin més recents o més desfasades.

Per fer-ho, parteixo del model de MPI en UNIX per explorar les eines que ofereix iOS. MPI s'ha convertit en la solució dominant en la informàtica distribuïda i HPC. MPI és una biblioteca de programació que proporciona una gran funcionalitat i estructures pels programadors que permeten un paral·lelisme escalable i està disponible en molts dels llenguatges preferits de programació com Fortran i C ++ [1]. És la més utilitzada actualment per a la programació concurrent en entorns en la que la comunicació es fa a través de missatges i se la considera l'estàndard de referència. L'objectiu principal de MPI és formar un estàndard que sigui àmpliament utilitzat en programes que requereixin utilitzar missatges per comunicar-se, per això, la interfície intenta ser pràctica, portable, eficient i flexible per tal d'incrementar la productivitat.

Partint de la informació anterior, l'objectiu principal del projecte és l'anàlisi de l'estat actual de la computació paral·lela i distribuïda en dispositius mòbils iOS. Aquest projecte proporciona una visió general de les característiques d'algunes llibreries primitives de MPI, i de les eines de suport que ens dóna Apple, per tal d'aconseguir l'òptima paral·lelització en la programació de dispositius iOS.

A partir d'aquí, es tracta de veure si un programador de MPI, és capaç de desenvolupar amb èxit la paral·lelització en dispositius iOS, sigui amb el propi MPI o només amb les eines que ens proporciona Apple.

1.2 Actors

Per a realitzar aquest projecte es troben diversos actors implicats. Són els següents:

- **Desenvolupador:** És la persona encarregada de dur a terme la investigació, documentació i implementació de tot el projecte. Aquest actor treballa amb uns terminis que ha de complir proposats pel director.

- **Director:** És la persona responsable de guiar, donar consells i ajudar al desenvolupador. Jordi Fornés, professor del departament d'Arquitectura de Computadors de la Universitat Politècnica de Catalunya, és la persona que ha actuat com a director.

- **Beneficiaris:** Atès que aquest projecte no tracta de crear un producte com a tal, pot semblar que no té cap beneficiari clar. No obstant això, ens permetrà veure en la situació que es troba MPI sobre iOS i, en el cas que no sigui rentable o possible utilitzar-ho en l'actualitat, quines alternatives a MPI existeixen per desenvolupar aplicacions en dispositius iOS. D'aquesta manera, es pot dir que aquest projecte va dirigit a tots els programadors experts en MPI que algun cop han pensat en la forma de programar aplicant paral·lelisme en els dispositius iOS.

1.3 Estat de l'art

L'estandardització de MPI [1] va començar-se a desenvolupar gràcies al patrocini del Centre d'Investigació en Computació Paral·lela de Williamsburg, Virgínia, Estats Units en l'any 1992. Va sorgir la primera versió preliminar, anomenada MPI1, que estava principalment enfocada a les comunicacions punt-a-punt, sense incloure rutines de comunicació col·lectiva. A causa de la utilitat i potencial d'aquest projecte, es va continuar desenvolupant aquest estàndard de comunicació. D'aquesta manera, en l'any 1994 es va publicar la MPI-1 i posteriorment, en el 1997 es va publicar la MPI-2.

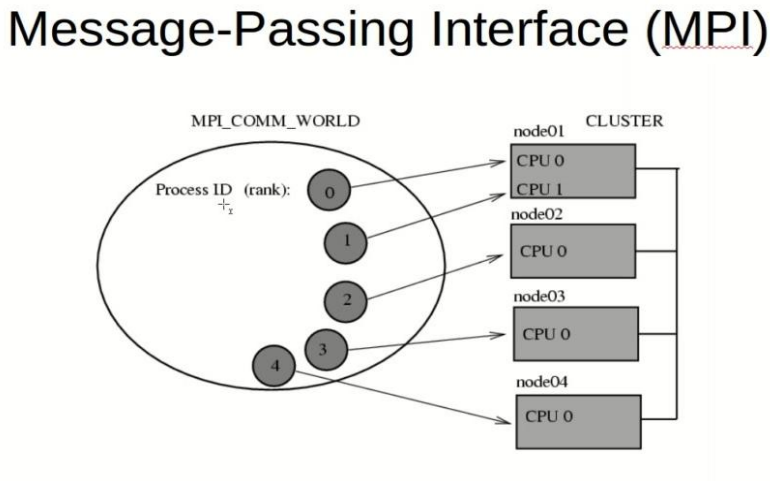

Figura 1: Message Passing Interface (MPI)

S'han fet alguns estudis sobre les característiques i les funcions de MPI en el sistema operatiu iOS, els quals s'aproximen en certs aspectes a l'estudi que vull realitzar. En un dels estudis, anomenat *Towards Energy Efficient Parallel Computing on Consumer Electronic Devices* [2], buscaven analitzar l'estat actual de la computació paral·lela i distribuïda d'energia eficient en dispositius electrònics mòbils i de consum. Per fer-ho, van crear una petita prova de concepte amb un *cluster* d'Apple TV de segona generació i van avaluar el rendiment en aplicacions estàndard. Per fer aquest experiment van utilitzar MPICH2 (és una implementació de MPI portàtil i disponible per sistemes operatius com Linux o MacOS). El punt clau d'aquest estudi, és la utilització del *jailbreak* en els Apple TV. Un *jailbreak* és el procés d'eliminar les limitacions imposades per Apple en dispositius que utilitzin el sistema operatiu iOS mitjançant l'ús de kernels modificats. És important saber si s'ha

de fer o no, ja que és una acció que s'ha de realitzar al principi del procés, abans de fer cap altra interacció amb el dispositiu iOS. Per desenvolupar el meu projecte m'ha servit molt aquest article, ja que he adquirit una gran base de coneixements sobre certs conceptes, tot i que el meu enfocament és just el contrari al que es descriu aquí. Per poder utilitzar MPICH2 en els AppleTV era necessari fer un *jailbreak* en els dispositius, mentre que jo no volia dependre dels *jailbreaks*. Vull fer servir les eines subministrades per Apple per assegurar-me la compatibilitat entre els dispositius, tenint en compte la capacitat d'evolució dels dispositius en el temps.

Trobem un altre estudi anomenat *Anatomy of a globally recursive embedded LINPACK benchmark* [3], on van fer una implementació d'un benchmark LINPACK en un iPad 2. El projecte es desvia una mica del meu estudi però, ens dóna informació bastant important. Per fer la prova de concepte en l'iPad 2, també van haver de fer un *jailbreak*.

Aquests dos estudis em permeten començar a veure que per utilitzar MPI en dispositius iOS serà necessari fer un *jailbreak*, ja que iOS no suporta MPI. Es pot pensar que hauré de buscar una altra alternativa a MPI per realitzar el paral·lelisme en els dispositius iOS, ja que el *jailbreak* no és una alternativa. Pel meu projecte hauré d'aprofundir en les eines de suport d'Apple en dispositius mòbils iOS.

1.4 Formulació del problema

1.4.1 Problema

Els dispositius iOS són dispositius mòbils molt potents que cada cop tenen més avantatges però, també tenen alguns desavantatges. Cada cop es dissenyen nous processadors més potents que els anteriors i que redueixen les diferències de rendiment que pot haver-hi amb els ordinadors. La majoria d'usuaris de telèfons intel·ligents es centren principalment en les funcionalitats bàsiques del telèfon, els gràfics, la disponibilitat de jugar a certs jocs però, sense esprémer al màxim les capacitats de rendiment que tenen els dispositius. A més, en iOS totes les aplicacions que instal·len els usuaris han de venir de l'AppStore, no es poden instal·lar aplicacions externes. Per una banda fa que sigui un sistema operatiu molt

més segur, però per altra banda això fa que hi hagi bastants limitacions i sigui complicat implementar nou *software* on Apple no et doni suport. Aquest fet ens fa pensar que amb la potència que tenen els dispositius iOS més recents, s'està desaprofitant un gran potencial que podria ser utilitzat de moltes altres maneres. Una de les maneres que crec que es poden aprofitar els dispositius, és utilitzar-los per a la computació paral·lela.

1.4.2 Objectius

El primer objectiu del projecte consisteix a establir les bases d'aquest, a partir d'analitzar en quin estat es troba actualment la computació paral·lela i distribuïda en els dispositius mòbils iOS i treure'n conclusions pròpies.

A partir de les conclusions tretes, en el segon objectiu es tracta de ser capaços d'implementar una aplicació utilitzant tècniques de computació paral·lela sobre dispositius que tinguin el sistema operatiu iOS. Per fer això, primer s'ha de configurar tot l'entorn de treball, des dels dispositius mòbils fins a les aplicacions requerides en macOS.

Finalment, l'últim objectiu és explicar el model de programació paral·lela i distribuïda de iOS des del punt de vista d'un programador MPI i, treure les conclusions de la computació paral·lela en els dispositius iOS a partir de l'aplicació desenvolupada, analitzant els avantatges i els desavantatges d'aquesta proposta respecte MPI.

1.5 Abast

Es vol utilitzar les tècniques de paral·lelisme sobre els dispositius mòbils per a millorar la computació paral·lela i distribuïda. Com s'ha vist en l'apartat anterior, totes les proves es realitzaran en dispositius amb sistemes operatius macOS i iOS. Un dels motius pels quals treballo amb dispositius iOS és que actualment ja s'han realitzat diversos estudis on es treballa amb MPI en dispositius Android. Les aplicacions tindran les funcionalitats necessàries per poder-se comparar amb la computació paral·lela que es pot fer actualment, mitjançant MPI. La idea és agafar un model de MPI per fer les aplicacions, i veure com es comporta en macOS i adaptar-lo perquè funcioni en una aplicació dissenyada per iOS.

Un possible obstacle que puc tenir és la quantitat necessària de dispositius mòbils iOS físics on podré fer les proves de rendiment. Tot i això, hi ha eines per solucionar aquests problemes com són els simuladors, ja que permeten executar aplicacions iOS en simuladors de diferents versions de dispositius mòbils iOS i simular que s'estan executant en un dispositiu físic.

1.6 Metodologia i rigor

1.6.1 Metodologia de treball

La metodologia de treball que es farà servir en aquest projecte serà la metodologia de cascada. Aquest mètode consisteix en un disseny seqüencial on les fases típiques d'aquest són les següents: requeriments, disseny, implementació, verificació i manteniment.

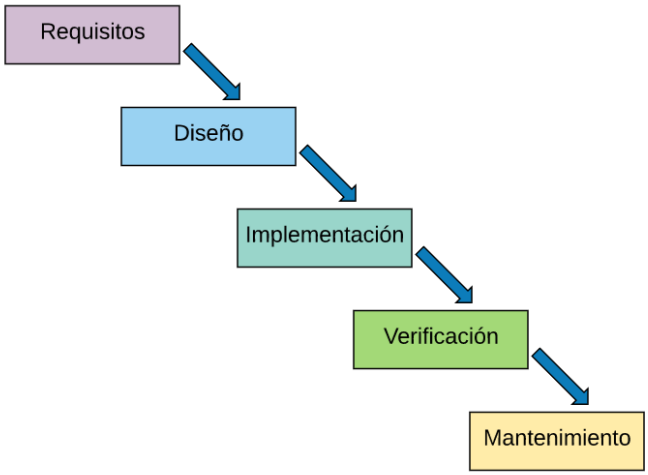

Figura 2: Fases de la metodologia en cascada

Una altra metodologia que podria haver fet servir és l'agile perquè ens dóna l'oportunitat de fer fases en paral·lel, però per a realitzar el meu projecte haig d'anar pas a pas seguint un ordre i com serà realitzat per una sola persona, necessito fer-lo el més fàcil d'entendre i utilitzar possible i, que l'administració del projecte no sigui complicada. Per això, s'ajusta més pel meu projecte la metodologia en cascada que l'agile.

Es realitzaran reunions periòdiques amb el director del projecte, ja sigui per modificar possibles errors en el desenvolupament, per donar consells d'implementació o per veure la correcta evolució de les diverses etapes de la metodologia emprada. Aquestes reunions es concretaran mitjançant correus electrònics, tot i que, en un principi es produiran cada setmana.

Cal aclarir que el desenvolupament de la memòria s'anirà desenvolupant progressivament al llarg de cada tasca, de manera que es pugui anar anotant tots els avenços i obstacles que es troben al llarg del projecte.

1.6.2 Eines de treball

Per a la realització d'aquest projecte s'intentarà disposar d'una eina de suport de la mateixa universitat. El iOS Developer University Program és un programa d'Apple pensat perquè les universitats puguin introduir als seus estudiants en el desenvolupament per a dispositius iOS sense haver de pagar la llicència de desenvolupador. Al realitzar un projecte dirigit cap a dispositius mòbils iOS, s'ha de tenir en compte aquest programa que ofereix la universitat, ja que a l'hora de córrer de forma física l'aplicació desenvolupada, pot ser útil aquest programa. Tot i això, la majoria de les proves a realitzar es poden fer perfectament en simuladors i, no està del tot clar que acabi sent necessari.

Per a la creació del codi es poden trobar una gran varietat d'editors. Algunes de les opcions més interessants són Sublime Text i Atom, que són dos editors de codi molt similars, fàcils d'utilitzar i força coneguts. Per altra banda també es troba l'editor Brackets, que funciona de forma molt similar als dos anteriors, però incorpora una funcionalitat com és una vista prèvia dinàmica, molt interessant pel desenvolupament de pàgines web entre d'altres. Per les característiques que busco, crec que Sublime Text és l'editor més adequat per mi a l'hora de desenvolupar codi relacionat amb MPI, tot i que es podria utilitzar qualsevol dels altres editors esmentats. A més, aquest codi s'anirà guardant en un repositori de GitHub [4] que m'anirà ajudant a tenir un control de versions i a fer un seguiment del desenvolupament del mateix codi. S'ha de tenir en compte però, que per desenvolupar aplicacions iOS hi ha una eina que ens proporciona Apple,

anomenada Xcode [5], que és clarament molt més útil que qualsevol altre editor de codi dels que he comentat, ja que Xcode és un entorn de desenvolupament integrat per a macOS que conté un grup d'eines de desenvolupament de *software* desenvolupades per Apple i destinades a macOS, iOS, iPadOS, watchOS i tvOS. També s'utilitzaran algunes eines de Google, com Google Calendar que es farà servir per posar data a les trobades amb el director i sincronitzar els actes, mentre que Google Drive es farà servir per mantenir la documentació al núvol i accessible des de qualsevol ordinador. Per últim, es farà servir TeamGantt [6] per dissenyar el Gantt, planificant la durada de cada etapa, com les dependències entre diverses tasques.

1.6.3 Validació

El projecte consisteix a realitzar una aplicació per dispositius iOS que sigui capaç d'aplicar tècniques de paral·lelisme. Aquesta aplicació es basarà en un model estàndard de paral·lelisme i, per tant, haurà de complir una sèrie de condicions per tal de poder fer la comparativa amb el model escollit. El desenvolupament d'aquesta aplicació em permetrà comprovar els avantatges i desavantatges respecte a la programació amb MPI. Per comprovar que l'aplicació funciona correctament es realitzarà una sèrie de proves, de manera que totes les parts de l'aplicació compleixin les tasques assignades.

Un cop s'hagi fet la validació de l'aplicació, es podrà veure si hi ha futur en la computació paral·lela en dispositius iOS i, en cas contrari, veure quines són les possibles solucions.

2 Planificació del projecte

2.1 Calendari

2.1.1 Durada aproximada del projecte

La durada prevista del projecte és d'uns vuit mesos, des de mitjans de febrer fins a finals de setembre de 2019.

2.1.2 Consideracions

Com que es tracta d'un projecte de recerca, cal assenyalar que la planificació inicial podria revisar-se i actualitzar-se com a conseqüència de l'evolució del projecte, incloent-hi algun canvi en algunes de les tecnologies que estic utilitzant.

2.2 Descripció de les tasques

La planificació del projecte està dividit en diferents fases o tasques que veurem a continuació.

2.2.1 Viabilitat del projecte

És la part inicial del projecte i es tracta d'una part molt important. S'ha buscat una gran quantitat de models implementats de manera similar on s'utilitzaven les llibreries de MPI en dispositius mòbils per tal de conèixer els diferents tipus d'estudis realitzats i per tal de determinar el punt de sortida.

A més s'analitzen les tecnologies que més utilitat poden tenir per desenvolupar el projecte i es trien les més adequades.

2.2.2 Planificació del projecte

És la tasca que consisteix bàsicament en documentar gran part de la memòria amb l'ajuda del curs de GEP. Es pot dividir en les següents etapes:

- Estat de l'art
- Abast del projecte
- Planificació del projecte
- Gestió econòmica i sostenibilitat

És necessari realitzar aquesta tasca un cop s'ha comprovat la viabilitat del projecte, ja que detallarà què es farà en les següents tasques i de la forma en què es farà.

2.2.3 Configuració inicial del sistema

Per a realitzar el desenvolupament del projecte, abans s'ha de configurar totes les eines necessàries per treballar-hi. Pot semblar una tasca senzilla, però és molt important, ja que es necessita fer tota la configuració correctament per tal que la resta del projecte es pugui desenvolupar de forma apropiada. En el meu calendari es pot veure que el nombre d'hores dedicades a la configuració inicial del sistema és bastant alta en comparació a altres tasques que semblen més importants.

Per desenvolupar el projecte, es necessita un dispositiu amb un sistema operatiu macOS, on es configurarà tot el *software* necessari per a la realització del projecte. On es desenvolupa la part important del meu projecte és en els dispositius mòbils iOS, per tant, necessitaré estudiar també, si em farà falta fer algun canvi, sigui instal·lant *software* addicional o eliminant algunes restriccions dels mateixos dispositius iOS o si ja em serveixen com vénen de fàbrica.

Per aquest motiu, es pot dir que és una tasca d'alt risc, ja que en triar tots els *softwares*, s'ha de comprovar que siguin compatibles entre ells o més endavant poden sorgir complicacions. Després d'instal·lar tot aquest *software*, un cop s'ha comprovat que tot funciona com s'esperava, podré començar a treballar en la fase més tècnica del meu projecte.

2.2.4 Desenvolupament del projecte

Aquesta és la tasca principal del projecte. Cobreix totes les tasques relacionades amb la implementació, proves i anàlisi de resultats de l'aplicació desenvolupada. Es divideix en les següents etapes:

- Adquirir coneixement iOS: Necessitaré uns coneixements bàsics i avançats per desenvolupar l'aplicació per dispositius iOS i, a més, em servirà per entendre més el funcionament i em facilitarà la realització de la següent etapa.
- Estructurar l'aplicació: És el primer pas, previ a la implementació de l'aplicació. Analitzaré com fer l'aplicació, tenint en compte quines

implementacions de MPI poden ser útils i, quines eines d'Apple poden anar millor per a la implementació de l'aplicació.

- Desenvolupament de l'aplicació: És la primera part que he d'implementar i, fins que no funcioni correctament no podré passar a la següent fase, és a dir, no podré realitzar les proves de rendiment corresponents. És una etapa d'alt risc, ja que no podré passar a fer cap prova fins que l'aplicació no estigui acabada.
- Tests de rendiment en dispositiu Mac OS X: Faré uns tests per comprovar el rendiment de l'aplicació en dispositius Mac OS X.
- Tests de rendiment en dispositiu iOS: Faré uns tests per comprovar el rendiment de l'aplicació en dispositius iOS.
- Anàlisi dels resultats: Compararé els resultats obtinguts de les proves anteriors. A partir de la comparació, trauré les conclusions necessàries per a les quals, fins ara, no s'ha utilitzat tècniques de paral·lelisme en dispositius iOS en gaire profunditat i, si té utilitat de cara al futur.

2.2.5 Finalització del projecte

En aquesta tasca es comprovarà que tot funciona com s'esperava i prepararé el lliurament del projecte, assegurant-me que tota la documentació és correcta i, finalment, preparant la presentació.

2.3 Temps estimat

A continuació es mostra una taula amb el temps de dedicació a cada una de les tasques descrites en la secció anterior. Aquests temps poden variar molt o poc segons les dificultats que es presentin.

Tasca	Temps Estimat (hores)
Viabilitat del projecte	20
Planificació del projecte	65
Configuració inicial del sistema	50
Adquirir coneixement iOS	35

Estructurar l'aplicació	25
Desenvolupament aplicació	180
Tests de rendiment MAC OS X	35
Tests de rendiment iOS	35
Anàlisi de resultats	40
Finalització del projecte	15
Total	**500**

Taula 1: Temps estimat per cada tasca

2.4 Diagrama de Gantt

A continuació, es mostra el diagrama de Gantt [6] resultant de les dependències per cada tasca i del temps estimat que es necessitarà per a la seva realització. He inclòs un marge de temps per una possible desviació en el cas que en sorgís qualsevol imprevist:

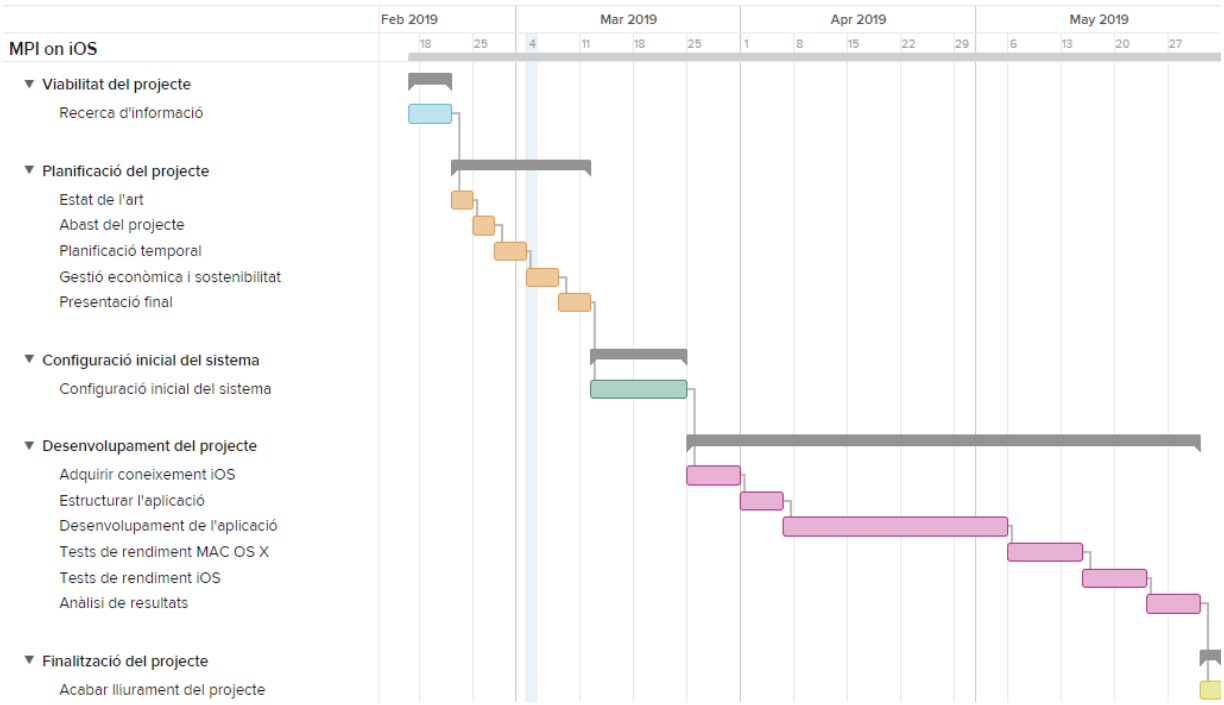

Figura 3: Diagrama de Gantt del projecte

Com podem veure en la imatge anterior, es produeixen moltes dependències. Per això, en apartats anteriors s'ha comentat que s'utilitzaria la metodologia de treball en cascada, on es necessita complir requisits per tal de poder avançar a la següent

tasca. Cada tasca estarà executada per una persona amb un rol diferent, amb un total de tres rols. Tota la planificació del projecte serà dissenyada per un dissenyador. El programador serà l'encarregat de configurar tot l'entorn i desenvolupar l'aplicació. Finalment, el tester serà l'encarregat de fer els tests necessaris i extreure els resultats corresponents.

2.5 Valoració d'alternatives i Pla d'acció

La primera idea és treballar com s'ha previst anteriorment, executant les tasques en l'ordre indicat en el diagrama Gantt, però sé que com en tots els projectes, poden sorgir diversos obstacles que facin que el pla hagi de canviar. Per descomptat, hi ha prioritats. Una d'elles ha de ser acabar l'aplicació completament per tal de fer les proves necessàries. Tot i això, sé que el temps no és il·limitat i tenim prioritats. Amb això estic dient que prioritzaré el correcte funcionament de l'aplicació per tal de poder fer les proves i, per tant utilitzaré algun dels models de MPI, per només haver de fer codi des de zero en l'aplicació iOS, que és la part important del projecte. En el cas extrem que no pogués executar l'aplicació en un dispositiu físic per falta del mateix dispositiu, podria seguir utilitzant emuladors de manera que no suposaria un gran problema a l'hora d'acabar el projecte en el temps establert.

Intentaré organitzar reunions amb el director del projecte cada setmana per anar comentant tot l'avenç del projecte i possibles dificultats. En el cas de finalitzar una etapa del projecte, també es realitzarà una trobada amb el director del projecte.

Per evitar qualsevol problema en cas que falti temps en l'evolució del projecte, he planificat una rutina a seguir en el dia a dia i una d'emergència. Ara estic seguint una rutina d'unes 28 hores setmanals (4 hores per dia laborable i 8 hores els caps de setmana) per desenvolupar el projecte, que es poden incrementar en el cas que sorgeixin complicacions, al voltant d'unes 35 hores setmanals (5 hores per dia laborable i 10 hores per caps de setmana). Amb aquesta última rutina hauria d'haver-hi suficient temps per acabar el projecte dins el termini especificat.

3 Pressupost i sostenibilitat

En aquesta secció s'explicarà i justificarà el pressupost del projecte. Trobarem una descripció detallada dels costos del projecte, on es descriu els costos materials i humans i l'anàlisi de com els diferents obstacles podrien afectar el meu pressupost. El pressupost descrit està subjecte a modificacions segons es vagi desenvolupant el projecte.

3.1 Estimació del pressupost

A continuació faré una estimació del pressupost necessari per fer el desenvolupament d'aquest projecte. El pressupost es pot dividir en tres seccions, depenent del tipus de recursos tenint en compte: *hardware*, *software* i recursos humans. Al final es mostrarà el pressupost total obtingut de la suma dels tres recursos anteriors.

3.1.1 Recursos *hardware*

A continuació es mostren els costs de tots els *hardwares* necessaris per poder desenvolupar correctament el meu projecte.

Producte	Preu	Vida útil	Amortització
MacBook PRO 13"	1800.00€	5 anys	120€
iPhone	800.00€	3 anys	88.89€
Total	**2600.00€**		**208.89€**

Taula 2: Costos de recursos *hardware*

3.1.2 Recursos *software*

A continuació es mostren els costs de tots els *softwares* que utilitzaré per desenvolupar el projecte.

Producte	Preu	Vida útil	Amortització
Mac OS X	0.00€	-	0.00€
Github	0.00€	-	0.00€

Sublime Text	0.00€	-	0.00€
MPI	0.00€	-	0.00€
Xcode	0.00€	-	0.00€
Google Docs	0.00€	-	0.00€
TeamGantt	0.00€	-	0.00€
Total	**0.00€**		**0.00€**

Taula 3: Costos de recursos *software*

3.1.3 Recursos humans

A continuació es mostren els costs dels recursos humans necessaris per desenvolupar el projecte.

Les tasques seran realitzades per diferents rols. En cada rol s'ha estimat el màxim a pagar per hora per tenir un control del pressupost a gastar amb recursos humans. El director del projecte només treballarà en la planificació del projecte. El dissenyador treballarà en tot allò que implica la programació, és a dir, la configuració de l'entorn i planificació de les aplicacions. El programador pot fer les mateixes tasques, a més del desenvolupament de l'aplicació. El tester participarà en algunes tasques de programació fent algunes proves per comprovar el correcte funcionament de l'aplicació, però la seva tasca principal és fer els tests de rendiment necessaris per treure els resultats corresponents. Coneixent el nombre d'hores que s'han de gastar en cada tasca i fent un càlcul aproximat utilitzant la participació de cada rol en cada tasca, surten les hores aproximades dedicades per a cada rol.

Rol	Preu per hora	Temps	Cost
Director del projecte	40.00€	100 h	4000.00€
Dissenyador	25.00€	120 h	3000.00€
Programador	30.00€	180 h	5400.00€
Tester	20.00€	100 h	2000.00€
Total	**/**	**500 h**	**14400.00€**

Taula 4: Costos de recursos humans

3.1.4 Pressupost total

A continuació es mostra el cost total del projecte, sumant tots els apartats anteriors.

Recursos	Cost
Hardware	208.89€
Software	0.00€
Humans	14400.00€
Total	**14608.89€**

Taula 5: Pressupost final

3.2 Control de gestió

El pressupost no és fix, ja que es poden realitzar diverses modificacions mentre es va desenvolupant el projecte i, això fa impossible que segueixi el pla establert. Tot i que podria necessitar més recursos de *software*, avui en dia hi ha moltes aplicacions gratuïtes que poden actuar com a substitut de gairebé qualsevol aplicació de pagament. A més, si es té un bon ús i manteniment dels dispositius no suposarà cap problema durant el projecte. Per tant, la secció en què s'ha de controlar el pressupost és la de recursos humans. En el desenvolupament de l'aplicació s'ha de tenir en compte que poden aparèixer problemes que faran invertir més hores de treball i, per tant, més despeses. A partir d'aquí es pot reservar un 20% més de pressupost per suposats problemes que sorgeixin i requereixin més hores de dedicació.

4 Fons tècnic

Per a la realització del projecte es necessitarà un dispositiu iOS i un dispositiu macOS. El dispositiu macOS serà utilitzat durant tot el projecte i, es treballarà amb l'arquitectura x86_64-apple-darwin18.6.0. En macOS s'executaran les aplicacions MPI i iOS. Per fer les aplicacions MPI s'utilitzarà models existents i es faran les execucions pertinents en la línia d'ordres. Per altra banda, trobem les aplicacions iOS, que es desenvoluparan en l'entorn de Xcode.

4.1 Xcode

Una gran part del projecte s'ha dedicat a la recerca d'informació i estudi de les eines que ofereix Apple. Tot i que Apple ofereix moltes eines, la principal sobre la qual ha girat quasi tot el projecte ha estat Xcode.

Xcode és un entorn de desenvolupament integrat (IDE) per a macOS que conté un conjunt d'eines de desenvolupament de *software* desenvolupades per Apple pel desenvolupament de *software* per a macOS, iOS, watchOS i tvOS [5]. En aquest entorn es realitzarà tot el procés de desenvolupar i executar l'aplicació. A més, l'execució també serà possible dins de Xcode, ja que es poden executar les aplicacions en simuladors del mateix entorn.

El meu objectiu és aconseguir executar una aplicació que apliqui tècniques de paral·lelisme en múltiples dispositius iOS. No hi ha una eina que em permeti fer-ho directament, per això he de buscar una forma d'aconseguir-ho.

Per aconseguir el meu objectiu, primer he de familiaritzar-me amb l'entorn de Xcode i, en la gran varietat d'opcions que té per desenvolupar les aplicacions. Per desenvolupar les aplicacions s'utilitzarà la versió 10.3 de Xcode. Sé que no puc incloure la llibreria MPI en la meva aplicació de forma automàtica, de manera que hauré de conèixer bé totes les opcions de les què disposa Xcode per tal de treure-li el màxim profit.

4.1.1 Llenguatges

A l'hora de crear una aplicació amb Xcode, estàs obligat a escollir un dels dos llenguatges que et proposa el programa. Aquests llenguatges són: Objective-C i Swift. Objective-C va néixer l'any 1984, mentre que Swift és molt més recent, concretament del 2014. Al principi, els llenguatges per desenvolupar aplicacions per dispositius Apple estaven molt limitats, tot i això, Objective-C pot incloure codi C i així es pot aconseguir desenvolupar aplicacions totalment en el llenguatge C.

Amb l'aparició de Swift, va aparèixer un nou llenguatge per programar aplicacions per dispositius Apple de forma més senzilla. De totes maneres, treballar amb el

llenguatge Swift et permet la coexistència amb Objective-C en cas de ser necessària.

4.1.2 Simulador

Tant simuladors com emuladors, permeten executar *software* en un entorn virtual, però ho fan de diferents maneres.

Un emulador funciona duplicant tots els aspectes del comportament del dispositiu original. Bàsicament simula tot el *hardware* que utilitza el dispositiu real, permetent que el mateix *software* es pugui executar sense modificar. Es pot copiar una aplicació del dispositiu original a l'emulador, sense que es notés la diferència.

D'altra banda, un simulador configura un entorn similar al sistema operatiu original del dispositiu, però no intenta simular el *hardware* del dispositiu real. Això pot provocar que alguns programes puguin funcionar una mica diferent i requerir alguns canvis.

A primera vista, un emulador sembla una solució molt millor però, té un inconvenient molt important: els emuladors són molt lents. Emular tot el *hardware* fa que el *software* s'executi més lent que l'original. A més, llançar l'emulador requereix que emulin tot el procés d'arrencada del dispositiu real, que també pot trigar un temps.

En el projecte s'ha utilitzat el simulador de iOS que trobem dins de Xcode. En un primer moment es tenia la idea d'utilitzar un emulador de iOS però, els que es poden trobar, no són oficials i els hi manquen bastants coses. Per aquest motiu, era més útil utilitzar les eines que ens dóna Apple, ja que el meu objectiu és fer-ho de la manera més clara i senzilla possible. El simulador ja es troba incorporat dins de Xcode i permet llançar les aplicacions en diferents versions iOS. Les versions més antigues es van quedant obsoletes però, avui en dia, comprèn des de l'iPhone 5S fins a l'actual (iPhone Xr). El simulador també funciona per diferents versions d'iPad.

D'aquesta manera, es pot dir que el dispositiu iOS no serà un dispositiu físic, ja que s'utilitzaran els simuladors esmentats. L'objectiu del projecte és proporcionar una nova forma de paral·lelisme en dispositius iOS, per tant, s'intentarà que funcionin les

aplicacions principalment en les versions més recents del simulador. Els simuladors poden ser de 32 i 64 bits. En el cas del simulador de 32 bits utilitza una arquitectura i386, mentre que el simulador de 64 bits utilitza una arquitectura x86_64. En el projecte es treballarà amb el simulador de 64 bits i, per tant, amb l'arquitectura x86_64.

4.1.3 Aplicació bàsica

Tot i que s'ha anat esmentant algunes de les característiques de Xcode, l'objectiu és desenvolupar aplicacions que apliquin paral·lelisme en els dispositius iOS. Per fer-ho, primer s'ha de saber crear una aplicació des de zero amb Xcode. A continuació, s'explicaran els passos a seguir.

Per crear un nou projecte un cop s'ha iniciat el programa, es fa seleccionant New> Project... des del menú File. També es pot crear un projecte amb la combinació Shift + Command + N.

Començar un nou projecte és senzill gràcies a les plantilles que ja vénen incloses. Es poden trobar diferents plantilles depenent de la plataforma per la qual es treballa. S'inclouen plantilles per projectes iOS, watchOS, tvOS, OS X o d'altres. Per crear una aplicació simple, es recomana utilitzar la Single View Application que és la més bàsica i fàcil d'utilitzar, com a mínim per familiaritzar-se amb l'entorn de Xcode i iOS.

Després de seleccionar la plantilla Single View Application i clicar el botó Next, Xcode presenta un llistat d'opcions per acabar de configurar el nou projecte:

- **Product Name**: El nom del producte serà el nom de la teva aplicació.
- **Organization Name**: El nom de l'organització pot ser el teu propi nom o el de la teva companyia.
- **Organization Identifier**: Xcode fa servir l'identificador de l'organització per crear l'identificador del paquet de l'aplicació. Apple recomana adoptar la notació del nom del domini invers per evitar col·lisions de nom.
- **Bundle Identifier:** L'identificador de paquet és la combinació de l'identificador de l'organització i el nom de producte.

- **Language:** Les versions més recents de Xcode admeten tant Objective-C com Swift. S'ha d'escollir una de les dues opcions.
- **Devices:** El menú de dispositius conté tres opcions, Universal, iPhone, i iPad. Aquesta configuració informa Xcode quins dispositius seran utilitzats per fer les proves durant el projecte. Seleccionant la Universal, el projecte utilitzarà tant iPads com iPhones.
- **Use Core Data:** Si se selecciona aquesta opció, Xcode crearà uns arxius addicionals i afegirà una plantilla de codi per ajudar al desenvolupador a començar a treballar amb Core Data, el *framework* de persistència d'Apple per OS X i iOS.
- **Include Unit Tests:** És un suport per a incloure testing unitari.
- **Include UI Tests:** És un suport per a incloure testing d'interfície d'usuari.

La majoria de les opcions poden ser fàcilment modificades un cop s'ha creat el projecte. Un cop s'hagi acabat de configurar el projecte, es clica el botó Next.

El pròxim pas serà per decidir la localització on s'anirà guardant el projecte cada cop que es facin canvis. Un cop establerta la localització, es clica en el botó Create per crear el projecte.

Al crear-se el projecte, es veurà a l'esquerra una sèrie de fitxers, on es pot trobar a dalt de tot el fitxer que conté la informació de tot el projecte. Si se segueix baixant, es troba una carpeta amb una sèrie de fitxers que s'hauran generat automàticament al crear el projecte i és on s'escriurà el codi. L'organització del codi es mostra de la manera següent:

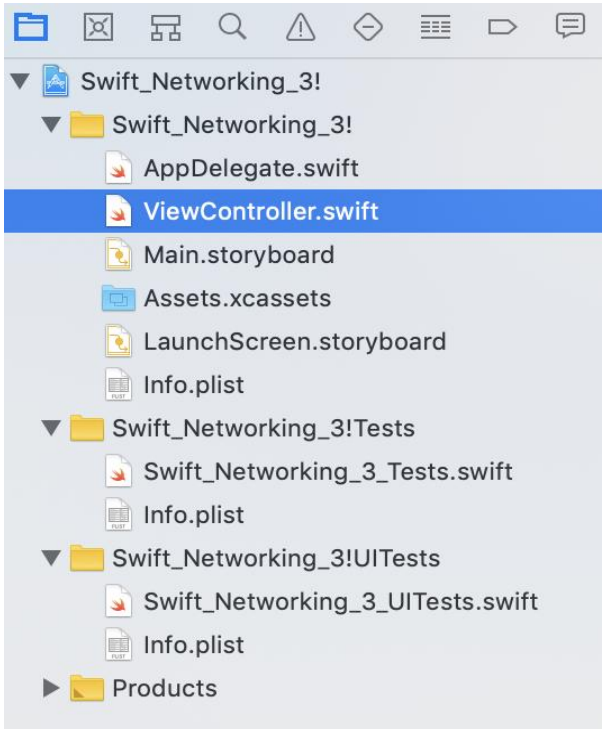

Figura 4: Fitxers de codi (Xcode)

Quan es treballa en el llenguatge Swift, s'acostuma a treballar en el fitxer ViewController.swift. En aquest fitxer es troba la funció `viewDidLoad()` que serveix per executar codi després de carregar una vista. En el cas del meu projecte, la part important és la que s'executarà en aquesta funció i no en la interfície d'usuari. L'objectiu és executar de fons la majoria d'operacions un cop s'ha carregat la vista principal.

Encara que no sigui l'objectiu, en el cas que es vulgui fer qualsevol prova per la interfície d'usuari, es pot realitzar a través del fitxer Main.storyboard.

A la barra de navegació de dalt, trobem un botó Run per fer la compilació i l'execució de l'aplicació. Al costat, hi ha el Stop que serveix per detenir l'execució. A més, en la mateixa barra de navegació, trobem el nom del projecte junt amb el dispositiu en el qual es produirà l'execució. No hi ha problema per utilitzar qualsevol dels simuladors que ofereix Xcode. Es veu de la manera següent:

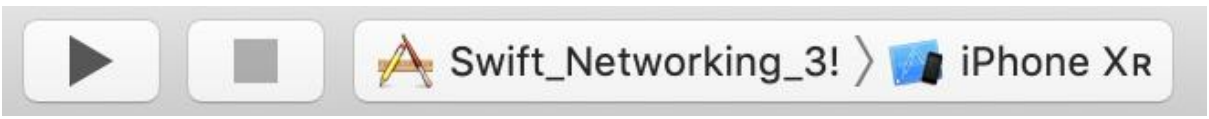

Figura 5: Barra d'execució (Xcode)

Després de fer un Run, ens retornarà un error en el cas que s'hagi trobat qualsevol problema durant el temps de compilació o execució i, en cas contrari, s'haurà executat l'aplicació correctament.

4.2 iOS

Per aconseguir realitzar aplicacions que apliquin tècniques de paral·lelisme, es requereixen dispositius iOS, siguin físics o simulats. En aquest projecte, s'ha de tenir en compte la gran varietat d'opcions que hi ha en tot moment i, veure si són realistes i útils o no tenen gaire sentit realitzar-les. De totes, hi ha una que pren molta importància i cal veure detalladament perquè no s'utilitzarà, ja que en alguns dels estudis que s'han analitzat sí que l'utilitzaven.

4.2.1 *Jailbreak*

S'ha comentat a l'apartat de l'Estat de l'art, en l'estudi anomenat *Anatomy of a globally recursive embedded LINPACK benchmark* [3], realitzen un *jailbreak* en els dispositius. Tot i que, al llarg del projecte s'ha comentat que es vol utilitzar només les eines que ofereix Apple i el *jailbreak* no és una d'elles, primer es necessita veure que és el *jailbreak*, quines avantatges i desavantatges té i, perquè es va descartar l'opció des del primer moment.

Un *jailbreak* és el procés d'eliminar les limitacions imposades per Apple en dispositius que utilitzin el sistema operatiu iOS mitjançant l'ús de kernels modificats [7].

Com a aspecte positiu o negatiu depenent de com es miri, iOS és un entorn tancat. Les aplicacions han de complir les directrius estrictes d'Apple per arribar a estar a l'App Store. En certa manera, aquesta restricció existeix per la protecció dels mateixos usuaris i, per mantenir la qualitat de les aplicacions en general.

Per a Apple, una aplicació s'ha de poder executar en els dispositius sense que es modifiqui cap paràmetre del mateix dispositiu. En cas de voler una aplicació que necessiti fer qualsevol mínima modificació en el dispositiu, ja deixa de poder-se

executar de forma convencional. Per aquest motiu, l'única opció possible, és fer un *jailbreak* en el dispositiu.

Al llarg dels anys han sortit molts tipus de *jailbreaks*. Cada un té els seus avantatges i inconvenients respecte els altres. No es poden fer tots els *jailbreaks* en totes les versions de iOS, trobem que algunes versions només admeten un *jailbreak* determinat. A més, a partir de la versió de iOS 12.1.3 ja no es pot fer cap mena de *jailbreak*.

El primer tipus de *jailbreak* que trobem és el *untethered jailbreak* que no requereix cap assistència quan es reinicia el sistema. El *kernel* es corregirà sense l'ajuda d'un ordinador o d'una aplicació. És una de les millors opcions, ja que no requereix connexió amb l'ordinador, tret del procés de *jailbreaking* inicial. Es pot reiniciar l'iPhone o l'iPod touch tant com es vulgui sense haver de connectar-lo a l'ordinador per arrencar. Si la bateria s'esgota en un dispositiu que té un *untethered jailbreak*, no és un problema tampoc, ja que només carregant-lo de nou s'iniciarà com de costum.

El *tethered jailbreak* és el contrari d'un *untethered jailbreak*. S'ha d'obrir un ordinador per engegar el dispositiu, ja que no s'iniciarà per si mateix. Són bastant incòmodes d'utilitzar perquè requereixen una connexió per arrencar un dispositiu iOS amb *jailbreak* a diferència dels dispositius que tenen *untethered jailbreak*. El motiu és el següent, cada vegada que es reinicia l'iPhone o l'iPod o la bateria s'esgota, s'ha de connectar el dispositiu iOS a l'ordinador de manera que el *hardware* pugui arrencar amb l'ajuda de l'aplicació de *jailbreak*. Després d'haver arrencat el dispositiu, el podreu desconnectar de l'ordinador i utilitzar-lo com de costum. Funcionen tant amb Mac OS X com Windows.

Una barreja entre un *untethered* i un *tethered jailbreak* és també una opció, és a dir, *semi-tethered* [8]. Aquest tipus de *jailbreak* permet a l'usuari tornar a arrencar el dispositiu, però no sense conseqüències. Les funcions que no necessiten *jailbreak* continuaran funcionant, com ara fer una trucada de telèfon, enviar missatges de text o utilitzar una aplicació de l'App Store però, no es podrà executar qualsevol aplicació

que hagi necessitat alguna característica del *jailbreak*. El dispositiu s'ha d'arrencar amb l'ajuda d'un ordinador com els que tenen un *tethered jailbreak*.

Per últim, tenim un tipus de *jailbreak* que s'ha anat utilitzant cada cop més, és el *semi-untethered*. Aquest tipus de *jailbreak* és com un *semi-tethered jailbreak* en el qual, quan el dispositiu es reinicia ja no té un *kernel* corregit però, el *kernel* es pot corregir copiant-lo mitjançant una aplicació instal·lada en el dispositiu i, per tant, sense l'ajuda d'un ordinador. Es realitza aquest *jailbreak* sobretot en les últimes versions de iOS, entre la versió 9.3.5 i l'última que té possibilitat de *jailbreak*, la versió 12.1.2.

Cal aclarir uns aspectes importants relacionats amb *jailbreaking*. Primer de tot, aclarir que fer un *jailbreak* és legal gràcies a les lleis relatives a l'elusió dels bloquejos digitals, com les lleis que protegeixen els mecanismes de gestió de drets digitals (DRM), exceptuant certs països que o no tenen aquestes lleis, o han exclòs el *jailbreak* d'aquestes. Per l'altra banda, fer un *jailbreak* al dispositiu iOS va en contra de les Condicions del servei que s'accepten quan s'inicia el telèfon per primer cop, per tant, si es vol portar un dispositiu a reparar amb la garantia, és necessari que es desfaci el *jailbreak* prèviament. Tots els tipus de *jailbreaks* són completament reversibles i es recupera el funcionament del dispositiu previ al *jailbreak*.

Un cop al dispositiu se li ha aplicat qualsevol dels *jailbreaks* comentats, necessitarem instal·lar Cydia. Cydia no és més que un gestor de paquets dpkg per iOS que utilitza una interfície gràfica. Cydia va ser creada per Jay Freeman, conegut en totes les xarxes com a Saurik [9]. Al principi Cydia servia únicament perquè els desenvolupadors oferissin el seu *software* de forma completament gratuïta, *software* que generalment havia estat rebutjat de l'App Store d'Apple per algun motiu però, ha acabat tenint un funcionament bastant similar al de l'App Store. La majoria d'aplicacions que podem descarregar són completament gratuïtes, moltes d'elles són aplicacions completament independents, tot i que hi ha moltes altres que depenen d'altres aplicacions (a vegades, natives de iOS) que el que fan és afegir noves funcionalitats o personalitzar diferents aspectes de l'aplicació.

Figura 6: Cydia

Utilitzant el cercador integrat es pot trobar l'aplicació que s'estigui buscant en el cas que coneguem el seu nom o podem navegar per les diferents categories veient les aplicacions disponibles, de manera semblant a com es fa en l'App Store. Un cop localitzada una aplicació que resulti interessant, s'ha de prémer el botó d'instal·lació i es passarà a tenir l'aplicació en el dispositiu.

Com a causa que no era viable l'equilibri entre ingressos i despeses des d'un temps enrere, la botiga de Cydia s'anava a tancar. A més, es va trobar un error en el codi de Cydia que va fer que Saurik decidís tancar-la abans del previst, concretament el desembre del 2018. Tot i això, Cydia seguirà funcionant i permeten instal·lar, desinstal·lar i administrar les nostres eines, temes de *jailbreak* existents i, a més, es podrà seguir navegant pels repositoris actuals. L'únic que es va tancar va ser la botiga.

4.2.1.1 Problemes *Jailbreak*

El *jailbreak* va anar perdent cada cop més pes amb la sortida de les noves versions de iOS [10]. En resum, implicava massa feina per obtenir molt poc a canvi. En les primeres versions on es podia fer *jailbreak,* les característiques que es podien obtenir eren molt més exclusives, en alguns casos eren gairebé la raó per la qual

algunes persones es compraven el telèfon però, cada cop més, només es rebien a canvi petites modificacions en el *software*. Aquest fet va començar a provocar un bucle, ja que va disminuir la quantitat de gent que feia *jailbreak* i va provocar que molts desenvolupadors deixessin de fer coses interessants per a la comunitat. I com menys desenvolupadors facin eines interessants, menys usuaris faran *jailbreak* als seus dispositius. A més les recompenses per les vulnerabilitats també tenen a veure. Quan es troba un forat en la seguretat es poden fer dues coses, o difondre'l a Internet i crear el *jailbreak*, o vendre aquest forat de seguretat a Apple o altres empreses. Per la primera opció la recompensa en diners és pràcticament nul·la, però per la segona, tant Apple com altres empreses de seguretat poden pagar molt bé al desenvolupador. Això ha ajudat al fet que els pioners del *jailbreak* s'han passat al costat bo. Tant Apple com altres empreses han sabut lluitar contra el jailbreak de forma intel·ligent. S'han dedicat a contractar a tots els hackers del panorama. Mitjançant suculentes ofertes de treball molts han preferit passar-se al costat bo i treballar directament per Apple descobrint i arreglant aquests forats de seguretat.

De totes maneres, no són els únics motius pels quals s'ha anat deixant d'utilitzar el *jailbreak*. La seguretat d'Apple és un altre motiu. Any rere any, iOS s'ha convertit en un sistema molt més segur. Amb cada actualització Apple cobreix més forats de seguretat. En les primeres versions de iOS era molt senzill entrar al sistema, mentre que ara és pràcticament impossible fer-ho. Seguint amb la seguretat, la seguretat del mateix usuari també és un motiu força important a tenir en compte. La raó que més ens ha d'importar, és la seguretat del dispositiu i com aquesta s'exposa al fer *jailbreak*. Totes les nostres dades poden ser llegides per tercers perquè al fer *jailbreak*, pràcticament obres les portes perquè pugui entrar qualsevol al nostre telèfon.

Ja han passat més de 10 anys des dels orígens del *jailbreak*. En tot aquest temps, s'han passat per diferents etapes, etapes en què ha tingut sentit defensar-lo i etapes en què pràcticament ja no hi ha raons de pes per fer-ho.

En els primers anys del *jailbreak* aquest es realitzava per esprémer el telèfon i poder treure-li el màxim partit a l'iPhone. Un exemple de la seva utilitat seria la següent, en les primeres versions es podien utilitzar algunes eines disponibles que ens

permetien enviar missatges multimèdia o enregistrar un vídeo amb la càmera de l'iPhone. Eren utilitats que al principi Apple no permetia i que posteriorment va anar implementant. Era la fase del *jailbreak* que ens permetia instal·lar eines que posteriorment Apple implementava a la seva manera, la fase en què t'avançaves a les futures versions de iOS.

Després d'això, els principals desenvolupadors de la comunitat es van adonar que Apple estava implementant cada vegada més i més novetats en iOS i que al mateix temps, obrien de forma segura el sistema perquè es puguin fer aquestes coses de manera legal per tercers. Arran d'això, els desenvolupadors van anar abandonant la comunitat. El seu relleu van ser grups de hackers anònims, grups que no revelaven la seva identitat i que per tant no sabíem a qui estàvem cedint l'accés als nostres dispositius.

L'última fase és la fase en què el *jailbreak* està pràcticament mort. El *jailbreak* va ser essencial en el seu moment per al desenvolupament de iOS i que iOS no hauria arribat on és ara si no fos pel *jailbreak*. Però ara, el *jailbreak* s'ha quedat com una cosa nostàlgica, essencial fa uns anys, però sense sentit ara mateix.
Al principi del projecte s'ha parlat sobre l'estat de l'art on es comentaven dos projectes que havien utilitzat computació paral·lela en iOS. Per fer el projecte, no es volia contemplar l'opció d'utilitzar-lo però, si es volia analitzar força exhaustivament aquesta opció per veure els motius pels quals els projectes esmentats en el capítol de l'estat de l'art es van decantar per fer *jailbreaks*. Tal i com s'ha comentat en tot el capítol del *jailbreak*, és important entendre que si et decantes per fer un *jailbreak*, estaries presentant una idea utilitzant una eina que cada cop utilitza menys gent, incloent-hi els mateixos desenvolupadors. L'objectiu és tenir en compte el present dels dispositius i el futur al màxim però, en cap cas ajudar-se d'eines que a poc a poc van perdent utilitat i quedant en desús. Amb tota aquesta informació, podem afirmar amb seguretat que no té cap sentit seguir utilitzant el *jailbreak* i menys en el projecte. Descartar fer un *jailbreak* no implica necessàriament descartar l'opció d'utilitzar MPI en els dispositius iOS, ja que l'objectiu en tot moment ha estat intentar utilitzar-lo però, mitjançant les eines d'Apple i, fent un *jailbreak* no es compleix aquesta condició.

4.3 MPI

4.3.1 Triar MPI

Primer de tot, per poder utilitzar MPI correctement, es necessita triar la versió correcta que funcioni per Mac OS X i, si és possible per iOS [1]. És important triar una versió havent mirat totes les possibilitats per assegurar que és la millor per realitzar el projecte.

Basant-me en el projecte del cluster d'Apple TV comentat en l'apartat de l'Estat de l'art, m'ha ajudat a determinar la implementació de MPI. Les implementacions de MPI més conegudes són OpenMPI, MVAPICH, MPICH, IntelMPI. De totes aquestes, utilitzaré MPICH, concretament MPICH 3.3 [11]. M'he decantat cap a la versió 3.3, ja que és una versió estable i no ha de contenir errors. La versió 4.0 o superior encara està en desenvolupament i no tinc cap garantia que sigui del tot estable i, em pot donar problemes en un futur.

4.3.2 Funcions MPI

MPI ofereix una extensa quantitat de funcions. A continuació veurem algunes de les més importants [12].

Primer de tot, trobem les funcions bàsiques on:

- `MPI_Init`: s'encarrega d'inicialitzar l'entorn d'execució de MPI
- `MPI_Finalize`: s'encarrega de finalitzar l'entorn d'execució de MPI. A més, ha de complir la condició de ser cridada per tots els processos que formen part del programa MPI.

Un element a destacar és el comunicador, el qual és un objecte que descriu una estructura de comunicació per a un grup de procesos [1]. Aquests processos dins del comunicador són capaços d'enviar-se i rebre missatges, essent una estructura bàsica en MPI. Els comunicadors formen part dels *handlers* (que són els apuntadors a les estructures de dades que MPI crea i manté per establir la comunicació entre els diferents processos del programa), i estan compostos per un *group* (els processos que participen en aquell comunicador en concret), i el seu context

(comunicació punt-a-punt o colectiva). El comunicador predefinit a MPI és `MPI_COMM_WORLD`, que engloba tots els processos que inicialment formen part d'un programa MPI.

Algunes de les funcions més utilitzades [12] per crear o gestionar els comunicadors i els processos corresponents són:

- `MPI_comm_size`: retorna el número de processos que té el comunicador.
- `MPI_comm_rank`: determina el rang del procés que es crida en el comunicador.

Un gran nombre de funcions MPI involucren la comunicació entre dos processos. Les més utilitzades per comunicacions punt-a-punt són les següents:

- `MPI_Send`: aquesta funció permet enviar, mitjançant un missatge MPI, les dades guardades en un *buffer*.
- `MPI_Recv`: si `MPI_Send` serveix per enviar missatges, es necessitarà una funció per rebre'ls. `MPI_Recv` té aquesta funció i, a més, guarda els missatges rebuts en un *buffer*.

Les funcions col·lectives involucren comunicacions entre tots els processos d'un grup de processos. Algunes de les funcions de comunicació col·lectiva que hi ha són:

- `MPI_Bcast`: aquesta funció fa que el procés identificat amb el rang *root* enviï a tots els procesos les dades del *buffer*.
- `MPI_Scatter`: implica un procés *root* designat que envia dades a tots els processos d'un comunicador. Té una funció molt similar a `MPI_Bcast` però, no són iguals. `MPI_Bcast` envia la mateixa informació a tots els processos mentre `MPI_Scatter` envia trossos diferents dels processos, és a dir, els processos no reben la mateixa informació.
- `MPI_Gather`: té la funció inversa a `MPI_Scatter`. En lloc de difondre elements d'un procés a molts processos, `MPI_Gather` pren elements de molts processos i els reuneix en un sol procés.

Les funcions orientades a la comunicació poden ser bloquejants o no bloquejants. Les bloquejants són funcions que es bloquegen fins que no es finalitza la comunicació. El bloqueig de la comunicació es fa mitjançant les funcions que he comentat abans: `MPI_Send()` i `MPI_Recv()`. Per altra banda, la comunicació no bloquejant es fa mitjançant funcions com `MPI_Isend()` i `MPI_Irecv()`. Aquestes funcions tornen immediatament encara que la comunicació encara no hagi finalitzat. Per veure si la comunicació ha acabat, s'ha d'utilitzar funcions com `MPI_Wait()` o `MPI_Test()`. Normalment les comunicacions no bloquejats s'utilitzen només si són necessàries, com per exemple: mentre has cridat `MPI_Isend()` i esperes que acabi amb `MPI_Wait()`, fas alguns càlculs. Això permet que els càlculs i la comunicació es superposin, la qual cosa generalment comporta un rendiment millorat.

En termes de sincronització, una funció important és:

- `MPI_Barrier`: és una funció especial de MPI dedicada a la sincronització de processos. Bloqueja fins que tots els processos del comunicador han arribat a aquesta rutina.

Hi ha moltes més funcions, però aquestes són les més bàsiques i necessàries a l'hora de desenvolupar una aplicació MPI.

4.3.3 Instal·lació MPICH

Com he comentat anteriorment, he triat la implementació MPICH, on utilitzaré la versió 3.3. Per poder instal·lar MPICH en MacOS X, tenim dues opcions: via *MacPorts* o via *Homebrew* [11].

MacPorts és una iniciativa de la comunitat utilitzant codi obert, que ha dissenyat un sistema fàcil d'usar per compilar, instal·lar i actualitzar *software* de codi obert basat en comandaments o interfície gràfica en el sistema operatiu Mac.

Homebrew és un gestor de paquets que simplifica la instal·lació, actualització i eliminació de programes en els sistemes operatius macOS d'Apple i GNU/Linux. És

considerada la manera més fàcil i flexible d'instal·lar les eines UNIX que Apple no inclou amb macOS.

En el present estudi m'he decantat per utilitzar aquesta última. Un cop triada, he instal·lat *Homebrew* i he seguit amb la instal·lació de MPICH.

Per instal·lar MPICH es va executar la comanda següent:

- `$ brew install mpich`

Per triar una versió concreta, només s'ha d'afegir el número de la versió que vols al final de la comanda.

Si no s'ha instal·lat qualsevol implementació de MPI mai, s'ha de modificar el fitxer `/etc/hosts` de l'ordinador, afegint una línia nova de la següent manera:

- `127.0.0.1 computername.local` on *computername* és el nom del MacBook.

Per saber el nom en un MacBook, selecciona el menú Apple i, al desplegar-se accedeix a preferències del sistema i fas clic a compartir. El nom del teu MacBook és el que es mostra a la part superior.

Després d'haver guardat els canvis del fitxer `/etc/hosts`, ja es pot executar MPICH en el MacBook. En cas de no modificar el fitxer `/etc/hosts` poden surgir errors quan s'intenti executar qualsevol mena d'aplicació que contingui MPI.

4.3.4 Alternativa a MPICH

Open-MPI és una altra implementació de MPI. Durant tot el projecte s'ha utilitzat MPICH però, hi ha l'excepció d'un cas concret on s'ha utilitzat Open-MPI. Tot i que aquest cas es comentarà més endavant en la secció Compatibilitat amb Xcode, a continuació es veu com fer la instal·lació.

Per poder utilitzar Open-MPI, va ser necessària la seva instal·lació, tot i que va ser puntualment, ja que un cop no es necessitava es va desinstal·lar. Per instal·lar Open-MPI es realitza mitjançant la comanda:

- `$ brew install open-mpi`

Un cop ja no es necessita més Open-MPI es desinstal·la mitjançant la comanda:

- `$ brew uninstall open-mpi`

4.4 MPI a macOS

4.4.1 Xcode

Es vol veure si és possible utilitzar MPI en dispositius iOS mitjançant les eines d'Apple. Per fer-ho, s'ha de realitzar la configuració des de macOS amb l'eina Xcode.

4.4.1.1 Build

En aquest apartat analitzaré en què consisteix fer un *build* dins d'un projecte de Xcode i l'entorn que es genera. A partir d'aquí, una de les opcions que estic buscant, és la d'analitzar l'estructura del codi generat i crear-lo des de la línia d'ordres que Xcode ofereix, de forma que es puguin afegir o modificar accions no permeses en l'entorn de Xcode.

Per accedir a la carpeta Build en Xcode, i es pot accedir de dues formes diferents:

- Des de la consola, on accedim a la ruta següent:

  `~/Library/Developer/Xcode/DerivedData/NomProjecte`

  i, posteriorment es podrà obrir el buscador en la carpeta corresponent. Per fer-ho s'executa la comanda:

  `$ open .`
- S'obré el buscador i s'accedeix a la ruta corresponent:

  `User/Library/Developer/Xcode/DerivedData/NomProjecte`

En cas d'haver canviat les preferències d'on es guardin els directoris de compilació, aquesta ruta no serà vàlida i t'hauràs de dirigir a la ruta escollida.

Dintre de la carpeta del projecte seleccionat, es troba diferent informació. Estic interessat només en les carpetes Build, Índex i Logs i en el fitxer info.plist.

Primer de tot ens fixem en el fitxer info.plist, que conté la ruta de l'espai de treball del present projecte, i ens permet saber on tinc guardat el meu projecte.

A continuació ens centrem en la part important, la carpeta Build.

Dins la carpeta Build podem trobar diferents carpetes, però hi ha dues importants. En la primera es poden trobar els fitxers .json amb la informació del projecte i, en la segona, els fitxers .d de cada fitxer .m que tenim en el projecte. En aquests fitxers .d es troben totes les dependències que necessiten i, així podem veure quins fitxers necessiten altres i quins no.

També podem veure els fitxers .plist que son fitxers que es generen amb codi XML(*eXtensible Markup Language*). Aquests fitxers tenen una estructura molt senzilla que facilita la seva manipulació o, crear-los manualment.

Si es vol fer la compilació del projecte des de la línia d'ordres he de seguir els passos següents:

- Obrir la consola i navegar fins al directori on resideix el meu <NOM> .xcodeproj.
- Executar la comanda:

```
$ xcodebuild -list -project <NOM> .xcodeproj /
```

- Un cop executada la comanda, se'ns mostrarà una llista d'esquemes. Hem de triar l'esquema desitjat i l'executem de la manera següent:

```
$ xcodebuild -scheme <NOM DE L'ESQUEMA> build
```

Per poder realitzar aquest últim pas sense problemes, prèviament has de seleccionar un equip de desenvolupament, per l'aplicació, dins l'editor de Xcode. Altrament el *build* fallarà.

A partir d'aquí, ja només ens faria falta fer el *deploy* en el dispositiu iOS. Per realitzar-ho, podem ajudar-nos de l'eina ios-deploy, i instal·lar-la mitjançant la comanda:

```
$ npm install -g ios-deploy
```

Un cop estigui instal·lada, només ens farà falta executar la comanda següent, afegint-li la ruta que obtenim al final de la informació obtinguda de la comanda `xcodebuild`:

```
$ ios-deploy --debug --bundle <RUTA APP>
```

Finalment, s'hauria d'iniciar l'aplicació en un dispositiu iOS connectat.

4.4.1.2 Compatibilitat amb Xcode

He comprovat que sempre que desenvolupem una aplicació *command line* per a macOS, podem inserir la llibreria mpi.h dins de Xcode i, funciona correctament sempre que s'utilitzi Open-MPI (MPICH no funciona). L'objectiu final d'aquest estudi és que també funcioni en aplicacions iOS, no només de macOS.

Per afegir la capçalera MPI dins de Xcode, es realitza de la mateixa manera tant per aplicacions de MacOS com de iOS. Per afegir-la i que la detecti correctament hi ha uns passos a seguir.

Primer de tot, s'ha d'instal·lar Open-MPI a l'ordinador. Cal mencionar que durant el projecte sempre s'utilitzarà MPICH en lloc de Open-MPI però, en aquest apartat concret, s'explica l'intent d'incloure la llibreria MPI dins de Xcode i que s'executi correctament i, per fer-ho només es pot amb Open-MPI. Com ja he comentat anteriorment, MPI es pot instal·lar via MacPorts o via Homebrew. S'ha de tenir en compte que si instal·les Open-MPI en un ordinador que tingui MPICH instal·lat o viceversa, es crea un conflicte entre els dos. No poden estar instal·lats a la vegada, per tant, hi ha dues solucions per resoldre el problema. La primera i més senzilla

seria desinstal·lar MPICH abans d'instal·lar Open-MPI. Altrament, pots tenir els dos instal·lats i invocar-los de forma desvinculada i, així no creant cap conflicte. Un exemple de com fer la crida, seria de la següent manera:

- `$(brew --prefix open-mpi)/bin/mpicc`

Pel present projecte, com només necessitem una de les dues llibreries, qualsevol de les dues opcions són vàlides.

A continuació, s'ha d'executar la comanda següent:

- `$mpicc -showme`

Aquesta comanda ens retornarà una informació que es necessitarà posteriorment a Xcode. El resultat retornat en el present cas és el següent:

- `clang -I/usr/local/Cellar/open-mpi/4.0.1_1/include -L/usr/local/opt/libevent/lib -L/usr/local/Cellar/open-mpi/4.0.1_1/lib -lmpi`

Com podem veure, ens retorna la ruta d'*includes*, la ruta de les llibreries i un *flag* (`-lmpi`). Amb aquesta informació, ja es pot afegir els camps necessaris dintre de Xcode. Per fer-ho, accedim al meu projecte de Xcode i anem a l'opció de *Build Settings*. Un cop dins s'ha d'afegir les rutes de la següent manera:

- "`/usr/local/Cellar/open-mpi/4.0.1_1/include`" dins de "Search Paths > Header Search Paths"
- "`/usr/local/opt/libevent/lib`" i "`/usr/local/Cellar/open-mpi/4.0.1_1/lib`" dins de "Search Paths > Library Search Paths"
- "`-lmpi`" dins de "Linking > Other Linker Flags"

A continuació, hem d'accedir a l'esquema del projecte, concretament a Edit Scheme per tal de modificar-lo. Accedim a través de Project>Scheme>Edit Scheme. S'obrirà una pestanya on es mostraran vàries opcions: Build, Run, Test, Profile, Analyze i Archive. Només necessitem modificar paràmetres dins de Run, la resta d'opcions no fa falta modificar-les.

Per a executar mpiexec necessitem saber diversos arguments. Un d'ells és el nombre de processadors i, l'altre, el fitxer executable. Accedim a la pestanya d'Arguments i afegim els arguments de la següent forma:

- "`-n X`" on X és el nombre de processadors que vols utilitzar.
- "`$(CONFIGURATION_BUILD_DIR)/NOM_APLICACIÓ`" que és la combinació de variables d'entorn que especifiquen el fitxer executable.

No sortim de Run i accedim a Info. Dins de Info canviem l'executable per defecte per Other…. Hem de buscar mpiexec, que es troba amb l'àlies d'orterun. Aquest executable es troba en una carpeta oculta. Un cop s'obra el buscador, podem obrir un buscador que també cerca carpetes ocultes prement cmd + shift + g. En el meu cas, "orterun" es trobava en la localització:

- "`/usr/local/Cellar/open-mpi/4.0.1_1/bin/`".

Per finalitzar, només falta afegir la capçalera `<mpi.h>` en el codi font on es vulgui utilitzar les seves funcions.

Com he comentat al principi de l'explicació, hem utilitzat Open-MPI i no MPICH, ja que MPICH no funcionava. El motiu és que no es pot seleccionar cap executable amb MPICH. La resta de passos es fan de la mateixa manera que si ho féssim amb Open-MPI, però un cop hem de seleccionar l'executable i anem a la localització corresponent, no es pot seleccionar cap executable com "orterun" en el cas d'Open-MPI i, per tant, podem fer el *build*, però no podem executar l'aplicació.

En iOS pots arribar a incloure la capçalera `<mpi.h>` en el codi font sense cap problema, seguint els passos que hem comentat per MacOS. En canvi, a l'hora d'executar si que dóna problemes i no deixa executar-se. Els dispositius iOS no admeten MPI i, cal recordar que, nosaltres tenim instal·lat Open-MPI en un dispositiu MacOS i no en el simulador de iOS de Xcode on estem executant el programa. Això també ho podem deduir a partir de la informació extreta a l'executar el programa, que ens dóna la informació d'error avisant-nos que el programa s'ha

creat per una plataforma que no suporta la seva execució. Per aconseguir que iOS si ho suportés, s'hauria de fer el *jailbreak* comentat anteriorment però, com ja s'ha dit, el projecte busca utilitzar les eines d'Apple per tal de garantir la compatibilitat entre els dispositius i de cara al futur.

4.4.1.3 Compatibilitat desde la consola

Tot i que no és una opció tan pràctica, ens quedava l'opció de comprovar si funcionava mitjançant la línia d'ordres. Aquesta opció no és tan pràctica, ja que s'ha d'afegir en cada execució una gran quantitat de paràmetres. A continuació veiem un exemple amb tots els paràmetres explicats.

- `xcrun --sdk iphonesimulator gcc -I/usr/local/Cellar/open-mpi/4.0.1_1/include -L/usr/local/Cellar/open-mpi/4.0.1_1/lib -lmpi --target=x86_64 -L/usr/local/opt/libevent/lib -o hello hello.c`

El primer paràmetre de tots és `xcrun` que ens permet executar una aplicació de Xcode des de la línia d'ordres. Els dos paràmetres que trobem a continuació són el `--sdk` junt amb el tipus de dispositiu que volem realitzar l'execució de l'aplicació. En el cas de treballar amb el simulador que ofereix Xcode, s'afegeix el paràmetre "iphonesimulator" com es veu en l'exemple.

L'objectiu d'aquesta comanda és fer l'execució de la mateixa manera que s'havia comentat en l'apartat anterior però, en lloc d'incloure MPI dins Xcode, agafem les rutes on tenim MPI i les posem a continuació de la comanda a executar. D'aquesta manera aconseguim que quan s'executi l'aplicació, busqui on està instal·lat MPI en l'ordinador i l'utilitzi.

Finalment queda explicar el paràmetre que té més repercussió en aquesta comanda. El paràmetre `--target` permet indicar el tipus d'arquitectura en el que s'executarà l'aplicació. Perquè tingui sentit aquesta comanda, s'ha de posar l'arquitectura del dispositiu que s'ha indicat prèviament que s'utilitzarà. Aquest

paràmetre et permet jugar fent múltiples proves d'execució en diferents dispositius. Per exemple, a la comanda xcrun li pots afegir:

- `--target=x86_64-apple-darwin18.6.0`

En aquest cas s'executa l'aplicació utilitzant l'arquitectura del macOS i, per tant l'aplicació fa la compilació i execució correctament. Òbviament, el que es vol és l'execució mitjançant l'arquitectura de l'iPhone però, ens serveix per comprovar que tots els paràmetres relacionats amb MPI s'han afegit i funcionen correctament. Per fer l'execució correctament afegim el paràmetre segons l'arquitectura de l'iPhone o si estem utilitzant el simulador. Com l'exemple és amb el simulador, s'estructura el `--target` de la forma següent:

- `--target=x86_64`

A partir d'aquí s'aconsegueix tenir una comanda on s'agafa MPI de l'ordinador, es compila l'aplicació per iOS i s'executa en el dispositiu iOS. En la pràctica no funciona. El problema és similar a quan treballem des de Xcode. La comanda permet fer la compilació amb el MPI que tenim instal·lat a l'ordinador. A partir d'aquesta compilació es crea un executable però, aquest executable s'ha d'executar en el dispositiu iOS. Per molt que utilitzem les rutes de MPI, són per fer la compilació des de l'ordinador i no ajuden en res en l'execució. L'executable s'envia al dispositiu seleccionat, el simulador en el meu cas, que posteriorment l'executa. Aquest dispositiu ja no utilitza les rutes de MPI a l'hora de fer l'execució i, provoca que el mateix dispositiu segueixi sense suportar MPI. Un cop s'ha executat l'aplicació, apareixen els mateixos errors que quan es va intentar des de Xcode.

Després de totes les proves que s'han realitzat, es demostra que actualment no es pot utilitzar MPI en dispositius iOS sense fer modificacions en l'iPhone mitjançant un *jailbreak*. Un cop descartada l'única opció per incloure MPI en dispositius iOS mitjançant les eines que ens dóna Apple, queda encara més clar que s'ha de buscar altres mètodes dins les eines que ens ofereix Apple per desenvolupar l'aplicació iOS i oblidar-nos d'utilitzar MPI. Al final, s'ha acabat fent l'aplicació amb Grand Central Dispatch (GCD) junt amb el *framework* Network. Més endavant es justifica tant l'ús

de GCD, com l'ús de Network *framework* en lloc d'utilitzar *sockets* que seria l'alternativa que sembla més lògica.

4.5 Objective-C

Objective-C és un llenguatge de programació orientat a objectes [13]. Va ser el principal llenguatge de programació suportat per Apple pels sistemes operatius macOS, iOS i iPadOS, i les seves respectives interfícies de programació d'aplicacions (API) Cocoa i Cocoa Touch fins a la introducció de Swift. Els programes portables d'Objective-C que no utilitzen Cocoa o Cocoa Touch, o els que utilitzin parts que puguin ser portables o reimplementades per a altres sistemes, també es poden compilar per a qualsevol sistema suportat per GNU Compiler Collection (GCC) o Clang.

Els fitxers que s'utilitzen en Objective-C tenen extensions .m, mentre que els fitxers en C tenen extensions .c.

4.6 Swift

Swift és un llenguatge de programació orientat a objectes creat per Apple per al desenvolupament d'aplicacions iOS, macOS, watchOS, tvOS entre d'altres [14]. És un llenguatge de programació segur, ràpid i interactiu que combina amb l'enginyeria d'Apple i les diverses aportacions de la seva comunitat de codi obert. El compilador està optimitzat per al rendiment. LLVM (*Low Level Virtual Machine*) és el compilador de Swift i, que té com a objectiu millorar l'optimització dels programes. Swift està inclòs dins de Xcode des de la versió 6, en el 2014. La combinació de seguretat i velocitat fan de Swift una excel·lent elecció per desenvolupar qualsevol mena d'aplicació. Apple pretenia amb el llenguatge Swift, donar suport a molts conceptes bàsics associats amb Objective-C, però d'una manera més segura, facilitant la detecció d'errors de *software*, a més de la coexistència entre codi Swift i Objective-C. Actualment tenim Swift 5 però ha canviat força des de la primera versió. Swift va tenir el canvi més dràstic i important a través de la versió 3.0, ja que la sintaxi de Swift va passar per una evolució significativa respecte a les versions anteriors. Com a resultat de totes les millores en les diferents versions de Swift, el codi és fàcil d'escriure però, a més, també és més fàcil de llegir i mantenir.

Tenint en compte els dos llenguatges que podem utilitzar en l'entorn de Xcode, m'he decantat per utilitzar Swift, a causa dels grans avantatges que representa respecte a treballar amb Objective-C, a més de tenir una coexistència amb Objective-C que ens permet incloure codi si és necessari.

4.7 Tipus de computació

4.7.1 Concurrència i Paral·lelisme

No s'ha de confondre el paral·lelisme amb la concurrència, ja que aquests conceptes estan relacionats però, no són iguals. El paral·lelisme només es pot aconseguir en dispositius amb diversos nuclis; mentre que un nucli executa una tasca, un altre nucli pot executar l'altra tasca simultàniament [15].

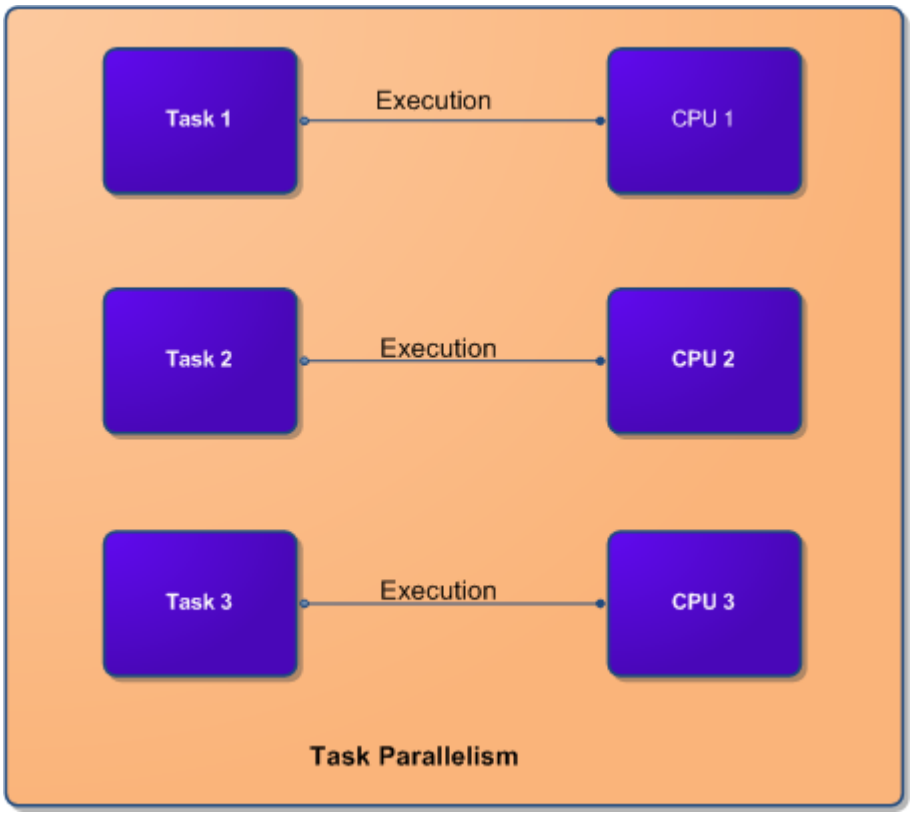

Figura 7: Concurrència i paral·lelisme 1

Es poden executar tasques concurrents mitjançant paral·lelisme però, el paral·lelisme no és necessari per a la concurrència. Amb dispositius d'un únic nucli, s'aconsegueix la concurrència mitjançant canvis de context: el nucli executa una

tasca durant algun temps, després passa a l'altra tasca o procés, l'executa, després torna a la tasca anterior, i així, fins que la tasca estigui completa.

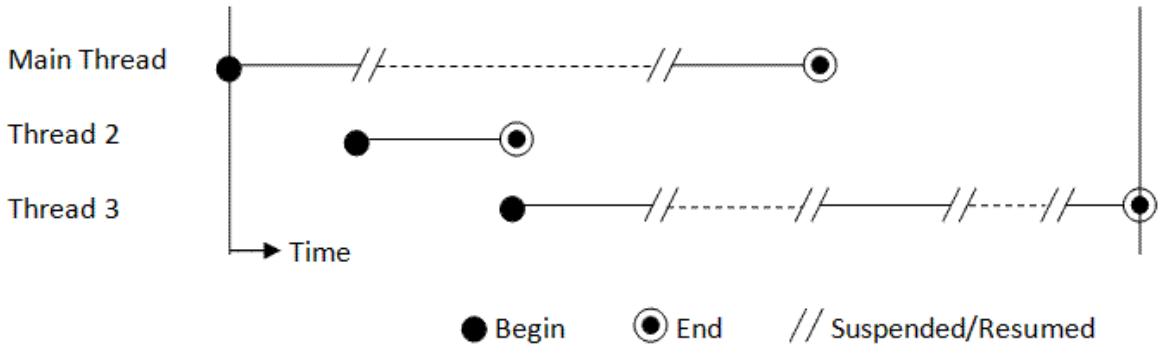

Figura 8: Concurrència i paral·lelisme 2

La concurrència funciona amb canvis de context constants i de forma ràpida. A més, un canvi de context requereix emmagatzemar i restaurar l'estat d'execució en canviar entre els fils, significant un cost addicional.

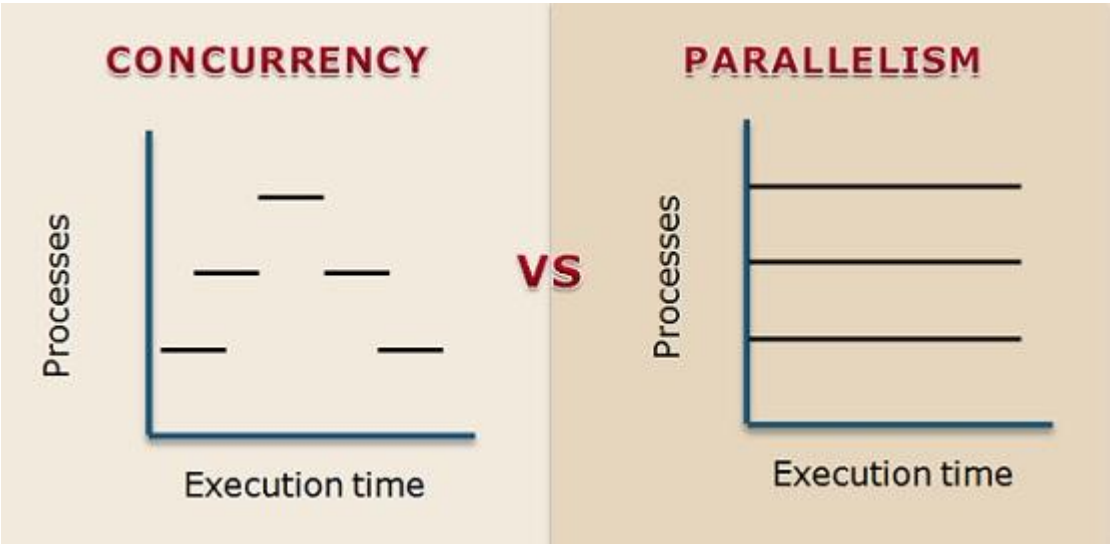

Figura 9: Concurrència i paral·lelisme 3

4.7.2 *Multithreading*

En iOS, la manera de realitzar la concurrència és mitjançant *threads*.

Un *thread* és la unitat més petita de processament que pot ser programada pels sistemes operatius, i que permet a un procés executar diferents tasques al mateix

temps. Cada *thread* té un procés que ha de ser executat. Aquesta característica dóna la possibilitat al programador de dissenyar un programa que executi diferents funcions concurrentment.

La tècnica de programació amb *threads* s'anomena *multithreading* i permet simplificar el disseny d'aplicacions concurrents i millorar el rendiment de la creació de processos. El *multithreading* és l'habilitat d'una unitat central de processament (CPU) (o d'un sol nucli en un processador de diversos nuclis) de proporcionar múltiples *threads* d'execució simultàniament, suportats pel sistema operatiu [16]. En una aplicació amb diversos *threads*, els *threads* comparteixen els recursos d'un o diversos nuclis, que inclouen les unitats de computació, les memòries cau de la CPU i la Translation Lookaside Buffer (la TLB és una memòria cau administrada per la unitat de gestió de memòria, que conté parts de la taula de paginació, la qual relaciona les adreces lògiques amb les físiques). En canvi, cada *thread* consta de les seves pròpies instruccions, la seva pròpia pila d'execució, s'executen a diferents velocitats i tenen el seu propi estat d'execució.

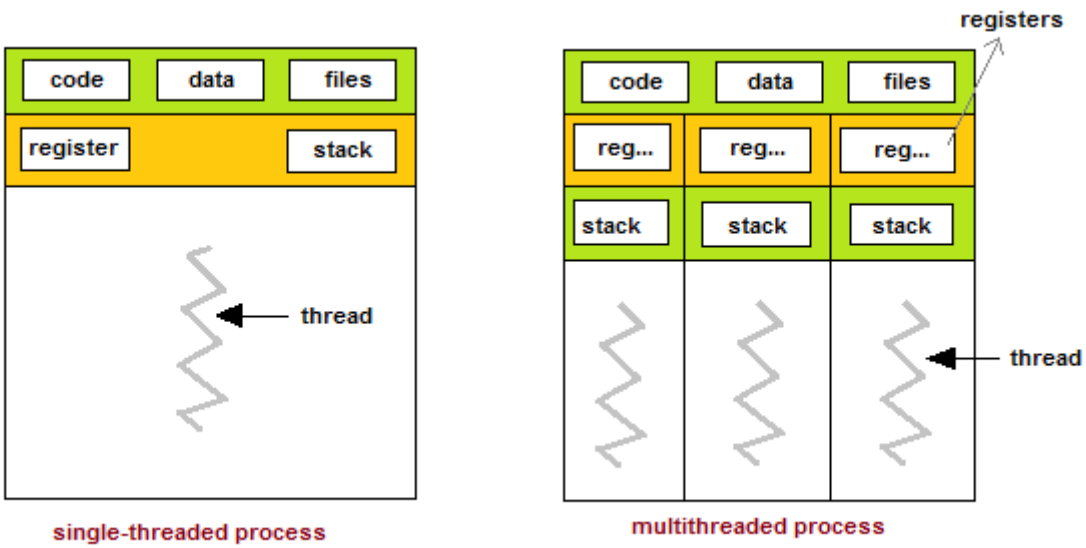

Figura 10: ***Multithreading***

En el cas de l'iPhone, sabem que té una CPU, la seva unitat central de processament. Tècnicament, una CPU només pot realitzar una operació a la vegada i, per això, s'introdueix el *multithreading*. El *multithreading* permet al processador

crear *threads* concurrents entre els quals es pot canviar, de manera que permeten executar múltiples tasques al mateix temps. Sembla com si els *threads* s'executin alhora però, el que passa és que el processador realitza canvis de context ràpidament entre ells i s'executen per parts. Els canvis de context es produeixen quan un *thread* que està al processador és eliminat i entra un altre. Fent canvis de context molt ràpidament donem la sensació que tots els *threads* s'executen simultàniament. Per tant, es pot dir que tots el *threads* s'executen concurrentment.

Un exemple molt comú on s'utilitzen *threads* és en les aplicacions client/servidor, on el servidor crea múltiples *threads* per tal de donar servei a múltiples clients a la vegada.

4.8 Grand Central Dispatch

Grand Central Dispatch (GCD) és una tecnologia desenvolupada per Apple en Mac OS X v10.6 per optimitzar el suport de les aplicacions per a processadors de diversos nuclis [16]. GCD és la llibreria que s'encarrega de la programació concurrent, és a dir, de com podem llançar diferents processos del nostre programa en diferents fils (*threads*) d'execució perquè s'executin al mateix temps. Quan tenim una CPU amb múltiples nuclis i fins i tot aquests nuclis tenen diversos fils, podem aprofitar aquests canals per llançar processos que puguin executar-se simultàniament i que aprofitin millor el rendiment dels nostres equips.

Hem de pensar que una CPU no treballa de manera lineal, sinó que fa petites pauses de processament per gestionar tasques secundàries. Si la CPU fes només una cosa a la vegada, ens trobaríem amb el problema que fins que un procés de càrrega, enregistrament, gestió o qualsevol altre no acabés, el nostre ordinador deixaria de respondre, no veuríem res a la pantalla, no es podria escriure al teclat, etc. La base de la computació des dels seus inicis és que les CPU fan petites pauses durant els seus processos per atendre altres peticions i amb això donar-nos la sensació que tot funciona a la vegada, però no és així. La CPU no fa moltes coses alhora, va canviant de tasca diverses vegades per segon per atendre a tothom i que tinguem la sensació que atén tots alhora.

Amb l'arribada dels processadors de múltiples fils i de múltiples nuclis, va ser quan els processadors van poder començar a fer tasques simultànies. I això és el que gestiona GCD: com gestionem o enviem tasques a fils diferents del principal (o fins i tot enviar tasques al fil principal estant en un fil secundari) per aconseguir ser més productiu i al seu torn, controlar com cada un d'aquests processos retorna (o no) possibles respostes.

4.8.1 Sèrie i Concurrència

Podem executar tasques enviades a la cua en sèrie o simultàniament [15]. De manera predeterminada, les tasques enviades a les cues s'executen en sèrie. En el mode *serial*, la tasca de la cua no s'executarà fins que la tasca enviada prèviament a la cua finalitzi la seva execució.

Podem crear cues concurrents per a l'execució de les tasques. Hem de definir la propietat .attributes de la cua a *concurrent*. En mode concurrent, les tasques de la cua s'envien una darrere l'altra, s'inicia l'execució immediatament i les tasques completen la seva execució en qualsevol ordre i, per tant, no es pot garantir l'ordre d'execució.

4.8.2 Quality of Service (QoS)

Hi ha algunes tasques que són més importants que d'altres, així que hem d'assegurar-nos que primer s'executaran les tasques més prioritàries [15]. Podem fer-ho assignant prioritats a les tasques que ho requereixin. Les tasques que s'executen en el fil principal i en l'Interfície d'Usuari sempre són d'una gran prioritat, mentre que les tasques que s'executen en el fil de fons són de menys prioritat. En definitiva, totes les tasques finalitzen la seva execució, però la prioritat decideix quina serà completada primera.

Hi ha una llista de prioritats que permet assignar la importància desitjada a certes tasques. Aquestes prioritats, ordenades de més a menys, queden de la manera següent:

- .userInitiated
- .default
- .utility
- .background
- .unspecified

Si no s'especifica la prioritat a la cua, s'assignarà per defecte la prioritat *default*.

4.8.3 Dispatch Queues

Quan utilitzem GCD, podem executar blocs de codi en diverses cues, que poden ser cues que hem creat nosaltres mateixos o les que el sistema ens proporciona [17].

Una de les cues que ofereix iOS és la *global dispatch queue* per executar les tasques en un *thread* de fons. Podem utilitzar-la si volem executar una tasca en segon pla com descarregar un fitxer, carregar dades o realitzar la cerca entre d'altres. També ens proporciona la *main dispatch queue* per executar les tasques en el *thread* principal / UI (Interfície d'usuari). No s'ha de realitzar operacions relacionades amb la UI en cues que no siguin la cua principal, en cas contrari el compilador mostrarà un avís. Necessitem la cua principal per realitzar operacions de UI.

També podem crear la nostra pròpia cua per a l'execució de tasques mitjançant la crida `DispatchQueue()`. Hem de proporcionar una etiqueta única per a la nostra cua per tal d'identificar-la. Per executar una tasca, s'ha d'assignar el mètode en què es vol realitzar l'execució d'aquella cua. S'assigna mitjançant els mètodes `.async` i `.sync`.

El mètode `.async` i `.sync` de la DispatchQueue indica al sistema com executar la tasca. Si s'escull el mètode `.sync`, l'execució de la tasca de la DispatchQueue no acabarà fins que s'hagi completat l'execució de la tasca sencera. Per tant, el codi escrit posterior, no s'executarà fins que no s'hagi completat l'execució de la tasca completament.

D'altra banda, el mètode `.async` torna de l'execució de la tasca immediatament després de l'enviament. Per tant, el codi escrit després de la tasca s'executarà, no s'espera al fet que finalitzi la seva execució.

4.8.4 Dispatch Groups

En molts casos, es necessiten executar diverses tasques llargues al llarg de l'execució. Per defecte, es poden executar aquestes tasques de manera seqüencial, és a dir, una tasca no comença fins que acaba la tasca anterior. Com es pot observar, no és un mètode molt eficient i, aquí és on entra DispatchGroup [18]. DispatchGroup ens permet que les tasques s'executin simultàniament, és a dir, les tasques no han d'esperar al fet que acabi la tasca anterior per poder començar a executar-se i, per tant, aconseguint reduir el temps total d'execució.

Per crear-ne un es fa mitjançant la crida `DispatchGroup()`. Un cop creat, s'utilitzen les crides `enter()` i `leave()` per executar les tasques de forma asíncrona. Aquestes crides es realitzen per cada tasca que es vulgui posar en el DispatchGroup.

4.8.5 Dispatch Work Item

DispatchWorkItem és una encapsulació que permet adjuntar un control complet o dependències d'execució [15]. En lloc d'escriure un bloc de codi, es pot crear un element de treball per a l'execució. S'executa DispatchWorkItem fent una crida al mètode `.perform()`. El mètode `.perform()` executarà l'element de treball al fil actual.

En general, les funcions Dispatch poden prendre com a paràmetre un bloc o un DispatchWorkItem. Es pot fer servir la que més convingui.

4.8.6 Execució retardada

En alguns casos, és necessari retardar l'execució del codi [15]. GCD ens permet fer-ho mitjançant un mètode especial anomenat `.asyncAfter`. S'ha d'establir la quantitat de temps per la qual es vol retardar l'execució. Es pot especificar el retard en segons, mil·lisegons, microsegons i nanosegons.

4.9 Persistencia de les dades

Core Data és un *framework* de persistència desenvolupat per Apple que ens permet simplificar la gestió del model de dades de les nostres aplicacions. Primer de tot hem de veure en què consisteix la persistencia [19].

La persistència és l'acció de preservar la informació d'un objecte de forma permanent (guardar-ho), unit a la possibilitat de poder recuperar la informació del mateix (llegir-ho) perquè pugui ser novament utilitzat.

Dit d'una altra manera, la persistència ens permet guardar les dades de les nostres aplicacions per a poder utilitzar-los quan l'usuari torni a executar la nostra app. Si desenvolupem una aplicació que no integri cap sistema de persistència, totes les dades que anem utilitzant durant l'execució de la mateixa simplement s'escriuran en memòria. Això vol dir que en estar únicament emmagatzemats en memòria i no en disc, en la següent execució de l'aplicació, totes aquestes dades es perdran.

En desenvolupament iOS hi ha diversos sistemes de persistència, diferents "eines" que podem utilitzar com a desenvolupadors per guardar les dades de les nostres aplicacions. Alguns d'ells són els següents:

- NSUserDefaults
- Property Lists
- NSFileManager
- SQLite
- Core Data

4.9.1 NSUserDefaults

Aquest mètode és molt útil quan volem emmagatzemar petites quantitats de dades com puntuacions, informació d'usuari, l'estat de la nostra aplicació, configuracions, etc.

És el mètode més ràpid d'utilitzar, ja que no requereix una base de dades, ni de fitxers, ni cap altra capa intermèdia. Les dades s'emmagatzemen directament en la

memòria del dispositiu dins d'un objecte de la classe NSUserDefaults. Cada valor que es guardi en NSUserDefaults ha d'estar associat amb alguna clau, a través de la qual es pot recuperar el mateix.

NSUserDefaults té una propietat de classe anomenada estàndard que retorna una instància de NSUserDefaults amb una configuració per defecte per emmagatzemar dades relacionades amb l'aplicació actual. Això és el que vols en la majoria dels casos.

NSUserDefaults pot emmagatzemar els tipus de dades següents:

- String
- Data
- NSNumber
- Date
- Array
- Dictionary

Qualsevol classe de dades que es vulgui guardar en la nostra aplicació i no estigui a la llista, no es podrà realitzar mitjançant NSUserDefaults. S'haurà de buscar una "eina" diferent per a guardar aquestes dades de forma persistent.

4.9.2 SQLite

SQLite és una biblioteca que implementa una base de dades relacional incrustada lleugera. Com el seu nom indica, està basat en l'estàndard SQL (S*tructured Query Language*) igual que MySQL i PostgreSQL.

La principal diferència amb altres bases de dades SQL és que SQLite és portàtil i molt lleuger. En lloc d'un procés separat al què s'accedeix des de l'aplicació client, SQLite no té servidor. En altres paraules, està integrat en l'aplicació o administrat pel sistema en què s'executa l'aplicació, el que significa que és molt ràpid.

Sens dubte és una opció freqüent per a l'emmagatzematge de dades del costat del client. L'avantatge de SQLite sobre treballar directament amb objectes és que

SQLite és molt més ràpid. Això es deu en gran part a com les bases de dades relacionals i els llenguatges de programació orientats a objectes difereixen fonamentalment.

Per acabar de funcionar de forma conjunta correctament SQLite i Objective-C, s'han creat diverses solucions de mapeig relacional d'objectes (ORM) al llarg del temps. L'ORM (*Object-relational Mapping*) que Apple ha creat per iOS i OS X es diu Core Data.

4.9.3 Core Data

Com ja hem comentat, Core Data és un *framework* de persistència desenvolupat per Apple que ens permet simplificar la gestió del model de dades de les nostres aplicacions. L'avantatge d'utilitzar Core Data és que treballa amb objectes en lloc de dades sense processar, és a dir, emmagatzema i administra dades en una interfície orientada a objectes. Però el valor que realment ens aporta, són una sèrie d'eines que ens faciliten tant la creació del model de la nostra app com la gestió posterior des del nostre codi. Les característiques principals inclouen el filtratge, la consulta, la classificació, la persistència de dades i la creació de relacions entre les dades.

Una altra gran característica és l'editor de models Core Data, integrat en Xcode. L'editor permet als desenvolupadors dissenyar el model de dades de l'aplicació a través d'una interfície gràfica.

Core Data sol utilitzar-se a través d'una base de dades de tipus SQLite. A més, aquesta capa que envolta la nostra base de dades SQLite ens permet que nosaltres com a desenvolupadors no haguem de treballar directament amb sentències SQL.

Trobem una serie de conceptes que són fonamentals a l'hora d'utilitzar el *framework* de persistència Core Data.

- **NSManagedObjectModel:** és la representació del nostre model en disc.

- **NSManagedObject:** és un objecte que representa un objecte únic emmagatzemat en Core Data.

- **NSManagedObjectContext:** representa una mena d'"espai de memòria temporal" on poder treballar abans de guardar les dades. Si penses en com guardar un objecte amb Core Data, podríem dir que es tracta d'un procés de dos passos. Primer insereixes l'objecte en el *managed object context* i una vegada que estàs segur pots confirmar el guardat de l'objecte emmagatzemant-lo en disc. Xcode genera automàticament un *managed object context* a les nostres aplicacions sempre que activem l'opció *Use Core Data* en crear el projecte. Concretament el *managed object context* s'emmagatzema en una propietat del *appDelegate*, de manera que quan necessitis utilitzar-lo, el primer que hauràs de fer serà crear una referència al *appDelegate* de la teva aplicació.

- **NSEntityDescription:** és un objecte que descriu una Entitat en Core Data. Una instància de NSEntityDescription determina el nom de l'entitat, els seus atributs i relacions i la classe per la qual està representada.

- **NSFetchRequest:** és la classe responsable de recuperar dades de Core Data. Per recuperar aquestes dades, utilitzarem peticions a les quals especificarem una sèrie de criteris. Aquestes peticions són bastant potents. Pots utilitzar fetchRequest per recuperar un conjunt d'objectes que compleixin unes determinades condicions. NSFetchRequest utilitza qualificadors per filtrar els resultats que es volen obtenir.

Una regla important que s'ha de complir quan treballem amb Core Data és la de no compartir contextos d'objectes gestionats entre els *threads*. Core Data no és segur utilitzar-lo en subprocessos, ja que espera ser executat en un sol *thread*.

4.10 *Sockets*

Un *socket* és un punt final d'un enllaç de comunicació bidireccional entre dos programes que s'executen a la xarxa [5]. Un *socket* està lligat a un número de port

de manera que la capa TCP pugui identificar l'aplicació a la qual es destinaran les dades.

Normalment, un servidor s'executa en un ordinador específic i té un *socket* que està lligat a un número de port específic. Per realitzar una connexió entre un client i un servidor, el servidor només espera, escoltant el *socket* a què un client faci una sol·licitud de connexió.

El client coneix el *hostname* de la màquina en què s'executa el servidor i el número de port on el servidor està escoltant. Per fer una sol·licitud de connexió, el client intenta reunir-se amb el servidor en la màquina i el port del servidor. El client també ha d'identificar-se al servidor de manera que es vinculi al client a un número de port local que s'utilitzarà durant la connexió. Normalment és assignat pel sistema.

Si tot va bé, el servidor accepta la connexió. Un cop acceptada, el servidor obté un nou *socket* lligat al mateix port local i també té el punt final remot configurat a l'adreça i al port del client. Es necessita un *socket* nou de manera que pugui seguir escoltant el *socket* original per a sol·licituds de connexió mentre està ocupat amb les necessitats del client connectat.

Al costat del client, si s'accepta la connexió, es crea correctament un *socket* i el client pot utilitzar-lo per comunicar-se amb el servidor. El client i el servidor poden comunicar-se escrivint o llegint des dels seus *sockets*.

4.11 Network.framework

Network.framework és una alternativa moderna als *sockets* [5]. Els *sockets* existeixen des de fa més de trenta anys i han estat molt útils però, el món va canviant i fa que vagin apareixent millors alternatives com és el cas del Network.framework. Avui en dia, és difícil utilitzar els *sockets* per escriure aplicacions per a Internet. Algun dels problemes de treballar amb sockets poden ser al moment d'establir connexions, en la transferència de dades, en la mobilitat dels propis sockets, etc. En el passat tot era molt més senzill però, avui en dia, els dispositius mòbils són increïblement potents i, alguns d'ells, es van movent de xarxa

en xarxa obligant a les aplicacions que dominin bé totes aquestes transicions per proporcionar una experiència perfecta als clients. Amb *sockets* és massa complex aconseguir-ho.

Amb Network.framework, Apple ha volgut assegurar-se que corregia totes les dificultats que provoca treballar amb *sockets*. Network.framework té un establiment de connexió increïblement intel·ligent, gestiona xarxes IPv6, *proxies,* té una ruta de transferència de dades increïblement optimitzada que permet anar més enllà del rendiment de qualsevol cosa que es pugui fer amb els sockets, té suport a la mobilitat i admet de manera predeterminada TLS (*Transport Layer Security*) i DTLS (*Datagram Transport Layer Security*).

A més de totes aquestes millores, Network.framework està elaborat de manera que sigui molt més clar i senzill, fent que extenses línies de codi treballant amb *sockets*, es redueixin a crides molt més senzilles.

5 De MPI a iOS

Ja hem vist que MPI no ha estat una opció vàlida a causa de l'obligatorietat de fer un *jailbreak* al dispositiu iOS perquè funcioni MPI correctament. Per tant, es vol comentar les principals diferències que suposa treballar amb MPI i amb les eines que ens ofereix Apple, concretament amb Network. D'aquesta manera, qualsevol programador que vulgui desenvolupar aplicacions aplicant tècniques de paral·lelisme en dispositius iOS i tingui força coneixements sobre MPI, pugui veure les similituds i les diferències entre les dues opcions. Hem de partir del fet que el *framework* Network no substitueix MPI en cap cas, sinó que és una alternativa que pot ser força interessant. A més, com no intenta substituir MPI hem de tenir en compte que pot haver-hi certes funcions de MPI que no estiguin disponibles o tinguin funcionalitats diferents en el *framework* Network i viceversa.

A diferència de MPI que requereix instal·lar-se en el dispositiu, Network és un *framework* presentat per Apple i que pots treballar amb ell de forma molt senzilla dins l'entorn de Xcode, facilitant la feina del desenvolupador.

Per treballar amb Network, es necessita importar la llibreria Network perquè posteriorment puguem utilitzar totes les funcions de les quals disposa. Un cop fet, no necessitem iniciar i finalitzar un entorn d'execució com es fa amb MPI amb les funcions `MPI_Init()` i `MPI_Finalize()` [20]. A partir d'aquí, ja es pot començar amb la comparació entre establir una connexió amb MPI o Network.

Network permet fer connexions entre diferents aplicacions connectades al mateix port. Per aconseguir establir connexió es requereixen els mateixos passos que si treballéssim amb MPI. Per una banda un *listener* que escolti peticions, i per l'altre les connexions que sol·liciten permís per connectar-se al port del *listener*. Network incorpora `NWListener` i `NWConnection` que agilitzen molt la creació de les dues bandes. `NWListener` és un objecte que s'utilitza per escoltar les connexions de xarxa entrants i funciona de forma similar a la funció `MPI_Comm_accept()` de MPI, amb la lleugera diferència que per preparar el `NWListener` per acceptar peticions en Swift, primer hem d'inicialitzar el `NWListener` i configurar una sèrie de paràmetres per tal que quedi preparat per acceptar les peticions, suposant un treball extra respecte si treballéssim amb MPI. Amb `NWConnection` passa exactament el mateix, s'ha d'inicialitzar a part de configurar una sèrie de paràmetres perquè es pugui connectar correctament. `NWConnection` permet una connexió de dades bidireccional entre un punt final local i un punt final remot i s'aproxima a la finalitat de la funció `MPI_Comm_connect()` de MPI. La principal diferència és que `NWConnection` permet una connexió bidireccional, per tant, requereix que s'estableixi aquesta connexió en les dues bandes. És a dir, en la banda del `NWListener`, que està escoltant un port concret, a l'arribar la petició de connexió ha de generar una `NWConnection` per tal de connectar-se a la connexió que acaba d'acceptar, en canvi, en MPI només crides la funció `MPI_Comm_accept()` per acceptar les peticions entrants i la resta es fa automàticament.

Un cop la connexió està establerta, toca estructurar l'enviament i rebuda de missatges. Tant l'enviament com la rebuda de missatges segueix una estructura similar a les comparacions anteriors. Per enviar i rebre missatges existeixen dues funcions però requereixen preparar-les bé perquè funcioni tot correctament i no es perdin els missatges. Per enviar un missatge tenim una funció dins `NWConnection`

anomenada `send()`. Aquesta funció ens permet l'enviament de dades on és necessari fer prèviament una conversió del missatge a enviar al tipus "Data". Si comparem amb MPI, trobem una funció similar anomenada `MPI_Send`. En MPI, hi ha altres funcions d'enviament com pot ser `MPI_Isend()`, `MPI_Bsend()`, `MPI_Ssend()`, etc. La majoria d'aquestes funcions es poden fer a partir de la funció `send()` però, afegint el codi necessari per què es diferenciïn d'un enviament normal amb `MPI_Send()`. Per rebre missatges trobem dues funcions diferents que segueixen l'estructura de la funció `send()`. Aquestes funcions són `receive()` i `receiveMessage()`. En la funció `receive()` s'ha d'indicar el màxim nombre de bytes que pot rebre, mentre que `receiveMessage()` rep el missatge complet. Aquestes dues funcions reben les dades com a tipus "Data", per tant, es requereix un posterior tractament de les dades pel receptor abans de poder treballar amb elles. Estem parlant de dues funcions que podrien equivaldre a `MPI_Recv()` i, com ja hem comentat abans, si es vol fer una rebuda de missatges de l'estil `MPI_Irecv()` s'ha de configurar prèviament.

Dins `NWConnection` existeix la funció `cancel()` que s'encarrega de cancel·lar la connexió entre les dues bandes, que equivaldria a `MPI_Comm_disconnect()` utilitzant MPI.

6 Implementació

Per realitzar la part més pràctica del projecte, s'ha realitzat una sèrie d'aplicacions basades en la multiplicació de matrius [21], amb l'objectiu principal d'aconseguir aplicar computació paral·lela en les mateixes aplicacions, mitjançant MPI o les eines que ens facilita Apple.

6.1 Multiplicació seqüencial

La primera de les aplicacions consisteix en una multiplicació de matrius de forma seqüencial. Aquesta versió permet veure la multiplicació de dues matrius sense aplicar els conceptes de concurrència i paral·lelisme. Com es pot apreciar en el pseudocodi de la imatge, consisteix en tres bucles, cadascun dins de l'anterior de manera que es vagin fent les multiplicacions de les dues matrius i guardant els

valors en una nova matriu. A continuació es veu, una forma de realitzar aquesta multiplicació sequencial:

```
for i in 0 ..< Matriu1.count {
 for j in 0 ..< Matriu1[i].count {
  for k in 0 ..< Matriu2[i].count {
   resultant[i][j]+=Matriu1[i][k]*Matriu2[k][j]
  }
 }
}
```

Com és una execució de forma seqüencial, totes les versions que es facin, sigui aplicant concurrència o paral·lelisme, han de ser més eficients que aquesta versió, per tal de treure'n beneficis en el rendiment respecte a aquesta versió. Per altra banda, no valdrà la pena fer cap modificació o implementació nova d'aquesta versió.

6.2 Multiplicació concurrent

A continuació trobem l'aplicació d'una multiplicació de matrius de forma concurrent desenvolupada dins de l'entorn de Xcode amb el llenguatge Swift. És una aplicació molt simple de realitzar gràcies al llenguatge Swift amb l'ajuda de GCD.

L'objectiu d'aquesta aplicació és aplicar els conceptes de concurrencia amb les eines d'Apple, concretament de GCD. Aquesta aplicació pretén millorar la versió seqüencial amb l'ajuda de les cues de les que disposem a l'utilitzar GCD.

En l'aplicació genero les matrius. A diferència de voler treballar amb MPI, en iOS no es necessita incloure cap llibreria externa. La diferència principal amb el model MPI, és la simplicitat que comporta treballar amb Swift. En molts casos, Swift s'encarrega de les gestions posteriors a cridar certes funcions i, fa que els usuaris els hi sigui molt còmode i fàcil, a la vegada de la rapidesa per programar de forma concurrent.

El codi necessari per fer la multiplicació de les matrius de forma concurrent no varia gaire del que s'escriu en la versió seqüencial. En la versió concurrent, la diferència

només és en la forma de fer els bucles. M'he ajudat d'una DispatchQueue per cada bucle fent la crida `DispatchQueue.concurrentPerform()` on s'executa cada bucle de forma concurrent. En el moment de fer la crida, s'ha de determinar el nombre d'iteracions que tindrà el bucle. A continuació es mostra un extracte del codi on es realitza la multiplicació de dues matrius:

```
let group = DispatchGroup()
group.enter()
  DispatchQueue.main.async {
   DispatchQueue.concurrentPerform(iterations: Matriu1.count) { index in
    DispatchQueue.concurrentPerform(iterations: Matriu1.count) { index2 in
     DispatchQueue.concurrentPerform(iterations: Matriu2.count) { index3 in
      res[index][index2] += Matriu1[index][index3] * Matriu2[index3][index2]
     }
    }
   }
   group.leave()
  }
```

Per utilitzar aquesta funció, és molt important tenir en compte les iteracions que ha de realitzar de cada bucle, pel simple fet que si s'utilitza aquesta funció en bucles amb poques iteracions, es crearan més despeses generals executant els *threads* de forma concurrent que no pas a l'executar el bucle *for* de forma seqüencial. S'ha d'utilitzar `DispatchQueue.concurrentPerform()` per iterar conjunts molt grans per tal de treure benefici en el rendiment. En el cas de la multiplicació de matrius, com més gran siguin les dimensions de les matrius respectives, més positiu serà utilitzar l'execució de bucles de forma concurrent.

6.3 Múltiples Hosts

Per a realitzar la multiplicació de matrius amb múltiples hosts, he escollit l'algoritme basat en el paradigma mestre-esclau [22]. La computació paral·lela basada en el model mestre-esclau, consisteix a dividir problemes computacionals en subproblemes (problemes més petits) perquè es calculin en paral·lel per un o més processos. És a dir, el format d'aquest model es basa a repartir la feina del procés mestre (principal) a la resta de processos, que són els esclaus (secundaris).

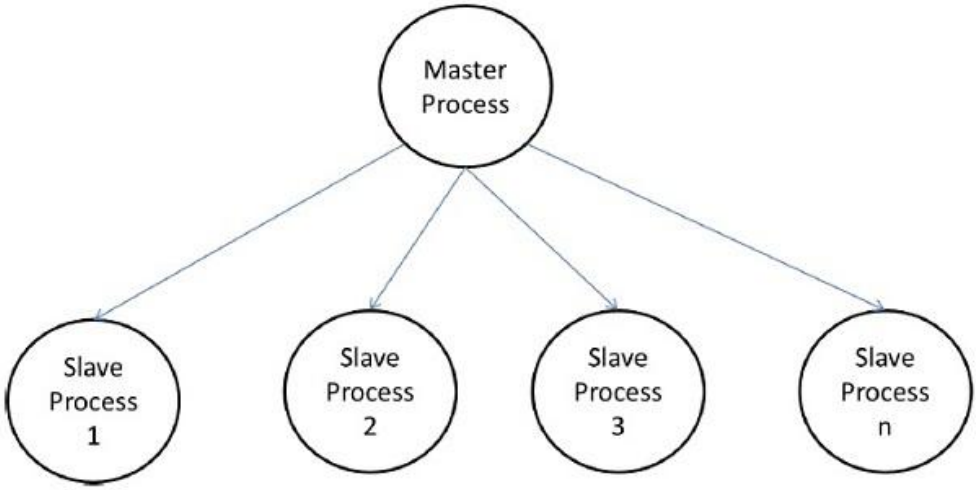

Figura 11: Model mestre-esclau

Les úniques interaccions entre el procés mestre i els processos esclaus, és quan el mestre inicia a tots els esclaus i, posteriorment, quan cada procés esclau retorna el resultat al mestre.

Per saber fins a on es pot aconseguir arribar amb les eines que ens dóna Apple, he agafat una aplicació que utilitza el model mestre-esclau en C utilitzant MPI. Aquest model s'explica a continuació per tal de poder veure posteriorment, les diferències amb el model que nosaltres hem implementat per a dispositius iOS.

6.3.1 Model MPI

La primera part, consisteix en una aplicació realitzada amb el llenguatge C, on s'introdueixen els conceptes previs de MPI amb algunes de les funcions esmentades que hagin estat necessàries per a realitzar la multiplicació de matrius aplicant els conceptes de paral·lelisme.

Per a realitzar la multiplicació de matrius amb MPI, primer de tot s'ha d'incloure la llibreria `<mpi.h>` per poder fer les crides corresponents. Per poder utilitzar MPI primer s'ha de fer la crida `MPI_Init()`, per inicialitzar l'entorn d'execució de MPI [21]. Un cop està inicialitzada, es fan dues crides necessàries per determinar les tasques. Primer es vol saber el nombre de processos dels quals es disposa, per tant, es fa la crida `MPI_comm_size()` que retorna el nombre concret. A més, s'ha

de poder diferenciar els processos per poder assignar-los tasques diferents. Per fer-ho, es necessita alguna informació addicional que s'obtindrà al fer la crida `MPI_comm_rank()` que retorna el rang del procés seleccionat.

A continuació s'ha de comprovar que el nombre de processos sigui com a mínim dos, ja que sinó ens serà impossible aplicar la paral·lelització i no es podrà seguir endavant amb l'execució del programa.

Un cop s'ha acabat la comprovació, es comença a preparar la multiplicació de les matrius. Per fer-ho, es divideix el codi en dues parts. Per diferenciar aquestes parts, servirà d'ajuda el rang dels processos que hem obtingut prèviament. El procés mestre (principal) executarà la primera part, on inicialitza la tasca principal i es prepara per dividir-se en subproblemes, mentre que els esclaus (la resta de processos) executaran la segona part de codi, on s'encarregaran de treballar pel procés mestre executen els subproblemes enviats pel mestre.

El procés mestre genera les dues matrius basant-se amb el nombre de files i columnes assignades. Un cop han estat les dues matrius generades, el procés mestre s'ha d'encarregar d'enviar la informació de les matrius a la resta de processos. Per enviar aquesta informació es fa mitjançant la crida `MPI_Send()`, tenint en compte les mides de la matriu, el repartiment de files per a cada procés, etc. Després de fer les crides `MPI_Send()` que hagin estat necessàries, farà falta fer la crida `MPI_Recv()` que és per rebre dades. En el cas del procés mestre, es fa aquesta crida per rebre la informació dels altres processos un cop hagin acabat la multiplicació per parts de les matrius i, es junta tota aquesta informació en el procés mestre on queda la matriu resultant de la multiplicació.

En la segona part del codi trobem els processos esclaus que esperen rebre la informació corresponent de les matrius per poder realitzar les multiplicacions, mitjançant la crida `MPI_Recv()`, que des del procés mestre s'envia per parts. Amb la informació rebuda, els processos executen la multiplicació de matrius i envien els resultats al procés mestre. Cada procés envia una part diferent, amb la crida

`MPI_Send()`, que finalment es junta en el procés mestre, on acaba sent la matriu resultant de la multiplicació.

Per acabar, es fa la crida `MPI_Finalize()` per finalitzar l'entorn d'execució de MPI. `MPI_Finalize()` ha de ser obligatòriament cridada per tots els processos.

6.3.2 Model iOS

Per altra banda trobem l'aplicació desenvolupada dins de l'entorn de Xcode amb el llenguatge Swift. En aquesta aplicació podrem veure les diferències que existeixen actualment entre desenvolupar una aplicació amb MPI o amb les eines legals que et dóna Apple. L'aplicació també realitza la multiplicació de matrius però amb les diferències que comporta que sigui destinada a un dispositiu Apple.

Com ja s'ha comentat anteriorment, en un dispositiu iOS no es pot aplicar cap mena de paral·lelisme, només concurrència mitjançant GCD entre d'altres. Per tant, si es té com a objectiu aconseguir fer una aplicació iOS amb paral·lelisme, ha de ser en més d'un dispositiu. Per aconseguir-ho, s'ha de desenvolupar una aplicació capaç d'enviar dades a través d'Internet entre els programes dels diferents dispositius, de manera que un dispositiu enviï les tasques a executar als altres dispositius disponibles i així alliberin de feina al programa principal. Les dues opcions que tinc per poder aconseguir-ho són mitjançant els *sockets* i amb el nou *framework* d'Apple anomenat Network.

En els últims anys, l'opció més lògica per aconseguir l'enviament de dades entre dues aplicacions eren els *sockets*. Però, en l'Apple WWDC (Conferència de desenvolupadors de tot el món) el 2018, Apple va presentar un nou *framework* anomenat Network com a alternativa moderna als sockets [5]. Fent balanç entre els avantatges i inconvenients de cada una de les alternatives, s'ha determinat que la millor opció per tractar l'enviament de dades és amb el *framework* Network. He tingut en compte que, tot i que és molt recent i no abunda la informació sobre el *framework*, l'objectiu que té és substituir els *sockets* i fer les connexions més simples. A més, Apple vol que la gent comenci a treballar a partir d'ara amb el nou *framework* que han desenvolupat en substitució dels *sockets*, de manera que a poc

a poc amb el pas del temps, Apple anirà facilitant el seu ús als usuaris o, complicant-los l'ús dels sockets. També s'ha de tenir en compte que els dispositius cada cop van sortint amb més potència que els anteriors i, si cada cop surt més a compte treure-li rendiment als dispositius, no interessa que es treballi amb eines més antigues havent-hi de noves que poden funcionar igual o millor.

Per fer la multiplicació de les matrius en més d'un dispositiu m'he basat en el model mestre-esclau amb MPI explicat en l'apart anterior, donant per fet que es treballa amb llenguatges diferents i amb eines totalment diferents. Tot i això, l'objectiu d'aquesta aplicació és aproximar-se al màxim possible del model anterior.

Per desenvolupar el model escollit, s'ha escrit el codi tot junt, i s'ha fet dues diferenciacions: servidor i client. Aquesta diferenciació està realitzada per diferenciar entre el procés principal o mestre (servidor) i els processos secundaris o esclaus (clients). Ho he fet d'aquesta manera, ja que l'aplicació està destinada a ser executada en diferents dispositius i s'ha de diferenciar les funcions que faran.

Endinsant-nos ja en el codi, a part de les classes del client i del servidor, he creat una classe extra, que es diu connexió i l'únic objectiu que té és d'establir tots els paràmetres d'una connexió per tal de facilitar la feina a l'hora d'escriure codi tant en el client com en el servidor.

El primer pas ha estat incloure el *framework* Network en l'aplicació. A partir d'aquí, els passos a seguir són molt similars a treballar amb *sockets* o amb MPI, amb la mateixa estructura a l'hora d'establir connexió, enviar dades, etc. A diferència de MPI no es necessita inicialitzar cap entorn d'execució, es pot treballar directament un cop Network estigui inclòs al projecte.

Per la banda del servidor o mestre, es vol escoltar les peticions de connexió entrants. Per fer-ho, he de crear un *listener* que s'ocupi d'aquesta tasca. El *listener* es crea mitjançant la funció `NWListener()`, on s'ha d'incloure una sèrie de paràmetres importants a l'hora d'establir la connexió amb el port. Entre aquests paràmetres a incloure, hi ha un molt important, ja que és per indicar el número de

port que es vol utilitzar per establir la connexió. També pots indicar el tipus de protocol que es vol utilitzar, en el meu cas he escollit el TCP. A continuació, s'ha d'acabar de configurar el *listener* correctament perquè funcioni. Per configurar-lo, el *listener* disposa d'una varietat de funcions, entre elles he de fer la crida `listener.stateUpdateHandler`. Aquesta opció ens permet rebre actualitzacions de l'estat en el qual es troba el *listener*. A continuació, veiem una forma d'implementar aquesta crida amb tots els estats que es poden configurar:

```
self.listener.stateUpdateHandler = self.stateDidChange(to:)

func stateDidChange(to newState: NWListener.State) {
            switch newState {
            case .setup:
                break
            case .waiting:
                break
            case .ready:
                print("Listener New State Ready")
                break
            case .failed(let error):
                print("server did fail, error: \(error)")
                self.stop()
            case .cancelled:
                break
            default:
                break
            }
        }
```

Un cop realitzada, toca utilitzar l'opció `listener.newConnectionHandler` que consisteix a desenvolupar un controlador que rep les connexions entrants. Amb aquest controlador, es podrà acceptar les connexions dels clients quan facin les seves sol·licituds. Finalment només farà falta fer la crida `listener.start()` que té la finalitat de començar a escoltar les peticions i, a més, estableix la cua on tots

els esdeveniments del *listener* són entregats. Aquesta crida es farà indicant la cua d'execució. En l'aplicació s'ha implementat de la manera següent:

- `self.listener.start(queue: .main)`

En la banda del client o esclau, es vol connectar al servidor o mestre per tal que enviï les dades necessàries per realitzar els càlculs. El primer pas ha de ser crear una nova connexió. La funció corresponent és `NWConnection()`, que té una sèrie de paràmetres importants que s'han d'indicar. Alguns ja els hem comentat en el servidor com el tipus de connexió i el port que han de ser iguals als que s'han establert en el servidor, ja que per fer una connexió es necessita tenir els dos dispositius connectats al mateix port i utilitzant el mateix protocol. A més, en el cas de crear una connexió, s'ha d'indicar en el paràmetre *host*, un nom o adreça que identifiqui un punt final de xarxa, estant disponible pel dispositiu en tot moment. Un exemple per fer la crida seria el següent:

- `let nwConnection = NWConnection(host: "192.168.1.39", port: 12548, using: .tcp)`

A continuació toca modificar el `connection.stateUpdateHandler`. Aquesta opció funciona igual que el `.stateUpdateHandler` del *listener* que s'ha comentat abans, i permet rebre actualitzacions de l'estat en el qual es troba la connexió, així es podrà determinar si el servidor accepta la connexió, si en un futur s'acaba la connexió, si surgeix qualsevol mena d'error, etc. A continuació s'ha de preparar la connexió per rebre dades. He creat la funció `setupReceiveServer()` que té també la funció de tractar les dades correctament a l'estar la connexió ja preparada per rebre les dades. S'aconsegueix gràcies a que fa una crida a `connection.receive()` de la manera següent:

- `self.nwConnection.receive(minimumIncompleteLength: 1, maximumLength: 65536)`

Dintre d'aquesta funció és on s'estableix correctament com es rebran les dades enviades per l'altra banda de la connexió.

Un cop es tenen tots els possibles estats de la connexió controlats, s'inicia la connexió i només falta que en l'altra banda s'accepti correctament.

Basant-nos en el model explicat, es vol reduir la feina del mestre. Per aconseguir-ho, he enviat les files de la primera matriu a diferents esclaus, junt amb la segona matriu sencera. El nombre de files que rep cada esclau de la primera matriu es calcula a partir del tamany de la matriu juntament amb el nombre d'esclaus. En l'aplicació es realitza la prova amb una matriu de dimensions 3x3 amb la connexió de tres esclaus, de manera que es reparteix una fila per esclau. A continuació veiem com s'ha enviat les dades des del mestre a cada esclau:

- `let data = Data(buffer: UnsafeBufferPointer(start: c1[i], count: c1[i].count))`
- `connection.send(data: Data(data))`

La crida `connection.send(data: Data(data))`, depèn de la funció de Network `send()`, que és l'encarregada d'enviar les dades correctament.

Com es pot deduir, cada esclau té la tasca de multiplicar la fila per la segona matriu i, posteriorment, retornar els resultats al mestre obtinguts a partir de la multiplicació. D'aquesta manera cada esclau multiplica una fila diferent de forma paral·lela i reduïm el temps que ens ocupa la multiplicació. Un cop acaba cada esclau de fer les multiplicacions, envien la fila obtinguda al mestre, que juntament amb les files obtingudes de la resta d'esclaus, obté la matriu resultant.

Aquesta versió serveix per demostrar que es pot assolir paral·lelisme amb dispositius iOS, tot i que d'una forma força diferent en comparació amb MPI. Tot i això, amb Swift també podem arribar una mica més lluny amb l'optimització. Aquesta aplicació conté tècniques de paral·lelisme però, sense aplicar conceptes de concurrència amb GCD, per tant, encara es pot optimitzar més.

6.4 Múltiples hosts amb GCD

L'última versió realitzada en el projecte és la multiplicació de matrius en múltiples hosts, aplicant els conceptes de concurrència en cada un d'ells. Per fer aquesta

aplicació, s'ha agafat com a model base l'aplicació anterior, on ja s'ha comentat que la multiplicació de matrius del mestre es dividia en subproblemes que executaven els esclaus. Tot i això, cadascun dels esclaus executava el subproblema de forma seqüencial. D'aquesta manera es pot observar que s'ha estat capaç de distribuir la feina en diferents dispositius mòbils però, en canvi, no s'utilitza tot el rendiment disponible que tenen els mateixos dispositius. Així, es vol fer una nova versió on cada esclau sigui capaç d'executar el seu subproblema de forma concurrent, concretament amb GCD que és una de les eines que ens ofereix Apple quan treballem amb Swift.

Aquesta versió es centra en modificar el codi de l'aplicació en la part dels esclaus, concretament en la funció on es multipliquen les matrius. Per passar de la multiplicació seqüencial a la concurrent, es parteix del codi escrit en l'aplicació feta prèviament, on s'ha realitzat la multiplicació de dues matrius de forma concurrent. Com ja s'ha explicat, s'utilitza GCD on es treballa principalment amb DispatchQueues i DispatchGroups. D'aquesta manera es pot gestionar la concurrència de les multiplicacions de les matrius, treballant de forma asíncrona però, sempre esperant al final de la multiplicació a què tota l'execució concurrent s'hagi acabat d'executar. Quan s'envia el resultat obtingut al mestre, s'ha d'haver acabat totes les execucions sinó, poden sorgir problemes de sincronització.

7 Anàlisi

Per realitzar les aplicacions amb múltiples hosts i amb múltiples hosts amb GCD, s'ha utilitzat el model mestre-esclau. L'aplicació de múltiples hosts amb GCD és la que es considera com a definitiva. Per desenvolupar-la s'ha necessitat quatre dispositius, un pel mestre i tres més pels esclaus. A l'hora de fer les comprovacions s'havia d'anar amb compte amb múltiples aspectes.

Per començar, trobem la connexió dels esclaus. L'aplicació ha d'estar preparada i ha de permetre que els esclaus estableixin la connexió amb el mestre. L'inconvenient és que no pots saber en quin ordre es realitzaran les connexions. Per aquest motiu, s'ha configurat l'aplicació de manera que no importi l'ordre de connexió dels esclaus i se'ls hi assigni un subproblema diferent. En l'aplicació

desenvolupada quan es parla dels subproblemes que el mestre delega als esclaus, es tracta de les diferents files en les que es divideix la matriu. S'assigna a cada esclau la fila o les files corresponents depenent de l'ordre en el qual han establert la connexió amb el mestre.

Un altre aspecte important a tenir en compte pel correcte funcionament de l'aplicació, és la sincronització entre els processos. És molt important que es respecti la sincronia i més, en una aplicació on hi ha processos que depenen d'altres. Dintre de la sincronització, és la part més òbvia però no menys important, també s'ha de mantenir una sincronització entre les connexions entrants. Com s'acaba de comentar, no importa l'ordre de les connexions entrants però, sí que s'estableixin. El procés mestre no pot començar a enviar dades si no té cap esclau que ha establert connexió. Ha d'esperar l'avís de les connexions perquè pugui procedir a enviar les dades. Seguint amb l'enviament de dades, els processos esclaus no poden començar a realitzar la multiplicació de matrius si no han rebut les dades del procés mestre, han d'esperar un avís que els permeti seguir endavant amb l'execució. De la mateixa manera, el procés mestre ha d'esperar les dades dels esclaus, on rep les files de la matriu ja multiplicades i, finalment les ajunta en la matriu resultant. Si no s'espera al fet que arribi la informació de tots els esclaus, el procés mestre pot obtenir una matriu que li faltin files.

A part de la sincronització en l'enviament de dades, també es treballa amb GCD que és una eina d'Apple que permet aplicar concurrència. En aplicar la concurrència s'ha de tenir en compte que podem tenir vàries execucions de fons que no hagin acabat. És el cas de la multiplicació de les matrius, on s'executa tota la multiplicació de forma concurrent i, un cop acabada, es vol enviar les dades resultants al procés mestre. Abans d'enviar les dades s'ha de comprovar que ha acabat totes les execucions concurrents i que no només ha acabat alguna d'elles. En aquest cas, s'està treballant amb una eina d'Apple com és GCD, que disposa de vàries maneres de controlar-ho, concretament mitjançant DispatchGroup. DispatchGroup té les funcions `enter()` i `leave()` [15] que serveixen per mantenir sincronitzades les tasques. A més, la funció `notify()`, permet estar notificat en tot moment de quan comencen i acaben les tasques. Amb l'ajuda d'aquestes tres funcions, s'ha utilitzat

un DispatchGroup per controlar l'execució de les tasques i assegurar-nos que en cap moment s'enviïn les dades abans d'acabar la multiplicació.

8 Treballs relacionats

He trobat alguns estudis relacionats amb el projecte a desenvolupar que ara comentaré breument:

El primer de tots, anomenat *DroidCluster: Towards Smartphone Cluster Computing* [23], tracta d'una prova de concepte en dispositius Android. En aquest estudi monten un cluster amb 6 dispositius Android. Per avaluar el *software*, van escollir executar Linpack, ja que és un *benchmark* estàndard per als sistemes HPC. Els *benchmarks* de Linpack són un sistema de mesura de la potència de càlcul de punts flotants d'un sistema i mesuren amb quina rapidesa un ordinador resol un sistema d'equacions lineals determinat. Per a realitzar aquesta prova de concepte, no es volia eliminar el sistema operatiu d'Android, de manera que es va crear una instal·lació de Debian ARM (*Advanced RISC Machines*) en una carpeta d'una targeta SD mitjançant debootstrap (Debootstrap és una eina que instala un sistema base de Debian en un subdirectori d'un altre sistema ja instal·lat) i, així, no interferint amb el *software* del dispositiu Android. En el sistema Debian es va instal·lar les llibreries de MPI i HPL (High Performance Linpack) per poder executar els *benchmarks* de Linpack i, finalment es va instal·lar un servidor SSH per configurar els dispositius Android a través de la xarxa, de forma més senzilla. Per realitzar els tests, van utilitzar tant Wi-Fi com cables USB connectats directament als dispositius Android, i van fer les proves amb diversos dispositius mòbils. Amb els resultats que obtenen, conclouen que poden utilitzar dispositius Android per distribuir certes tasques computacionals, sempre amb l'inconvenient de la limitació de la bateria dels dispositius mòbils.

A continuació trobem l'estudi anomenat *MMPI a Message Passing Interface for the Mobile Environment* [24], és molt interessant, ja que van utilitzar la llibreria MMPI (*Mobile Message Passing Interface*) junt amb la tecnologia Bluetooth per enviar i rebre missatges entre ordinadors, dispositius mòbils i PDAs (Personal Digital Assitant). Aquest estudi és bastant antic però, és útil veure utilitzar una llibreria

diferent de MPI a la que s'havia pensat utilitzar, ja que pot servir d'ajuda a l'hora de fer el millor plantejament.

Un altre estudi anomenat *High Performance Computing Over Parallel Mobile Systems* [25] va analitzar els estudis *DroidCluster: Towards Smartphone Cluster Computing* [23] i *MMPI a Message Passing Interface for the Mobile Environment* [24], entre d'altres, i van fer una prova de concepte en dispositius Android basant-se en aquests estudis. Per a realitzar l'estudi es van ajudar d'una llibreria de C anomenada Bionic. Aquest estudi és molt interessant, ja que està explicat pas a pas tot el procediment que van realitzar per muntar el cluster de dispositius Android. Com a punt final, van creure que seria possible utilitzar clusters de dispositius mòbils en un futur pròxim.

Finalment he trobat altres estudis que tenen algunes similituds i em poden servir d'idees, però es desvien bastant a l'hora de l'objectiu principal del projecte o del *software* i *hardware* utilitzat. Tot i això, m'he fixat en aquests estudis, ja que utilitzen llibreries de MPI en clusters de dispositius.

El més interessant d'aquests estudis, es diu *The Computational Performance and Power Consumption of the Parallel FDTD on a Smartphone Platform* [26], i van fer una prova de concepte basant-se en la viabilitat de l'ús d'un dispositiu Android per a la computació d'alt rendiment (HPC) per a diverses disciplines científiques, en el context del modelatge electromagnètic utilitzant el mètode FDTD (*Finite-difference time-domain method*). El mètode FDTD és una tècnica d'anàlisi numèrica per a la modelització de l'electrodinàmica computacional que consisteix a trobar solucions aproximades a un sistema associat d'equacions diferencials. En aquest estudi van fer una comparació de l'eficiència energètica del FDTD paral·lel en diferents plataformes. El que ens interessa d'aquest estudi, és com ho van fer en els dispositius Android i com aprofiten les característiques dels mateixos dispositius. Com a conclusió de l'estudi, el FDTD es pot desplegar fàcilment en paral·lel en un dispositiu Android i s'utilitza per a un modelatge electromagnètic rudimentari a petita escala, tot i que els resultats no són bons si es fan a gran escala. La conclusió no és del tot rellevant pel meu estudi, ja que utilitzen *hardware* i *software* que no s'utilitzarà

però, sí que em pot servir per tenir un punt de partida de la potència i de l'abast dels dispositius mòbils actuals.

També trobem un altre experiment, anomenat *Building a large low-cost computer cluster with unmodified Xboxes* [27], on van utilitzar les llibreries de MPI per muntar un cluster de videoconsoles Xbox, a diferència dels altres estudis que he esmentat. Aquest estudi el menciono, ja que treballa amb les llibreries de MPI com he comentat anteriorment i, sobretot, perquè va servir d'ajuda per a realitzar l'estudi *Towards Energy Efficient Parallel Computing on Consumer Electronic Devices* [2] que he explicat en l'apartat de l'Estat de l'art i m'ha servit de punt de partida del meu projecte. En aquest estudi fan una prova de concepte amb videoconsoles Xbox a l'avaluar que amb Plays Stations sortiria més car, ja que es necessita *software* addicional. D'aquest estudi, van treure que és factible muntar un cluster de Xbox amb uns bons resultats obtinguts en les proves de rendiments i, a més a més, força interessant de cara al futur, com a conseqüència del baix preu que tenen les Xbox.

9 Planificació final

9.1 Calendari

A continuació explicaré justificadament els canvis que ha patit el calendari del projecte respecte a la planificació que vaig plantejar inicialment.

La fase que ha patit les modificacions més significatives ha estat la de desenvolupament del projecte.

- Adquirir coneixements iOS
- Estructurar l'aplicació
- Desenvolupament de l'aplicació
- Tests de rendiment Mac OS X
- Tests de rendiment iOS
- Anàlisi de resultats

Adquirir coneixements iOS és la tasca que ha tingut més canvis significatius. El motiu principal ha estat la gran quantitat d'informació juntament amb la suma

d'inconvenients que m'he trobat. Per fer correctament el projecte, necessitava assegurar-me de totes les opcions possibles i triar la millor possible. La majoria de les opcions han estat comentades durant la memòria. L'exemple més clar és el *jailbreak*, on es va destinar un número bastant elevat d'hores a veure si era una solució factible que m'ajudes en el projecte però, finalment es va descartar. Aquest motiu entre d'altres, em va fer descartar l'opció d'incloure MPI en l'aplicació iOS i, per tant, dedicar-li força temps a adquirir els coneixements necessaris per treballar amb sockets o el *framework* Network.

La tasca d'estructurar l'aplicació ha vingut condicionada de la tasca d'adquirir coneixements iOS. La meva idea era utilitzar sockets en el cas que MPI no fos possible però, el *framework* Network, que és molt recent i va costar hores descobrir-lo i trobar informació corresponent, va passar a ser una nova opció per desenvolupar l'aplicació.

A partir d'aquí, la tasca de desenvolupament de l'aplicació s'ha modificat lleugerament. Com he disposat de més temps, he decidit fer dues aplicacions diferents, necessitant algunes hores més pel desenvolupament de les que tenia previstes en el primer moment.

Vist el projecte, he afegit una tasca que considero essencial. Consisteix en una comparació entre MPI i el *framework* Network, perquè d'aquesta manera tots els programadors de MPI puguin veure com programar amb iOS amb Network i facilitar d'alguna manera el traspàs d'un a l'altre. A més, que serveixi com una ajuda per poder veure les similituds i les diferències entre les dues formes de programar, entenent que Network és un *framework* molt recent i poc conegut, requerint algunes explicacions molt ben detallades pels nous programadors d'aquest *framework*.

A partir de tota la informació obtinguda de la tasca anterior i del desenvolupament de l'aplicació es vol fer una sèrie de proves per tal de veure la funcionalitat d'aquesta aplicació que he dissenyat. Les tasques de rendiment Mac OS X i iOS s'han juntat en una sola tasca que li he dit Proves. La finalitat d'aquestes tasques és la comparació entre l'execució de MPI en MacOS X i l'aplicació en iOS i, crec que té

més sentit si faig les dues tasques conjuntament, ja que m'ajudarà a veure els diferents comportaments tant de MPI com de l'aplicació.

La tasca d'anàlisi de resultats no s'ha modificat, simplement s'analitzaran els resultats obtinguts durant el desenvolupament del projecte i, s'aprofundirà sobretot en els resultats de la tasca anterior.

Tot i que en la planificació inicial estava planejada el desenvolupament de la memòria durant el projecte, no hi havia una fase explícita que ho indiqués sinó que cada tasca implicava unes hores a desenvolupar la memòria. Té més sentit mostrar el desenvolupament de la memòria de forma clara en el Gantt afegint una nova fase al projecte, tot i que a efectes pràctics és el mateix.

A la taula següent podem veure el nombre d'hores que hem realitzat finalment:

Tasca	Temps Estimat (hores)
Viabilitat del projecte	20
Planificació del projecte	65
Configuració inicial del sistema	50
Adquirir coneixement iOS	180
Estructurar l'aplicació	95
Desenvolupament aplicació	200
Comparació MPI/iOS	30
Proves	35
Anàlisi de resultats	40
Memòria del projecte	100
Finalització del projecte	15
Total	**830**

Taula 6: Temps estimat per a cada tasca definitiu

A continuació es mostra el diagrama de Gantt de les tasques finals, incloent-hi totes les modificacions que acabo de comentar.

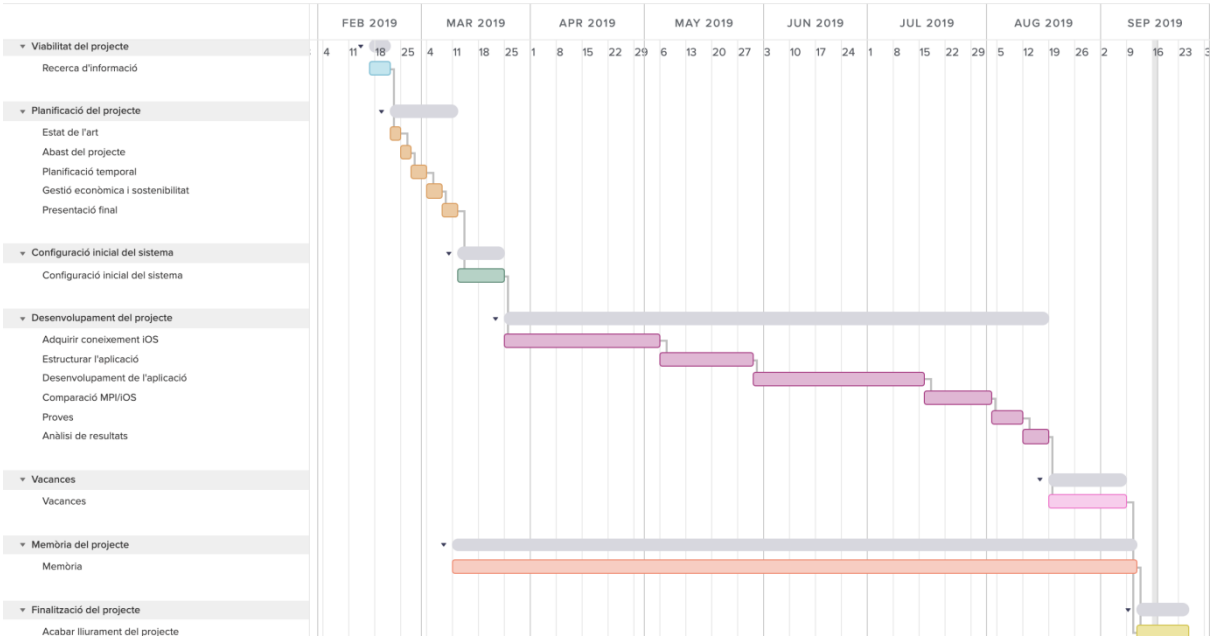

Figura 12: Diagrama de Gantt definitiu

9.2 Gestió econòmica

Un cop finalitzat el projecte, ja es pot calcular quin ha estat el cost real.

9.2.1 Recursos *hardware*

A continuació es mostren els costs de tots els *hardwares* que s'han necessitat pel projecte. Com es pot veure en la taula, s'ha eliminat el cost de l'iPhone. El motiu són els simuladors iOS. Aquests simuladors han servit per a realitzar totes les proves necessàries per comprovar que les aplicacions funcionen correctament en múltiples versions de iOS.

Producte	Preu	Vida útil	Amortització
MacBook PRO 13"	1800.00€	5 anys	120€
Total	**1800.00€**		**120€**

Taula 7: Costos de recursos *hardware* definitius

9.2.2 Recursos *software*

A continuació es mostren els costs de tots els *softwares* que s'han utilitzat per desenvolupar el projecte. Només trobem dues diferències respecte al plantejament inicial. La primera diferència és que s'ha concretat la utilització de MPICH i Open-MPI, ja que al principi encara no es sabia quina o quines implementacions de MPI s'utilitzarien. L'altra diferència es troba en el fet que s'ha afegit la utilització d'un simulador de iOS. El simulador que s'ha utilitzat finalment es troba dins de l'entorn de Xcode però, com és un dels recursos més importants del projecte, el mencionem a part.

Producte	Preu	Vida útil	Amortització
Mac OS X	0.00€	-	0.00€
Github	0.00€	-	0.00€
Sublime Text	0.00€	-	0.00€
MPICH	0.00€	-	0.00€
Open-MPI	0.00€	-	0.00€
Xcode	0.00€	-	0.00€
Simulador iOS	0.00€	-	0.00€
Google Docs	0.00€	-	0.00€
TeamGantt	0.00€	-	0.00€
Total	**0.00€**		**0.00€**

Taula 8: Costos de recursos *software* definitius

9.2.3 Recursos humans

A continuació es mostren els costs dels recursos humans que s'han necessitat pel desenvolupament del projecte.

Rol	Preu per hora	Temps	Cost
Director del projecte	40.00€	120 h	4800.00€
Dissenyador	25.00€	240 h	6000.00€

Programador	30.00€	300 h	9000.00€
Tester	20.00€	140 h	2800.00€
Total	**/**	**800 h**	**22600.00€**

Taula 9: Costos de recursos humans definitius

9.2.4 Pressupost total

A continuació es mostra el cost total del projecte un cop finalitzat, sumant els costos obtinguts en tots els apartats anteriors. El cost en *hardware* s'ha vist reduït mínimament però, a l'augmentar el nombre d'hores dedicades al projecte, incrementa força el cost dels recursos humans. El cost dels recursos *software* no s'ha modificat, ja que totes les modificacions fetes han estat de la utilització de *software* gratuït.

Recursos	Cost
Hardware	120.00€
Software	0.00€
Humans	22600.00€
Total	**22720.00€**

Taula 10: Pressupost final definitiu

Com es pot veure, el cost final és superior al cost estimat inicialment. Aquest increment és la conseqüència de l'augment del cost dels recursos humans.

10 Lleis i regulacions

Aquest projecte presenta una forma d'aplicar tècniques de paral·lelisme en dispositius iOS mitjançant dues aplicacions que he desenvolupat. Aquestes aplicacions es limiten a fer càlculs amb dades internes per demostrar l'objectiu sense cap mena de necessitat d'utilitzar cap classe de dada personal. D'aquesta manera, les funcionalitats de les aplicacions no es veuen afectades per cap llei.

Tot i això, s'ha de tenir en compte que es vol utilitzar el codi com a exemple per a programadors que vulguin experimentar el que faci falta. Per tant, aquest projecte si que necessita la llicència de codi obert. Les llicències de *software* són les que permeten reconèixer codi com a codi obert.

11 Sostenibilitat

En aquesta secció s'explica i es justifica la sostenibilitat del projecte. Trobarem una avaluació de la sostenibilitat del projecte separada en tres àmbits diferents: àrea ambiental, àrea econòmica i àrea social.

11.1 Sostenibilitat ambiental

11.1.1 Projecte posat en producció

Ha estimat l'impacte ambiental que tindrà la realització del projecte? T'has plantejat minimitzar l'impacte, per exemple, reutilitzant recursos? [Fita inicial]
En aquest projecte solament hi participa una persona. Una persona de mitja consumeix 0.1kWh, aquesta persona compta amb un portàtil que consumeix 0.1kWh (MacBook Pro), en total 0.2kWh. Es calcula que es pugui utilitzar un iPhone físic sobre unes 10-20 hores del projecte, tot i que és molt probable que no sigui necessari. Per ara, obviarem el consum de l'iPhone. Es dedicaran 500 hores en total al projecte, per les quals es consumiran aproximadament uns 100kW.

Durant tot el desenvolupament del projecte, es tindrà una computadora en funcionament. Els dispositius utilitzats, l'energia gastada i el paper utilitzat per imprimir la documentació seran els únics recursos utilitzats.

S'ha pensat a agafar alguns models de MPI per a fer tests i, així evitar fer més codi del necessari pel projecte. Això implica invertir menys temps en el projecte i, per tant, es necessita menys energia. També s'estalviarà en la utilització del dispositiu mòbil durant gran part del projecte, ja que m'ajudaré de simuladors per fer les comprovacions i, així, evitant gastar energia necessària. Per altra banda, l'ordinador serà necessari en tot moment per a la realització del projecte.

Has quantificat l'impacte ambiental de la realització del projecte? Quines mesures has pres per reduir l'impacte? Has quantificat aquesta reducció?

Durant el desenvolupament del projecte s'ha pres la mesura d'eliminar la utilització d'un dispositiu iPhone físic per tal de minimitzar l'impacte el màxim possible com ja havíem insinuat en un primer moment. Finalment, la duració del projecte ha estat de 800 hores. Es consumien 0.2kWh, per un total de 800 hores, surt un total de 160kWh.

Si fessis de nou el projecte, podries fer-ho amb menys recursos?

Aquest projecte es va planificar intentant utilitzar els mínims recursos possibles, per tant, si el fes de nou els recursos necessaris serien els mateixos.

11.1.2 Vida útil

Com es resol actualment el problema que vols abordar (estat de l'art)? En què millorarà ambientalment la teva solució a les existents? [Fita inicial]

Durant la vida útil dels dispositius que estan relacionats amb el projecte, podem pensar que la gent no llançarà els seus dispositius com si res, ja que a l'afegir un valor addicional, en trauran millor profit si els venen o els reutilitzen. En qualsevol dels dos casos evites que es generin més residus. En molts casos si tens un objecte, acostumes a evitar comprar-ne un altre de similar i, així, evites generar encara més residus.

Quins recursos estimes que s'usaran durant la vida útil del projecte? Quin serà l'impacte ambiental d'aquests recursos?

Per poder realitzar la computació paral·lela en iOS, es requerirà principalment alguns telèfons iOS depenent de l'aplicació a executar. Per desenvolupar aquesta aplicació també es requerirà un ordinador amb macOS per poder desenvolupar-ho correctament.

Per calcular el consum total, s'hauria de sumar el consum de l'ordinador més el consum de tots els telèfons mòbils utilitzats.

Pel manteniment del producte només serà necessari utilitzar l'ordinador cada cop que es vulgui fer modificacions en les aplicacions. Per tant, el consum total pel manteniment només tindrà en compte l'ordinador en cas que s'hagi fet qualsevol modificació.

El projecte permetrà reduir l'ús d'altres recursos? ¿Globalment, l'ús del projecte millorarà o empitjorarà la petjada ecològica?

La idea del projecte és permetre reduir l'ús d'alguns ordinadors per a certes tasques utilitzant els dispositius iOS, tot i que també pot permetre que els mateixos dispositius iOS s'afegeixin a l'ús dels ordinadors. En el cas de substituir els ordinadors per dispositius iOS dependrà del consum de cadascun per saber si millora o empitjora la petjada ecològica. Per altra banda, si els dispositius iOS es sumen als ordinadors, incrementarà clarament.

11.1.3 Riscs

Podrien produir-se escenaris que fesin augmentar la petjada ecològica del projecte?

Podria augmentar la petjada ecològica en un sol escenari. Si es volgués utilitzar una aplicació que seguís el meu concepte, però en múltiples dispositius físics, fent que aquests dispositius gastin molt més que uns simples simuladors. Això és a causa de que la gent podria tornar a utilitzar dispositius que actualment no estan gastant res, ja que els tenen abandonats.

11.2 Sostenibilitat econòmica

En aquest document s'avaluen els costos del projecte, tenint en compte tant els recursos materials com humans.

11.2.1 Projecte posat en producció

Ha estimat el cost de la realització del projecte (recursos humans i materials)? [Fita inicial]

El cost indicat a l'apartat del pressupost total d'aquest document podria ser l'únic cost invertit en el projecte, ja que l'objectiu és que comprin la idea de cara a un futur, o en cas negatiu la desestimin, de manera que no es necessitarà manteniment.

Tanmateix, és possible que es necessiti un major nombre de proves en un futur pròxim, ja que les tecnologies van avançant ràpidament i poden aparèixer noves eines per realitzar de millor forma el projecte. Un clar exemple pot ser l'arribada del 5G a la majoria dels telèfons mòbils, que farà una connexió molt més ràpida que l'actual i podria ajudar a millorar els resultats obtinguts amb les tecnologies actuals. Crear una nova versió o millorada del programa suposaria un major cost per al seu desenvolupament.

La inversión en recursos humans es podrien reduir, en el cas que no fossin necessàries totes les hores de dedicació assignades per a desenvolupar el projecte, o augmentar si es necessitessin més hores. En els recursos *software*, tot és gratuït, per tant no necessitem reduir cap cost. En el camp dels recursos *hardware* tampoc es pot reduir, ja que són mínims els recursos utilitzats. L'ordinador és necessari per a desenvolupar l'aplicació o executar els models MPI, entre d'altres.

El cost total del projecte és de 14608.89€.

Has quantificat el cost (recursos humans i materials) de la realització del projecte? Quines decisions has pres per reduir el cost? Has quantificat aquest estalvi?

El cost final del projecte ha estat de 22720.00€.

Durant el plantejament d'aquest projecte ja es va buscar estalviar pel que fa als costos del projecte i l'únic punt on hem aconseguit reduir costos és en els recursos *hardware* evitant utilitzar un iPhone de forma física. Per altra banda, hi ha hagut un augment necessari d'hores que ha produït un augment significatiu en els recursos humans fent que incrementi el pressupost total del projecte.

¿S'ha ajustat el cost previst al cost final? Has justificat les diferències (lliçons apreses)?

Els canvis de planificació han fet que el cost final estigui per sobre el cost esperat, tot i el 20% extra que teníem reservat per afegir en cas que fos necessari. Amb l'experiència d'aquest projecte, es pot determinar que és impossible controlar tots

els factors que envolten un projecte. És necessari per projectes futurs dedicar més temps a l'estimació dels costos perquè no es produeixin aquests errors.

11.2.2 Vida útil

Com es resol actualment el problema que vols abordar (estat de l'art)? En què millorarà econòmicament la teva solució a les existents? [Fita inicial]

Actualment, no existeix cap solució òptima respecte a la utilització de la MPI o cap altra tècnica de paral·lelisme en dispositius mòbils iOS. Per tant, s'ha de veure si és possible aquesta solució i, un cop fetes les comprovacions, exposar la idea als sectors que puguin estar interessats, com pot ser a la mateixa companyia Apple, per tal que incorporin o adaptin les eines necessàries en els nous dispositius mòbils per a fer-ho el més fàcil possible. Econòmicament, et pots beneficiar traient-li partit a mòbils que molta gent podria pensar a canviar o llançar, ja que tindrien una funcionalitat addicional. Si evites comprar altres dispositius que et facin la feina, estalvies grans quantitats de diners.

Quin cost estimes que tindrà el projecte durant la seva vida útil? Es podria reduir aquest cost per fer-lo més viable?

El projecte no tindrà cap cost extra durant la seva vida útil. Aquest projecte presenta una forma de fer paral·lelisme amb dispositius iOS, probablement desconeguda per una gran majoria de programadors. A partir d'aquest projecte, es vol ajudar a introduir aquesta nova idea, amb les expectatives de treure-li un gran rendiment en el futur. Aquest rendiment ha de ser a partir de les noves funcionalitats que implementin els mateixos programadors ajudant-se de tota la nostra informació però, sense cap necessitat de fer modificacions o avenços en les aplicacions del projecte.

¿S'ha tingut en compte el cost dels ajustos / actualitzacions / reparacions durant la vida útil del projecte?

Com ja s'ha comentat, és la introducció a una forma d'aplicar paral·lelisme en dispositius iOS. L'aplicació no està pensada per fer cap mena d'actualització, reparació o ajust, ja que no està destinada a la comercialització. Està pensada com a punt d'inici, on posteriorment altres programadors facin altres aplicacions similars que els hi serveixin pels seus propis beneficis.

11.2.3 Riscs

Podrien produir-se escenaris que perjudiquen la viabilitat del projecte?

La viabilitat del projecte està subjecte als canvis que es puguin realitzar en un futur a qualsevol dispositiu iOS. Aquest projecte requereix la disponibilitat dels iPhones per la computació paral·lela i, Apple sempre pot tenir l'última paraula decidint fer canvis que perjudiquin la viabilitat. Per exemple, s'ha treballat amb el *framework* Network que és molt nou i pot anar evolucionant i millorant al llarg dels pròxims anys, però també es pot decidir eliminar aquest *framework* pel motiu que sigui.

11.3 Sostenibilitat social

11.3.1 Projecte posat en producció

Què creus que et va a aportar a nivell personal la realització d'aquest projecte? [Fita inicial]

Aquest projecte m'aportarà una gran quantitat d'aspectes satisfactoris en l'àmbit personal. L'aspecte més clar que aporta aquest projecte és el coneixement. Actualment el més important és anar adquirint nous coneixements i, sobretot, si són sobre temes relativament nous i importants. Treballar amb llenguatges que mai he utilitzat com Objective-C o Swift per desenvolupar aplicacions iOS és molt interessant, ja que t'endinses en un camp completament nou. També, aprendre més exhaustivament MPI del que ja tenia certs coneixements sembla força interessant, sent MPI molt utilitzat actualment.

També hi ha altres aspectes importants no relacionats amb la informàtica com pot ser l'organització o la superació entre d'altres. Per desenvolupar un projecte com aquest, es necessita un mínim d'organització per anar desenvolupant-lo correctament i, no tenir un gran caos amb tota la informació a la qual es té accés. A més, en un projecte tan llarg com aquest, normalment apareixen certs obstacles i no sempre tens una persona al costat que et pugui ajudar a solucionar-los, necessites aprendre a superar-los.

La realització d'aquest projecte ha implicat reflexions significatives a nivell personal, professional o ètic de les persones que han intervingut?

A vegades quan surten obstacles, també surten dubtes. Sigui el camp que sigui, tots tenen les seves dificultats. De totes maneres, en el meu cas he après a superar aquests obstacles a part de gaudir una gran experiència desenvolupant un projecte que mesos enrere no m'hagués plantejat de fer. A més, gràcies a la realització d'aquest Treball de Fi de Grau he pogut aplicar una gran quantitat de coneixements adquirits al llarg del grau, i sobretot els adquirits durant l'especialitat de Tecnologies de la informació.

Un altre aspecte important és la gran quantitat d'informació que hi ha. Aquesta situació m'ha fet pensar sobre la quantitat de coneixement que em falta per adquirir en aquest camp. Abans mai m'havia plantejat desenvolupar aplicacions amb iOS, ja que tenia coneixements d'Android i em semblava més útil però, he descobert forces coses interessants sobre el funcionament de iOS i, actualment em sembla més interessant desenvolupar amb iOS que Android.

11.3.2 Vida útil

Com es resol actualment el problema que vols abordar (estat de l'art)? En què millorarà socialment (qualitat de vida) la teva solució a les existents? [Fita inicial]

Actualment utilitzem ordinadors per a l'execució de certs programes. Amb el projecte s'intenta comprovar si és factible fer-ho amb dispositius mòbils iOS. Aquest fet, facilitaria en alguns casos a la gent a l'hora de ser més pràctics, ja que un mòbil és un dispositiu molt més còmode i fàcil de portar que un ordinador, ja sigui pel pes, mida, etc. A més, pot permetre una segona oportunitat a dispositius que la gent ja tenia oblidats perquè els considerava antiquats o sense cap mena d'ús.

Existeix una necessitat real del projecte? [Fita inicial]

La necessitat de molts projectes no semblen necessaris fins que s'instauren a la societat i crec que el meu projecte podria ser similar. Cada cop els dispositius tenen noves funcionalitats que la societat no sap ni que poden arribar a existir fins que arriben al mercat. Abans d'aparèixer els telèfons intel·ligents, la majoria de gent donava un ús als mòbils de forma completament diferent de l'actual. Si aconseguíssim utilitzar els dispositius iOS per a la computació paral·lela, a més de

ser cada cop més potents, en un futur es podria arribar a prescindir dels ordinadors en certs casos en favor dels telèfons intel·ligents.

Qui es beneficiará de l'ús del projecte? Hi ha algun col·lectiu que es pot veure perjudicat pel projecte? En quina mesura?

Des del meu punt de vista, el projecte pot ser molt útil per a empreses que no vulguin destinar, per exemple, ordinadors molt potents a realitzar tasques no gaire importants, i aquí és on entrarien els dispositius mòbils que tinguin la capacitat de realitzar aquestes tasques.

En quina mesura soluciona el projecte el problema plantejat inicialment?

Per exemple, si tens un ordinador i un telèfon mòbil i reparteixes les tasques entre ells, dedicaràs menys temps que si només les executes en un ordinador i, per tant tindràs un rendiment superior. En el projecte es donen els conceptes, mitjançant exemples de formes de repartir tasques mitjançant tècniques de paral·lelisme en dispositius iOS. A partir d'aquí, els programadors interessats poden desenvolupar noves aplicacions utilitzant tots aquests conceptes pels seus propis beneficis.

11.3.3 Riscs

Podrien produir-se escenaris que fesin que el projecte fos perjudicial per a algun segment particular de la població?

Actualment, encara hi ha grans diferències entre ordinadors i telèfons intel·ligents. Tot i que el projecte pot ser una idea per casos en els quals faltin ordinadors o per substituir-los, hi ha una gran quantitat d'avantatges dels ordinadors respecte als telèfons mòbils que, ara per ara, no fa pensar en cap cas que perdin cap mena d'importància independentment del meu projecte. Es fa difícil pensar en cap cas que el projecte pugui ser perjudicial.

Podria crear el projecte algun tipus de dependència que deixés als usuaris en posició de debilitat?

Sembla bastant difícil que aquesta situació es produeixi. El motiu principal és que vull oferir una nova forma de fer computació paral·lela, fent que només programadors interessats amb el tema puguin treure-li profit. El meu codi en cap cas

és el punt de sortida de futurs projectes, simplement és la demostració que es pot fer però, cada programador desenvoluparan les seves aplicacions. A més, en cap cas s'utilitzen eines per recopilar cap mena d'informació dels usuaris en el projecte.

11.4 Autoavaluació online

És molt important quan parlem de sostenibilitat a l'hora de desenvolupar un projecte tenir un gran coneixement sobre el tema i, està clar, que no se'ns informa de la manera correcta. Hi ha un desconeixement molt gran sobre les conseqüències de desenvolupar un projecte, i més, sobre la sostenibilitat d'aquest projecte. Se'ns hauria de conscienciar sobre la gran majoria d'efectes que poden provocar. Alguns coneixements els podem tenir integrats, com fer un projecte evitant els màxims despeses possibles, però d'altra banda, la gran majoria de gent és incapaç de saber, per exemple, com es mesura l'impacte ambiental d'un projecte. En resum, ens falta coneixement respecte a un tema que és molt important en l'actualitat.

12 Conclusions

12.1 Recapitulació

S'ha volgut veure totes les opcions de les quals disposa Apple per aplicar paral·lelisme en dispositius iOS. Tant les eines d'Apple utilitzades com utilitzar MPI té els seus avantatges i desavantatges.

La computació paral·lela en dispositius iOS no és molt reconeguda actualment i, al llarg del projecte s'ha anat trobant certs inconvenients que confirmaven els possibles motius.

El primer inconvenient que es va trobar és la utilització de MPI en iOS que no és una opció recomanable, ja que obliga a fer modificacions en el dispositiu. Entre aquestes modificacions es troba la més important que és el *jailbreak*, on es va acabar concloent que no era gens recomanable realitzar-lo. No només pels motius explicats sobre la deteriorització del propi *jailbreak* sinó, pel fet que els nous models d'iPhone tampoc l'admeten. Al no suportar el *jailbreak* els nous models, estan obligats a

instal·lar versions més antigues del sistema operatiu en els dispositius, una acció una mica extranya, ja que l'objectiu del projecte era mostrar les eines més útils de cara al present i futur i en cap cas fer retrocedir als usuaris de iOS.

A partir d'aquest inconvenient, es van prendre decisions a l'hora de triar una forma d'aconseguir la computació paral·lela en dispositius iOS. L'opció més clara per desenvolupar l'aplicació eren els *sockets*. Posteriorment, es va trobar el *framework* Network que era una eina molt recent d'Apple i la presentaven com una alternativa moderna als *sockets*. Es va fer anàlisi dels possibles avantatges i inconvenients de Network respecte als *sockets* però, finalment es va decidir utilitzar Network. Tot i que actualment no hi ha gaire informació respecte al nou *framework* d'Apple, gràcies a la documentació d'Apple es va aconseguir fer una implementació correctament.

Del projecte es destaquen dues aplicacions que realitzen una multiplicació entre matrius, sent una la versió millorada de l'altra.

La primera aplicació és la multiplicació de matrius en múltiples hosts. Per fer aquesta aplicació, es va necessitar l'eina Xcode junt amb el simulador que ja porta integrat. A més d'utilitzar el *framework* Network. Aquesta aplicació es basa en el model mestre-esclau que divideix un problema en subproblemes. En l'aplicació, el mestre fa la divisió en files d'una de les matrius i, posteriorment, les envia als tres esclaus. Aquests esclaus resolen els subproblemes corresponents i envien els resultats de la multiplicació al mestre. Finalment, el mestre junta els resultats restants en la matriu resultant.

La segona aplicació és la multiplicació de matrius en múltiples hosts amb GCD. GCD és una tecnologia d'Apple que permet aplicar concurrència en el llenguatge Swift. Aquesta aplicació té la finalitat de mostrar totes les eines d'Apple juntes en la mateixa aplicació, per millorar el rendiment al màxim possible. Per desenvolupar-la, es parteix de la primera aplicació però, utilitzant també GCD. La principal diferència respecte a la primera aplicació es troba en la multiplicació de les matrius per parts dels esclaus. En aquesta versió es realitza concurrentment per aconseguir la màxima millora del rendiment possible.

12.2 Conclusions dels objectius

- **Analitzar en quin estat es troba la computació paral·lela i distribuïda en els dispositius iOS.**

  Abans de començar a entrar amb dissenys i implementacions, era necessari estudiar molt a fons l'estat en el qual es trobava la computació paral·lela i distribuïda en els dispositius iOS. Amb aquest estudi, es va descobrir una sèrie d'eines per aplicar paral·lelisme. Un cop es veu que és possible la computació paral·lela en dispositius iOS, l'objectiu es considera assolit.

- **Analitzar, dissenyar i implementar l'entorn de treball.**

  Primer era necessari determinar tot el *software* necessari a instal·lar en el dispositiu macOS per desenvolupar el projecte correctament. El primer pas important, era triar quina implementació de MPI s'utilitzaria. Amb la implementació de MPI triada, era necessari escollir un model de MPI que apliqués paral·lelisme. Aquest model és el que posteriorment s'havia de traspassar a l'aplicació a desenvolupar per dispositius iOS. També era necessari configurar tot l'entorn per la creació i testeig d'aplicacions iOS, inclosa la configuració dels simuladors. A més, era important establir un entorn d'execució de programes MPI en macOS per fer les proves que fossin necessàries. En el moment que es pot executar programes MPI en macOS i desenvolupar qualsevol aplicació per iOS en l'entorn desitjat, es considera que s'ha assolit l'objectiu.

- **Analitzar, dissenyar l'aplicació per iOS.**

  En l'aplicació per dispositius iOS, era necessari un estudi previ per establir com fer l'estructura i la seva implementació. Com es va determinar de fer l'aplicació amb eines d'Apple, s'havia de trobar una opció per utilitzar MPI dintre de l'aplicació sense realitzar un *jailbreak* o buscar altres eines que substituïssin MPI. Un altre aspecte a analitzar, eren les eines que utilitzava Apple per aplicar concurrència en les seves aplicacions. L'últim aspecte era analitzar l'entorn de desenvolupament de l'aplicació Xcode, per tal de dominar totes les característiques particulars i poder tenir el màxim coneixement possible a l'hora de desenvolupar l'aplicació. Un cop es decideix

utilitzar el *framework* Network en substitució de MPI, l'ús de GCD per a la concurrència en iOS i, l'anàlisi de Xcode s'ha realitzat en profunditat, l'objectiu es considera assolit.

- **Implementar l'aplicació per iOS que apliqui tècniques de paral·lelisme.**
  Per realitzar l'aplicació, es van utilitzar totes les eines i coneixements adquirits prèviament. Durant el desenvolupament, es va utilitzar els coneixements adquirits sobre el *framework* Network i GCD i, sobre l'entorn de Xcode. L'objectiu era aplicar tècniques de paral·lelisme i, per tant, en el moment que s'aconsegueix aplicar el model MPI seleccionat en una aplicació iOS per a múltiples dispositius iOS a més d'afegir-li GCD, l'objectiu passa a considerar-se assolit.

- **Analitzar el model de programació paral·lela i distribuïda iOS des del punt de vista d'un programador de MPI.**
  Aquest objectiu tenia com a finalitat explicar a un programador expert en MPI com es diferencia la programació amb MPI de la programació paral·lela en iOS. Per complir aquest objectiu, era necessari veure totes les diferències entre les dues formes de programar en paral·lel i, sobretot, veure quins avantatges té la proposta que es presenta respecte MPI. Quan s'acaba de desenvolupar l'aplicació i funciona correctament, es té tota la informació necessària per veure les facilitats i les dificultats que pot suposar passar d'una forma de programar a l'altra. Un cop s'explica detalladament totes les característiques de la programació paral·lela per iOS amb les comparacions necessàries per facilitar als programadors de MPI entendre millor el funcionament d'aquesta nova proposta, aquest objectiu es considera assolit.

12.3 Implementacions de cara al futur

Ara mateix, fer aplicacions que utilitzin Network amb dispositius iOS i macOS barrejats no és del tot fàcil. El *framework* Network té suport per les dues plataformes però, no s'ha de configurar les connexions entre el mestre i l'esclau de la mateixa manera.

Volia comprovar si era possible executar l'aplicació desenvolupada en iOS en l'entorn de macOS sense fer cap modificació a excepció, de configurar el codi per a l'execució en macOS. Cal tenir en compte que les aplicacions no s'executen igual en iOS que en macOS i per tant s'ha de fer una configuració previa dins de l'entorn de Xcode.

Tot i que Network és compatible amb els dos sistemes operatius, a l'intentar executar l'aplicació desenvolupada per a macOS no funciona correctament. No dóna errors de compilació a causa de la compatibilitat de Network amb macOS però, no s'executa correctament.

De cara al futur pot ser interessant un estudi per veure les diferències de comportament de Network depenent de si treballes amb iOS o macOS. A primera vista, sembla que no tingui molt sentit utilitzar Network per aplicacions de macOS, ja que actualment existeix MPI i pel que hem anat veient al llarg del projecte, sembla més fàcil d'entendre que Network. Tot i això, s'ha de tenir en compte que Network és una eina que ofereix Apple, és molt nova i pot anar rebent actualitzacions que facilitin encara més la seva implementació. A més, si s'aconsegueix establir connexions amb Network entre dispositius iOS i macOS també poden sorgir utilitats força interessants. Actualment els ordinadors encara són més potents que els dispositius mòbils i si distribuïm les tasques més pesades en dispositius encara més potents que els dispositius mòbils, es pot arribar aconseguir grans millores de rendiment.

També, de cara al futur s'haurien d'introduir noves funcionalitats en el *framework* Network si el que es vol és crear una alternativa als *sockets* força més útil. Els passos a seguir per establir les connexions no es diferencien dels *sockets*, per tant s'han de buscar altres maneres de millorar el *framework*. Una opció molt interessant seria afegir altres funcions o alguns paràmetres nous a les funcions actuals, ja que al comparar Network amb MPI s'ha vist que hi ha una gran varietat de funcions MPI que no es poden trobar en el *framework* Network, com per exemple en l'enviament i rebuda de dades. En MPI tens diferents formes de tractar les dades que vols enviar, fent que et permeti tenir més flexibilitat a l'hora de treballar de forma síncrona o asíncrona.

12.4 Conclusions finals

Un cop s'ha finalitzat el projecte, es pot concloure que encara queda molt camí a recórrer per millorar les tècniques de paral·lelització per a dispositius iOS, a part de facilitar més eines en els mateixos dispositius per poder treballar de la manera més senzilla i clara possible. Això és a causa de que existeixen una sèrie d'eines de gran utilitat per desenvolupar aplicacions amb paral·lelisme però, no són suportades pel dispositius iOS i, per tant, s'ha de treballar amb les eines d'Apple on algunes són força recents i els hi falta informació.

De tota manera, es pot dir que els objectius proposats a l'hora de desenvolupar el projecte, han estat complerts amb èxit tot i la desviació d'hores establerta inicialment. Sobretot, gràcies a la gran quantitat de temps dedicat a la cerca d'informació relacionada amb el projecte i, posteriorment, al desenvolupament de les aplicacions.

Aquest projecte ha resultat una molt bona eina d'aprenentatge. En l'àmbit acadèmic, m'ha servit per aprendre una gran quantitat d'informació de la programació d'aplicacions Apple, tant per iOS com per macOS, incloent dos llenguatges de programació i un *framework*, anomenat Network, molt recent però que sembla que tindrà força futur. També he après sobre temes que no estan relacionats amb Apple com MPI, sobre el qual gira tot el projecte. També m'ha servit per anar una mica més enllà i descobrir camps recents i força interessants, els quals no hagués après en la mateixa universitat i possiblement per voluntat pròpia. En l'àmbit professional m'ha servit per aprendre a gestionar correctement els recursos, la càrrega de treball, les dificultats, etc., de cara a treballar en grans projectes.

13 Referències

1. MPI Forum. (n.d.). Recuperat 24 d'Abril, 2019, de https://www.mpi-forum.org
2. Kranzlmüller, D. ; Toja, A. M. (2011). Information and communication on technology for the fight against global warming : first International Conference, ICT-GLOW 2011, Proceedings (Vol. 6868 LNCS), 1-9.

3. Dongarra, J. ; Luszczek, P. (2012). Anatomy of a globally recursive embedded LINPACK benchmark. 2012 IEEE Conference on High Performance Extreme Computing, HPEC 2012, (January 2016).
4. Github. (n.d.). Recuperat 21 de Febrer, 2019, de https://github.com/
5. Apple Developer Documentation. (n.d.). Recuperat 1 d'Abril, 2019, de https://developer.apple.com/documentation/
6. Online Gantt Chart Software | TeamGantt. (n.d.). Recuperat 28 de Febrer, 2019, de https://www.teamgantt.com/
7. How to Jailbreak. (2019). Recuperat 1 d'Abril, 2019, de https://www.idownloadblog.com/jailbreak/
8. Untethered Jailbreak vs. Tethered Jailbreak vs. SemiTethered Jailbreak — What's the Difference? (2011). Recuperat 1 d'Abril, 2019, de https://www.idownloadblog.com/2011/10/22/untethered-jailbreak-vs-tethered-jailbreak-vs-semitethered-jailbreak/
9. Freeman, J. (n.d.). Table of Contents (saurik). Recuperat 20 de Maig, 2019, de http://www.saurik.com
10. El jailbreak en iOS ya no tiene sentido, así es cómo Apple y otras compañías han acabado con él. (2017). Recuperat 1 d'Abril, 2019, de https://www.applesfera.com/ios/el-jailbreak-en-ios-ya-no-tiene-sentido-asi-es-como-apple-y-otras-companias-han-acabado-con-el
11. MPICH | High-Performance Portable MPI. (n.d.). Recuperat 1 de Març, 2019, de https://www.mpich.org/
12. Web pages for MPI Routines. (n.d.). Recuperat 15 d'Abril, 2019, de http://mpi.deino.net/mpi_functions/index.htm
13. Objective-C Tutorial. (n.d.). Recuperat 21 de Març, 2019, de https://www.tutorialspoint.com/objective_c/
14. Swift Tutorial. (n.d.). Recuperat 25 de Març, 2019, de https://www.tutorialspoint.com/swift/index.htm
15. Medium – a place to read and write big ideas and important stories. (n.d.). Recuperat 27 de Març, 2019, de https://medium.com/
16. Grand Central Dispatch: Multi-Threading With Swift – LearnAppMaking. (2019). Recuperat 1 de Juny, 2019, de https://learnappmaking.com/grand-central-dispatch-swift/

17. Swift Multi-Threading using GCD (n.d.). Recuperat 1 de Juny, 2019, de https://hackernoon.com/swift-multi-threading-using-gcd-for-beginners-2581b7aa21cb
18. Dispatch Groups in Swift 3. (n.d.). Recuperat 1 de Juny, 2019, de http://jordansmith.io/dispatch-groups-in-swift-3/
19. Attention Required! | Cloudflare. (n.d.). Recuperat 10 de Maig, 2019, de https://code.tutsplus.com/es/tutorials/ios-from-scratch-with-swift-data-persistence-and-sandboxing-on-ios--cms-25505
20. Pacheco, P. S. (1997). Parallel Programming with MPI. Performance Computing.
21. ElEnin, S. A. ; ElSoud, M. A. (2011). Evaluation of Matrix Multiplication on an MPI Cluster.
22. Nakkeeran, M. ; Chandrasekaran, R. M. (2011). Parallel matrix multiplication implementation in distributed environment through RMI.
23. Büsching, F. ; Schildt, S. ; Wolf, L. (2012). DroidCluster: Towards smartphone cluster computing - The streets are paved with potential computer clusters. Proceedings - 32nd IEEE International Conference on Distributed Computing Systems Workshops, ICDCSW 2012, 114–117.
24. Doolan, D. C. ; Tabirca, S. ; Yang, L. T. (2009). MMPI a message passing interface for the mobile environment, 1-10.
25. Attia, D. E. ; ElKorany, A. M. ; Moussa, A. S. (2016). High Performance Computing Over Parallel Mobile Systems. International Journal of Advanced Computer Science and Applications, 7(9), 99–103.
26. Ilgner, R. G. ; Davidson, D. B. (2015). The computational performance and power consumption of the parallel FDTD on a smartphone platform. Applied Computational Electromagnetics Society Journal, 30(12), 1262–1268.
27. Guillot, B. J. ; Chapman, B. ; Pâris, J.-F. (2004). Building a large low-cost computer cluster with unmodified Xboxes, 1–16.

Annex I: Implementació final de l'aplicació

Codi de suport pel mestre i els esclaus:

```
class Connection {

    init(nwConnection: NWConnection) {
        self.nwConnection = nwConnection
        self.id = Connection.nextID
        Connection.nextID += 1
        self.recv_msg = []
    }

    private static var nextID: Int = 0

    let nwConnection: NWConnection
    let id: Int
    var recv_msg: Array<UInt32>

    var didStopCallback: ((Error?) -> Void)? = nil

    func start() {
        print("connection \(self.id) will start")
        self.nwConnection.stateUpdateHandler = self.stateDidChange(to:)
        self.setupReceive()
        self.nwConnection.start(queue: .main)
    }

    func send(data: Data) {
        self.nwConnection.send(content: data, completion: .contentProcessed( { error in
            if let error = error {
                self.connectionDidFail(error: error)
                return
            }
            //print("connection \(self.id) did send, data: \(data as NSData)")
            //let msg = String(data: data, encoding: String.Encoding.utf8) ?? "Data could not be printed"
            print("connection \(self.id) did send, data: \(data as NSData)")
        }))
    }

    func receive_msg_Server() {
        self.setupReceiveServer()
    }

    func stop() {
        print("connection \(self.id) will stop")
    }
```

```
private func setupReceive() {
    self.nwConnection.receive(minimumIncompleteLength: 1, maximumLength: 65536) { (data, _, isComplete,
        error) in
        if let data = data, !data.isEmpty {
            print("connection \(self.id) did receive, data: \(data as NSData)")
            //let text = String(data: data, encoding: String.Encoding.utf8) ?? "Data could not be
                printed"
            //print("connection \(self.id) did receive, data: \(text)")
            var arr2 = Array<UInt32>(repeating: 0, count: data.count/MemoryLayout<UInt32>.stride)
            _ = arr2.withUnsafeMutableBytes { data.copyBytes(to: $0) }
            //print("arr2.count: \(arr2.count)")
            for i in 0 ..< arr2.count {
                //print("Printing recieved array: \(arr2[i])")
                self.recv_msg.append(arr2[i]);
            }
        }
        if isComplete {
            self.connectionDidEnd()
        } else if let error = error {
            self.connectionDidFail(error: error)
        } else {
            self.setupReceive()
        }
    }
}

    private func setupReceiveServer() {
        self.nwConnection.receive(minimumIncompleteLength: 1, maximumLength: 65536) { (data, _, isComplete,
            error) in
            if let data = data, !data.isEmpty {
                print("connection \(self.id) did receive, data: \(data as NSData)")
                //let text = String(data: data, encoding: String.Encoding.utf8) ?? "Data could not be
                    printed"
                //print("connection \(self.id) did receive, data: \(text)")
                var arr2 = Array<UInt32>(repeating: 0, count: data.count/MemoryLayout<UInt32>.stride)
                _ = arr2.withUnsafeMutableBytes { data.copyBytes(to: $0) }
                //print("arr2.count: \(arr2.count)")
                for i in 0 ..< arr2.count {
                    //print("Printing recieved array: \(arr2[i])")
                    self.recv_msg.append(arr2[i]);
                }
            }
            if isComplete {
                self.connectionDidEnd()
            } else if let error = error {
                self.connectionDidFail(error: error)
            } else {
                self.setupReceive()
            }
        }
    }
}
```

```swift
private func stateDidChange(to state: NWConnection.State) {
    switch state {
    case .setup:
        break
    case .waiting(let error):
        self.connectionDidFail(error: error)
    case .preparing:
        break
    case .ready:
        print("connection \(self.id) ready")
    case .failed(let error):
        self.connectionDidFail(error: error)
    case .cancelled:
        break
    default:
        break
    }
}

private func connectionDidFail(error: Error) {
    print("connection \(self.id) did fail, error: \(error)")
    self.stop(error: error)
}

private func connectionDidEnd() {
    print("connection \(self.id) did end")
    self.stop(error: nil)
}

private func stop(error: Error?) {
    self.nwConnection.stateUpdateHandler = nil
    self.nwConnection.cancel()
    if let didStopCallback = self.didStopCallback {
        self.didStopCallback = nil
        didStopCallback(error)
    }
}
```

Codi del mestre:

```
class Server {

    init() {
        self.listener = try! NWListener(using: .tcp, on: 12548)
        self.timer = DispatchSource.makeTimerSource(queue: .main)
        self.timer_kill = DispatchSource.makeTimerSource(queue: .main)

        self.Matriu1 = [[UInt32]](repeating: [UInt32](repeating: 0, count: 3), count: 3)
        self.Matriu2 = [[UInt32]](repeating: [UInt32](repeating: 0, count: 3), count: 3)
        self.c1 = [[UInt32]](repeating: [UInt32](repeating: 0, count: 3), count: 3)
        self.c2 = [[UInt32]](repeating: [UInt32](repeating: 0, count: 3), count: 3)
        self.c3 = [[UInt32]](repeating: [UInt32](repeating: 0, count: 3), count: 3)
        self.resultant = [[UInt32]](repeating: [UInt32](repeating: 0, count: 3), count: 3)
        self.connections = []
    }

    let listener: NWListener
    let timer: DispatchSourceTimer
    let timer_kill: DispatchSourceTimer

    var Matriu1: Array<Array<UInt32>>
    var Matriu2: Array<Array<UInt32>>
    var c1: Array<Array<UInt32>>
    var c2: Array<Array<UInt32>>
    var c3: Array<Array<UInt32>>
    var resultant: Array<Array<UInt32>>

    var connections: [Connection]


    func start() throws {
        print("server will start")
        self.listener.stateUpdateHandler = self.stateDidChange(to:)
        self.listener.newConnectionHandler = self.didAccept(nwConnection:)
        self.listener.start(queue: .main)

        self.timer_kill.setEventHandler(handler: self.stop)
        self.timer_kill.schedule(deadline: .now() + 180.0)
        self.timer_kill.activate()
    }
```

```swift
func stateDidChange(to newState: NWListener.State) {
    switch newState {
    case .setup:
        break
    case .waiting:
        break
    case .ready:
        print("Listener New State Ready")
        break
    case .failed(let error):
        print("server did fail, error: \(error)")
        self.stop()
    case .cancelled:
        break
    default:
        break
    }
}
```

```
private var connectionsByID: [Int: Connection] = [:]

private func didAccept(nwConnection: NWConnection) {
    let connection = Connection(nwConnection: nwConnection)

    self.connections.append(connection)
    print("self.connections: \(self.connections)")
    self.connectionsByID[connection.id] = connection
    connection.didStopCallback = { _ in
        self.connectionDidStop(connection)
    }
    connection.start()
    print("server did open connection \(connection.id)")

    //Inicialitza les matrius
    self.init_matrices()

    //Envia les matrius
    self.send_msg(c1: self.c1, c2: self.c2, c3: self.c3)

    //Es prepara per rebre la resposta
    connection.receive_msg_Server()

    //Espera la resposta
    DispatchQueue.global(qos: .background).async {
        while (connection.recv_msg.count < 3) {
            //print("connection.recv_msg.count: \(connection.recv_msg.count)")
        }
        //Obtenim els resultats de la connexio
        self.matrix_results(connection: connection)

        //Print matriu resultant
        self.print_matrix(matrix: self.resultant)

        print("Final de la feina del servidor")
    }
}

private func connectionDidStop(_ connection: Connection) {
    self.connectionsByID.removeValue(forKey: connection.id)
    print("server did close connection \(connection.id)")
}
```

```
private func stop() {
    self.listener.stateUpdateHandler = nil
    self.listener.newConnectionHandler = nil
    self.listener.cancel()
    for connection in self.connectionsByID.values {
        connection.didStopCallback = nil
        connection.stop()
    }
    self.connectionsByID.removeAll()
    self.timer.cancel()
    //self.timer_kill.cancel()
}

private func init_matrices(){
    var cont : UInt32 = 1
    //Matriu1
    for i in 0 ..< self.Matriu1.count {
        for j in 0 ..< self.Matriu1[i].count {
            if(i == 0){
                self.c1[i][j] = cont
            }
            else if(i == 1){
                self.c2[i][j] = cont
            }
            else { //i == 2
                self.c3[i][j] = cont
            }
            self.Matriu1[i][j] = cont;
            //print("Matriu 1: \(self.Matriu1[i][j])")
            cont+=1
        }
    }
    //Matriu2
    for i in 0 ..< self.Matriu2.count {
        for j in 0 ..< self.Matriu2[i].count {
            cont-=1
            self.Matriu2[i][j] = cont
            //print("Matriu 2: \(self.Matriu2[i][j])")
        }
    }
}
```

```
private func send_msg(c1:Array<Array<UInt32>>, c2:Array<Array<UInt32>>, c3:Array<Array<UInt32>>) {
    self.print_size()
    //let size = self.connectionsByID.count
    //let timestamp = Date()
    //print("server send_msg, timestamp: \(timestamp)")

    //Per cada connexió del servidor enviem els paràmetres de la matriu
    for connection in self.connectionsByID.values {
        //let data = "send_msg, connection: \(connection.id), timestamp: \(timestamp)\r\n"
        print("CONNECTIONS : \(connection.id)")
        //let data = "Hello from Server"
        //connection.send(data: Data(data.utf8))
        if(self.connectionsByID.count == 3){
            if(connection.id == 0){
                //Enviem c1
                for i in 0 ..< c1.count{
                    //print("c1: \(c1[i])")
                    let data = Data(buffer: UnsafeBufferPointer(start: c1[i], count: c1[i].count))
                    connection.send(data: Data(data))
                }
            }
            if(connection.id == 1){
                //Enviem c2
                for i in 0 ..< c2.count{
                    //print("c2: \(c2[i])")
                    let data = Data(buffer: UnsafeBufferPointer(start: c2[i], count: c2[i].count))
                    connection.send(data: Data(data))
                }
            }
            if(connection.id == 2){
                //Enviem c3
                for i in 0 ..< c3.count{
                    //print("c3: \(c3[i])")
                    let data = Data(buffer: UnsafeBufferPointer(start: c3[i], count: c3[i].count))
                    connection.send(data: Data(data))
                }
            }

            //Enviem Matriu2
            for i in 0 ..< Matriu2.count{
                //print("Matriu2: \(Matriu2[i])")
                let data = Data(buffer: UnsafeBufferPointer(start: Matriu2[i], count: Matriu2[i].count))
                connection.send(data: Data(data))
            }
        }
    }
}
```

```
private func matrix_results(connection:Connection) {
    print("matrix results")
    if(connection.id == 0 && connection.recv_msg.count == 3){
        print("connection.id == 0: \(connection.id)")
        print("connection.id == 0: \(connection.recv_msg)")
        //Rebem c1
        for j in 0 ..< connection.recv_msg.count{
            self.resultant[connection.id][j] = connection.recv_msg[j]
        }
        connection.recv_msg = []
    }
    if(connection.id == 1 && connection.recv_msg.count == 3){

        print("connection.id == 1: \(connection.id)")
        print("connection.id == 1: \(connection.recv_msg)")
        //Rebem c2
        for j in 0 ..< connection.recv_msg.count{
            self.resultant[connection.id][j] = connection.recv_msg[j]
        }
        connection.recv_msg = []
    }
    if(connection.id == 2 && connection.recv_msg.count == 3){
        //Rebem c3
        print("connection.id == 2: \(connection.id)")
        print("connection.id == 2: \(connection.recv_msg)")
        for j in 0 ..< connection.recv_msg.count{
            self.resultant[connection.id][j] = connection.recv_msg[j]
        }
        connection.recv_msg = []
    }
}

private func print_matrix(matrix:Array<Array<UInt32>>){
    for i in 0 ..< matrix.count{
        for j in 0 ..< matrix[i].count{
            print("matrix final en el servidor: \(matrix[i][j])")
        }
    }
}

private func print_size() {
    print("SIZE : \(self.connectionsByID.count)")
}
```

```
    static func run() {
        let listener = Server()
        try! listener.start()
        //dispatchMain()
    }
}
```

Codi dels esclaus:

```
class Client {

    init() {
        let nwConnection = NWConnection(host: "192.168.1.39", port: 12548, using: .tcp)
        self.connection = Connection(nwConnection: nwConnection)
        self.timer = DispatchSource.makeTimerSource(queue: .main)
        self.matr = []
        self.Matriu1 = []
        self.Matriu2 = []
    }

    let connection: Connection
    let timer: DispatchSourceTimer
    var matr: Array<Array<UInt32>>
    var Matriu1: Array<Array<UInt32>>
    var Matriu2: Array<Array<UInt32>>

    func start() {
        self.connection.didStopCallback = self.didStopCallback(error:)
        self.connection.start()

        DispatchQueue.global(qos: .background).async {
            while (self.connection.recv_msg.count < 18) {
                //print("self.connection.recv_msg.count: \(self.connection.recv_msg.count)")
            }
            self.tractament_dades()
            self.send_msg2()
            print("El client ja ha fet la seva feina!!")
        }
    }

    func didStopCallback(error: Error?) {
        if error == nil {
            exit(EXIT_SUCCESS)
        } else {
            exit(EXIT_FAILURE)
        }
    }
```

```swift
//Envia missatge des del client
func send_msg2() {
    //let timestamp = Date()
    //print("client send_msg, timestamp: \(timestamp)")
    for i in 0 ..< self.matr.count {
        if(self.matr[i][i] != 0){
            let data = Data(buffer: UnsafeBufferPointer(start: self.matr[i], count: self.matr[i].count))
            connection.send(data: Data(data))
        }
    }
}

//Multiplicació de matrius amb les dues matrius que ens han enviat
func mult_matrices(Matriu1:Array<Array<UInt32>>, Matriu2:Array<Array<UInt32>>) -> Array<Array<UInt32>>{

    var res = [[UInt32]](repeating: [UInt32](repeating: 0, count: 3), count: 3)

    //Multiplicació Concurrent
    let group = DispatchGroup()
    group.enter()

    DispatchQueue.main.async {
        DispatchQueue.concurrentPerform(iterations: Matriu1.count) { index in
            DispatchQueue.concurrentPerform(iterations: Matriu1.count) { index2 in
                DispatchQueue.concurrentPerform(iterations: Matriu2.count) { index3 in
                    res[index][index2] += Matriu1[index][index3] * Matriu2[index3][index2]
                }
            }
        }
        group.leave()
    }
    group.notify(queue: .main) {
        print("Res: \(res)")
    }
    return res
}
```

```
    private func tractament_dades() {

        var i = 2
        //Omplim Matriu2
        while i < self.connection.recv_msg.count/2 && self.connection.recv_msg.count/2 > 2 {

                Matriu1.append([self.connection.recv_msg[i-2],self.connection.recv_msg
                [i-1],self.connection.recv_msg[i]])
            i+=3
        }
        //Omplim Matriu2
        while i >= self.connection.recv_msg.count/2 && i < self.connection.recv_msg.count &&
            self.connection.recv_msg.count/2 > 2 {

                Matriu2.append([self.connection.recv_msg[i-2],self.connection.recv_msg
                [i-1],self.connection.recv_msg[i]])
            i+=3
        }

        //Multiplicació de matrius
        self.matr = self.mult_matrices(Matriu1: Matriu1, Matriu2: Matriu2)

        //Print Matriu Multiplicada
        for i in 0 ..< self.matr.count {
            for j in 0 ..< self.matr[i].count {
                print("Res (Multiplicada) - Client: \(self.matr[i][j])")
            }
        }
    }

    static func run() {
        let client = Client()
        client.start()
        //dispatchMain()
    }
}
```

Annex II: Resultats de l'execució

Execució del mestre:

Swift_Networking_3!

```
server will start
Listener New State Ready
```

Swift_Networking_3!

```
server will start
Listener New State Ready
self.connections:
    [Swift_Networking_3_.ViewController.Connection]
connection 0 will start
server did open connection 0
SIZE : 1
CONNECTIONS : 0
connection 0 ready
```

Swift_Networking_3!

```
server will start
Listener New State Ready
self.connections:
    [Swift_Networking_3_.ViewController.Connection]
connection 0 will start
server did open connection 0
SIZE : 1
CONNECTIONS : 0
connection 0 ready
self.connections:
    [Swift_Networking_3_.ViewController.Connection,
    Swift_Networking_3_.ViewController.Connection]
connection 1 will start
server did open connection 1
SIZE : 2
CONNECTIONS : 1
CONNECTIONS : 0
connection 1 ready
```

```
server will start
Listener New State Ready
self.connections:
    [Swift_Networking_3_.ViewController.Connection]
connection 0 will start
server did open connection 0
SIZE : 1
CONNECTIONS : 0
connection 0 ready
self.connections:
    [Swift_Networking_3_.ViewController.Connection,
    Swift_Networking_3_.ViewController.Connection]
connection 1 will start
server did open connection 1
SIZE : 2
CONNECTIONS : 0
CONNECTIONS : 1
connection 1 ready
self.connections:
    [Swift_Networking_3_.ViewController.Connection,
    Swift_Networking_3_.ViewController.Connection,
    Swift_Networking_3_.ViewController.Connection]
connection 2 will start
server did open connection 2
SIZE : 3
CONNECTIONS : 0
CONNECTIONS : 2
CONNECTIONS : 1
connection 2 ready
```

```
connection 0 did send, data: <01000000 02000000 03000000>
connection 0 did send, data: <00000000 00000000 00000000>
connection 0 did send, data: <00000000 00000000 00000000>
connection 0 did send, data: <09000000 08000000 07000000>
connection 0 did send, data: <06000000 05000000 04000000>
connection 0 did send, data: <03000000 02000000 01000000>
connection 2 did send, data: <00000000 00000000 00000000>
connection 2 did send, data: <00000000 00000000 00000000>
connection 2 did send, data: <07000000 08000000 09000000>
connection 2 did send, data: <09000000 08000000 07000000>
connection 2 did send, data: <06000000 05000000 04000000>
connection 2 did send, data: <03000000 02000000 01000000>
connection 1 did send, data: <00000000 00000000 00000000>
connection 1 did send, data: <04000000 05000000 06000000>
connection 1 did send, data: <00000000 00000000 00000000>
connection 1 did send, data: <09000000 08000000 07000000>
connection 1 did send, data: <06000000 05000000 04000000>
connection 1 did send, data: <03000000 02000000 01000000>
connection 0 did receive, data: <1e000000 18000000 12000000>
connection 1 did receive, data: <54000000 45000000 36000000>
connection 2 did receive, data: <8a000000 72000000 5a000000>
matrix final en el servidor: 30
matrix final en el servidor: 24
matrix final en el servidor: 18
matrix final en el servidor: 84
matrix final en el servidor: 69
matrix final en el servidor: 54
matrix final en el servidor: 138
matrix final en el servidor: 114
matrix final en el servidor: 90
Final de la feina del servidor
```

Execució de l'esclau 1:

Networking_client_1

```
connection 0 will start
connection 0 ready
```

```
connection 0 will start
connection 0 ready
connection 0 did receive, data: <01000000 02000000 03000000
    00000000 00000000 00000000 00000000 00000000 00000000
    09000000 08000000 07000000 06000000 05000000 04000000
    03000000 02000000 01000000>
Res: [[30, 24, 18], [0, 0, 0], [0, 0, 0]]
Res (Multiplicada) - Client: 30
Res (Multiplicada) - Client: 24
Res (Multiplicada) - Client: 18
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
El client ja ha fet la seva feina!!
connection 0 did send, data: <1e000000 18000000 12000000>
connection 0 did end
```

Execució de l'esclau 2:

Networking_client_2

```
connection 0 will start
connection 0 ready
```

```
connection 0 will start
connection 0 ready
connection 0 did receive, data: <00000000 00000000 00000000
    04000000 05000000 06000000 00000000 00000000 00000000
    09000000 08000000 07000000 06000000 05000000 04000000
    03000000 02000000 01000000>
Res (Multiplicada) - Client: 0
Res: [[0, 0, 0], [84, 69, 54], [0, 0, 0]]
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 84
Res (Multiplicada) - Client: 69
Res (Multiplicada) - Client: 54
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
El client ja ha fet la seva feina!!
connection 0 did send, data: <54000000 45000000 36000000>
connection 0 did end
```

Execució de l'esclau 3:

```
connection 0 ready
connection 0 did receive, data: <00000000 00000000
    00000000 00000000 00000000 00000000 07000000 08000000
    09000000 09000000 08000000 07000000 06000000 05000000
    04000000 03000000 02000000 01000000>
Res: [[0, 0, 0], [0, 0, 0], [138, 114, 90]]
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 0
Res (Multiplicada) - Client: 138
Res (Multiplicada) - Client: 114
Res (Multiplicada) - Client: 90
El client ja ha fet la seva feina!!
connection 0 did send, data: <8a000000 72000000 5a000000>
connection 0 did end
```